Ian Stroud

Boundary Representation Modelling Techniques

Springer

Boundary Representation Modelling Techniques

Ian Stroud

Boundary Representation Modelling Techniques

Springer

Ian Stroud (EPFL), STI-IPR-LICP
Ecole Polytechnique Federale de Lausanne
Lausanne 1015
Switzerland
ian.stroud@epfl.ch

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Control Number: 2005938356

ISBN-10: 0-387-84628-312-4
ISBN-13: 978-1-84628-312-3

Printed on acid-free paper.

Printed in the United States of America. (MVY)

9 8 7 6 5 4 3 2 1

springer.com

This book is dedicated to my wife and family, as well as to numerous colleagues past and present.

Preface

This book is concerned with aspects of boundary representation (B-rep) solid modelling, describes several algorithms illustrating both general principles of modelling algorithms and their versatility and outlines the general principles behind the development of such algorithms and the extensions for handling information in models. It goes on to describe feature modelling, graphics, model input and output, and applications. It represents a collection of work partly by the author but also by many other people whose results are described.

The purpose of the book is not to give a general introduction to solid modelling. A variety of techniques are available and to go into detail about each one would require much more space than is reasonable. This book attempts to cover basic techniques for one branch. It is not intended to be a review of commercial systems nor to describe any one system. The aim is to describe the techniques behind different systems.

The first two chapters describe the background to solid and product modelling and what has already been done. Chapter 1 gives a brief outline of computer modelling, and chapter 2 describes in more detail the background to B-rep modelling specifically, because this forms the basis for the book.

Chapter 3 describes an idealised, or recommended B-rep modelling environment, the basic operations or tools that provide a basis for the work described in the book. This idealised representation can be implemented in several ways, so the chapter first describes the various options and the basic datastructures available when designing a modelling system. Datastructure definitions of the B-rep entities are described in Appendix A. The chapter goes on to identify and describe various basic tools, i.e. elementary modelling operations, which provide a uniform functional shell surrounding any particular implementation. Chapter 4 also describes basic tools, the Euler operators, and the theory behind decomposition of modelling operations into sequences of Euler operators. Chapter 5 describes variations on the basic datastructure, the non-manifold, degenerate, and partial object representations.

Chapter 6 describes several modelling algorithms for manipulating B-rep models, including modelling with degenerate models. These algorithms demonstrate the versatility of the B-rep technique. The algorithms are not a complete set, because one characteristic of B-rep systems is that they can be adapted to provide a diversity of tools suitable for different users and ap-

plication environments. Instead the algorithms are intended to represent a broad selection of algorithms to illustrate B-rep modelling and the principles of stepwise construction. Chapter 7 outlines some general principles for developing modelling algorithms and demonstrates the use of these principles with two detailed examples.

Chapter 8 describes some techniques for creating and maintaining information and auxiliary structures with B-rep models. These extra facilities are necessary to extend the model so that it can be used for a more complete communication in a distributed application environment.

Chapter 9 discusses some aspects of features which are an extension to the basic representation domain to include sets of model elements with some meaning. There are two basic approaches: feature recognition to recover feature information from a model and "design by/with features" to include feature information directly. These approaches are briefly discussed as well as some other aspects, such as feature verification and feature datastructures.

Chapter 10 describes some basic graphics techniques for presenting models. Chapter 11 describes disc formats and disc transfer of models. Chapter 12 describes command interpreter techniques. Chapter 13 describes some techniques for introducing free-form geometry into the model and chapter 14 describes, sketchily, some application areas. Chapter 15 describes the medial axis transform calculation for volumetric reasoning applications. Finally, chapter 16 describes some miscellaneous aspects of modelling: tolerances and debugging.

Contents

Chapter 1

Introduction

1.1 Background

Geometric modelling of objects might be defined as "the software representation of physical objects for some purpose".

This definition is deliberately vague because both the means of representing the objects and the purposes for which the representation has been used have changed considerably over the years. The original simple graphics models gave way to computer models of lines and nodes in the early 1960s, so-called wire-frame models. These were commercialised during the 1970s in the form of drafting systems, which were seen as capable of making the design process more efficient. These latter systems were essentially 'electronic drawing boards', capable of making pictures.

However, the gaps between the lines are also important, and computer systems for representing surfaces were also developed during the 1960s and 1970s. Surface modelling systems were developed partly to generate automatically the control code (NC-code) for machining complex shapes for aircraft and cars (see [2], for example). The technique has a long history and is more-or-less independent from the subject of this book, which is concerned with one branch of solid modelling. However, there are historical links between surface and solid modelling and a continuing relationship.

Another development was to model complete solids, a technique that emerged at the beginning of the 1970s, and that has replaced the earlier drafting systems. As with the earlier development of surface modelling, solid modelling systems expanded the potential of the computer tool. They made it possible to compute interactions between models, for example, and made possible new automatic calculations such as mass properties. As the solid models contain surface information, techniques for generating NC-code from solid models were also developed. Several other application areas have also grown up around the representation of solids, such as automatic finite-element mesh generation, dimensioning and tolerancing, part classification,

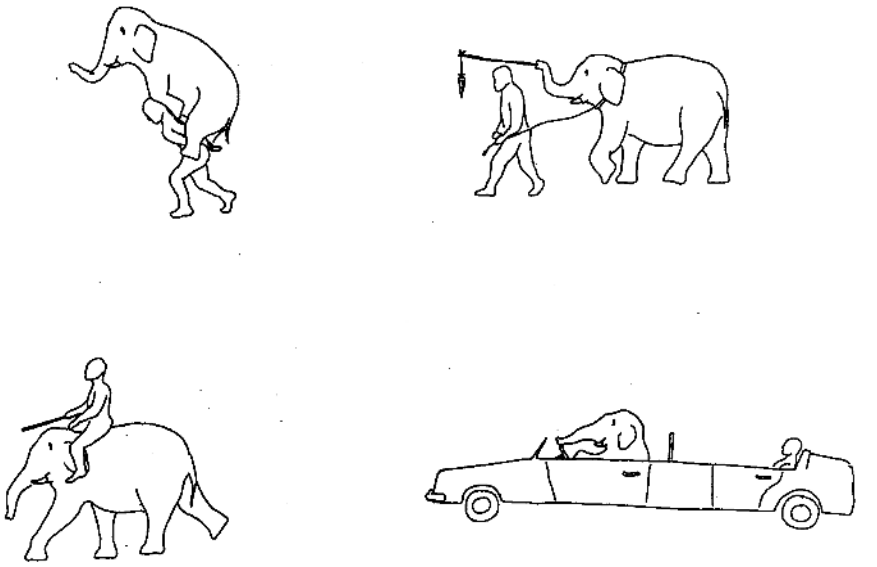


Figure 1.1: Symbolic comparison of modelling system types

and process planning.

The potential offered by solid models for both shape representation and automating manufacturing has stimulated much research into uniting design and manufacture. However, it was realised towards the end of the 1970s that simple shape representations were not enough to provide a complete production basis, and hence the idea arose of having a product model. Although the concept is not clearly defined yet, the product model was conceived of as providing much more than just a shape model; however, the shape of a part forms an important element of it. As part of this development, the former solid model has been annotated and extended to include information other than just geometry.

These various stages can be represented, perhaps not entirely seriously, by the pictures of elephants in figure 1.1, where the elephants represent computer systems. For wire-frame modelling systems, most of the work has to be done by the user, the user has to 'carry' the system. For surface modelling systems, the system is more like a partner for the user; it can do a lot of work, but it has to be led around. Solid modelling systems can do much more for the user, they can provide a high level of support, in effect carrying the user, although they need to be directed, and sometimes need a lot of coercion to produce the desired result. Product modelling is still too young to be classified exactly; the hope is that the user will be able to direct the system more efficiently – to do it more quickly.

1.2 Representations for solid modelling

That is, briefly, the background to computer modelling. This book is concerned with techniques for solid and product modelling. Different representation techniques are used to represent the solids. The main representation techniques are as follows:

1. **Cell decomposition:** Explicitly represents the material of which the solid is composed by dividing space into a set of elements, each full of material.
2. **General sweeping:** Represents objects in terms of two-dimensional shapes extruded along general curves.
3. **Set theoretic:** There are two variants: one representing solids as combinations of primitive shapes (CSG), and the other representing objects by defining a set of surfaces bounding the object.
4. **Boundary representation (B-rep):** This represents objects in terms of their 'skin', the boundary between model and non-model.

These techniques are described briefly in this chapter, but the reader should refer to, for example, Mortenson [88], Jared and Dodsworth [62], or Mäntylä [82] for more details.

1.2.1 Cell decomposition

The straightforward cell decomposition method is to divide object space into unit-sized elements (cubes or spheres) and to represent shapes as collections of these. Because of practical limitations in computers, it is not possible to divide space into infinitely small elements, so cell decomposition methods have to be approximate representations. A refinement of the technique using variable sized elements is called the octree technique. Using this method, objects are represented as trees of elements. Each element is classed as being full of material, partially filled or empty. Partially filled elements are subdivided, and the smaller elements are again classified as filled, partially filled, or empty. The elements are always subdivided into eight smaller elements, hence the name octree. Figure 1.2 shows a quarter cylinder as it might be represented by a simple cell decomposition scheme. Figure 1.3 illustrates the progressive development of an octree model of the same object.

1.2.2 General sweeping

This method represents objects in terms of two-dimensional shapes extruded along general curves. The method can represent only a limited class of object shapes, but it preserves information about how the shape was derived. This can be of help for applications, for example for NC-code generation.

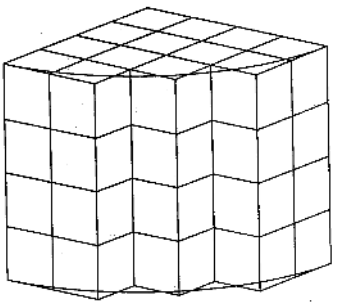


Figure 1.2: Simple cell division representation of quarter cylinder

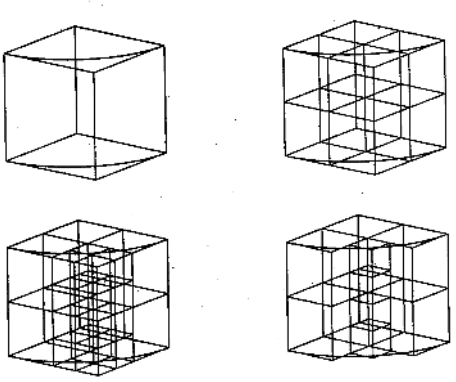


Figure 1.3: Creating an octree representation of a quarter cylinder

1.2.3 Set theoretic

There are two variants of this technique: one called constructive solid geometry (CSG), and the other dealing with so-called "half-spaces". They have in common that they deal with point sets; CSG defines a point set in terms of primitive shapes (defined by a set of half-spaces), and half-spaces split the modelling space into two point sets, one on either side of a surface. (See Requicha and Voelcker [111] or Okino and Kubo [93]).

CSG represents objects as trees of the point-set primitives, combined with the three Boolean operators: ADD, SUBTRACT, and INTERSECT. Originally the primitives were simple shapes, cubes, cylinders, spheres, cones, and so on, but later primitives with free-form shapes have been included to model complex surfaces. The modelling method is a 'one method modeller', applying only Boolean operations to the primitives, so that any other operation (such as sweeping) has to be simulated as combinations of these. Another version of this type of modelling uses only single objects and unary operators (NEVER, ALWAYS, etc.) to build up models (Katainen [68]). The method has the advantage that the models are mathematically correct, and operations such as mass property calculation and shaded image generation are easier than with, say, boundary representation. However, the single method modelling technique is restrictive and can be clumsy to use, especially for making small changes to models, which is a common need for incremental design.

The half-space modelling technique is related to surface modelling in that shapes are defined in terms of collections of bounding surfaces. The result object is then visualised by intersecting it with a series of parallel planes to produce section lines round it, which can then be drawn. As with the CSG modelling technique the method uses Boolean operations on point sets, defined by the surfaces. For example, a plane bounding the object cuts space into two parts: one 'outside' the object, and the other containing the object. The object is then the set of points common to all half-spaces. Obviously, bounded surfaces, such as spheres, are complete in themselves, but surfaces such as planes, cylinders, and cones define an infinitely large portion of space and must be closed off with other surfaces if the object is to be realisable.

1.2.4 Boundary representation

The method used for representing and manipulating solid models in this book is the so-called B-rep technique. This technique represents objects in terms of the 'skin' surrounding them. See Braid [10][11], Baumgart [4] or Mäntylä [82]. The skin is composed of a set of adjacent bounded surface elements, called faces. Faces are bounded by sets of edges, which are portions of curves lying on the surface of the faces on either side of the edge. The points where several faces meet are called vertices. The boundary representation datastructure will be described in more detail in chapter 3 and Appendix A.

Boundary techniques can be used to describe both solid models and a vari-

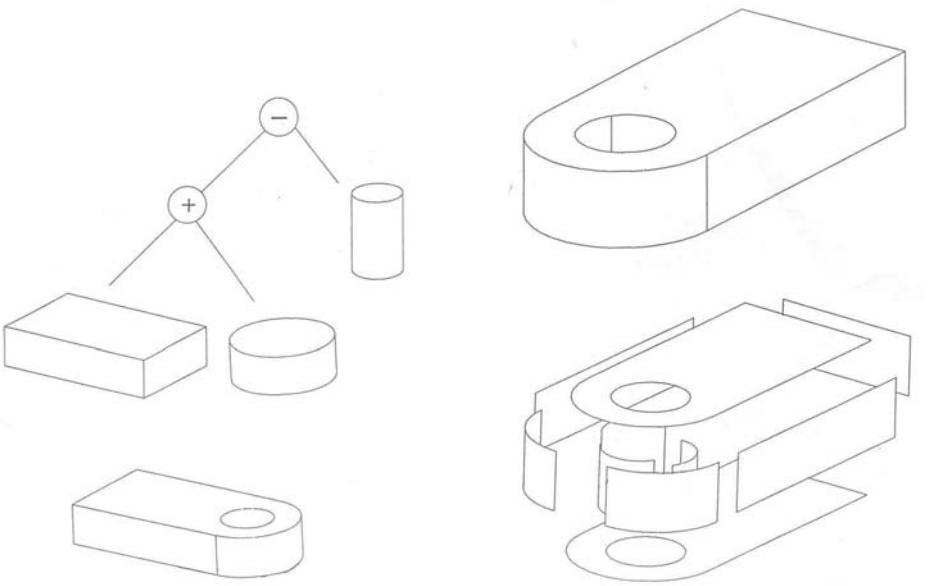


Figure 1.4: CSG and B-rep models (after Kjellberg [69])

ety of simplified forms that can be termed: "degenerate models". Extensions to the datastructure allow special types of models called "non-manifold models" to be represented that can be used for different purposes. The degenerate and non-manifold models will be described further in chapters 3 and 5.

Figure 1.4 shows a model as it might be represented using CSG and B-rep techniques.

1.3 Implications for commercialisation

The development and commercial exploitation of solid modelling has several implications. In general, a system provides a user with a standard set of modelling tools with which to create a design or to perform an application. A common problem, though, is that these tools are not sufficient, because each company has slightly different requirements. A common solution is to provide a programmable interface to the modeller, or to provide a macro-language with the help of which, application code can be developed.

Initially, solid modelling based systems were still viewed as drafting systems, albeit with extra functionality. Three-dimensional tools for directly creating solid models in three dimensions also exist but do not correspond to traditional design methods, so they have been slower to appear. Research systems have demonstrated that solid modelling is a very flexible tool; the problem is that the people who can adapt it tend to be software engineers,

whereas the people who know what tools are, or might be, useful are designers and manufacturing engineers. Software companies distributing modelling software have to make a 'best guess' to try and determine the commonly useful tools and leave the development of the others to user companies, or contract separately to produce the modelling code. In general, this policy means that a few special operations, such as chamfering and blending are provided and the 'operation set' is closed with general techniques like the Boolean operations or sweeping. As a result the user is forced to interpret what he or she wants to do in terms of these, rather than being allowed to make changes that are logical for an application.

Another aspect of the commercial application of modelling is that extra facilities are necessary for a user. Design tools, such as construction geometry to provide a geometric framework for a design, and tools for adding engineering information have typically been provided as a layered set surrounding the modelling system, not integrated with it. This causes data management problems when trying to keep track of the separate pieces of information during the course of the design process.

Application of modelling systems in practical engineering environments has been patchy, leading to so-called 'islands of automation' because different commercial systems have been developed for different application areas, and companies have had to buy a mixture of systems to supply their needs. Unfortunately, at least to begin with, these commercial systems did not communicate with each other because they used different formats, and even different bases for the underlying modelling software. There are several 'standard' exchange formats for communicating information (picture, surface, solid, etc.) between systems, such as IGES, XBF, SET, VDA-FS, CAD*I, and STEP, but communication is still not fluent. With the need for communication, there has been a move to enhance computerised solid models so that they can replace and extend the role of engineering drawings as information carriers. This trend leads to what is called "Product Modelling".

Alternatively, some commercial systems offer a wide range of applications compatible in modelling terms, but these can be expensive. Another problem is that dedicated application systems can be more dynamic and innovative than application packages in monolithic systems.

1.4 Product modelling

The product model is intended to be a complete information basis for production of a product. It is supposed to act as a communications basis between the application environments within the production environment. To illustrate the sorts of areas between which the product model is to act as a communications medium, a diagrammatic representation of such an environment, from the Production Systems Laboratory, IVF/KTH in Stockholm, Sweden, is shown in figure 1.5. It is difficult to be definite about the contents of such

a product model because products and production methods vary widely and because research projects tend to concentrate only on limited aspects of production rather than on the complete process. However, it is possible to make some general observations about the sorts of information that might be useful and examine the implications of these. Possible components of a product model might include:

- Product specification and information
- Annotated shape model
- Kinematic model
- Analysis models and results
- Process plan
- Assembly plan
- Production facility layout

Obviously, some of these components are outside the realm of extensions to solid modelling. It is clear that an annotated shape model, containing information about such things as the material from which an object is made, or surface finish, for example, is part of the product model for many companies. These kinds of information are necessary for communication between application areas. A kinematic model is also related to extended shape modelling, describing how parts of an assembly are related. The other areas may or may not affect the model directly. Analysis models may be linked to part models, or just be passive, slave models derived from the original. Process plans and assembly plans may also be linked to the model or have their own separate, related models. However, it is clear that annotated shapes and kinematic models are needed.

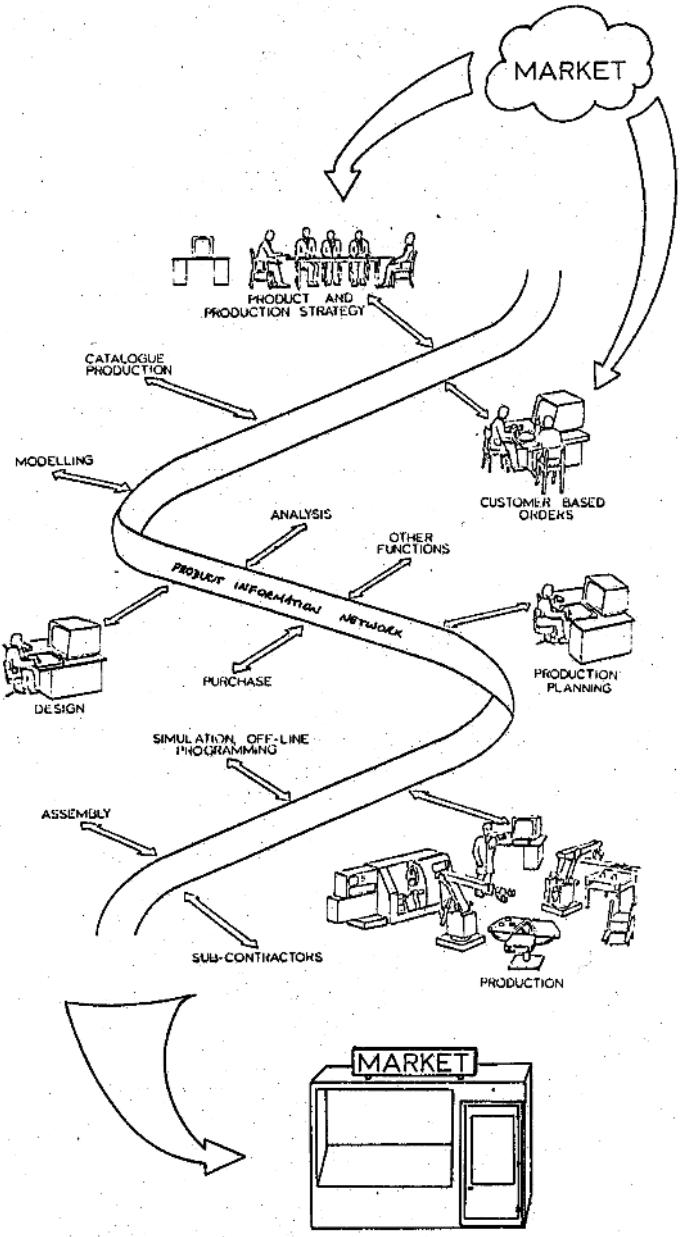


Figure 1.5: Elements in a product modelling environment

Chapter 2

Modelling Background

This book deals with techniques for modelling using the boundary representation (B-rep). The basic technique for CAD with boundary representation models was developed by Braid in the early 1970s, and presented in his dissertation "Designing with volumes" [10], this work being demonstrated with the BUILD-1 system. The use of the winged-edge pointer method was used by Baumgart [4] on polyhedral representation of objects for computer vision. Braid's work continued during the 1970s and culminated in a report, "Notes on a geometric modeller" [11], containing information about various aspects and operations of the then BUILD modelling system, BUILD-3. Building on Baumgart's definition of Euler operators, and later work by Braid et al. [13] and Eastman and Weiler [30] at Carnegie Mellon University, Mäntylä [81] formalised their use, and established several important theoretical aspects of B-rep. Mäntylä and Sulonen [83] describes the Geometric WorkBench (GWB), a modeller based on Euler operators.

In France, work on B-rep modelling, initially for analysis and later commercialised for design as well, resulted in the EUCLID system [14]. In Japan, work started on the boundary modeller GEOMAP [60] and has continued, investigating various aspects of B-rep modelling and its application in various areas of design and manufacturing. In America, related work for architecture and engineering was carried out at Carnegie Mellon University (see e.g. [5]). Building on earlier work with GLIDE, Weiler went on to examine various aspects of connectivity in a model [143], and on non-manifold modelling [144]).

Starting during the middle of the 1970s, and continuing today, commercial systems based on the B-rep technique have appeared. Although several such systems have been developed, the work of the company Shape Data is especially noteworthy, not only because of the original links between Cambridge University and the company, but also because of the continuing development of B-rep modelling techniques. Shape Data produced the Romulus modelling kernel (and Remus!) and later Parasolid. Three of the original founders of

Shape Data Ltd., Drs. Braid, Grayer, and Lang, went on to found another commercial company, Three-Space Ltd., which produced the modeller ACIS.

Also during this period work was going on other aspects of boundary representation modelling. Dimensioning and tolerancing was investigated by Hillyard [57] . There was an investigation of the analysis of a model to extract features (Kyprianou [74]), aimed at part classification. There was also work done on integrating B-spline surfaces into BUILD (Solomon [120]), and the automatic generation of tetrahedral elements for finite element analysis (Wördenweber [148]). The work on features, originally by Kyprianou and later by Anderson (reported in Parkinson [95]) during the early 1980s has proved more generally useful than solely for part classification. Feature information provides a useful high-level handle for interrogation of the model and prompted the addition of feature datastructures in the model.

Towards the end of the 1970s an internordic project, Geometric Product Models (GPM), was set up to examine the use of modelling in companies, and to produce basic software for enhancing industrial CAD/CAM systems. The project analysed design methods in several companies participating in the project, producing specifications for four software modules: an assembled plate construction module (APC), a sculptured surface module (SS), a geometric design program (GDP), and a volume module (VOL) (GPM information brochure [50]). The volume module contained not only methods for solid modelling, based on Braid's BUILD system, but also extra facilities for special representations for sheet- and wireframe models.

During the 1980s, there were several developments based on earlier work with the BUILD system in Cambridge. One development was to extend the work of Kyprianou on feature recognition to provide a high-level 'handle' on a model, mentioned above. Other work was done on automatic generation of NC-code from solid models [96], following earlier work by Alan Grayer on a previous version of BUILD [51]. This work led to a project to examine methods for generative process planning based on solid modelling (Jared [63]).

Further work on solid modelling, based on the BUILD system, was carried out in Hungary on the FFSolid project. Part of this work involved developing methods for integrating free-form surfaces with solid models [138], on blending [142], on automatic NC-code generation for 3- and 5-axis milling [42], and various related enhancements. In an associated project, Luo examined the development of non-manifold representations [78], [79], which could also be used for representing features.

Yet more work was done on Euler operators in Japan by Chiyokura and Kimura. They were used as a basis for an object sculpting method [18] and for recording modelling operations for redoing and replaying [19]. See also [20].

In Finland work started on the Geometric WorkBench (GWB), a modelling system based on Euler operators. This work has expanded with several joint

projects with the Kimura laboratory as well as work in Finland.

The above is only a sketchy outline of the history of B-rep modelling, and the picture is becoming more and more fuzzy as the number of centres increases and work diversifies. The main influences on the work described in this book have been the BUILD system, its extensions in FFSolid, and the GPM Volume module, so the history of B-Rep modelling reported here is biased towards what happened with these systems.

2.1 The BUILD system

In the beginning of the 1970s, Ian Braid developed a solid modelling system for designing engineering parts [10]. Before arriving at the final version of the scheme, Braid considered and rejected a cell decomposition representation method, and a method of primitive instancing, where objects were composed of collections of simple shapes that were drawn as though united. The final scheme was the Boundary Representation scheme, which was later refined in the light of Baumgart's work in America.

The original system offered the user a set of primitives (CUBE, WEDGE, TETRAHEDRON, CYLINDER, SEGMENT, and FILLET) or allowed the user to define shapes composed of straight lines and circular arcs that could then be extruded (swept) to produce solids. These could be combined, using Boolean operations, to produce complex shapes in an incremental manner. The system offered modelling commands to:

- DEFINE objects, primitives, or arbitrary shapes with unit thickness.
- MODIFY objects to alter their position in space.
- DELETE objects.
- NEGATE objects.
- MERGE objects, to join objects at specified faces.
- COMBINE objects, for merging sets of coincident faces.
- INTERSECT objects, the Boolean operations on a negative primitive and an arbitrary object.
- SAVE objects on a disc file.
- GET objects back from a disc file.

The system also contained a variety of drawing commands (for drawing wireframe-, locally hidden-, fully hidden-, or shaded-pictures of objects on screen or plotter), printing objects, or controlling the system. The system was implemented in FORTRAN and in assembly code, with a primitive user interface allowing input of typed commands.

Braid's representation scheme consisted of eight lists, containing information about: 1) Transformations; 2) Vertices; 3) Faces; 4) Edges; 5) Straight edges; 6) Curved edges; 7) Intersection curve information; and 8) Edge parameter limits. In addition, a separate array contained geometric information for faces, in the form of matrix surface equation in a standard orientation and a reference to a transformation matrix (from list 1) to define the face orientation in its desired orientation.

Although this datastructure may seem crude, it contains many of the elements of the more sophisticated datastructure that emerged in the second version and was later refined. Evident from this structure are the separation of the object structure (topology) and the shape (geometry), at least for faces and edges, vertex coordinates being stored with the vertex. Loops (the mechanism for representing multipli-connected faces) were present implicitly; the edges bounding faces were held in circuits, representing the outer boundary and any inner boundaries, if present. Braid also included a tag mechanism with his original datastructure, an important facility for many modelling tools.

Braid's original work demonstrated the practicality of representing and manipulating models in this way. However, the basic structures of the program were limited in size. They were revised to take advantage of the representation developed by Baumgart for solid modelling for computer vision.

2.2 Baumgart and the winged-edge representation

Baumgart's principle concern was modelling real objects for computer vision. In his thesis [4], he describes a model structure for representing polyhedral objects that influenced Braid's early work. The contributions of Baumgart to B-rep modelling can be summarised as follows:

1. The winged-edge structure
2. The Eulerian structure for models
3. The strict division into manifold and non-manifold models
4. The Euler operators

The winged-edge structure provided a method whereby all elements in the model structure were of fixed size. The name, "winged-edge", comes from the way that all edges refer to four neighbours only, and it is shown in its later form, due to Braid et al., in figure 2.2.

The Eulerian nature of the model was one of the pre-conditions for valid models in Baumgart's system. The Eulerian formula, F - E + V = 2 - 2 * H, relates elements in the model. The validity of the structure was guaranteed by manipulating it with so-called Euler operators (see Chapter 4) that manipulated only combinations of elements, not isolated elements.

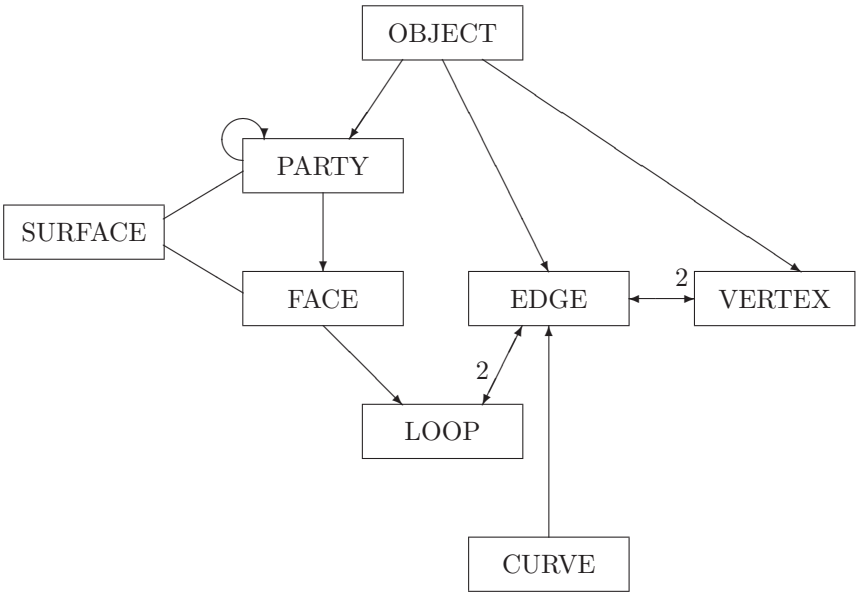


Figure 2.1: The BUILD single object datastructure

The 'manifold' condition is that, at every point on the boundary of an object, a small enough sphere will cut the object boundary in a figure "homeomorphic to a disc". This essentially means that valid objects enclose a portion of space, or three-dimensional point set.

An interesting feature of Baumgart's modelling operations is that they were built up using a set of basic operations called Euler operations, or Euler primitives. The Euler primitives are necessary because the elements from which the model are built are constrained by the Euler formula, Baumgart's first condition for polyhedra, mentioned above. The Euler primitives were modelling 'building blocks'. The main modelling routines consisted of sequences of Euler primitives, creating temporary models not adhering to Baumgart's polyhedral conditions but leading to new models that did.

Baumgart was mainly interested in modelling for computer vision, so much of his other work involved integration of the modelling system with the visual sensing system. However, his winged-edge datastructure, his characterisation of models in terms of Eulerian graphs, and the Euler primitives had an important influence on B-rep modelling.

2.3 BUILD system: Developments and extensions

Braid reimplemented the BUILD system (in Algol68C) in terms of the winged-edge structure, having already identified his original structure as being too limited. In addition to Baumgart's model entities: faces, edges, and vertices, an extra element, the loop, was added to allow multipli-connected faces. Later, a face grouping mechanism, the party, was added. The final datastructure for a single object, at the end of the 1970s, is shown in figure 2.1 in SYSDOC form. The boxes in the diagram represent entity classes and the arrows represent connections. Thus, for example, the arrow from entity class OBJECT to entity class EDGE indicates that there is a pointer from an OBJECT to an EDGE (the head of a chain) as part of the datastructure. A double-headed arrow indicates that there is a many-to-many relationship. A pointer with an associated number indicates a restriction on the number of entities in the relationship. Assemblies could also be constructed, as collections of 'instances', each with a transformation matrix, associated with a prototype object or sub-assembly.

The addition of the loop necessitated changes to the winged-edge structure, (Braid's version is shown in figure 2.2), and to the Euler formula stated by Baumgart, and thus to the Euler primitives used to build complex operations. The extended formula is the Euler–Poincaré formula:

$$v - e + f - h = 2 * (m - g)$$

where v, e, f, h, m, and g represent the number of vertices, edges, faces and hole-loops, the multiplicity, and the genus, respectively. The genus is an attribute of a model, not represented directly in the datastructure.

Braid et al. [13] and Eastman and Weiler [30] describe how a set of five operators can be chosen as a 'spanning set', a combination of which can be used to create any Eulerian graph. Chapter 4 describes the Euler primitives, or Euler operators, in more detail. The meaning of the genus and multiplicity are illustrated in figure 2.3. The sphere in figure 2.3a has genus zero, the torus in figure 2.3b has genus one, and the object in figure 2.3c has genus two. The cube in figure 2.3d has multiplicity one, and the cube with an internal cavity in figure 2.3e has multiplicity two, indicating two separate shells or closed face sets. A collection of seperate objects also has multiplicity greater than one, so a collection of ten cubes would have multiplicity ten. Genus can be calculated using adjacencies as described in Braid et al. [13]. Multiplicity can be represented explicitly, although this was not done in BUILD. The GPM system, described in section 2.4, used a convention that top-level facegroups were to used represent separate shells explicitly.

Parties were added to BUILD to share surfaces between groups of faces. In addition, they, and objects, contained a simple measure of spatial occupancy, the 'bubble', a centre-point and the radius of a sphere surrounding the elements contained in the party or object. Both the surface sharing mechanism

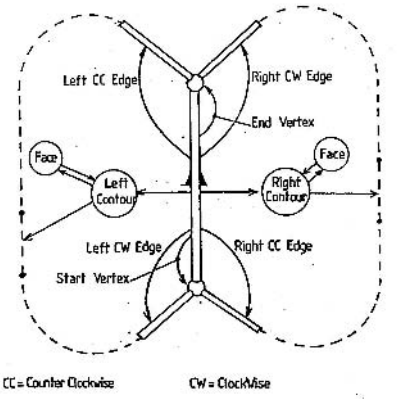


Figure 2.2: The winged-edge structure (after Braid et al. [13])

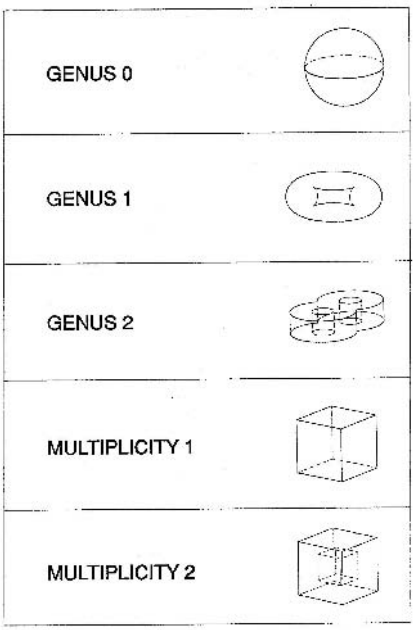


Figure 2.3: Illustration of genus and multiplicity

and the spatial occupancy mechanism helped to improve the efficiency of the Boolean operations, as investigated by Smith (private communication). One of Smith's demonstrations of the improvements to Boolean operations was to take a long thin block and subtract a series of cylinders from the block. Instead of taking longer and longer as each cylinder was subtracted, and the number of faces increased, each Boolean operation took approximately the same time. Bubbles and other spatial occupancy description methods will be discussed later (in section 3.1.4).

Another extension to Baumgart's polyhedral structure was to add curves and curved surfaces to the model structure, following Braid's original BUILD system. The second version of BUILD contained general quadric surfaces, spline curves (for the conics), and B-spline curves to represent non-planar intersection curves. As a space-saving measure, the system contained an implicit straight line representation and the shorthand arc description of Sabin [114] for commonly occurring curves that could be shared. Later on, in BUILD III and BUILD IV, toroidal surfaces, B-spline patches (Solomon [120]), and double-quadratic geometry (Várady [138]) were also added, potentially allowing any shape to be represented.

As well as reimplementing the operations of the original BUILD system, another development was to extend the functionality of the system. Baumgart's straight and rotational sweep operations, the pyramid building operation, and the dual operation were incorporated into BUILD. In addition many new shape creation and modification functions were implemented, demonstrating the flexibility of the B-rep. Around the modelling system a flexible command interpreter was added, which created both a convenient testing environment and embodied the BUILD modelling philosophy. Braid [11] describes the whole of BUILD. Several design tools, the so-called 'local operators' are described in Jared and Stroud [65]. Braid also investigated implicit blends, implemented by marking edges as rounded, and attaching a blend radius.

During the 1980s development work continued, although the emphasis shifted away from pure solid modelling towards applications. As well as Várady's work on integration of free-form surfaces with solid models, mentioned above, some work was done on removing so-called 'fake' edges from the model. Fake edges are extra edges added to break up curved faces into subfaces, and they are described further in section 3.1.4. They were originally included in BUILD because faces were not allowed to extend through more than 180 degrees. However, they are artificial elements, tending to fragment the model, so it is desirable not to have them, if possible. Feature recognition and representation were also examined to provide high-level information 'handles' on the model for applications.

In parallel with the work on BUILD in Britain, activity started at the Computer and Automation Institute at the Hungarian Academy of Sciences, transforming BUILD into the FFSolid system. During the course of this work, more was done on improving the basic facilities in the modeller, geometric intersection code, for example; on implementing extra tools, such as that for

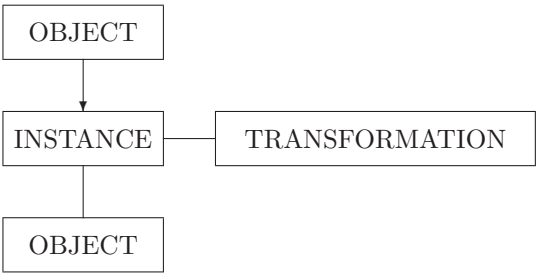


Figure 2.4: The BUILD assembly datastructure

'blending' or rounding off sharp edges and vertices; and on applications. The spherical spatial occupancy measure added with parties was supplemented with a face boxing measure, again to improve the efficiency of the Boolean operations [139].

The assembly structure is as shown in figure 2.4. An object could either be a single object representation or contain a set of instances, chained together. Each instance contains a reference to an object, which could be a single object or a collection of instances. This relatively simple hierarchical structure allows the definition of complex assemblies.

2.4 The GPM project

The GPM Information Brochure [50] describes the GPM project and the resulting software modules mentioned at the beginning of this chapter. The main concern, here, is the Module for Volume Geometry.

The project started in 1977 with an initial design phase involving the examination of the use of volumetric models, as well as different solid representation techniques. Having examined several different systems with various representation techniques, boundary representation was chosen for the system. However, as well as representing solids, there was a requirement that wireframe and sheet models should also be integrated into the system. The analysis phase and philosophy behind the use of these computer aids in production is described by Kjellberg [69]. The final, single-object datastructure, shown in figure 2.5, is related to that of the BUILD system on which it was based, but there are some differences to cope with the extra model types, as well as with the volume model datastructure. The module was implemented

between 1980 and 1982 as a joint development between Sweden and Finland. It was distributed as a subroutine package, implemented in so-called "Compatible FORTRAN", a subset of FORTRAN IV.

Graph models were added as simple wireframe representations that could be used freely to build up design sketches, as well as for special purposes, such as the representation of pipework centre-lines. They consisted simply of edges and vertices, together with their associated geometry. The only specific limitation on their use was to forbid two vertices to be connected by more than one edge. This restriction was imposed to prevent creating faces bounded by only two edges, because graph models were also seen as a mechanism for defining two-dimensional shapes to be extruded into solids using 'sweep' or 'swing' type operations.

Laminae and shell-objects, which together are termed here sheet models, were added to represent idealisations of thin parts, such as sheet metal products. There was a special module, the GPM assembled plate construction (APC) module, solely for this purpose. The sheet object representation in the volume module provided a parallel facility, although less well developed, which could be used for communication with the APC model. The sheet models were degenerate models, still locally two-manifold; the edges at the boundaries could be considered as being homeomorphic to a 'folded' disc. There was a convention that the faces in the two sides of the sheet object were assigned to different facegroups, so that there was an explicit representation of 'sidedness'.

Volumetric models were basically the same as BUILD objects. However, the face grouping mechanism GPM's FACEGROUP, corresponding to the BUILD party, was used, among other things, for explicitly representing shells in objects. A shell is a closed set of faces representing a complete object boundary. Objects such as a cube have a single shell. Objects with an internal cavity, such as that shown on the bottom row of figure 2.3, have two shells. Each separate shell in a volumetric model was represented as a separate top-level facegroup, facilitating their handling in, e.g., Booleans. Another difference with the BUILD system was that there was an explicit representation for a point, the vertex geometry. This was done so that complete geometric frameworks could be set up independently of topology as design structures. Also, the GPM volume module contained loop-edge links, called winged-edge links, defined in 1980 as part of the basic datastructure. These were not used for non-manifold modelling, though, and there was a maximum of two for each edge.

Geometrically, GPM used only quadric, plane, and toroidal surfaces; conic curves; and B-splines for intersection curves, because the GPM sculptured surface module provided free-form geometry. Connection between the two modules is described in Fjällström et al. [38]. Explicit representations were used for several surface and curve types. Available surface types were plane, sphere, cone, cylinder, general quadrics, and toroidal surfaces. Curve types were: straight line, circular arc, circle, ellipse, parabola, hyperbola, and closed

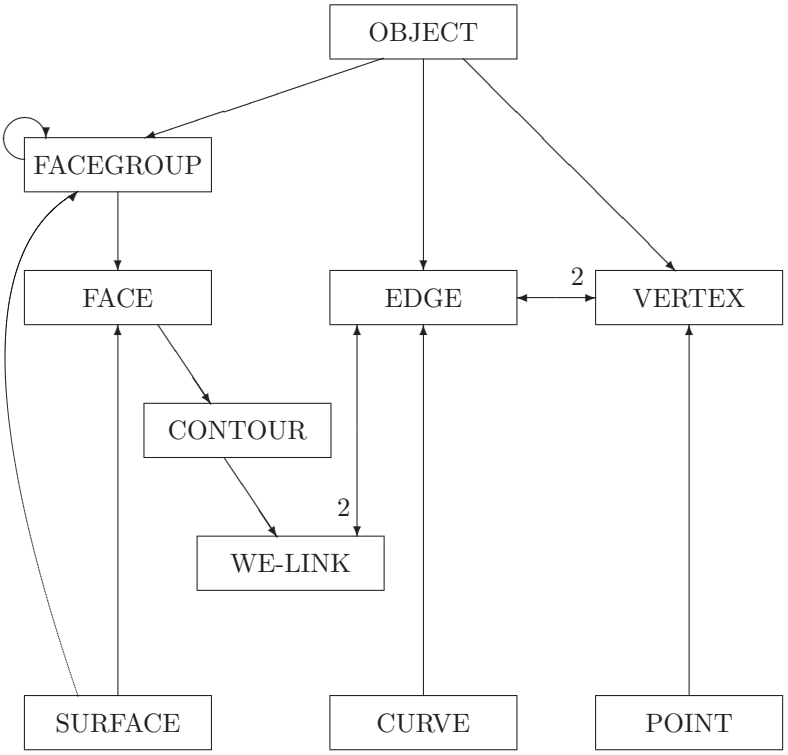


Figure 2.5: The GPM Volume module datastructure

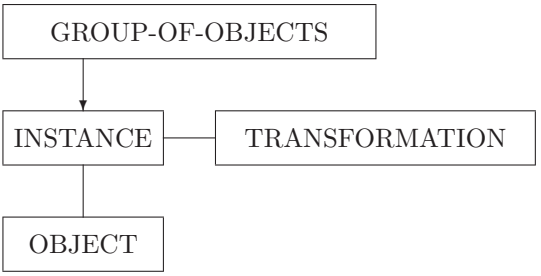


Figure 2.6: The GPM assembly datastructure

and open B-splines. Having the common geometric types explicitly represented allowed special cases to be identified and handled simply (Pfeifer [99]).

The extra model types included with the system meant that new modelling operations had to be developed for integrating all three. Closed graph models, with exactly two edges at every vertex, could be set in surfaces to make sheet models. Graph models could also be swept to produce sheet models. Sheet models could be swept to produce volume models or given 'thickness' by expanding them. Sheet models could be converted back into graph models by removing the face information, and volume models could be converted into closed sheet objects. Three of these operations, sweeping graph (or wireframe) models, sweeping or swinging laminae, and expanding sheet models, are described in chapter 6. Although the graph models are non-Eulerian, and so limitations were imposed on their use, the sheet models were Eulerian objects. They were, in effect, treated like volumetric objects, being locally manifold on the interior, with the exterior edges handled specially.

As well as the single object representations, the module also provided facilities for creating product models, with facilities for creating geometric design frameworks, mechanisms for annotating models, and creating linked assemblies. As indicated by the project title, Geometric Product Models, the software was intended to be part of complete product modelling systems. To this end, these extra mechanisms were intended to supplement the basic shape model with the aim of making a more complete basis for production. However, the modelling algorithms were developed primarily to handle the geometry and topology of a model's shape, and this information was ignored. Some limitations were imposed so that a user had to be aware of information in the model, but no automatic handling methods were developed. Nor was there any attempt to create a general kinetic package with object links; see Tilove [136], for example, but experiments with dynamic mechanism simulation were performed with a linked robot model by Wingård, and Palm [146] and others.

2.5 Feature recognition and features in modelling

In the late 1970s, Kyprianou [74] developed a method of feature recognition, based on the BUILD solid modeller, which decomposed models into sets of faces corresponding to a basic shape (the root) and various delimited sub-units of the basic shape called 'features'. Originally, features had to be completely bounded by sets of concave or convex edges, but the method was later improved by Anderson [95] to allow overlapping features to be recognised.
The use of features in modelling is, however, a complex process, and much other work has been done on alternative approaches. In addition to feature recognition techniques, another popular idea is to make a basic shape and to

introduce features into it, so-called "design by features". Another technique, investigated by Wingård [145], is to pick features from other objects and to fill in the rest of the design between these. This can be used to construct objects that have to fit into assemblies. Another variant on this, for cutting and pasting features, is described by Ranta et al. [104]. For more information about features see, for example, Shah et al. [121] for a survey of feature modelling.

The various feature techniques, feature recognition, design-by-features, and building designs from feature sets, all have advantages and disadvantages. Feature recognition has the advantage that the shape model is the source of the feature information, so changing the model and modifying the features in it is unconstrained because this information is always recovered anew when needed. A disadvantage, though, is that the same features may be re-recognised several times, or that isolated shape elements introduced specifically into a design have to be recognised; the information about what they are is lost. Design-by-features and building objects from feature sets retain the information that a feature is a feature, but they fix the features so that modification may corrupt the feature model.

One thing is clear, features have an important role in modelling applications. Kyprianou's original work [74] was to determine automatically part classification codes needed for large part databases. These are, in effect, numerically coded shape descriptions that can be used to check if a part or something similar to it has been made before. Although this is an important end in itself, the classification of shape through the use of features has a wider use. Boundary representation models are essentially local in nature, composed of faces, edges, and vertices, as already described. Features provide a higher level 'handle' on a model that can be used for applications such as process planning, orientation classification, or assembly.

Features are described in more detail in chapter 9.

2.6 Non-manifold modelling

As described in section 2.2, one of the basic properties defined by Baumgart for valid polyhedra was that they be manifold. The definition of 'manifold-ness' is that, at every point on the object boundary, a sphere with a small enough radius cuts the boundary in a figure homeomorphic to a disc. It is a condition that is necessary if the model is to be physically realisable. However, for some purposes, for models that are not meant to be made, this requirement can be relaxed.

A formal datastructure for non-manifold modelling was introduced by Weiler [143]. More recently, Luo and Lukacs [78] and Luo [79] have worked on formal aspects of Euler operators for non-manifold modelling and for using these representations for feature modelling.

One main element of Weiler's datastructure was the radial edge, or star

representation. With the datastructures of Braid and of Baumgart, edges lay between two faces (although the left and right face pointers could refer to the same face). The radial edge structure allows more than two faces to meet at the edge. Instead of having pointers to a left and right face, each edge refers to a list of radial edge elements, each referring to a separate face. The radial edge links are ordered round the edge according to the geometry of the faces. The edge contains the common information; the links define the connectivity. Weiler also defined some Euler operators for manipulating this extended datastructure.

Luo and Lukacs extended this scheme and proved several basic theorems about the structure and their extended Euler formula. Each edge is connected to a set of wedges, each between two faces. Each vertex is connected to a set of one or more bundles, each bundle connecting a sequence of faces round the vertex. Bundles are, in fact, a logical dual of the loop used for multiply-connected faces. The scheme was also extended to include joints, coincident face overlapping face sets. This structure allowed objects to be decomposed into volumetric elements corresponding to features (Luo and Lukacs [78]).

Non-manifold objects are specialised, useful for particular applications but not as exact models of physical objects. Weiler suggested that they may be useful for such purposes as representing finite element models, the radial edge structure providing a single method for representing both manifold models and non-manifold ones. Such structures can cause problems for modelling operations, not so much because of the extra cases that have to be handled, but more because of the ambiguity introduced into the model. They will be discussed in more detail later (in chapter 5).

2.7 Modelling system implementation

As with many other questions modelling system implementation, which computer languages and styles are used is the subject of strongly held, well-entrenched views in some quarters. I read in one paper that LISP was now the de facto standard language with which to write modelling systems. As the authors seemed to have based their finding on a representative sample of LISP-based modellers, it may be true, for them. However, in general, there are no such easy answers. The question of which language to use to implement a modelling system, and whether or not it should be "object oriented" is tied up with personal viewpoints, rather than with scientific necessity. All it is reasonable to say is that B-rep modelling systems need well-developed datastructuring facilities. They could be implemented in any language really, but it is harder to write good modelling code in assembler or in FORTRAN IV, than it is to write in a modern high-level language. Compilers take a lot of the drudgery out of trying to manipulate datastructures. Debuggers make interactive languages less necessary for being able to monitor the functioning of a system. Personally I like Algol68C, but it has its drawbacks (like the lack

of support by computer suppliers). There are lots of languages that facilitate development to a greater or lesser extent, so there should be no need to presuppose a particular language for the purpose of understanding the processes involved in modelling. The datastructure definitions in Appendix A may look like Algol68 STRUCTs, although there are several similar datastructure definitions in other languages, but they could even be mapped onto arrays if necessary. Similarly, pointers can be translated into integers (array addresses) if desired. The aim is to avoid talking about implementation details in the book.

2.8 Research directions

A simplified view of research is shown in figure 2.7. Of course it is not that simple, but the intention is to show some of the trends and their relation to solid modelling. Also, solid modelling began before 1975 and has its roots even earlier; the diagram tries to demonstrate relations.

Solid modelling as a research theme was strong in the 1970s but during the 1980s attention turned towards applications. Research and development on solid modelling is much rarer to find outside the commercial companies supplying solid modelling packages.

Work on machining from solids had already been done by Grayer during the 1970s and work on this continued related to features during the early 1980s.

Work on features began in the late 1970s and, fortunately, has continued to develop since then. Features are a way of 'understanding' an object at a higher level than as simply a set of faces, edges, and vertices. Features are of interest in many application areas, from design to manufacture. The same sort of 'geometric reasoning' techniques are also applicable more widely for analysis of shape, symmetry, and so on.

Research on surfaces and geometry in general has been around for longer than solid modelling, but there has been a lot of work on complex geometry in solids. Sculpting methods are an interesting example of the combination of surfaces and solids. Surface technology also turns up in reverse engineering, for surface segmentation, for example.

Analysis in various forms, from calculation of mass properties to automatic finite element meshing, has also been the subject of much research. More recently, medial axis techniques have also been applied in this area. The medial axis transform is a way of computing volumetric properties.

Multiple representations have also been a topic of long-standing interest, both for design and for other applications. Design has provided stimulus for feature research as well as other aspects of product modelling, constraint-based modelling, dimensioning and tolerancing, surfaces, and so on.

B-rep to CSG conversion was a subject that was of great interest during the 1980s but has been worked on less since, although CSG is still around.

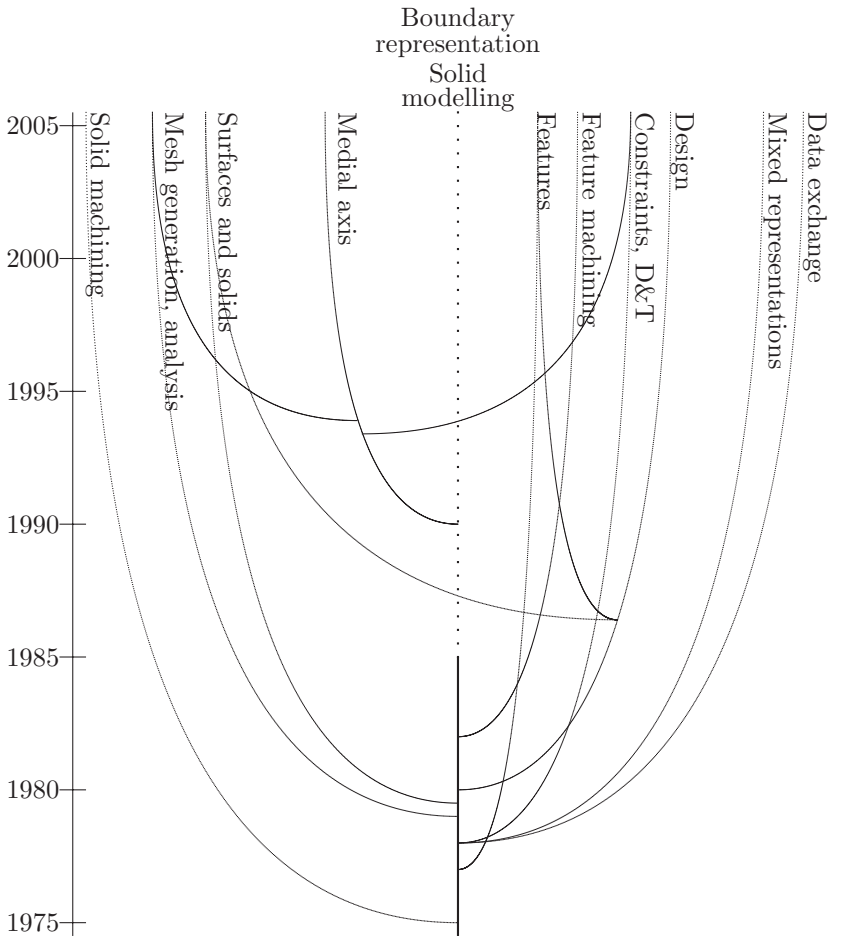


Figure 2.7: Tangled tree of solid modelling and relations in the forest of knowledge

There is some connection between this and the medial axis.

Geometric reliability, or geometric robustness, is another topic that has been worked on during the years. This is still a problem due to the numerical constraints of computer representations. It has faded a little as the memory sizes have increased.

Reverse engineering is still flourishing. As well as being of interest for mechanical parts, this is also relevant for bio-medical and for aesthetic parts. Both of these latter applications present new challenges because of the complexity of the shapes involved. This is in addition to the other technical challenges that have already been and are being researched.

The information around the geometric shape of a part is also a continuing subject of interest. Setting modelling into the cadre of its application in industry means that there is more information than just the shape to be dealt with. The complete information set is usually termed the 'Product Model' and means that design and manufacturing demands are influencing solid modelling.

Final in this brief overview it is necessary to mention multi-material models and flexible models. Two basic assumptions behind boundary representation are that the model is homogenous and that it is rigid. Composite materials were recognised as a potential problem right at the beginning of boundary representation modelling. Flexible parts are also a potential problem, especially in micro-engineering where material properties may make mechanical assemblies unnecessary.

It would be difficult to cover all of the research topics in an adequate depth in one book. If done, the pocket version of such a book would probably need a crane to lift it. It is more appropriate to have a set of books rather than to try and cram everything into one. The intention of this book is to cover the techniques in boundary representation modelling, currently the basis of many commercial systems. As well as explaining the internal workings of a modeller, chapter 14 describes some aspects that are related to applications. This is intended to explain how it is possible to work with solid models, the interfaces between the model and the applications. Tools for data extraction from models and traversal methods are also described.

Chapter 3

Datastructures and tools

Because of the large amount of work that has been done on boundary representation (B-rep) modelling, there are several different variants in system designs. The basic decisions about the datastructures and geometry of the modelling system have several different implications for the basic tools and algorithms available in the modelling system. Although this variation does not alter the basic characteristics of the algorithms and tools, it does have practical 'house-keeping' implications for the modelling system. It is neither possible nor, necessarily, desirable to place strict limitations on the actual representations for the datastructure. The variations can be overcome by standardising the way that the datastructure is addressed by the modelling algorithms. This chapter defines an 'ideal' datastructure and the functional interface between this and modelling algorithms.

Section 3.1 describes the datastructures that form the object representations, section 3.2 describes the different ways of representing the model connections, and section 3.3 describes the basic modelling operations, the basic tools, from which more complex operations are built.

3.1 Modelling datastructures

Boundary representations represent objects in terms of their 'skin', the boundary between 'model' and 'non-model'. The skin is divided up into surface portions or faces. The faces are surrounded by sequences of edges, which are portions of curves between two adjacent surface portions. Edges, or curve portions, are delimited by vertices, which are also where several faces meet (figure 3.1). The datastructure can be divided into two basic groups: one responsible for defining the structure of the object (the topology) and one the form or shape of the object (the geometry) (figure 3.2a). The main elements, mentioned above, are the faces, edges, and vertices, together with their geometric forms: surfaces, curves, and points (figure 3.2b). The 'normal' datas-

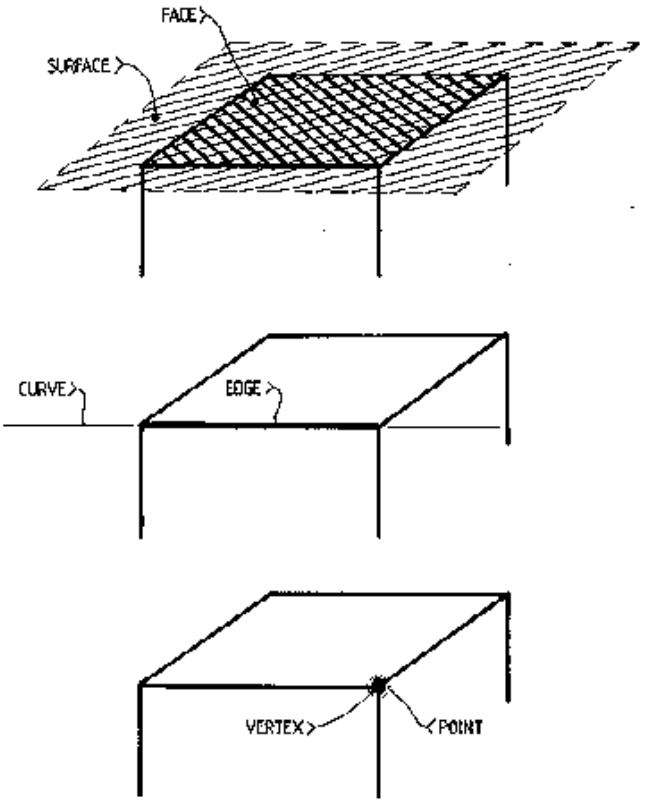


Figure 3.1: Faces-surfaces, edges-curves, and vertices-points

tructure kernel contains other elements, for various purposes (figure 3.2c). Several additional facilities can be added around this kernel for representing such things as features and for annotating the model. In addition, it is desirable to include facilities for creating assemblies and for implicitly copying objects. It can be useful to associate supplementary geometric entities with models or parts of models for various purposes.

Section 3.1.1 summarises various questions that should be considered when designing a modeller. Section 3.1.2 describes the model elements in more detail. Section 3.1.3 describes the geometric forms and various implications when choosing the geometry set. Section 3.1.4 describes the extra elements of the datastructure.

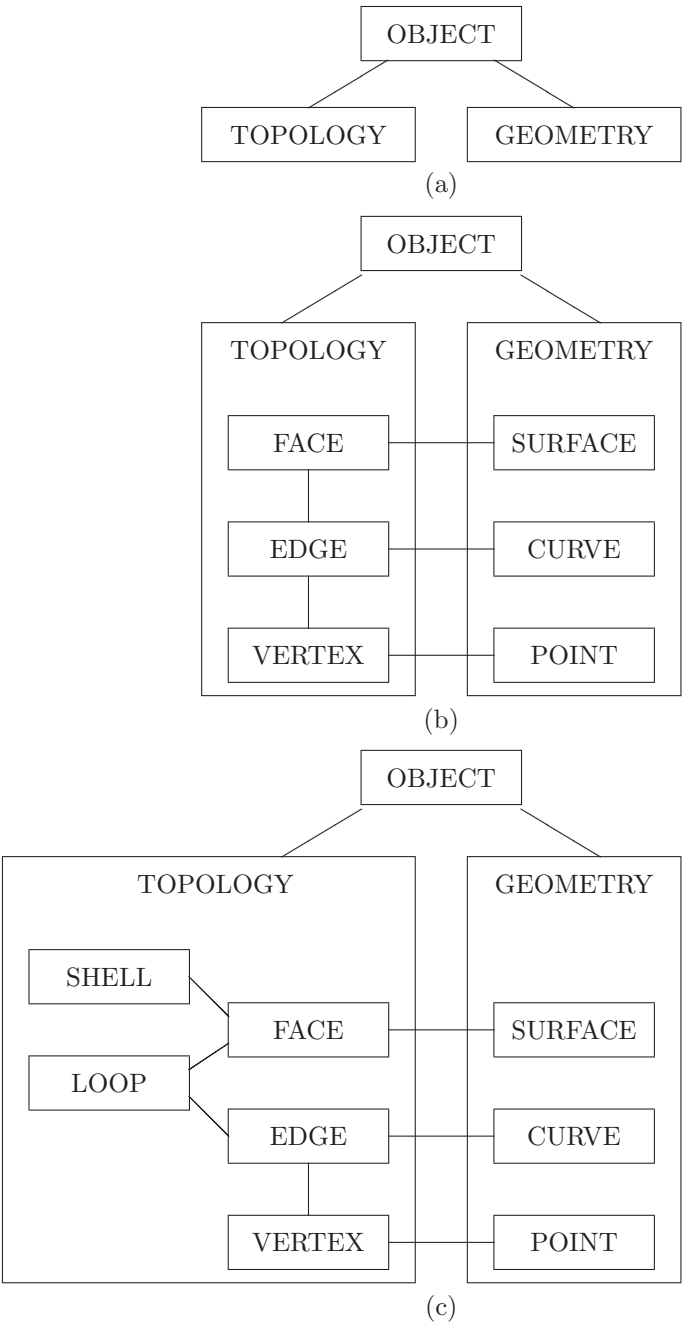


Figure 3.2: Basic data structures

3.1.1 Open questions

There is no single B-rep model datastructure that is superior to all others. For example, the winged-edge structure described in sections 2.2 and 2.3 can be implemented either directly (using direct pointers) or using links (see question 4). Assuming that the basic face-edge-vertex boundary representation is used, there are several options that remain. These are summarised below.

1. **Embedded or separate geometry:** Should the geometry be included in the topology elements, or should it be separate? The Euler operators described in section 3.3 and the algorithms described in chapter 4 assume separate geometry. Separating geometry and topology gives more flexibility for modelling operations, but it requires that some measure of the relevant portions of geometry be communicated for some geometry algorithms. Embedded geometry is only really relevant for facetted modellers that use planar approximations rather than exact geometry. With such models, only vertex positions need be recorded. The edge geometry can be determined from the positions of its end vertices, and the face surface can be inferred from the positions of three corner points. This type of modeller, though, seems to have only limited use, so separate geometry will be assumed in the following.

2. **Multiply or singly connected faces:** Should there be loops in faces? The Euler operators in chapter 4 and the algorithms in chapter 6 assume that these are present. Singly connected faces, with only a single set of bounding edges for a face, imply a restriction on the models that can be handled, whereas a multiple connection includes the case where a face has a single boundary. For this reason, multiply connected faces are assumed. Singly connected faces need a sort of 'fake' edge (see question 8) to connect inner edge-loops with the outer loop.

3. **Edge loop equivalents for vertices:** Loop equivalents for vertices allow, for example, two bodies to be joined at a single vertex. They also mean that the duals of faces with loops (see section 6.10) can be represented correctly, instead of representing them with coincident but distinct vertices. An alternative is to link coincident vertices with pointers, so that each each vertex is a normal 'manifold' vertex, but more information is available in the datastructure. This is discussed in more detail in chapter 5 describes non-manifold modelling in general.

4. **Topology connections:** The exact implementation of the connections in the topology can vary. This can cause problems if algorithms access the datastructure directly; hence, it is better to access the datastructure via a functional shell, as described in this

chapter. However, such implementation details must be decided as part of the basic modeller design in order to be able to write the functional interface. The same basic connectivity is present, but the form has to be decided. One option is the direct pointer structure as in BUILD [11]. Another alternative is to use loop-edge links, so-called 'half-edges' (see Mäntylä [82], for example). These connections are described in section 3.2.

5. **Structuring mechanisms:** These mechanisms, such as feature and shell representations and faceset grouping must also be decided as part of basic modeller design. They offer benefits, and possibly an improved model, but increase the cost of the modelling algorithms. It is assumed in the 'ideal' datastructure that such elements will be present in the model, because their absence implies a simpler task for the modelling code.

6. **Extra model types:** These provide extra information about the state of a particular model, but also incur extra 'housekeeping' costs for the modeller. As information about the model status is relevant, even if the model has to be analysed to retrieve it, it seems better to include them as part of the basic model.

7. **Geometry set for the models:** This is also part of the basic modeller design. There are three basic questions:

   1) Exact or approximate geometry? 2) What curve and surface types will be included? 3) How will the geometry be implemented?

   There is an important restriction that the geometry is closed under the set of modelling operations, but otherwise it is a matter of modeller design. Making the choice is largely a matter of trade-offs. Approximate geometry improves the robustness of a modeller by simplifying the geometry but risks errors where the approximation is insufficient, and causes fragmentation of the model. An increased number of geometry types gives more information about a model but increases the number of cases that have to be dealt with. The representation of the geometry can take various forms, such as multiple representation, parametric, and point based. See section 3.1.3.

8. **Fake edges:** An important decision is how the point-in-face interrogation will work. Once this has been decided, it is known whether or not fake edges are needed. If faces are not allowed to extend through more than 180 degrees, then they can be projected unambiguously onto some plane. 360 degree faces are more 'natural' in some sense but cause various problems for modelling algorithms. See also section 3.1.4.

9. **Extra information in the model:** Although adding information to a model may be desirable to improve the information flow to other applications using the model, it is pointless if the information is going to be lost or invalidated during modelling. This restricts the use of information to annotating a finished model or requires significant changes to modelling algorithms. See chapter 8 that describes possible information associated with a model, and strategies for handling it.

3.1.2 Elements of the datastructure

The datastructure elements for a boundary representation modeller, and wireframe extensions, are:

VERTEX	POINT
EDGE	CURVE
LOOP	
FACE	SURFACE
FACEGROUP	
SHELL	
WIREFRAME_OBJECT	
SHEET_OBJECT	
VOLUME_OBJECT	

INSTANCE
GROUP-OF-OBJECTS

FEATURE MECHANISM_CONSTRAINT
SHAPE_MODIFIER CONSTRAINT_DATA
SUPPLEMENTARY_GEOM PASSIVE_DATA
TRANSFORMATION GENERAL_GROUP

These are defined as follows:

- VERTEX: A vertex is a node of the datastructure, lying at a point in space.
- EDGE: An edge is a segment of a curve, running between two vertices. In an Eulerian model (i.e., not a wireframe model) edges lie in two loops, or possibly occur twice in the same loop (such as edge 5 or edge 10 in figure 3.33).
- FACE: Faces are portions of surfaces. Faces are bounded by loops, which are ordered sets of edges.

- LOOP: In its simplest form, the model datastructure consists only of faces, edges, and vertices. However, this does not allow multiply connected faces, where there is an outer boundary and one or more inner boundaries. To allow for this kind of model, the edges bounding a face are divided into closed circuits of edges, called Loops.

- FACEGROUP: Faces can be grouped together as logical units called facegroups. Each facegroup forms a sub-unit according to some modelling criteria, e.g., that the faces in the facegroup have a common surface, or that they were produced in the same basic operation. Facegroups are for use in the modeller, not for a user. The feature structures, defined later, are more suitable for use outside the system.

- SHELL: Each closed set of faces in the object forms a Shell. It is useful to represent these shells explicitly in some way in a model, rather than having to retrieve the information by traversing a faceset to see if it is closed.

- WIREFRAME_OBJECT: Wireframe objects are collections of edges and vertices only, face and loop information being ignored.

- SHEET_OBJECT: Sheet objects are degenerate models of, e.g., thin plate objects, with the small side faces represented by edges. They are considered in this book as being Eulerian (see section 3.3) and locally manifold.

- VOLUME_OBJECT: Volume objects are 'complete' solid models, with closed sets of faces bounding volumes of space. The models considered here are Eulerian manifold models.

- POINT: A zero-dimensional entity, a position in 3D Euclidean space defining the position of a vertex.

- CURVE: A one-dimensional entity defining the shape of an edge.

- SURFACE: A two-dimensional entity defining the shape of a face.

- INSTANCE: An example of a prototype model (single object or group of objects) with an associated transformation. Note that this is virtually the same as the use of instancing in graphics. See e.g. Newman and Sproull [91].

- GROUP-OF-OBJECTS: An assembly or collection of instances creating a single logical unit.

- FEATURE: As well as the facegroup structures for grouping faces, it is advisable to have an extra face grouping mechanism for preserving feature interpretations in the model. These differ from the facegroup mechanism in that a face can belong to several different feature facesets,

either because the same face is allocated as part of two different features or because there are parallel feature interpretations of a model. These feature datastructures should be in the form of 'frames' (Minsky [86]) to preserve the interpretations of the roles of the various faces grouped into a feature, or as simple groups of faces.

- MECHANISM_CONSTRAINT: For defining relationships between sub-models in a group-of-objects, or between sub-models and supplementary geometry.
- SHAPE_MODIFIER: For defining implicit modifications to model elements, e.g., implicit blends on edges, or screw-threads for circular holes or projections.
- CONSTRAINT_DATA: For defining relationships such as distance or angle between elements in the model.
- SUPPLEMENTARY_GEOMETRY: Supplementary geometry is used as a design aid, and for helping specify constraints between instances in a group-of-objects.
- PASSIVE_DATA: Information intended to be communicated with the model. For example, material, colour, and price.
- TRANSFORMATION: Transformation data describe how a model is to be modified, reflected, scaled, translated, or rotated. Other transformation data are possible, shearing, for example, but they are generally complicated to apply in practice. Transformations should always be invertible; hence, zero scaling in any direction should be disallowed. Valid transformations affect the geometry, not the topology, and so the task of handling them belongs to the functional interface to the low-level modeller. Transformations in assemblies should be limited to rotations and translations that do not change the shape only the position of objects.
- GENERAL_GROUP: A useful general mechanism for associating elements for some purpose.

Suggestions for datastructure definitions for these are contained in Appendix A, section A.1.

In this 'ideal' boundary representation, the geometry and topology are kept separate, as mentioned earlier. This allows more flexibility in the modeller, but by divorcing the parts of the modeller, there is risk of ambiguity when the parts have conflicting information. The structure of a model, the faces, edges, and vertices, in a general class of models called Euler objects, can be kept consistent by using basic operators called Euler operators, described in chapter 4. However, the separate information means that it is possible that, for example, the edge between two adjacent faces might lie in a curve that lies in only one, or neither, surface of the faces. Similarly, the end vertices

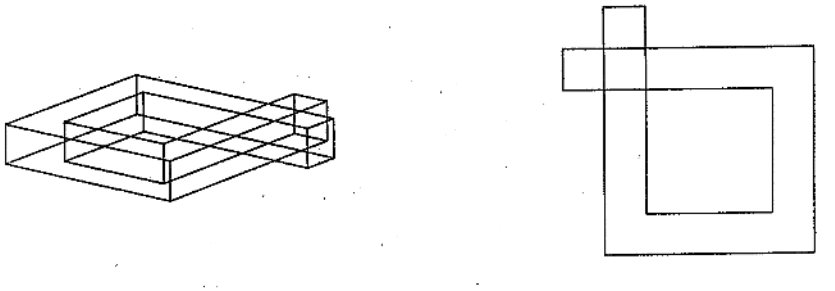


Figure 3.3: Self-intersecting object

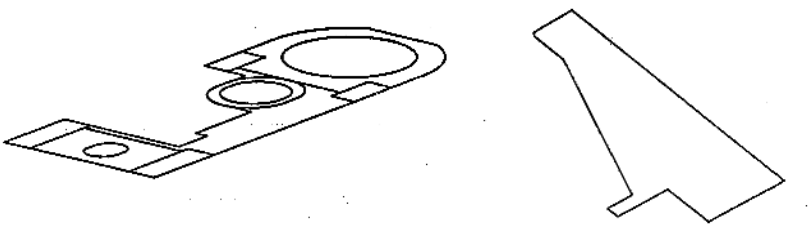


Figure 3.4: Two-dimensional objects

of an edge are not specifically constrained to lie on the curve of the edge. It is also possible to create so-called 'self- intersecting' objects, where one part of the object intersects another geometrically without having corresponding topology, as in figure 3.3. However, this separation allows more flexibility in the modelling system. For example, it is possible to represent two-dimensional objects, like those in figure 3.4.

Geometry, in its pure form, is considered as being unbounded, the relevant portion being bounded by other geometry related through topology. This has practical problems for intersection, for example, so the relevant portion of space has to be specified in some way. Bounding spheres or boxes can be associated with topological elements to define the interesting portion of geometry to improve efficiency and limit some geometric calculations. Bounding spheres are uniform shapes that can be handled easily, whereas boxes provide a closer fit to geometric elements. Another type of bounding measure, fat-sectors (see Várady [139]) provide a hollow measure that can be used to approximate curved hollow shapes.

Note that if, instead of the topological-based structures described above, the modelling system is designed so that the geometry is kept consistent, and the model structure related to that, the system becomes similar to CSG systems. The geometry would be controlled through surfaces, because curves and vertex positions are dependent on these and the role of the topology becomes unclear.

There are several methods for representing the topological connections between the elements in the model. One way is with the direct pointer implementation of Baumgart's winged-edge structure; each edge record in the model contains pointers to four adjacent edges, the start and end vertices, and the left and right loops (and hence the left and right faces). To avoid variable size entities, each vertex points to only one edge at the vertex, and each loop to only one edge in the loop; the other edges are found via the edge links. See figure 2.2. An alternative to having direct pointers is to arrange the edges into sequences around a loop using links (called, for example, "half-edges" or "coedges" or "winged-edge links"). This is an interpretation of the winged-edge mechanism, illustrating an alternative to the direct pointer implementation. Each loop points to a list of these links, one for each edge. This makes traversing loops of entities easier, but it is slightly less efficient for accessing adjacent edges. It is also possible to have a similar vertex-edge link, using these to relate connected edges in the same way as with loop-edge links. Topological connections are described further in section 3.2.

For convenience, in addition to the winged-edge structure, all vertices, edges, and faces can be linked into lists accessed directly from the object rather than by crawling over the datastructure. Vertices and edges can be linked into simple chains. Similarly, faces can be linked into a chain, or accessed as a tree structure using the FACEGROUP structure. This is an important extra facility as it allows fast access to the elements of an object for various modelling purposes. The presence of these direct chains is not strictly necessary. It is possible to use the face structure to find edges and vertices (as in the ACIS modeller). Having such extra lists increases the work that has to be done to maintain them, but there are benefits to having them; they provide duality that can be useful for checking as well as for fast access.

As an illustration of existing datastructures, the BUILD single object datastructure was shown in figure 2.1, and the GPM volume module datastructure was shown in figure 2.5, in SYSDOC form.

In addition to these datastructure elements for single objects, it is also important to be able to collect objects into groups, or assemblies. A group-of-objects consists of a set of instances of single objects or groups-of-objects; thus a group-of-objects can have a tree structure of single objects. The only requirement is that a group-of-objects does not contain an instance of itself as part of this tree stucture.

Besides the basic shape description elements of the model, there are various additional information entities that are necessary to be able to annotate the

model, to represent mechanisms, and to represent other properties. These entities are as follows:

1. MECHANISM_CONSTRAINT
2. SHAPE_MODIFIER
3. CONSTRAINT_DATA
4. PASSIVE_DATA
5. SUPPLEMENTARY GEOMETRY
6. FEATURES

These entities are discussed further in chapter 8.

As a final point about the basic datastructure elements for models, it should be noted that it is useful to include an identity tag or tags for the elements. A naming mechanism is essentially passive for modelling and provides a user 'handle' on the model. In addition it is also important to have a system identity number for each entity for system use. This is used, for example, for the dual algorithm in section 6.10, for copying objects (see Appendix C section C.10) or for disc input/output (see chapter 11).

3.1.3 Geometry

Three basic questions concerning geometry were identified in section 3.1.1:

1. Exact or approximate geometry?
2. What types of geometry will be included?
3. How will the geometry be represented?

The first question has already been dealt with; exact geometry is considered here. Determining what types of geometry will be included is partly a basic design decision in itself and influenced by the operational characteristics of the modeller. The basic requirement on the geometry is that it forms a closed set under the modelling operations included in the modeller. For example, suppose it has been decided that only straight lines and planar surfaces be represented in the modeller; then operations such as the Boolean operations and sweeping pose no problem, but swinging (sweeping a profile round an axis) is not allowed because it would be possible to create rotational surfaces and circular curves, such an operation would have to be approximated so that the rotational shape is approximated by a set of wedge-shaped elements.

A 'reasonable' set of surfaces for modelling is as follows:

PLANES
SPHERES

CYLINDERS
CONES
GENERAL QUADRICS
TOROIDS
FREE-FORM SURFACES

and the set of curves:

STRAIGHT LINES
CIRCLES
ELLIPSES
PARABOLAS
HYPERBOLAS
FREE-FORM CURVES

This set contains special representations for the most common geometries, with the general quadric and free-form surfaces as closures for the surface set, and the free-form curves to close the curve set.

The actual choice of geometry is variable, and for the purposes of this book, it is not relevant. As with topology, the geometry should not be handled directly but via a functional interface that deals with the general classes: SURFACE, CURVE, or POINT rather than with specific geometry types. It should not matter whether, for example, toruses are represented as special surface types, or as free-form surfaces. In fact, all geometry could be handled using free-form representations, if required, although this is somewhat extreme. What is interesting, here, is the way in which the geometry is addressed, not the actual geometry. See section 3.3.4. Example geometry definitions are contained in Appendix A, section A.2.

The question of whether or not to use parametric representations for curves and surfaces is also a matter of philosophy. Both parametric and non parametric representations are possible and have their own characteristics. Parametric interrogation of a curve or surface is a common necessity but can be calculated for non parametric curves and surfaces as well. As stated, the important thing for general modelling algorithms is to provide a geometric functionality.

Finally, practical points to note concern defining the geometric domain and tolerances.

When using unbounded geometry, it can be useful to define the relevant portion of space in which a geometric operation is to be applied. For example, when calculating an approximate intersection curve from two unbounded surfaces, it is important, for the sake of efficiency, to be able to tell when intersection points are not required, such as when the intersection curves are infinite.

Geometric tolerance problems are notorious and not easily solved, because of the basic nature of computer representation of real numbers. Solomon performed experiments using fractions in geometric representations. Although

he did not pursue the experiment because of efficiency problems, the method has since become more practical. Techniques using real numbers still predominate, though. These, too, have improved as hardware costs have fallen and the former 'double precision' has become the norm. These improvements, however, have not eliminated the problem, merely alleviated it, and more work needs to be done on geometric stability.

3.1.4 Arbitrary representations

This section concerns arbitrary representations and associated structures. This means the decisions made by the implementer have to be made and treated consistently but are not dictated by technology. One question already mentioned associated with the geometry and with the topology is whether to use 'fake' edges in the modeller. These edges are needed if faces extending through more than 180 degrees or through a whole circle are not allowed in the model. If faces are not allowed to extend through more than 180 degrees, then the edges limit the extent of a curved face so that it can be uniquely projected onto a plane. If a face can extend through 360 degrees, but must have a single fake edge, then a ray extended through the face is guaranteed to cut at least one edge. If no fake edges are required, then the ray may lie entirely within the face, so care has to be taken when determining the ray direction. Figure 3.5 shows examples of cylinders with different options for fake edges. In figure 3.5a, the cylinder is represented with three fake edges around the curved side. In figure 3.5b, there is a single fake edge, and in figure 3.5c, there are no fake edges. Faces with no fake edges make some operations simpler, such as circular sweeping, but can add a great deal of complexity elsewhere. Trying to check the orientation of a face extending through 360 degrees can be really difficult. Having fake edges can lead to object fragmentation, which is difficult for a user to understand.

In addition to the basic datastructures described in section 3.1.2, it is possible to add extra information to a model for communication purposes. Unfortunately, this tends to make models a little like overly dressed Christmas trees, so this facility should be used with care. With current techniques, it is best to add these details after modelling has been completed to avoid information conflicts and loss during modelling.

The trend to add information to models has come as they have developed from purely shape descriptions towards product models. The problem for solid modelling is that the information has no direct connection with the shape description that is dealt with by the modeller. If, for example, a user wants to add two objects, one marked as being made of steel and the other marked as being made of wood, then the modelling algorithms will happily produce a combined shape, but it is not clear what happens to the material tags. The result might be steel or wood or both or neither, depending on how the tags are handled. Chapter 8 deals with the topic of information handling in modelling in more detail, for now it is enough to consider how

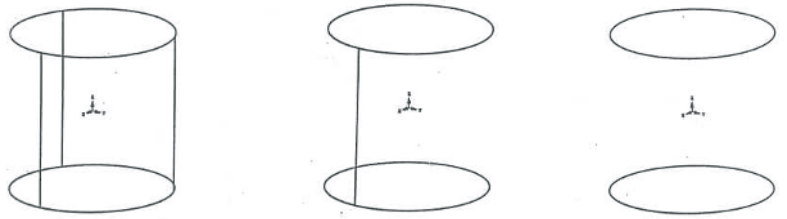


Figure 3.5: Fake edge options

such mechanisms may be implemented.

One important kind of tag, which has already been mentioned at the end of section 3.1.2, is name tags. These allow users to refer to parts of an object by some meaningful name, but they are not of use for the modelling algorithms. There is a slight drawback to names in that they can allow the user to trace what happens to entities during arbitrary modelling operations. If, for example, a named face is split by adding a new edge, then unless the user is required to specify which part is the new face, the name could be assigned to either part. It is possible that the name does not remain with the part that the user wants. It is unnecessarily restrictive to remove all names on modified entities because, if for example a user wants to inscribe a shape on a named face, then it is useful that the exterior face remains named. The way that names are handled should be built into the modelling system so that arbitrary modelling operations on named entities are made to require more information from the user. Names could be handled as passive data (see chapter 8), but this implies that there could be several entities with the same name. This is not impossible to handle, though, but could require user intervention if an operation is specified on a named entity where there are several entities with that name.

There is a topic, termed persistent naming, on how to name entities and maintain names during modelling. However, the topic is dynamic because it is the subject of research; hence, it will not be dealt with further here, and the reader should refer to specific articles for more details.

3.2 Topological connections

As described previously in this chapter, the topological connections in the model can be represented in several ways. Three ways of connecting edges together are "winged-edge" links, loop-edge links, and vertex-edge links. These are described in this section. The reader should note, again, that this low-level representation should be surrounded by a functional interface of topological manipulation and traversal routines, as described in this chapter, to avoid needing to be too dependent on one type of structure.

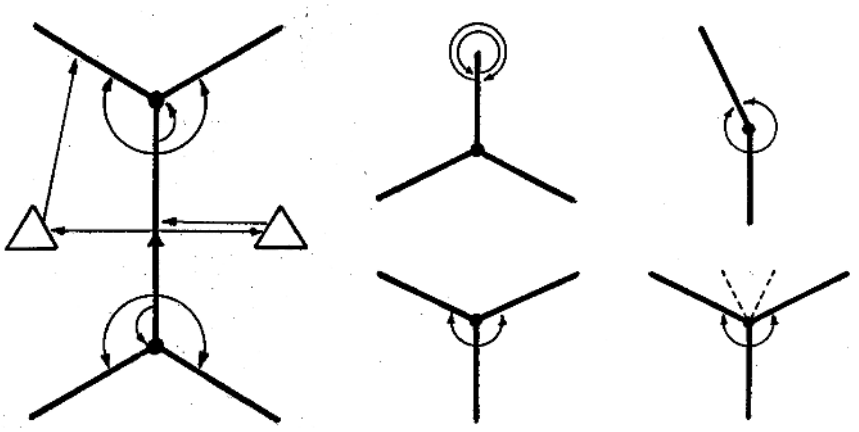


Figure 3.6: Winged-edge pointer connections

With the winged-edge link datastructure, the edge is the link to everything. There are eight connectivity pointers:

1. The right loop
2. The left loop
3. The start vertex
4. The end vertex
5. The right clockwise edge
6. The right counter-clockwise edge
7. The left clockwise edge
8. The left counter-clockwise edge

Figure 3.6 shows these pointers and various special cases that have to be considered. Figure 3.6 (top middle) shows the winged-edge pointers when there is just one edge at the vertex (a spur vertex), figure 3.6 (top right) shows the winged-edge pointers when there are just two edges at the vertex, figure 3.6 (bottom middle) shows a trihedral vertex, and figure 3.6 (bottom right) when there are more than three edges at the vertex. Note that in a properly connected edge, there are always two pointers around the vertex, even if they point back to the edge itself, or if they point to the same edge or if there are more than three edges at the vertex. The pointers always refer to the next edge immediately adjacent to the edge around the vertex.

With the loop edge connection method, the edge has two link entities attached to the right and left loops. The connection information is thus

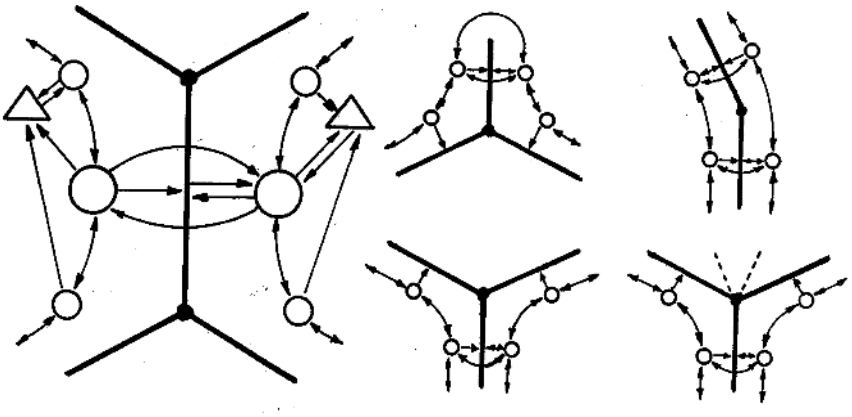


Figure 3.7: Loop-edge link connections

contained in the links, whereas the edge retains the common information, the curve reference, data references, and so on. The links have various names; "half-edges" is one well-known one; they were called "winged-edge links" in the GPM volume modeller, but the function is the same. Loop-edge links can be connected together around the edge so that more than two faces are allowed to meet at the edge, making a so-called "non-manifold" edge. This is described in chapter 5.

Figure 3.7 is the equivalent of figure 3.6 for loop-edge links. When there is only one edge at the vertex, the loop-edge links of the edge are adjacent to each other in the loop (figure 3.7 top middle), with two edges at the vertex (figure 3.7 top right), a trihedral vertex (figure 3.7 bottom middle), and when there are more than three edges at the vertex (figure 3.7 bottom right).

Vertex-edge links are the 'dual' of loop-edge links. Figure 3.8 shows the arrangement of the links and the same four special cases as in figures 3.6 and 3.7.

As an illustration of the types, the following figures show how a cube may be built up. The winged-edge sequence is as in the BUILD modeller, which builds a cube by creating a minimal object (body with one face and one vertex), sweeping this to a straight edge, sweeping this edge to make a square lamina, and then sweeping the lamina up as in figure 3.9.

The final cube, with face, edge, and vertex numbers, is shown in figure 3.10. Numbered triangles represent faces (and loops), numbered squares represent edges, and circles represent vertices.

The following figures show the construction sequences for datastructures based on winged-edge pointers (figure 3.11 to figure 3.19), loop-edge links (figure 3.20 to figure 3.24), and vertex-edge links (figure 3.26 to figure 3.31). The sequences use Euler operators, which are described in chapter 4. The first makes a basic body with one face and one vertex, the simplest possible

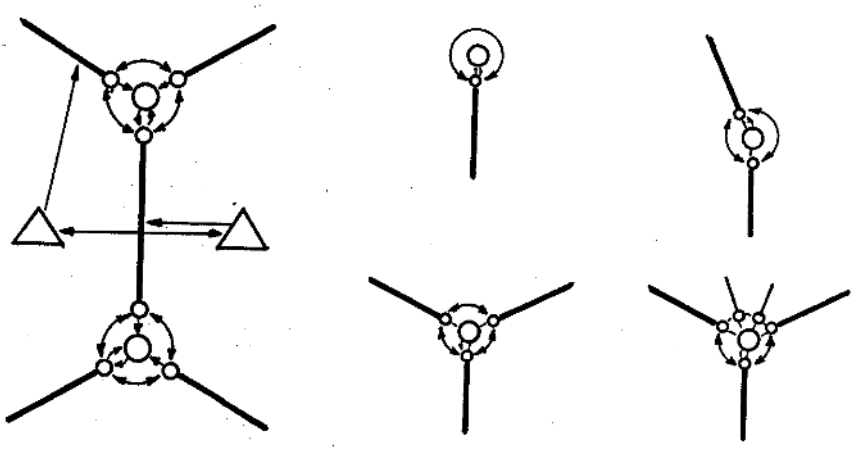


Figure 3.8: Vertex-edge link connections

Eulerian body, and then the remaining vertices, edges, and faces are added with Make Edge and Vertex and Make Face and Edge operations. A more concise illustration of this is shown in figure 4.5.

The first step in each sequence is to create a minimal object consisting of one face and one vertex. As stated before, it should be possible to connect the vertex and the loop of the face directly, not just via edges so this case can be represented exactly. The second step is to add an edge and a vertex, making a spur edge with two spur vertices. For a structure based on loop-edge links, this creates a loop with two links, both pointing to the same edge. Two more edges and vertices are added, and a third edge is added creating a new face. The new face is added on the right of the new edge and points to the new edge as the first in the loop. Proceeding clockwise around the faces, the first face has edge order: 1, 3, 4, 2, and the second has the order: 4, 3, 1, 2. The order of traversing the edges, clockwise or counter-clockwise, is a matter of convention only. Here the loop-edge link figures show a clockwise traversal; the vertex-edge links also show clockwise traversal.

From the base, two upright edges are added from vertex 4 and vertex 3, respectively, and the two new spur vertices (vertices 5 and 6) are connected with a third new edge (edge 7), creating another new face. Another new upright spur edge is created from vertex 1, and the spur vertex (vertex 7) is connected to vertex 6 with a new edge, creating face 4. The last upright is added from vertex 2, and the new spur vertex (vertex 8) is connected to vertex 7 creating face 5. Finally an edge is added between vertex 8 and vertex 5 creating face 6.

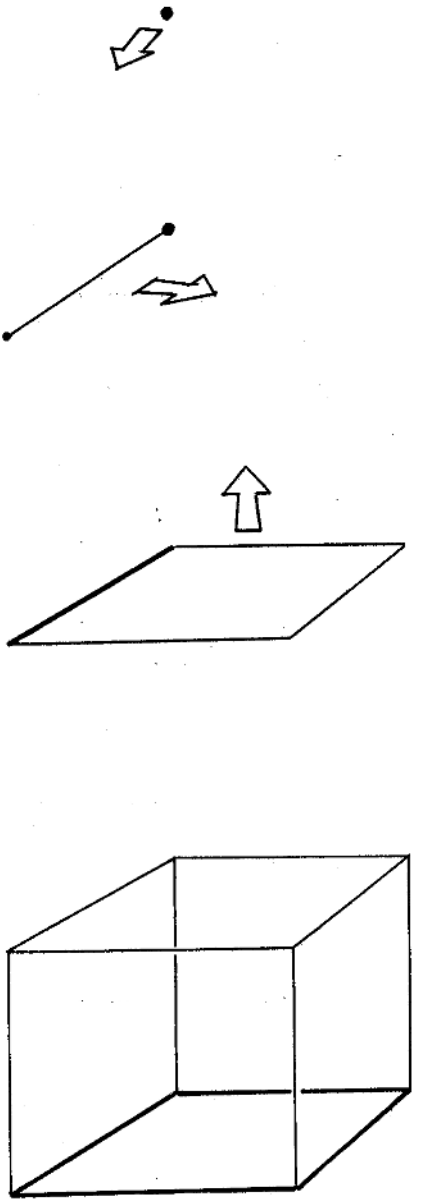


Figure 3.9: Creating a cube in BUILD

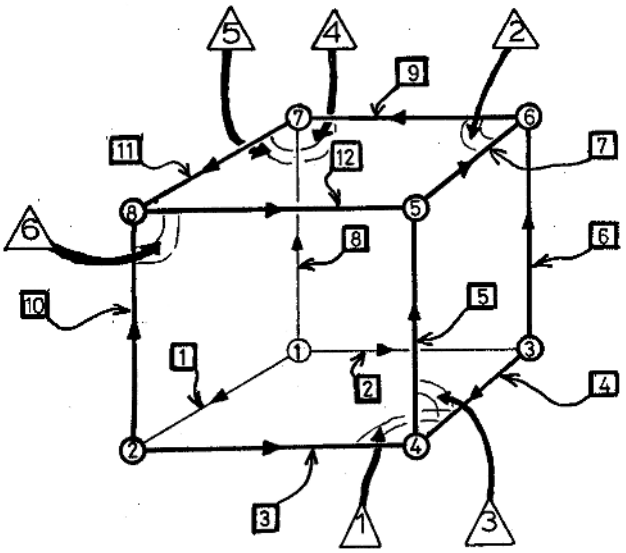


Figure 3.10: Final BUILD cube structure

As can be seen, all of these representations work in that the model connections can be made using them. The functional interface described in the next section should isolate the datastructure level from the modelling system so that the basic connection representations need not be known. The loop-edge link structure is currently popular because of its ability to represent non-manifold connections, but both the winged-edge pointer and the vertex-edge link structures could be made to represent non-manifold objects as well. Vertex-edge link structures are not, as far as I know, used in any system, but the representation is just as valid as the other two.

3.3 Modelling facilities

The questions identified in section 3.1.1 cannot be answered solely on the basis of the B-rep modelling technique; they are bound up with the functionality required of the modeller. This means that there is a certain amount of uncertainty about the datastructure, and hence about how modelling tools will be implemented. However, several basic tools help to offset the variations imposed when choosing particular answers to these questions. These tools provide a more standard functionality as an interface between the modelling algorithms and the datastructure. This section describes several of these.

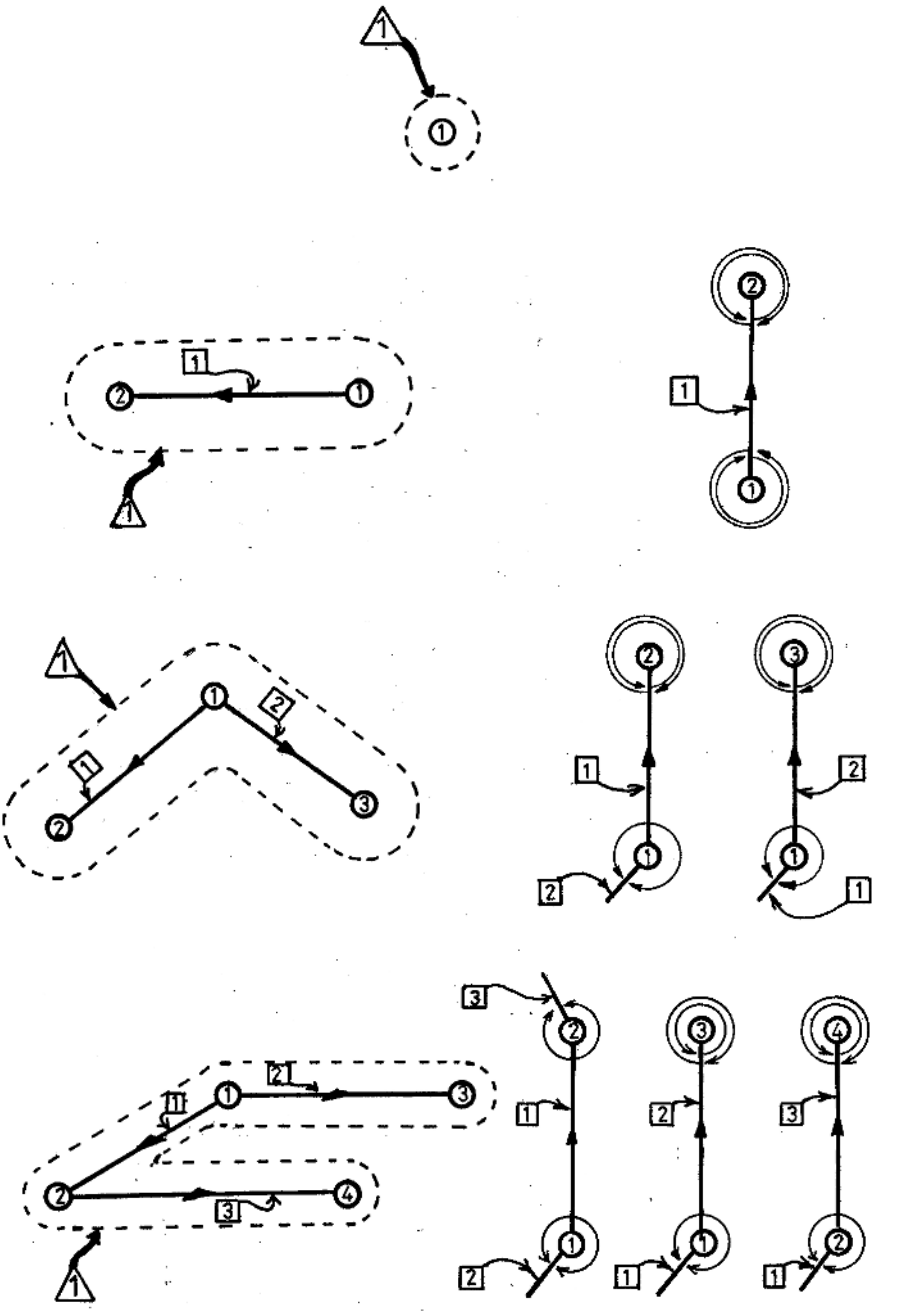


Figure 3.11: Building a cube with winged-edge links (1)

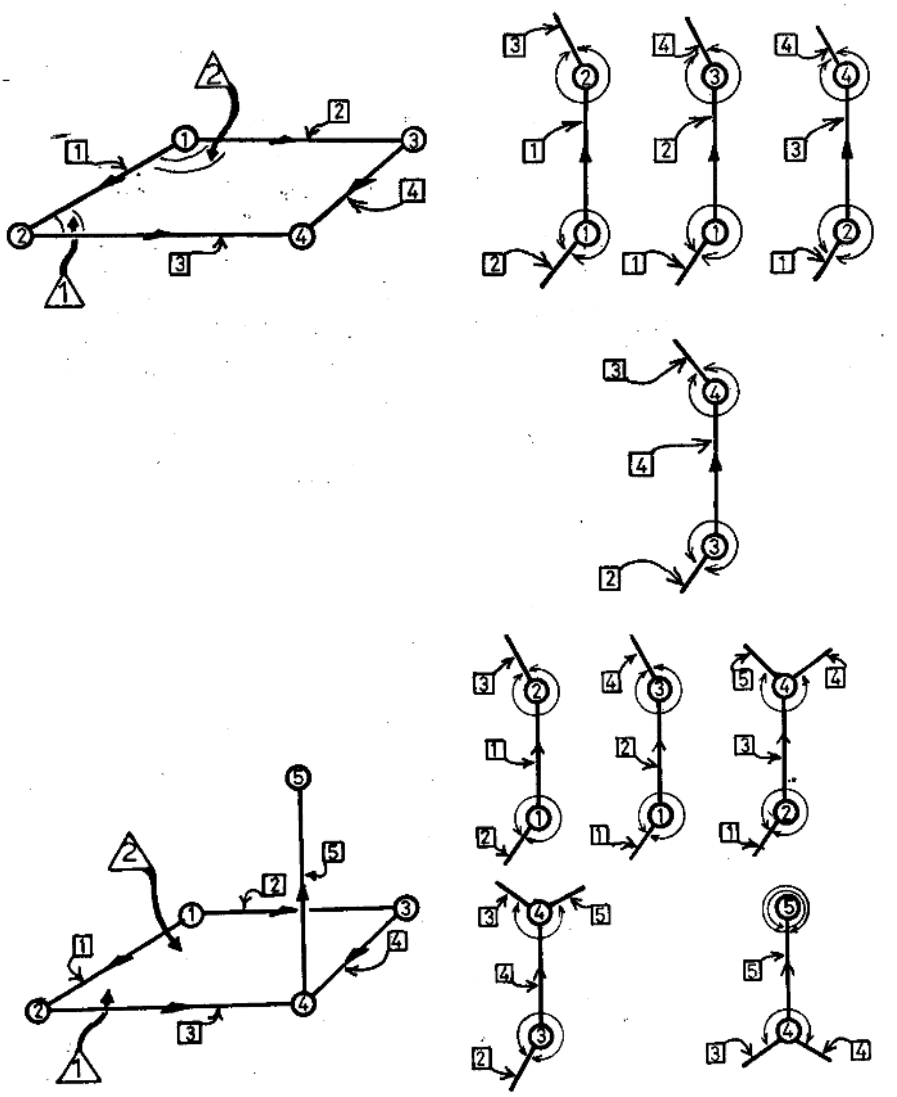


Figure 3.12: Building a cube with winged-edge links (2)

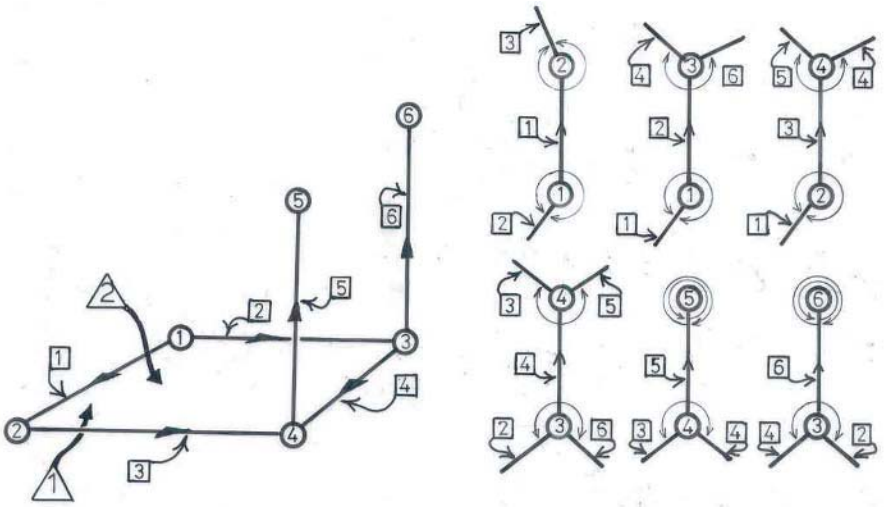


Figure 3.13: Building a cube with winged-edge links (3)

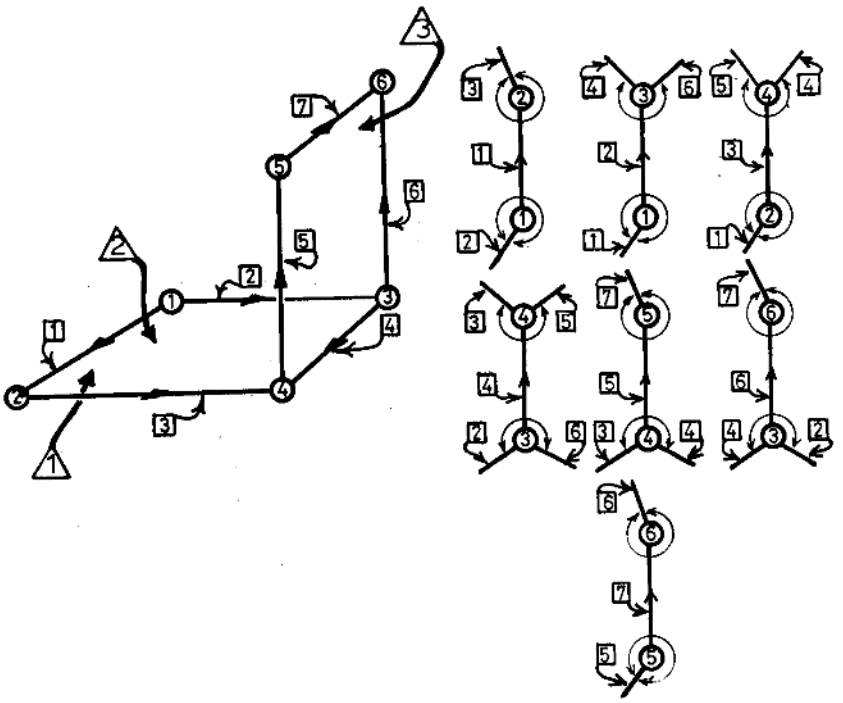


Figure 3.14: Building a cube with winged-edge links (4)

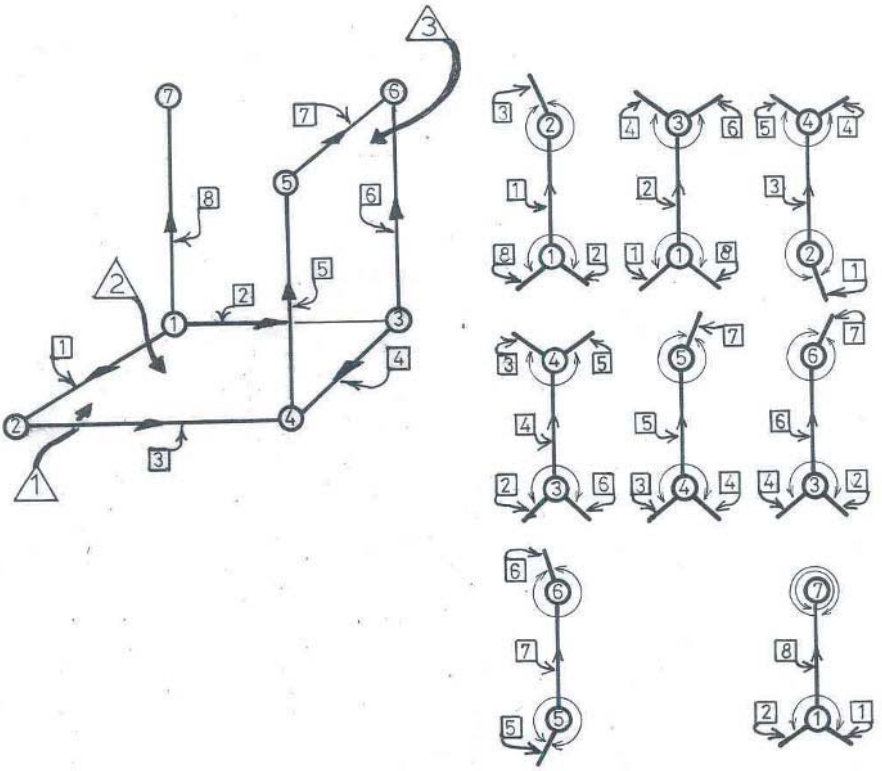


Figure 3.15: Building a cube with winged-edge links (5)

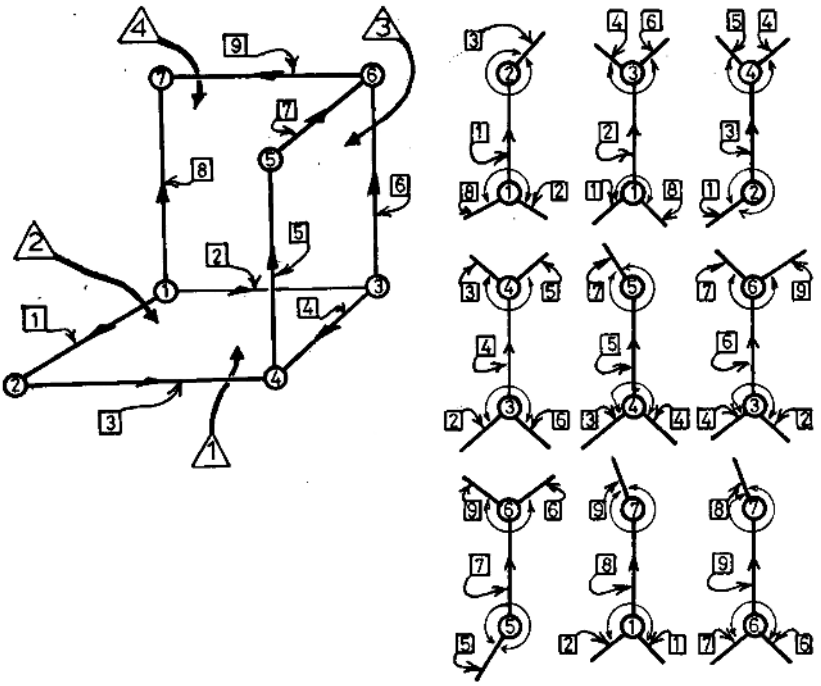


Figure 3.16: Building a cube with winged-edge links (6)

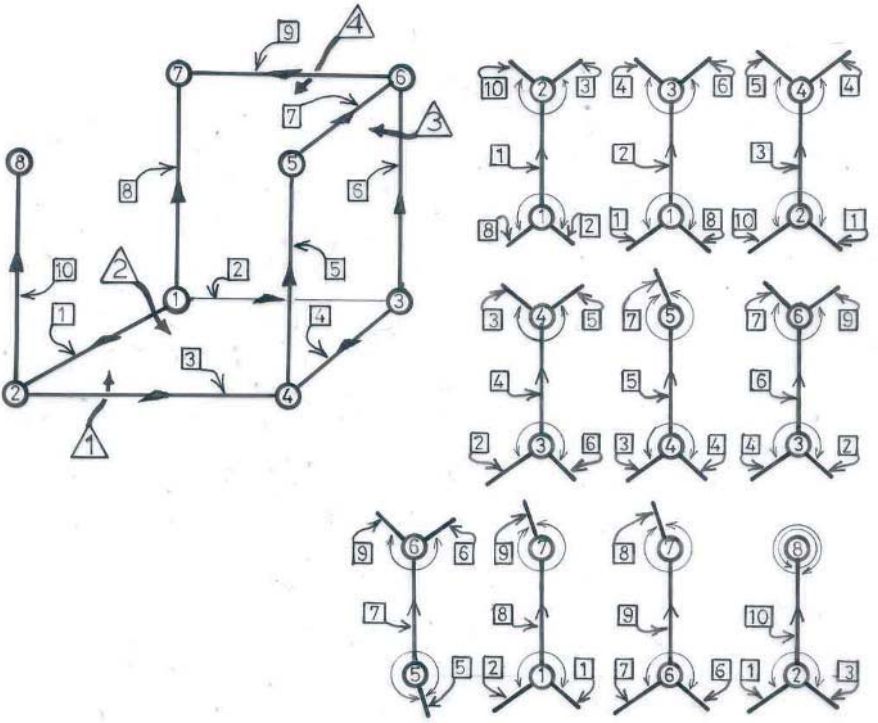


Figure 3.17: Building a cube with winged-edge links (7)

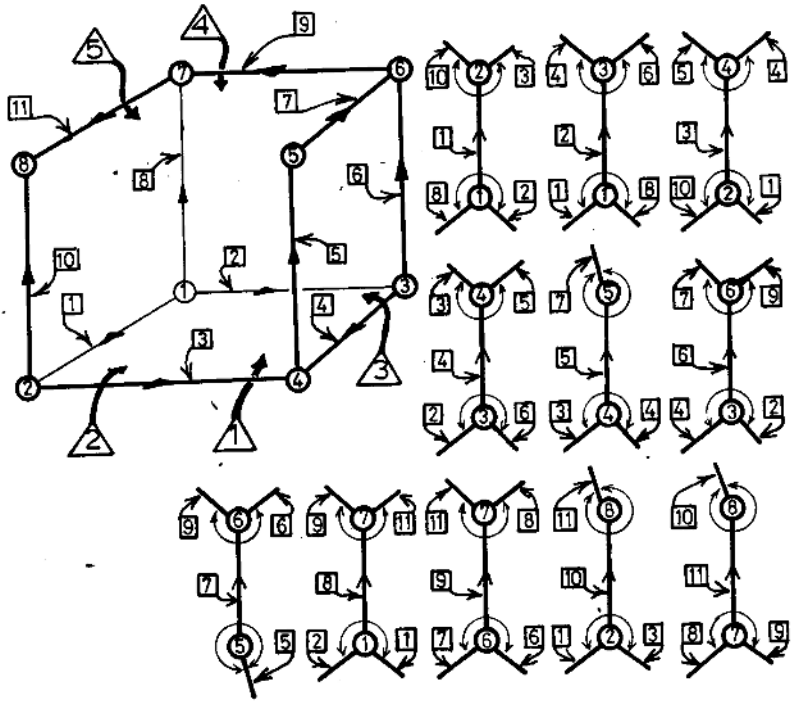


Figure 3.18: Building a cube with winged-edge links (8)

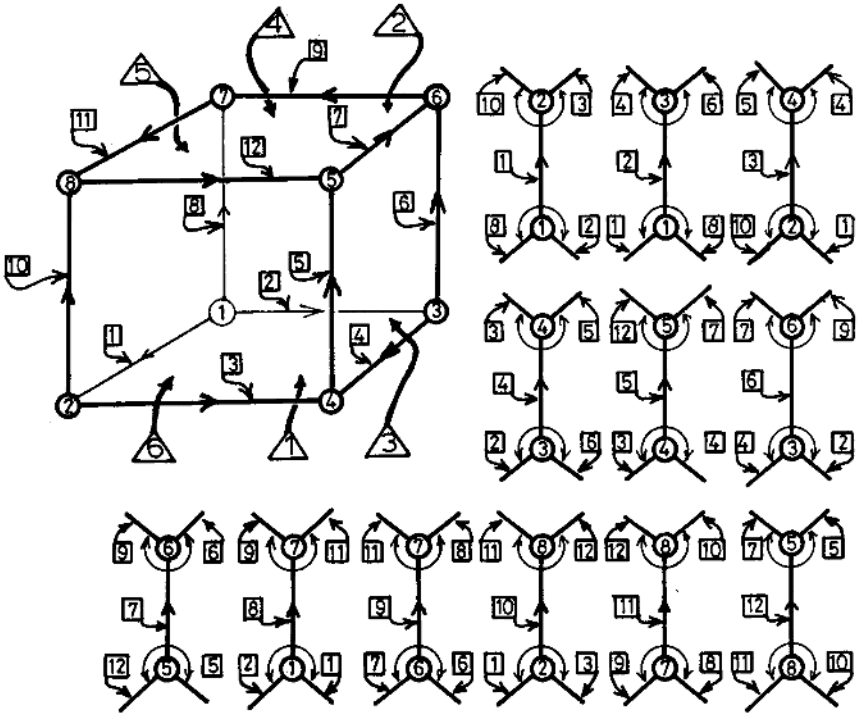


Figure 3.19: Building a cube with winged-edge links (9)

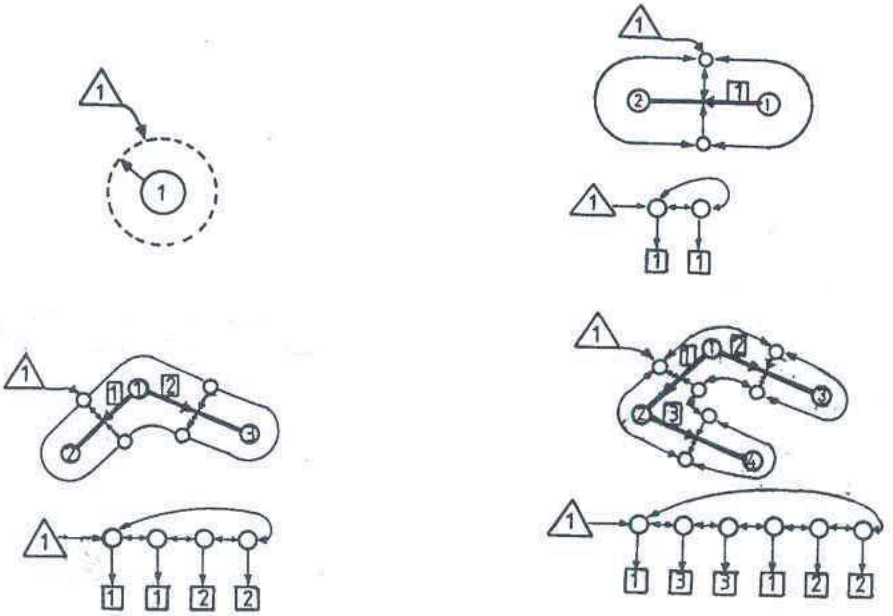


Figure 3.20: Building a cube with loop-edge links (1)

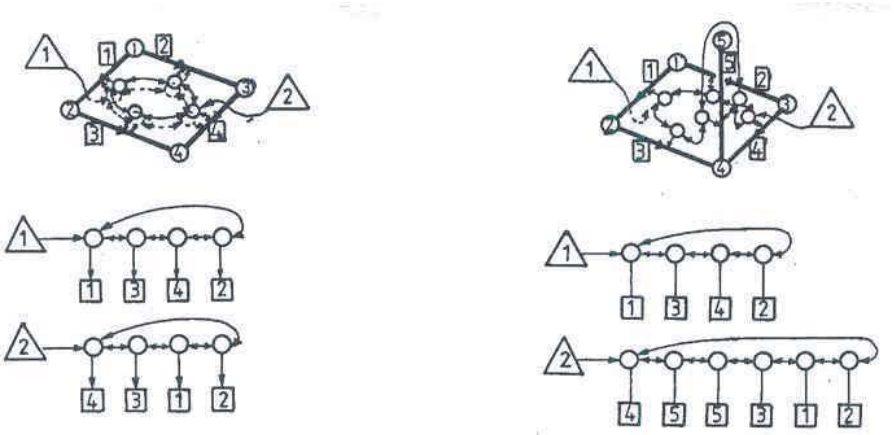


Figure 3.21: Building a cube with loop-edge links (2)

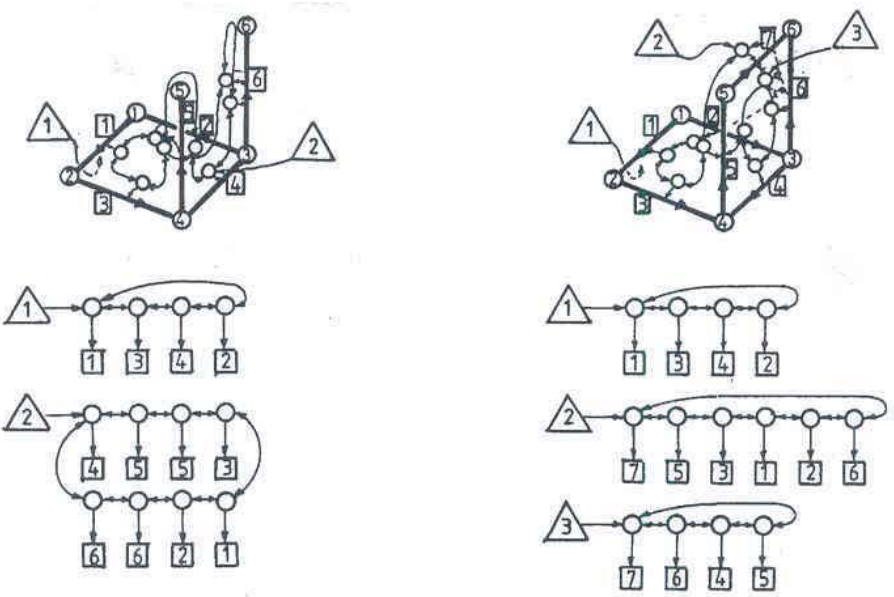


Figure 3.22: Building a cube with loop-edge links (3)

3.3.1 Traversing all related elements in a structure

The structure traversal procedures provide a standard way of accessing the datastructure without having to know the exact relationship between the owner entity and the subsidiary entities. They have a similar role to the "identify connected entities" function in the CAM-I Application Interface Specification (AIS) (Braid [12]).

The safest way of traversing model structures is to use the datastructure connections to traverse the model while building a separate list containing the identified elements, and then to process that list applying some function to each element. It is also possible to traverse the structures directly, processing each entity as it is found, in effect using the model datastructure as a dynamic list. The disadvantage of doing this, though, is that if the structures are altered while processing, for example, by deleting an entity, then the structure being traversed will change. Producing a separate list is less efficient, but it is safer than traversing the structure directly because the list structure is independent of the model datastructures that will be changed during modelling operations.

Basic modelling operations such as drawing and printing objects use the traversal procedures to traverse complete objects, applying procedures that deal with a single entity (i.e., "print edge" or "draw face"). They can also be used to find particular entities in the datastructure, such as "edge 23" or

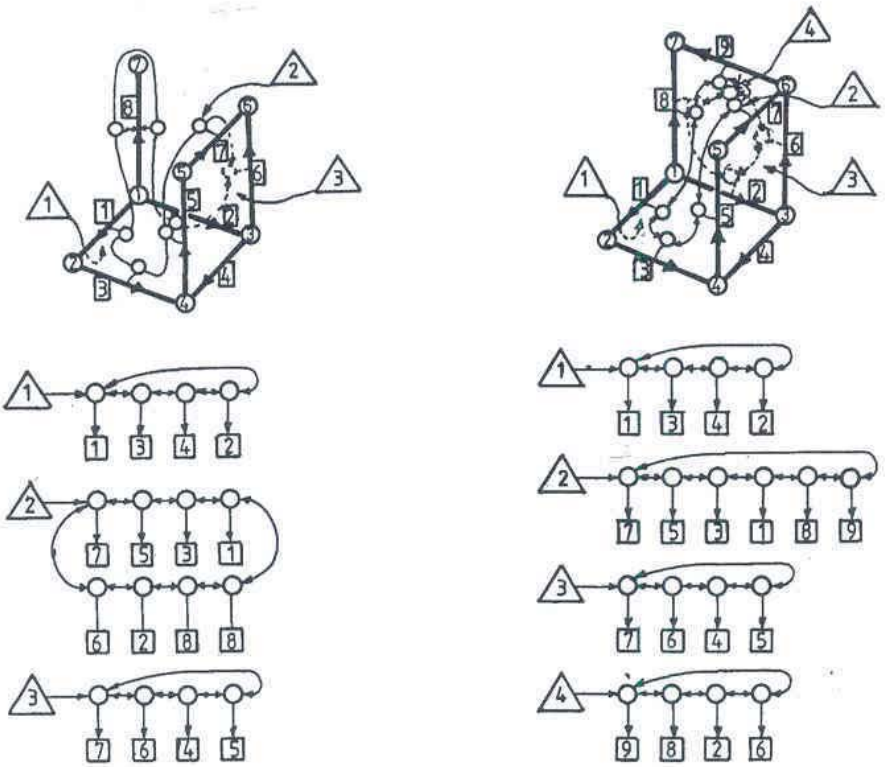


Figure 3.23: Building a cube with loop-edge links (4)

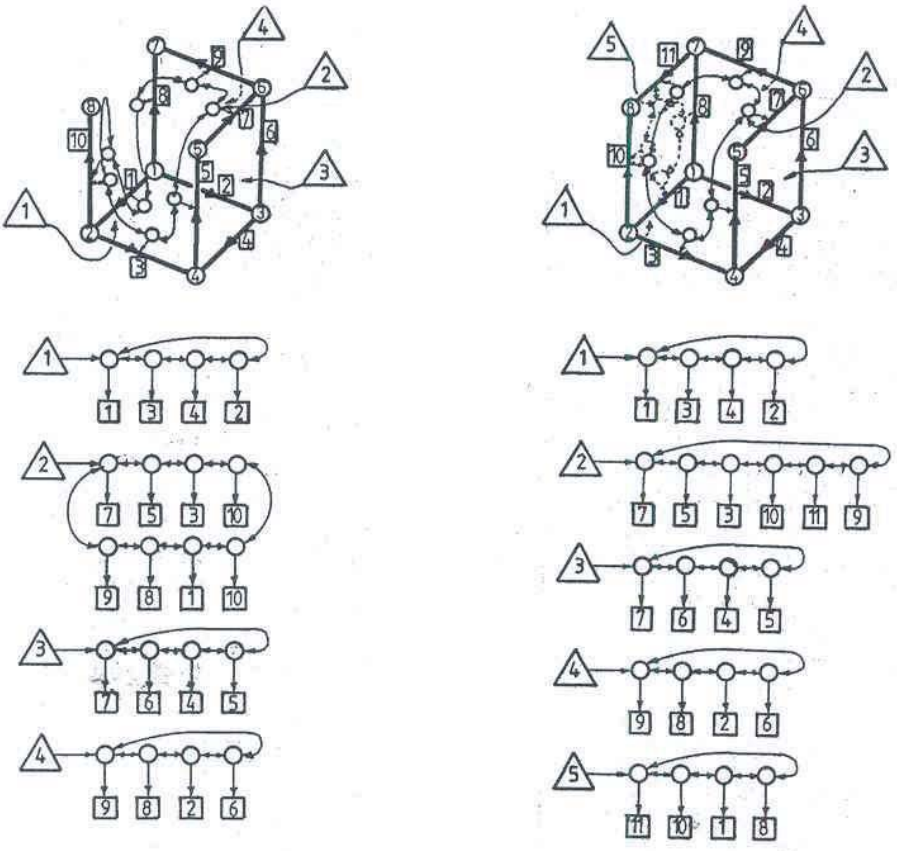


Figure 3.24: Building a cube with loop-edge links (5)

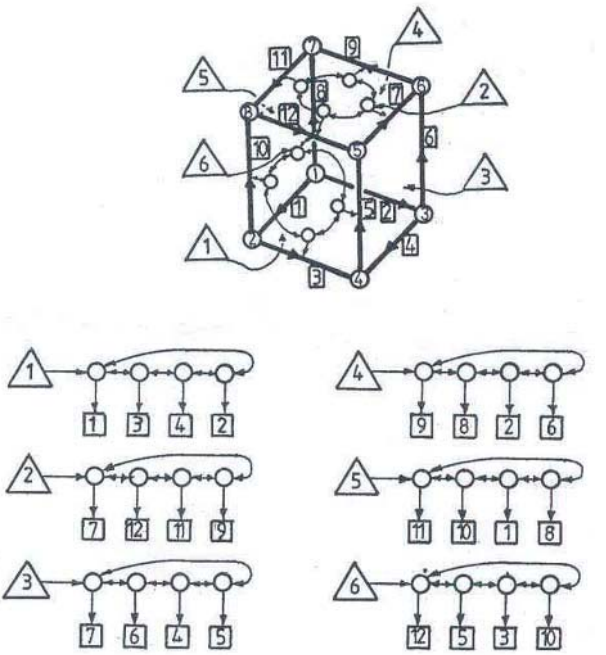


Figure 3.25: Building a cube with loop-edge links (6)

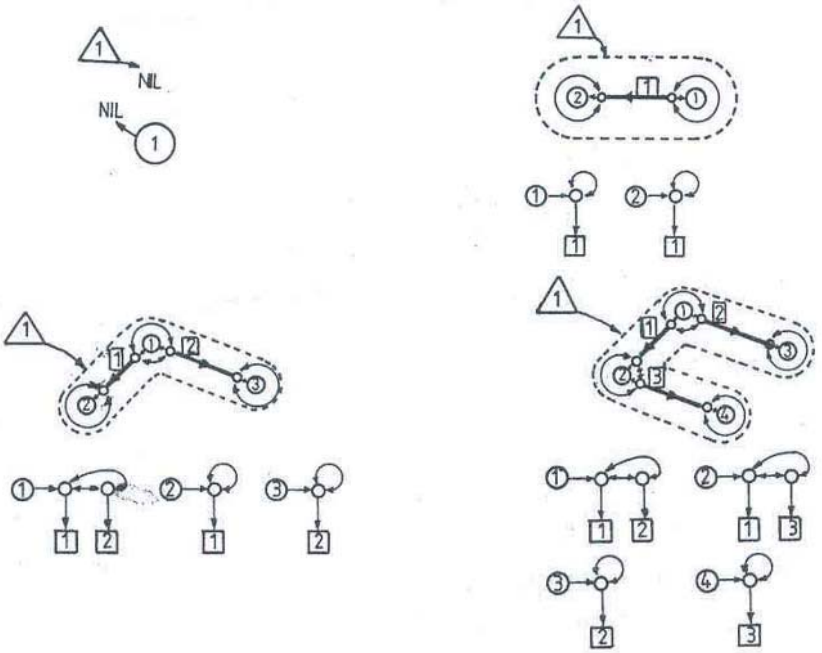


Figure 3.26: Building a cube with vertex-edge links (1)

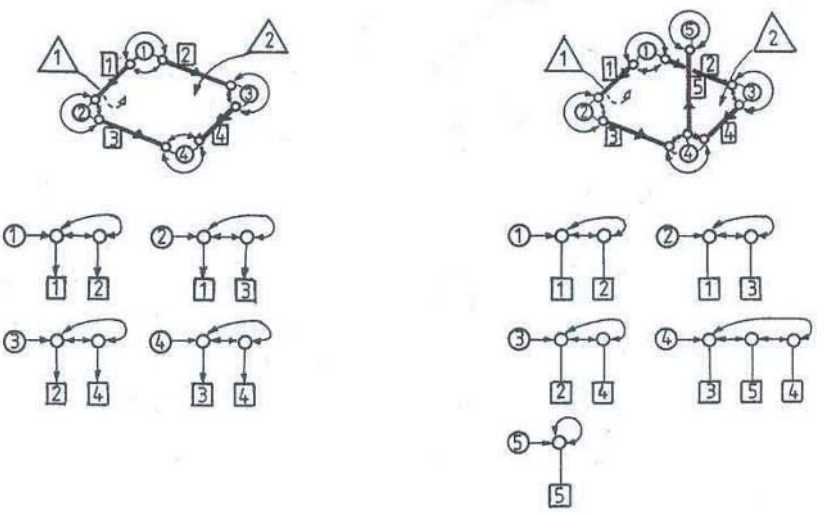


Figure 3.27: Building a cube with vertex-edge links (2)

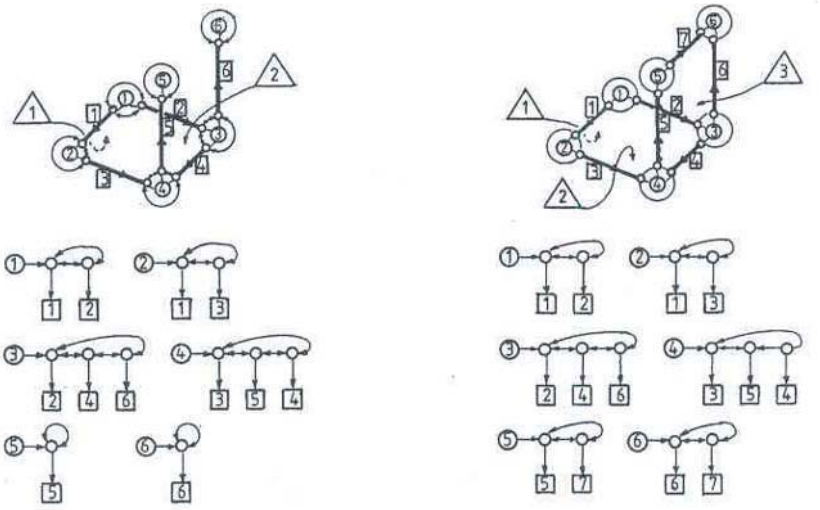


Figure 3.28: Building a cube with vertex-edge links (3)

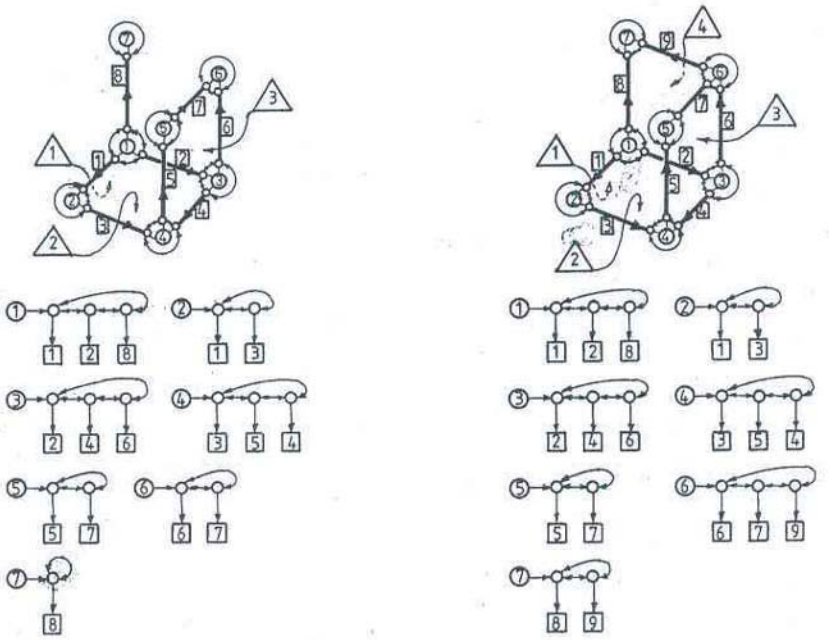


Figure 3.29: Building a cube with vertex-edge links (4)

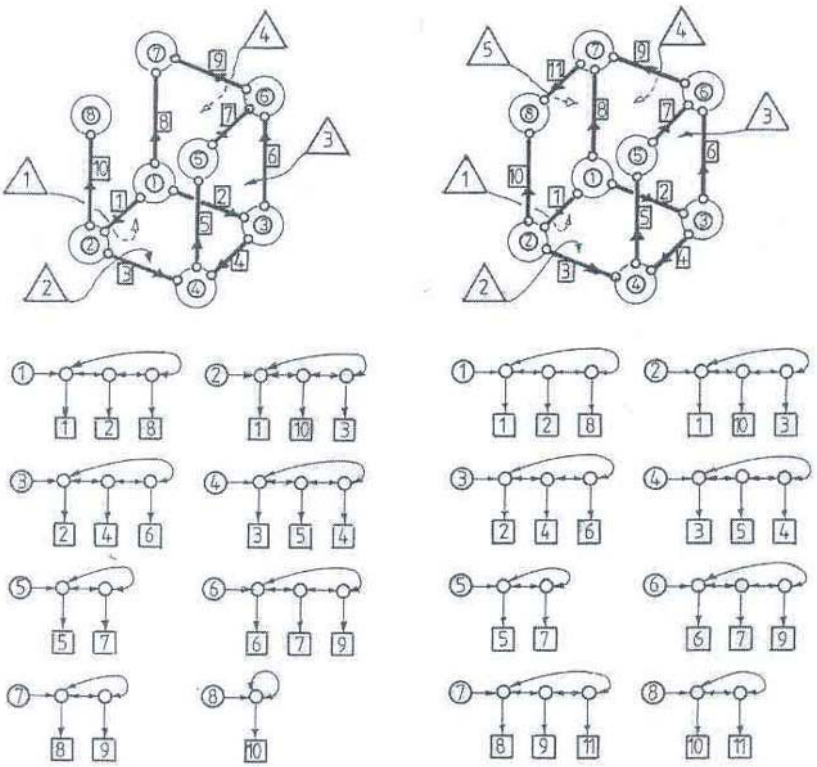


Figure 3.30: Building a cube with vertex-edge links (5)

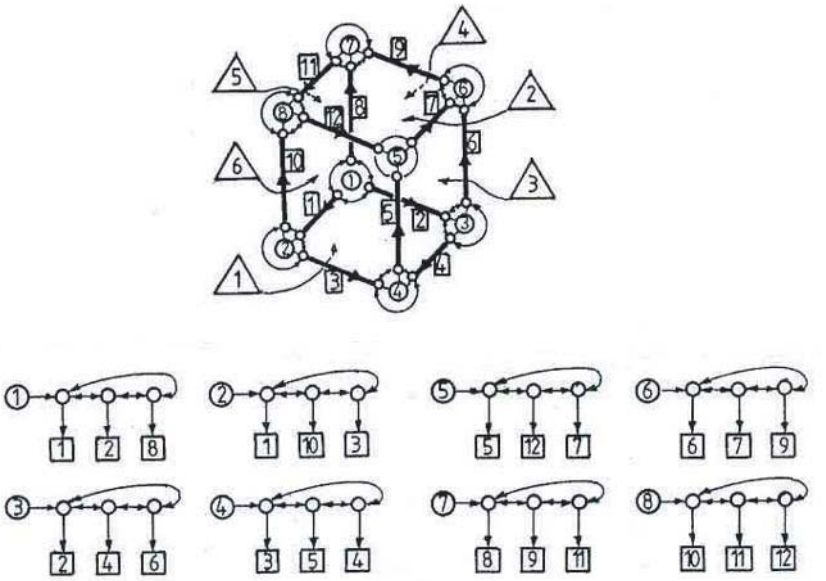


Figure 3.31: Building a cube with vertex-edge links (6)

"the face with name: BASEFACE". The traversal procedures can be useful for setting up an operation (such as marking entities) and in the finishing phase.

The traversal procedures can be divided into four groups based on the datastructure described in section 3.1.2. These are listed below.

Owner to single-level structure following:

- Vertices in body
- Edges in body
- Loops in face
- Faces in facegroup containing only faces
- Instances in a group of objects
- Notes in entity

Shared entity traversal:

- Surfaces in object

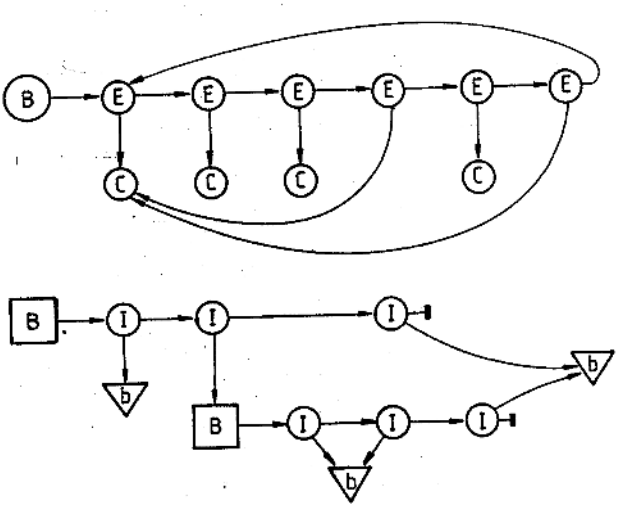


Figure 3.32: Topological structures

- Curves in object
- Single objects in a group of objects

Tree-structure traversal:

- Facegroups under object or facegroup
- Faces in body
- Faces in facegroup where the facegroup contains facegroups
- Instances under body

Topological set traversal:

- Edges in vertex
- Edges in loop
- Faces, edges and vertices in a shell

The shared entity traversals are for coping with structures where an entity might be found more than once, such as the geometry of a body. For example, the curves in a body are found by traversing all edges in the body, but some curves may be shared between several edges if, say, a single curve is broken into several pieces. A possible structure is illustrated in figure 3.32, top. This can

be a problem when transforming a body with a rotation, say, because it would be wrong to transform a curve once for each edge, which refers to it. It is also unnecessary to write the same curve description several times to disc when saving an object, because this will inevitably cause extra work when reading the curve back. It is also possible that there are shared instances in assemblies, as illustrated in figure 3.32, bottom, so a shared-instance traversal is also needed for accessing these. The figure shows a structure with six instances (shown as circles with the letter I), two assemblies (squares with the letter B), and three basic objects (triangles with the letter b). The top-level assembly consists of three instances, the first referring to a basic, unshared object; the second to a sub-assembly; and the third to a basic object also referred to from the sub-assembly. The sub-assembly contains three instances, two referring to the same object, and the third to the same object referred to from the instance at the top level.

The tree-traversal algorithms are more complicated in that they can perform 'post-order' or 'pre-order' traversals depending on the operation to be performed. For example, when printing the facegroup structure of an object, it is useful to print in 'pre-order'; that is, the parent is printed first followed by any child facegroups. However, when deleting a complete facegroup structure, it is easiest to delete the facegroups in 'post-order', that is, to delete the facegroup tree below a facegroup before the owner facegroup to preserve the backward references up the tree. Another complication with tree-structure traversal is provided by the instance structure traversals because each instance has an associated transformation. For these, it is necessary to provide an associated list of transformations, or to supply the transformation information in some way.

The last group of traversal functions to be described here uses topological relationships to traverse structures. The first two are more-or-less straightforward, but the third is more interesting, necessary to separate the shells in bodies.

The basic algorithms for traversing all edges in a loop and all edges meeting at a vertex start at a predefined starting edge, referred to by the vertex or loop, and use the winged-edge pointers, or their equivalents, to find the other edges, using the vertex or loop for orientation. There are some problems in that the datastructure allows edges to have the same loop on both sides (wire edges), or the same vertex at both ends. These two conditions are, in fact, duals of each other, and they can be handled using extra entities (a vertex when traversing edges in a loop, and a loop when traversing edges at a vertex). If links are used round loops, then the loop traversal becomes almost a simple list traversal, except that wire edges have to be dealt with. Similarly, if edges are connected to vertices using links, then the traversal of the edges meeting at the vertex becomes simpler.

The final topological traversal is to traverse all connected faces, edges, and vertices. This is a more complicated traversal than the others because it uses the adjacency relationships between elements in the datastructure for

the traversal rather than subsidiary lists. The traversal, though, is crucial for several operations, for example, Boolean operations, and even for low-level operations such as the Euler operation to make a face and kill a hole-loop or its inverse (see Appendix F, sections F.9 and F.13). It is used for identifying separate shells using topological connectivity.

Examples of algorithms for performing these traversals, based on the datastructure defined in Appendix A, are given in Appendix B, section B.1.

3.3.2 Single entity traversals

The single entity traversals take two elements and find a third according to the orientation of the given two (usually). Among other uses they are used by local operators for traversing the datastructure while modifying it. They are specified using the relative arrangements of elements rather than using the direct connections.

Edges are linked in a specific order around both loops and vertices so it is possible to use an edge and a vertex or an edge and a loop to find another entity. A loop and a vertex can also be used to find another edge. Edges also form a 'bridge' between both vertices and loops, so an edge and an adjacent vertex or loop can be used to find the other adjacent entity. The list of these traversal routines, identified by their BUILD mnemonics for convenience, is as follows:

Orientation using an edge as a 'bridge'

vopev Find the vertex opposite a given vertex along a given edge.

lopel Find the loop opposite a given loop across a given edge.

Using an edge and a loop to find an edge or vertex

ecwel Find the edge clockwise around a given loop from a given edge.

eccel Find the edge counter-clockwise around a given loop from a given edge.

vcwel Find the vertex clockwise around a given loop from a given edge.

vccel Find the vertex counter-clockwise around a given loop from a given edge.

Using an edge and a vertex to find an edge or a loop

ecwev Find the edge clockwise around a given vertex from a given edge.

eccev Find the edge counter-clockwise around a given vertex from a given edge.

lcwev Find the loop such that the given edge is clockwise around that loop from the given vertex.

lccev Find the loop such that the given edge is counter-clockwise around that loop from the given vertex.

Using a loop and a vertex to find an edge

ecwlv Find the edge clockwise from a given vertex around a given loop.

ecclv Find the edge counter-clockwise from a given vertex around a given loop.

Miscellaneous

eclev Find the edge clockwise or counter-clockwise from a given edge around a loop in the direction of a given vertex.

vve Find the edge connecting two given vertices (if any).

Obviously, if supplied entities are not adjacent to each other (except for the utility to find the edge connecting two vertices), an error is generated as no third entity can reasonably be delivered. Edges whose left and right loop pointers refer to the same loop (wire edges) are unoriented with respect to loops, and cause problems. Similarly, edges that start and end at the same vertex are unoriented with respect to vertices and should generate errors.

Algorithms for these single entity traversals are given in Appendix B, section B.3.

3.3.3 Topological utilities

Apart from the single entity traversals, described in the previous section, there are several topological utilities for manipulating or interrogating the nature of entities in the datastructure. They, too, serve to envelop the datastructure with a functional 'shell' to avoid accessing the datastructure directly. They also make it unnecessary to include direct checks for conditions that are controlled by convention and that can differ in different modellers. An example of this latter type is the way hole loops are recorded. An enquiry routine such as number 8, below, removes the need for directly including a test for a particular convention into several places in the code. See also Appendix C.

1. **Creating and deleting entities**
   Exactly how entities are chained together in the datastructure is not fixed because of the variable nature of the connections, but it is useful to have extra lists besides the sufficient connections. When creating entities, it is useful to add them to a structure so that they are not left hanging. For example, new edges in an object can be created and immediately added to the 'all edges in body' chain. The topological connections are made elsewhere, by the Euler operators (described in section 3.3) or by the modelling operators. Another related topic concerns how dynamic allocation is handled. If a modeller includes a memory handling mechanism using 'free-lists', or lists of deleted entities, then these have to be handled by such creation and deletion utilities so that old entities are reused.

2. **Moving entities to new owners**
   These perform the same sort of 'housekeeping' operation as the above, but for moving them between owners.

3. **Separate a closed sequence of edges into a new loop**
   This is used for partitioning portions of loops after a new edge has been added. It is necessary to traverse one portion, adjusting the loop pointers of the connected edges to point to the new loop. Traverses the new loop portion in the same way as the 'for all edges in loop' traversal, above.

4. **Separate a sequence of edges belonging to a new vertex**
   This is used for separating edges around a vertex when splitting the vertex, analogous to the way edges are separated into new loops, above. Uses the same sort of traversal as the 'for all edges in vertex' traversal, above.

5. **Separate a set of adjacent faces, edges, and vertices into a new shell**
   This is used to separate portions of objects once they have been topologically separated. This is needed to pull apart the lists of entities in an object. It uses the 'for all edges in shell' traversal, above, to access the connected entities, and the entity moving routines.

6. **Check whether an edge is a wire edge**
   This is for checking edges in Eulerian objects. The condition for an edge to be a wire edge is that it refers to the same loop as both the left and right loops. Edge 5 in figure 3.13 is a wire edge. It is important to know whether edges are wire-edges, for traversing edge sequences, for the Euler operators, and for the modelling algorithms.

7. **Check whether an edge or vertex is a spur**
   A spur vertex is a vertex with only one edge attached. A spur edge is one where one or both end vertices are spur vertices, i.e., that the edge is the only edge at its start or its end vertex. Edge 10 in figure 3.13 is a spur edge, and vertex 9 is a spur vertex.

8. **Check whether a loop is a hole loop**
   Checks whether the loop is a hole loop in the datastructure. It is not a geometric check, merely a check of the status of the loop in the datastructure.

9. **Checks for special object types**
   Two interesting special object types are minimum Eulerian objects and objects with no faces, edges, or vertices. A minimum

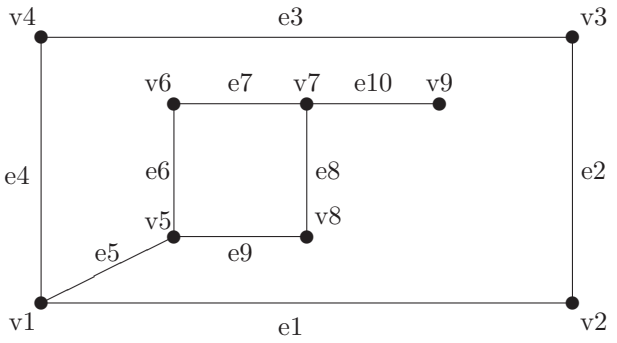


Figure 3.33: Illustration of special topological elements (wires, spurs)

Eulerian object is one with a single vertex, a single face, and no edges. An object with no faces, edges, or vertices can be a degenerate result that should be deleted.

10. **Copy an object**
    This is used, for example, when reflecting an object and in Booleans for copying multiply instanced objects, as well as being a useful as a user tool.

11. **Modify object with transformation**
    This uses the geometry modification utility described in the next section but is responsible for coordinating the changes for the whole object. As described in section 3.3.1, curve, surface (and possibly coordinate) geometry can be shared within an object. This utility is responsible for modifying each geometric entity once only, and for handling the spatial approximation measures, bubbles, face boxes, and so on.

3.3.4 Geometric interrogations

The general philosophy in BUILD, and later in the GPM volume module, has been to treat geometry in terms of the general classes: point, curve, and surface, with points being handled as vector positions in BUILD. The geometry is handled by several tools that reduce the number of places where it is necessary to know the exact geometry formats. This reduces the amount of work needed to introduce new geometry or to change existing geometry. The main tools are listed below. See also Appendix D.

1. **Geometric intersection package**

To intersect two complete geometric entities of the general classes: point, curve or surface. This is a much used facility, necessary for many basic modelling operations. The six possible intersection combinations are as follows:

Point-point int.	Effectively, "check if two points are coincident".
Point-curve int.	Check if a point lies on a curve. This could be combined with tool 5, see below, to return the parameter value (or values) of the point if it lies on the curve. This is a practical question, so the tools are differentiated here.
Point-surface int.	Check if a point lies on a surface. As above, this could return the result in the form of the parameter values of the point if it lies on the surface, but the tools are differentiated here.
Curve-curve int.	Check (and return intersection results) if two curves cut each other at points, are coincident, or are partially coincident.
Curve-surface int.	Check (and return intersection results) if a curve lies on a surface, cuts through it, or is partially coincident with it.
Surf.-surf. int.	Check (and return intersection results) if two surfaces cut each other along a curve, touch at a point, are coincident, or partially coincident.

The interface sorts out which intersections to use from the format of the geometry. It looks at the geometry types and decides whether, say, a cylinder–cylinder intersection is required, or a circle–plane intersection. A delimiting measure (sphere or box, as mentioned in section 3.1.2) is needed when approximations to infinite curves need to be calculated.

In what form the results are returned is not fixed. They could be given as a set of geometry, points, curves, or surfaces, but extra information can be provided as well. A convenient result set, based on that in BUILD, is as follows:

- COINCIDENCE (same orientation)
- COINCIDENCE (reverse orientation)
- POINT
- CURVE
- SELF-INTERSECTING CURVE

- SURFACE

or a set of these. Partial coincidence can be handled by returning a new geometric entity (curve or surface) representing the coincident subpart of the curve or surface.

2. **Calculate a curve tangent**
   This calculates the tangent direction at a point or parameter value on a curve. It is useful if the tangent vector is explicitly made to be in the direction of the curve at the calculation point as this aids determination of operations such as when intersecting a face with a curve.

3. **Calculate a surface normal**
   The surface equivalent to the above utility.

4. **Calculate the parameter value of point on curve**
   Parameter information may be supplied directly, if, say, a parametric geometric form was given to the intersection package, or might have to be calculated specifically if a non-parametrised geometric form is used. Parameter values are used, say, for sorting intersection points along curves, so the function is necessary for this if they are not supplied from intersection utilities.

5. **Calculate the parameter values of a point on a surface**
   For calculating parameter values on parametric surfaces.

6. **Calculate coordinates from a parameter value on a curve**
   The inverse function to number 4, above. The function is useful, among other things, for calculating intermediate points on a curve between known points.

7. **Calculate coordinates from parameter values on a surface**
   The inverse function to number 5, above.

8. **Create a surface by sweeping an edge in a straight line, circular arc, or along a curve**
   This is the main geometric creation utility for sweeping. It uses the curve of the edge and the sweep direction to create the surface shape, and the orientation of the edge in the loop to determine the orientation.

9. **Modify geometry**
   As well as being used to change the shape or position of complete objects, without changing the topology, geometry modification is also used for such operations as sweeping or swinging and bending. This may even change the form of some geometry, say, changing a circle to an ellipse, with non-uniform scaling.

10. **Reparametrise a curve**
    Reparametrisation is necessary to ensure that consistency is maintained if the parametrisation of a curve is linked to an edge, and the start and end positions of the edge are changed.

11. **Produce an offset curve from a given curve**
    This is necessary for producing offset geometry when giving a sheet object thickness, for example; see section 4.7.

12. **Produce an offset surface from a given surface**
    As above. This is used when giving sheet objects thickness.

3.3.5 General utilities

The final utilities described in this section are general background utilities not specially topological or geometrical in nature, or a mixture of both. See also Appendix E.

1. **Marker bits**
   The marker bit mechanism is very heavily used in the algorithms described in chapter 4 for recording special conditions.

2. **Matrix and vector packages**
   As both matrices and vectors are used for geometry representation, it is important to have general packages for handling them.

3. **Determine whether a point is inside or outside an object**
   This utility is used, for example, for determining how to handle objects in Boolean operations where the boundaries do not intersect.

4. **Determine whether a point lies in a face**
   This is important for testing to which faces internal loops belong, for example. The best method is to use a ray test, but care is needed to cope with special cases if no fake edges (see section 3.1.4) are included in the model.

5. **Determine whether a loop lies in a face**
   This is important for correct partitioning of hole-loops when a face has been divided. In its simplest form, it may only require that a point on the loop be tested to see if it lies in a face. If more complicated conditions can occur, then it may be necessary to return more than a Boolean, indicating, perhaps, whether a loop intersects or coincides with the boundary of a face, as well as testing whether the loop lies completely inside or completely outside a face.

6. **Intersect a face with a curve and sort the results**
   Basically this involves intersecting the curve with all of the edges bounding a face, i.e., for the exterior loop and all hole-loops, and sorting out the results. Once all of the intersection points have been found, they are sorted, basically by parameter value but with extra checks for coincidence. In the algorithms described in chapter 6, this is used for Boolean operations and sectioning, for imprinting, and for bending. The same basic process is also useful for drawing silhouette curves for faces.

7. **Determine whether a curve lies between two edges**
   This is used to determine geometrically the order of edges around a vertex. It is used by the imprinting operation when there are more than two edges at a vertex adjacent to the face being imprinted. A similar test is used in the Booleans. It is also useful for user-friendly Euler operations to avoid requiring orientation edges from a user to determine where a new edge should lie.

8. **Check if an edge is convex, concave, smooth, or sharp at a point**
   This check is partly topological, because it refers to edges, and partly geometrical because it involves the surfaces meeting at the edge. The test uses the face normals on either side of the edge to perform the check. As edges can change character along their length, it is necessary to specify a point on the edge for the test.

9. **Calculate the orientation and approximate area of a loop**
   This is very useful for checking the orientation of faces and individual loops. It can be used for checking orientations of loops in low-level Euler operations as well as for higher level operations. This is only suitable for faces extending through less than 180 degrees.

3.4 Assembly structures

Assemblies are organisational structures for uniting single objects and groups of objects into groups. The classic structure is shown in figure 2.4 and figure 3.34 shows a slightly updated version.

Assembly structures should be treated carefully as there is a very common problem that they look correct, but that the structure is misleading. In a correct structure, each unique object appears only once and that object is referenced, or instanced, as many times as needed. Each object has an associated use count that records how many times it is referenced. This information can then be used directly to construct the bill of materials (BOM). It is much harder to compare geometric shapes to see if they are identical and a waste of

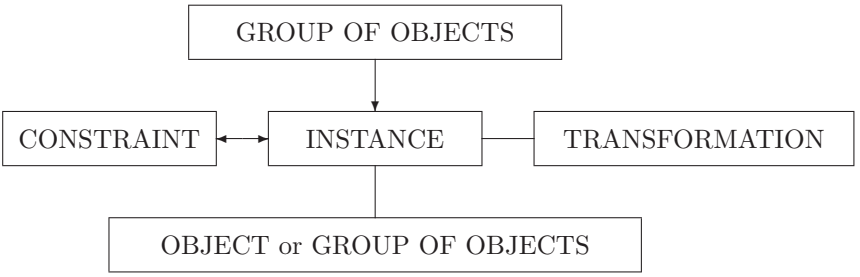


Figure 3.34: General assembly datastructure

time if the assembly structure is correct. Other problems, such as modifying multiply-used objects, are also easier in a correct structure. How to handle instances for modifications is described later.

3.4.1 Determining the assembly structure

The exact structure is determined by the user. Some illustrations are given to illustrate this. In figure 3.35, all parts are different, and so the assembly structure is flat.

In the next example, figure 3.36, there is a group of objects that appears twice in the same configuration. This can therefore be defined as a sub-assembly, and this sub-assembly is referenced twice.

In the third example, figure 3.37, standard parts, here represented by simple nut-and-bolt models, are referenced several times.

The fourth example, figure 3.38, shows a special assembly structure created for kinematic reasons.

The lesson to be learned is that there are many possibilities for creating assembly structures. This means that it is impossible to predefine how to create an assembly structure, which must be left to the judgement of the user. One way of deciding a structure is to make it reflect the physical nature of the realised product. So, functional sub-assemblies of the final product should be sub-assemblies of the model datastructure. It may well be best to do this as part of the early phase of design using a layout tool such as that of Csabai [25] which defines the assembly structure before detail design commences. To simplify horribly, Csabai allowed the chief designer and design team to set up a product structure defined hierarchically in terms of design spaces (simple primitive shapes such as cues or cylinders). These could be

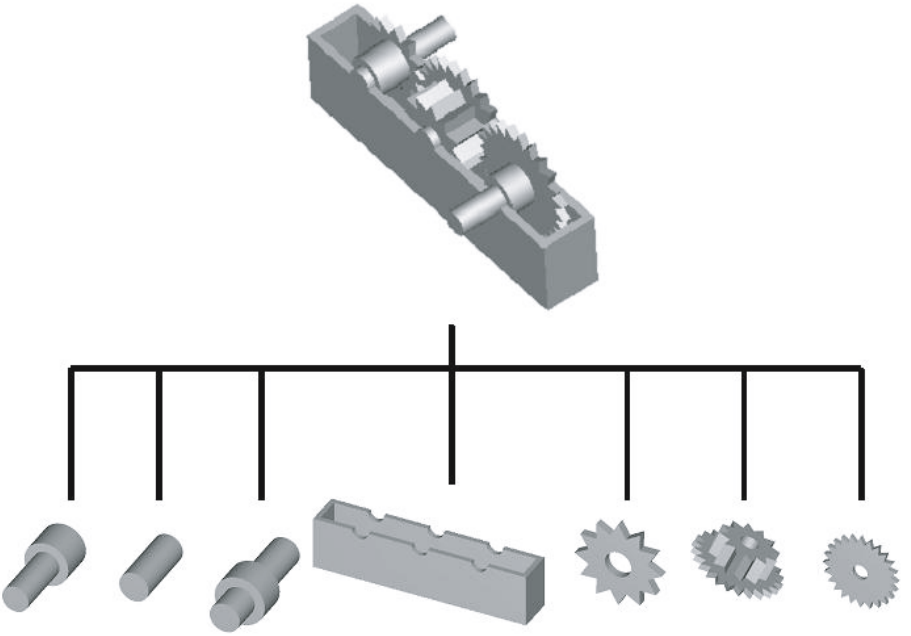


Figure 3.35: Single-level assembly

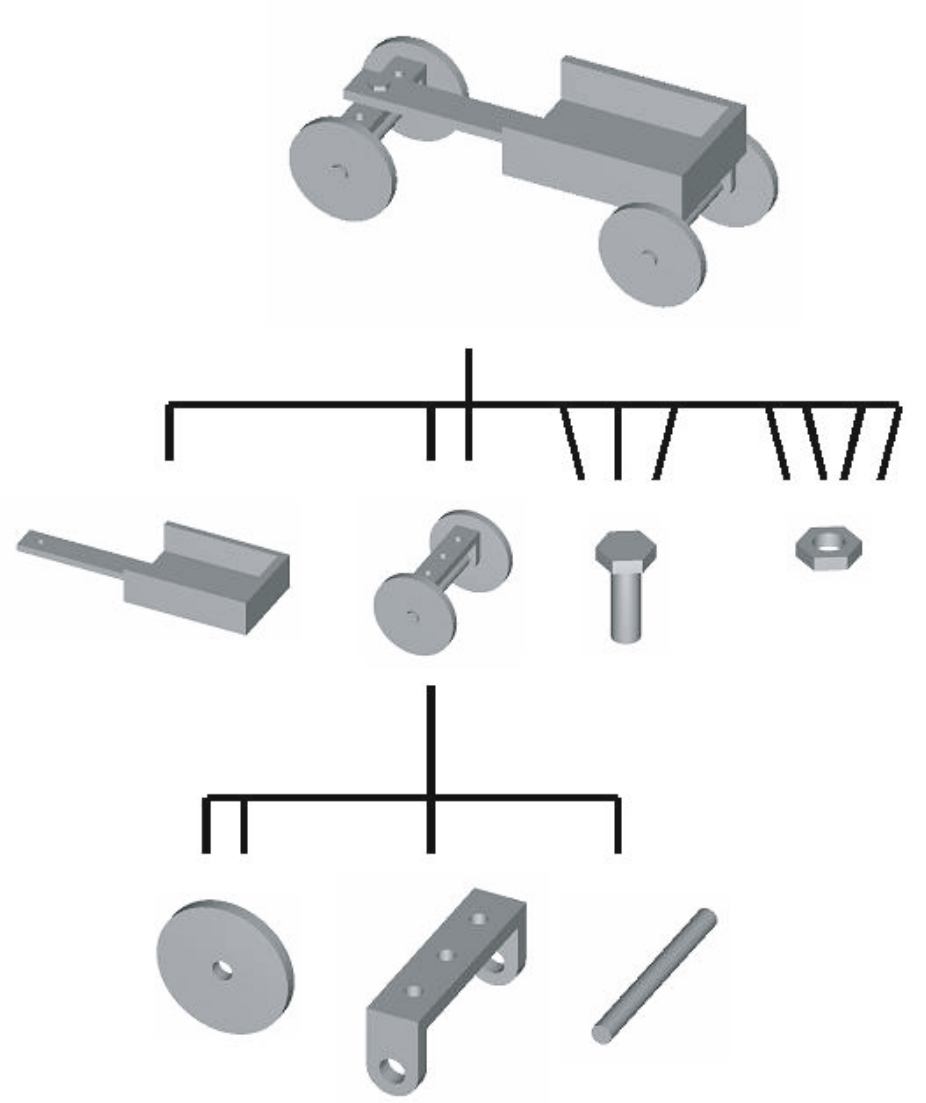


Figure 3.36: Multi-level assembly

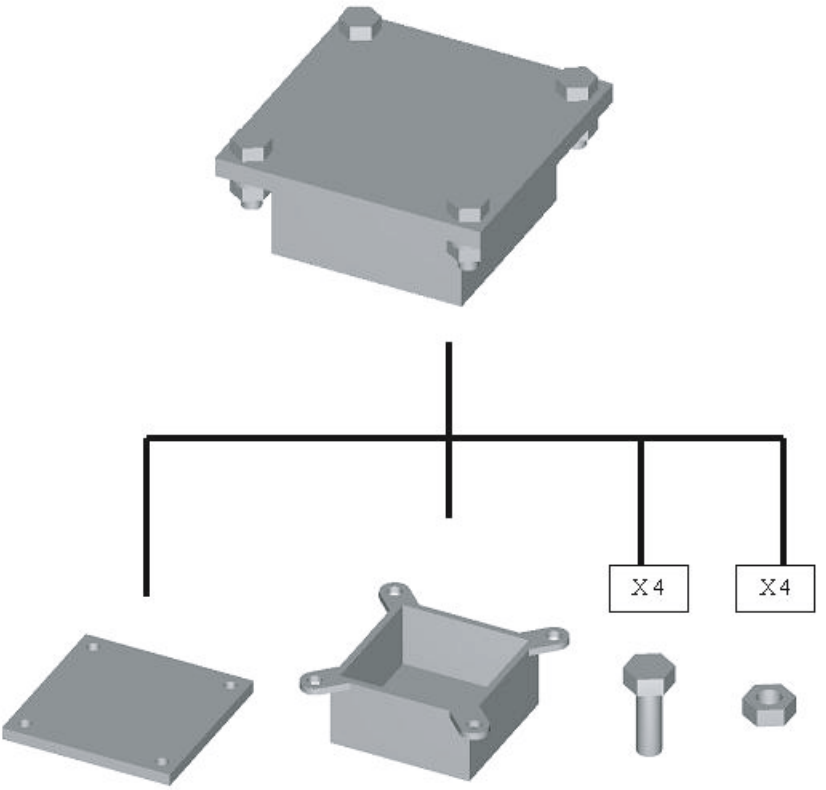


Figure 3.37: Assembly with standard parts

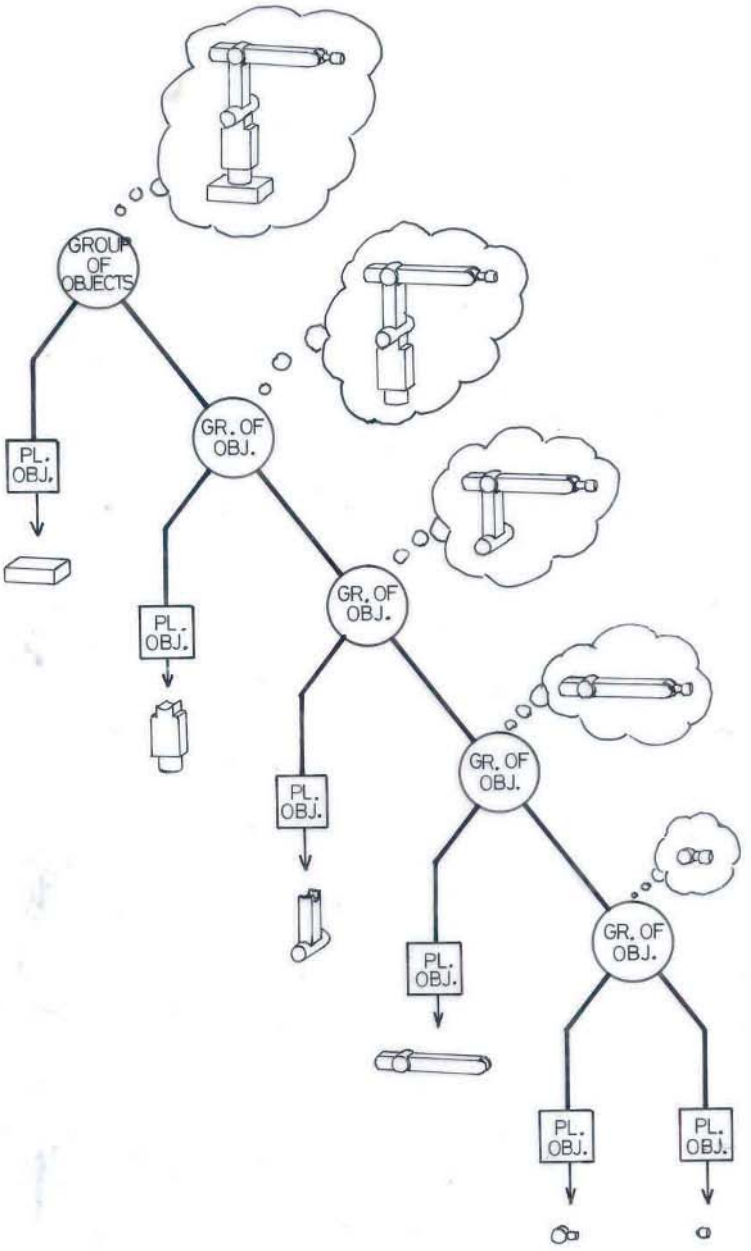


Figure 3.38: Kinematic/functional assembly

linked together to provide a kinematic model of the product before design of part shapes. This means that the assembly structure could, also, be defined as a set of assemblies and instances that is passed over to the CAD system as part of a predefined design environment. This, though, is outside the scope of what is presented here, which is more to do with how the assembly structures are represented internally.

3.4.2 Handling instances

As stated, each unique object should appear only once in the assembly. It is common that, during assembly, design errors may appear, such as collisions and shape mismatches. Modifications are then necessary, but it is possible to modify at two different levels: instance modification or object modification.

With instance modification, only one particular instance is modified and all others remain. If the instanced object is only used once, then this is the same as modifying the object. If it is used more than once, then it is necessary to copy the object instanced (thus reducing the use count by one), to modify the copy, and to replace the reference in the instance to the modified copy.

With object instancing, the object is modified directly and, hence, all instances of the object will change. This may also be useful and, hence, cannot be excluded.

Which one of these two cases is needed depends on what the user wants, and so it is necessary to be able to distinguish this in the user interface.

The next problem, though, is harder to solve because it goes outside the modelling methods. If a modified object is saved, then it is necessary to know whether the object is used in other product assemblies. If the object is simply saved back into the same file, then this implies that all products using that part will be changed, possibly making them invalid. Some parts, such as standard nuts and bolts, for example, should never be modified. Parts such as these, though, should be labelled or named anyway so that a BOM can reference them correctly. Other parts, such as company parts that already form part of another product, may be blocked in the same way. This, however, is part of the company data management methods. Another possibility is to add a second use count to single objects and groups of objects to note the external use of the object.

3.4.3 Constraints or connections

Constraints, or connections, are the current method of creating relations between instances. They allow kinematic simulations to be performed and maintain the relationships as objects are repositioned during assembly. Constraints have to be applied between instances. Note that in chapter 8, these types of entity are termed “connections” to differentiate them from constraints between single model elements. The term “constraint” is widely used in practice, so it is used here too.

Constraints relate simple geometric objects, lines, planes, and points. They do not relate complex geometries, so it is possible to constrain, for example, a cylindrical protrusion within a cylindrical hole that is too small, or even too large. The constraints concern the cylindrical centre lines, not the radii.

The combinations available are as follows:

1. point–point
2. point–line
3. point–plane
4. line–line
5. line–plane
6. plane–plane

The constraint contains references to two instances together with the geometric definitions. The definitions are transformed with the instance transformations and the relationships checked. If the relationship is not satisfied, then one of the instances is moved so that the relationship is satisfied. In order to do this, it is necessary that at least one object in the assembly is fixed, or "grounded"; otherwise, there is a risk of looping.

Using constraints is not always easy for users. Objects have six degrees of freedom, and constraints remove these in different ways according to their character. The problem for users is to understand how to reduce these degrees of freedom without creating conflicts. If degrees of freedom are removed more than once, then the assembly is over-constrained. If not enough degrees of freedom are removed, then the assembly is under-constrained. Constraints corresponding to practical mechanical solutions do not always constrain the mathematical assembly model in the same way.

Constraints should also have notes of which degrees of freedom they remove. If conflicts arise, then the superfluous degrees of freedom need to be ignored. Remaining degrees of freedom are then also evident.

3.4.4 Mechanism libraries

I know of no mechanism library as a support for assembly construction in the way intended here, but because it is technically feasible, it is mentioned here as a possible idea for development.

Mechanism libraries are a way of facilitating constraint definitions by predefining mechanisms. These mechanisms would consist of a set of empty instances (with simple graphic attributes) with predefined constraints. The user would then indicate the model elements corresponding to the constrained

elements (points, lines, planes), and the empty instances would be replaced by the real instances in the assembly.

At the moment it is just an idea.

Chapter 4

Euler operators

In mathematical terms the standard boundary representation (B-rep) datastructure is a graph with certain properties. In the datastructure described so far, volume objects, sheet objects, and combinations of these are called Eulerian objects. The name comes from graph theory because the elements in the datastructure form a graph in mathematical terms. Recent developments in non-manifold modelling mean that the nature of the graphs representing a model may be non-Eulerian. Non-manifold datastructures will be discussed in chapter 5. This chapter describes the basic Euler operators for the simpler datastructure. Extensions for creating basic manipulation operators for non-manifold datastructures are fairly straightforward. The importance of Euler operators lies in their use for low-level manipulation of the datastructure. They preserve the topological integrity of the object, making minimal changes only.

For Eulerian objects, the numbers of elements in the datastructure for a valid object or objects are related by a series of rules, described by Braid et al.[13] as follows:

1. $\underline{v},\underline{e},\underline{f},\underline{h}, \underline{g},\underline{b} >= 0$ *[This is the condition that a valid object cannot have a negative number of elements.]*

2. if $\underline{v} = \underline{e} = \underline{f} = \underline{h} = 0$, then $\underline{g} = \underline{b} = 0$ *[This condition means that an object with genus 1, say, but with no other elements is disallowed.]*

3. if $\underline{b} > 0$ then a) $\underline{v} >= \underline{b}$ and b) $\underline{f} >= \underline{b}$ *[A valid object must have one or more vertices, and one or more faces.]*

4. $\underline{v}$ - $\underline{e}$ + $\underline{f}$ - $\underline{h} = 2(\underline{b}$ - $\underline{g})$ *[The Euler–Poincaré formula.]*

where $\underline{v}$ is the number of vertices, $\underline{e}$ is the number of edges, $\underline{f}$ is the number of faces, $\underline{h}$ is the number of inner-, or hole-loops, $\underline{g}$ is the genus, and $\underline{b}$ is the multiplicity or number of shells

The Euler–Poincaré formula defines a five-dimensional network in the six-dimensional space defined by the six topological parameters. The nodes of this network, at positive integer values of the parameters, represent the valid Eulerian objects. The operators to transform the Euler object corresponding to one node into another object at any adjacent node are termed Euler operators. These were described by Baumgart [4], by Braid et al. [13], by Eastman and Weiler [30], and by Mäntylä [81], [82], and an extension to handle non-manifold models is described by Luo [79]. The description given by Braid et al. is most appropriate here, so what follows is based on that work.

The 99 possible Euler operators that change the number of any element by at most one (i.e., perform transitions between adjacent nodes in Euler space) are listed in Appendix F, together with a reduced set created by removing Euler operators that are obvious combinations of others, together with implementation details of a useful set of these. Any complicated change can be described in terms of combinations of these. As the 'null' point, where there are no topological elements, is part of the network, it also follows that any object can be built using a sequence of these.

The numbers of vertices, edges, faces, and hole-loops can be easily determined from the datastructure. The multiplicity can be counted if object shells are recorded explicitly, but the genus is more difficult to determine. Robin Hillyard developed a package to calculate the Betti numbers (see Giblin [43] and Braid et al. [13]) from adjacencies in a model. This allows the genus to be determined explicitly, rather than implicitly, to balance the Euler equation of an object. If the datastructure does not have a way of representing separate shells that form cavities within a body, then their existence has to be determined in some way. However, if the shells are recorded explicitly, then low-level operations that potentially split off parts of an object, such as the operation to make a face and kill a hole-loop, have to make sure that a new shell has not been created.

From a practical point of view, Euler operators form a convenient building block from which to construct complex modelling operations. They are also interesting in that their use maintains consistency of the topology of a model. However, not all of the Euler operators need to be implemented. Indeed, the effect of some of them is obscure. Section 4.1 describes spanning sets and decompositions of objects and modification operations into sequences of Euler operators. Appendix F describes implementation of the operators, and section 4.2 contains a cautionary note about their use.

4.1 Spanning sets and decompositions

As described by Braid et al. [13], it is possible to choose a set of five Euler operators that form a 'spanning set' combinations of which can be used to create or modify the topology of Eulerian objects. To choose a spanning set, it is necessary to find five independent vectors. One possible set is as follows:

(1, 1, 0, 0, 0, 0) - MEV, Make an Edge and a Vertex
(0, 1, 1, 0, 0, 0) - MEF, Make a Face and an Edge
(1, 0, 1, 0, 0, 1) - MBFV, Make a Body (new shell), Face and Vertex
(0, 0, 0, 0, 1, 1) - MGB, Increase the Genus and Make a Body (shell)
(0, 1, 0,-1, 0, 0) - MEKH, Make and Edge and Kill a Hole

together with their inverses.

Writing the Euler operators in matrix form, together with the final row that corresponds to the coefficients of the Euler–Poincaré formula, gives the matrix A:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 1 & -1 & 1 & -1 & 2 & -2 \end{bmatrix}$$

Euler objects, and transitions to change one Eulerian object into another can be described as combinations of these primitives; thus:

$$q = pA$$

where q is a vector representing the numbers of the elements in the Euler object or the change in the numbers of elements, and p is the vector of the numbers of times each primitive is to be applied. Multiplying by the inverse matrix A^{-1} gives

$$qA^{-1} = pAA^{-1} = p$$

The inverse matrix to that representing the spanning set is:

$$1/12 \begin{bmatrix} 7 & -5 & 4 & -2 & -1 & 1 \\ 5 & 5 & -4 & 2 & 1 & -1 \\ -5 & 7 & 4 & -2 & -1 & 1 \\ 5 & 5 & -4 & 2 & -11 & -1 \\ 2 & 2 & -4 & 8 & -2 & 2 \\ -2 & -2 & 4 & 4 & 2 & -2 \end{bmatrix}$$

For a cube, say, with topological element vector:

$$(8,12, 6, 0, 0, 1)$$

the vector describing the number of times each primitive is to be applied is:

(7, 5, 1, 0, 0, 0)

A cube can thus be created with seven Make and Edge and Vertex (MEV) operations, five Make a Face and Edge (MFE) operations, and one Make Body, Face and Vertex (MBFV) operation. As the cube has no hole-loops and a genus of zero, the other operators are not needed. Also, if any operator is to be applied a negative number of times, $-n$, say, this is equivalent to applying its inverse n times.

The order in which the operations should be applied is not defined. Different strategies may be applied as guidelines, and a further rule to add to the set defined by Braid et al. [13] above, i.e., that, when building an object:

5. if any of v, e, f, h or g $>$ 0, then b $>$ 0.

This is the 'common-sense' rule that faces, edges, vertices and hole-loops cannot exist without a body.

This means that when building an object, the first operator applied must be the MBVF operator to make a vertex, a face, and a new shell. Figures 4.1, 4.2, 4.3, 4.4, and 4.5 show five different strategies for building a cube from the seven MEV, five MFE and one MBFV operations. In figure 4.1, the strategy is to try and build complete (i.e., for a cube four-sided) faces. In figure 4.2, the strategy is to create trihedral vertices. Figure 4.3 shows an extreme case where all MEV operators are applied first followed by the MFE operators. Figure 4.4 shows the other extreme case where all MFE operations have been applied followed by the MEV operations. Figure 4.5 shows the actual sequence used to create a cube using sweep operations. The problem of Euler operation application will be discussed further in section 4.2.

Another problem is that operators such as the MGB, as illustrated in figure 4.6, have a complex interpretation that are difficult to specify in a convenient form. Operators that manipulate single edges, vertices, or faces are easier to specify, implement, and use. In practice, it is possible that local operators, as described in chapter 6, may perform an Euler operation or some simple combination of Euler operators as a preliminary step. These are 'hidden' operators in the modeller because they are not explicitly available.

This diversity of function for the same basic Eulerian changes makes it difficult to specify exactly how a particular Eulerian change is to be applied in all cases. Appendix F describes implementations of various Euler operators to illustrate functional considerations. The implementations are intended to provide examples rather than a complete set. Also, for practical reasons, it is sometimes desirable to include extra Euler operators for performing basic changes rather than using a sequence of several operators from the spanning set. This is because the spanning set essentially defines a way of traversing the Euler network, and the extra operators represent 'short cuts'.

The set of operators described are as follows:

1. Make an edge and vertex (spur vertex and edge)

2. Make an edge and vertex (split an edge)

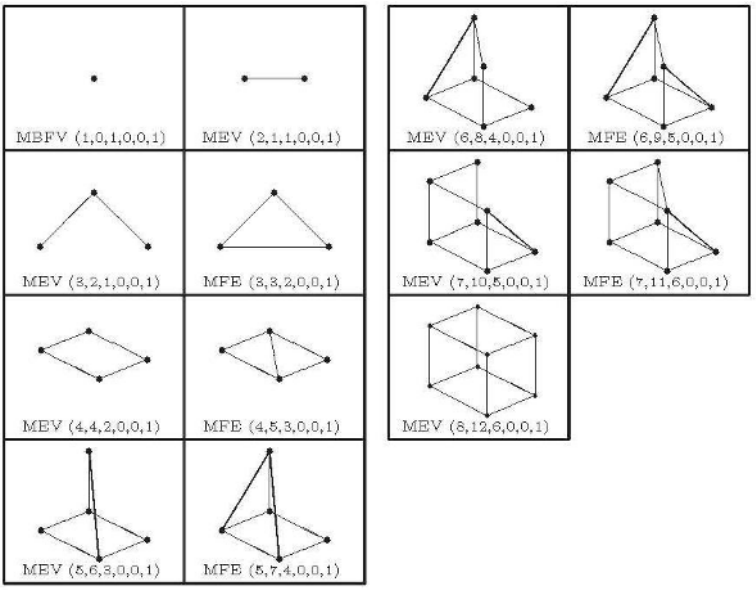


Figure 4.1: Creating a cube with Euler operators (1)

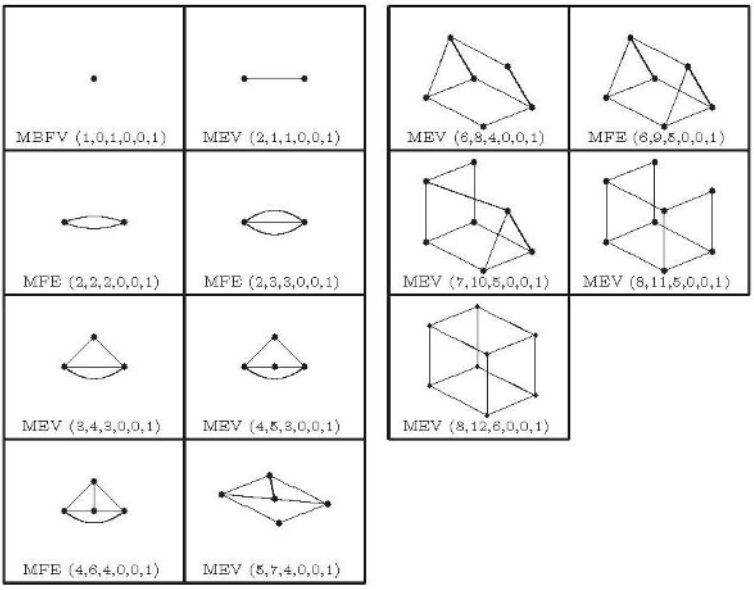


Figure 4.2: Creating a cube with Euler operators (2)

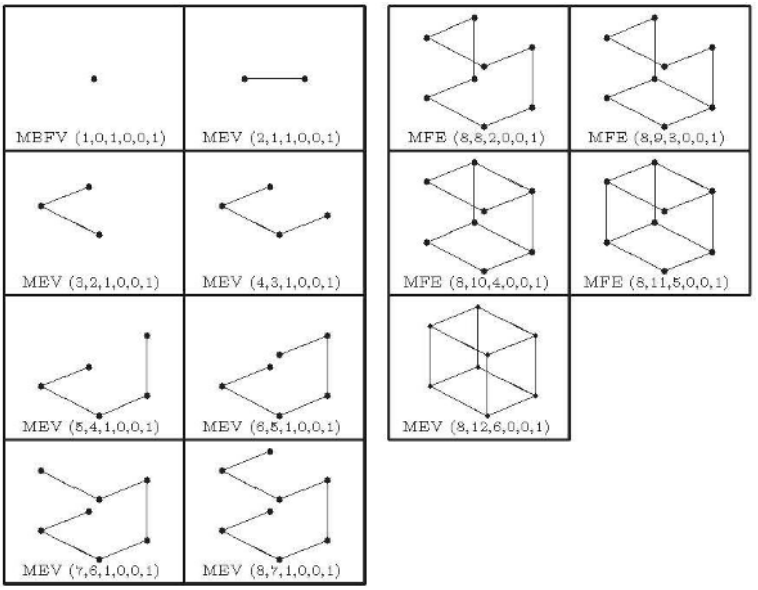


Figure 4.3: Creating a cube with Euler operators (3)

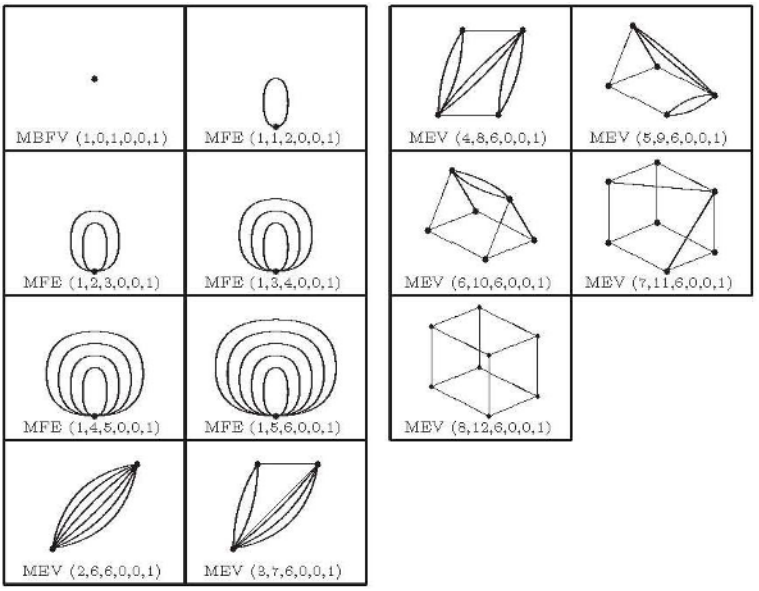


Figure 4.4: Creating a cube with Euler operators (4)

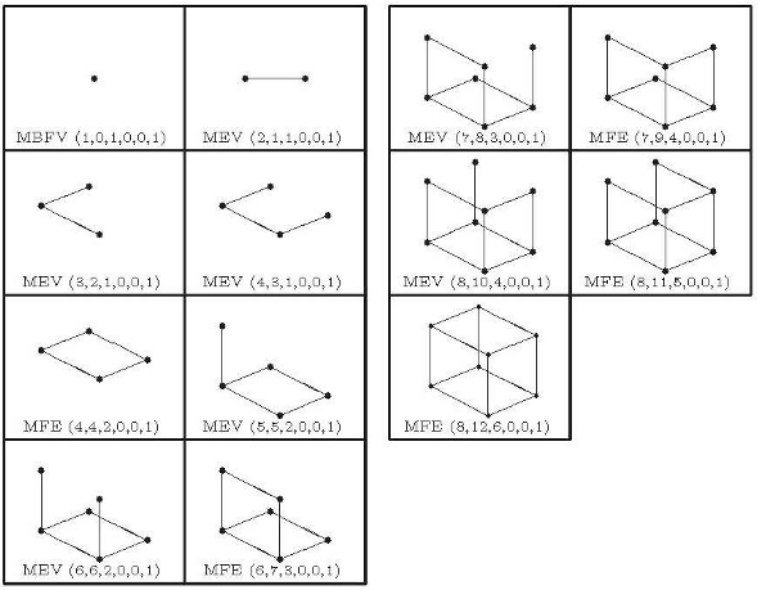


Figure 4.5: Creating a cube with Euler operators (5)

3. Make an edge and vertex (split a vertex)
4. Make an edge and a face
5. Make an edge and a face (slicing an edge)
6. Make a vertex, a face, and a new object
7. Make an edge, and kill a hole loop
8. Make an edge, and kill a face and body (shell)
9. Make a hole-loop, and kill a face
10. Make a vertex and a hole-loop
11. Kill an edge and vertex
12. Kill an edge
13. Kill a hole-loop, and make a face
14. Kill a vertex and hole-loop
15. Merge coincident vertices
16. The null operator

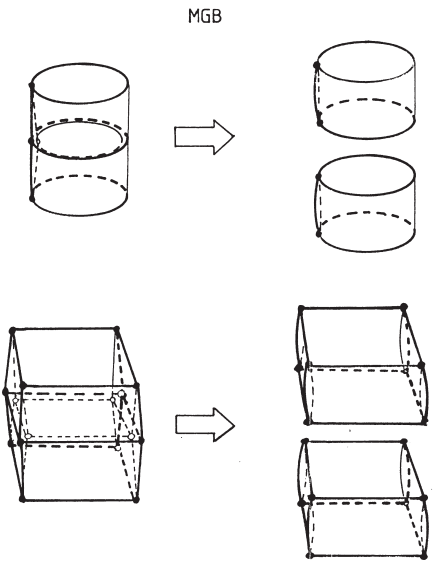


Figure 4.6: Illustration of the MGB Euler operator

4.2 Restrictions

Euler operators form a useful set of basic tools with which to build up more complex modelling operations. In the examples shown in figures 3.11 to 3.31, the Euler operators take the burden of handling the connection implementation away from the user. However, several points about them should be noted, as follows:

1. They are not generally suitable as user tools without extra geometric checks, and with limited application domain.
2. There are no hard and fast rules for when, where, and how to apply operators in modification sequences.
3. The set of potentially useful Euler operators is not limited to a spanning set.
4. There are three elements in the model: topology, geometry, and information. An important question is how these are to be integrated.

These points are examined below. In defence of Euler operators, it should be noted that their use is as basic tools from which to build up more complex modelling operators extends beyond the basic B-rep datastructure to the non-manifold structure described in the next section. A mathematical treatment for non-manifold Euler operators can be found in the dissertation by Luo [79]. Here it is more important to understand their modelling function as the extensions for non-manifold operators can be readily grasped once the basic operators are understood.

1. Euler operators as user tools

Except for very limited changes, it is very tedious to use Euler operators for modifying models. To sweep a four-sided face requires eight Euler operator applications: four MEV operations and four MFE operations. Requiring a user to perform these operations explicitly is more likely to hinder a design than to aid it. The role of Euler operators is mainly as a system development and application programmer's tool. This means that they should be reasonably efficient, and that there is probably information about how and where they should be applied readily available. User tools should incorporate geometric checks to determine this information to avoid forcing a user to have a detailed understanding of what they do, and to ensure model consistency as far as possible.

2. Application of Euler operators

As demonstrated in the Euler operator application sequences shown in figures 4.1 to 4.5, the order in which Euler operators is applied is not fixed. Deciding which one to apply and how is something of a 'black art' which is

difficult to describe. It is easier to implement higher level operations, using Euler operators and other basic tools, and make these available as modelling tools as well as the basic Euler operators. These higher level operators in effect insert sub-sequences of Euler operators into the complete sequence, but they provide a more comprehensible tool. The basic Euler operators can be used to set up the operation and to finish it off, say, whereas the higher level operators create the middle steps.

3. Which Euler operators to implement?

It is not desirable to implement only a spanning set of Euler operators, because there are many different useful ones according both to taste and to applicability as logical steps in a more complex operation. Several useful examples have been described in Appendix F, and some of these, like the kill-edge operator, cover several Euler operators, the actual case being determined by the connection state of the parameters (i.e., if an edge is a spur-edge, a wire-edge, or neither). The Euler operators are used in B-rep modelling systems as convenient steps from which to create complex modelling tools. Which ones to implement depends partly, therefore, on the useful steps with which to build the complex tool. It is also important to note that the axes of the six-dimensional Euler space represent the numbers of elements in the Euler–Poincaré formula, not their arrangement. Because of this, it is possible that a change to the number of elements in the model can be interpreted in several ways, as for example, the operators to add an edge and a vertex to the model (sections F.1, F.2, and F.3). This also means that topological operators that make no change to the numbers of topological entities in the model actually perform useful tasks.

4. Integration of topology, geometry, and information

Finally, it should be noted that the B-rep model comprises (currently) three main parts: topology, geometry, and information. Strictly, the Euler operators affect only the topology. However, it is, in general, necessary to perform the other modifications as well, somewhere, to preserve model consistency. It may be possible to ignore the geometric effects with some types of geometry, i.e., straight edges or geometry unrelated to the topology, but thought has to be given to the problem, if only to dismiss it. Operations can be implemented as a 'layered set', with the pure Euler operators at the lowest level, and then one or two more levels handling geometry and information. This would allow other modelling functions to 'choose' what degree of modification is required but is less efficient than combining the functions. This is part of the basic design of the modeller and should be handled consistently throughout.

Chapter 5

Special representations

[The material in this chapter is based on papers that first appeared in the CAD Journal, in 1990 and 1994 ([123] and [125]) and several figures from these papers are reproduced here by permission of Elsevier.]

One of the original conditions for boundary representation (B-rep) models (see Baumgart [4]) was the manifold condition, i.e., that, at any point on an object, a small enough sphere would cut the surface of the object in a figure homeomorphic to a disc. Although this condition is necessary if the object is to be thought of as a connected set of points, it is not absolutely necessary for performing modelling operations, although care must be taken to identify the special cases arising. Various types of special representation that do not conform to the manifold condition have been used in modelling systems. This chapter describes three basic types and explains their meaning in terms of modelling operations.

The three common types of representation used to handle special cases in B-rep solid modelling. These are as follows:

1. Partial models: These are partially complete. One side is completely defined, topologically and geometrically, and the other side is either topologically defined but geometrically meaningless, or is not even defined topologically. These were used, for example, in the BUILD system as representations for swept wire objects. See Braid [11].

2. Degenerate models: These are completely defined Eulerian models, but elements are allowed to coincide. These were used as special representations of idealisations of thin plate models in the GPM volume module; see Kjellberg [69] or Stroud [123].

3. Non-manifold (star) models: These are models allowing multiple faces meeting at an edge. As the degenerate models, above,

these are completely defined, but coincident edges and vertices are merged. See Weiler [144], Luo and Lukacs [78], or Luo [79].

These three types have different characteristics, and hence, they are useful in different ways.

1. **Partial models**

   With partial models, one 'side' is completely defined, geometrically and topologically, and the other side is either missing or consists of a single face with undefined geometry. This means that the model is smaller, in terms of computer memory requirements, than the other types. It is useful for representing surfaces and parts of models 'pulled off' for some reason, e.g., features. Because one side of the model is undefined, that side, and the edges bordering the undefined region have to be treated specially. The interior of the defined region can be treated as 'locally manifold'; that is, it can be treated as though there were solid material behind it. This makes it possible to perform local operations and even Boolean operations on such models. By implication, partial models are 'flat' shapes, or shapes composed of flat pieces. Although they may be derived from volumetric models there is, by definition, an undefined portion; hence, the face or faceset of the partial model is not closed.

2. **Degenerate models**

   These are useful for representing idealisations and temporary results. These are very similar to 'normal' volume models except that at some points, a small enough sphere cuts the object boundary in a figure homeomorphic to a pair of discs. Alternatively, if the object is two-dimensional or has two-dimensional parts, the sphere would cut the boundary in a figure homeomorphic to a folded disc. These were used in the GPM volume module as a representation for idealisations of thin plate objects (see [123]). Such objects were flattened shapes, but the technique can be extended to cover cases where parts of volumetric objects are allowed to coincide. As with the partial models, degenerate models are locally manifold, so local operators can function as they do with volumes, but care should be taken not to create 'mixed' models that have sheet and volumetric parts.

3. **Non-manifold (star) models**

   With non-manifold (star) models, coincident parts of models are merged so that more than two faces can meet at an edge, unlike partial and degenerate models where only two faces may meet at any edge (or one for the boundary edges of partial models). At

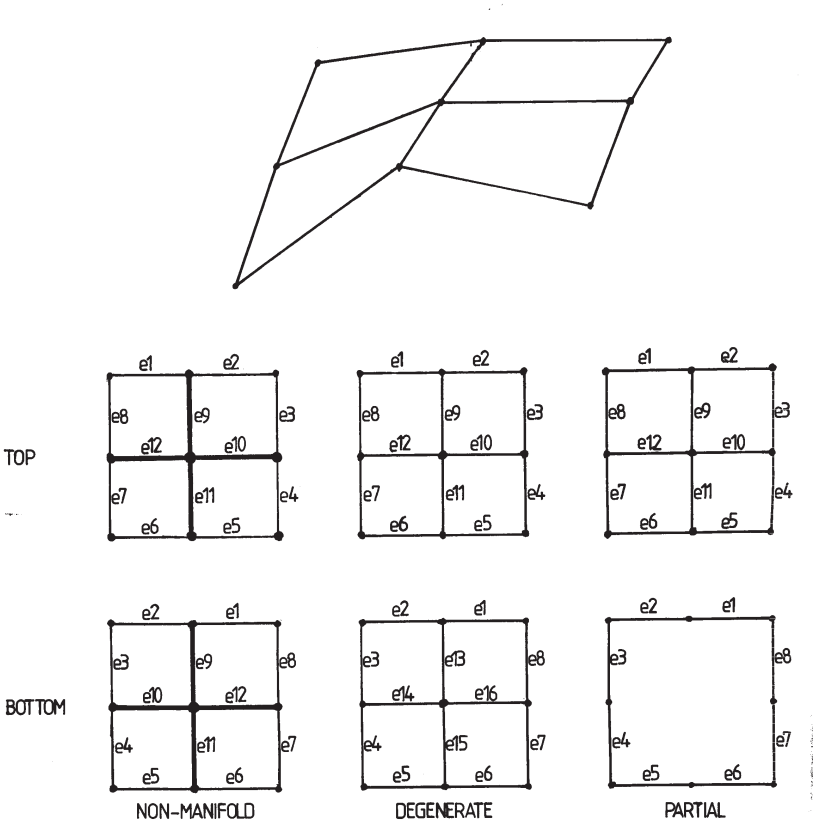


Figure 5.1: Illustration of star, degenerate, and partial models (from [125])

each edge, there should be an even number of faces that meet, for a correct non-manifold model. This was achieved using a so-called "radial edge" or "star" structure (developed by Weiler), where each edge has a set of loop-edge links arranged around it instead of a pair of edge-loop pointers. An advantage of radial edge type mechanisms is that matching parts of the model can be linked, retaining the information that they match; however, the role of non-manifold representations and modelling operations is not well defined. Luo [79] describes Boolean operations for non-manifold models.

Figure 5.1 illustrates how the three model types represent a flat plate object, shown at the top of the figure.

The two figures on the left at the bottom show the top and the bottom of a non-manifold representation. The edges in the middle, labelled 9, 10, 11,

and 12 and drawn thicker, are non-manifold edges appearing on both sides of the object. Each edge is adjacent to four faces, one pair on either side of the object. The central vertex is a 'non-manifold vertex'.

The two figures in the centre show a degenerate model version of the shape. In this type of model, each edge is adjacent to two faces only, so the internal edges on the top, 9, 10, 11, and 12, are matched by four others on the bottom, 13, 14, 15, and 16. The vertex at the centre is also duplicated.

For the partial model, shown on the right at the bottom, one side is undefined, so the four internal edges, 9, 10, 11, and 12, are adjacent to two faces only but have no matching edges on the other side. The bottom of the model is undefined, topologically or geometrically. If geometrically undefined, the edges e1 to e8 are adjacent to one face only.

Whereas partial and degenerate models can use the standard boundary representation datastructure, non-manifold structures need an extension to be able to handle multiple faces meeting at an edge. The usual way of doing this is with a loop-edge link (see Appendix A, sections A.1.2 and A.1.3). However, for a non-manifold structure, extra fields are needed from the definition given in section A.1.3. A simple non-manifold (N-M) edge link structure is as follows:

edge	EDGE ptr.	Pointer to the edge owning the link.
loop	LOOP ptr.	Pointer to the loop owning the link.
next	N-M LINK ptr.	Pointer to the next link around the loop.
prev	N-M LINK ptr.	Pointer to the previous link around the loop.
assoc	N-M LINK ptr.	Pointer to the next link around the edge.
left	LOGICAL	For noting whether the link is a left or right link of the edge.

This is illustrated in figure 5.2.

The links around the edge could also be doubly linked, but generally there will not be a large number, so the extra pointer field is probably wasteful. The links should also be ordered around the edge in some way, depending on the incidence angle of the faces meeting at the edge. In addition, it is necessary to have an orientation flag, because there can be more than two links around the edge. An edge has a pointer to a single loop-edge link, instead of two pointers.

The vertex definition has also to be extended to cope with multiple independent edge sets. A vertex needs a pointer to a chain of vertex-edgeset links (not as in section A.1.4), each of which has a pointer to a loop-edge link, instead of to one edge to facilitate traversal. See figure 5.3.

There is also a need for extensions to the model traversal utilities described in sections 3.2.1 and 3.2.2 and Appendix B to handle non-manifold structures. For example, the "loop opposite a given loop across a given edge" (lopel) has to cope with more than two faces at the edge, and extra functions, such as loop-edge link clockwise or counter-clockwise around an edge from a given loop-edge link.

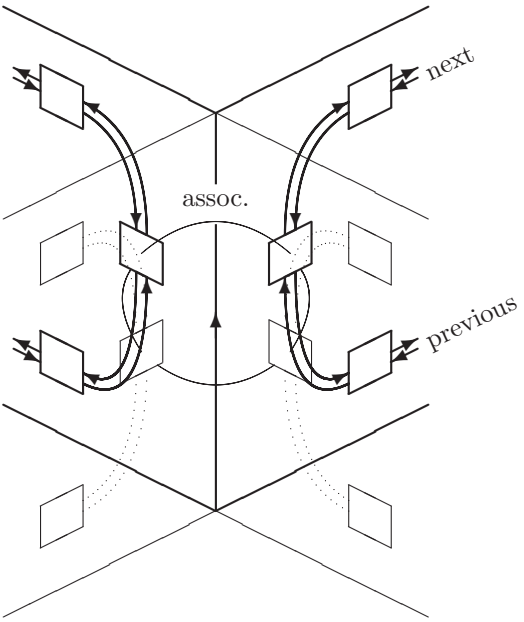


Figure 5.2: Edge links in a non-manifold model

As explained in the "conclusions" section, non-manifold models can be represented in other ways than by using the non-manifold datastructures described above. The edge and vertex datastructure definitions in Appendix A include an extra pointer field for associating matching edges and vertices with each other instead.

5.1 Conversions between representations

The reason for describing conversions between these representations is to show that there is no need to be pedantic about the use of one or another type to the exclusion of the others. There are some possible overheads with the conversions, but nothing too obviously difficult to cause major problems. In practice, for example, a part or parts of a model may become degenerate and a user may require these to be converted into full non-manifold portions. This could be achieved automatically, using a technique similar to Boolean operations, or by manually identifying the parts to be merged. Alternatively, a user may require that non-manifold 'star' edges representing idealisations be 'fattened' for practical purposes, such as process-planning. This is also relatively easy to accomplish by scanning the model for non-manifold edges, converting them to degenerate, locally manifold parts, and applying local operators to

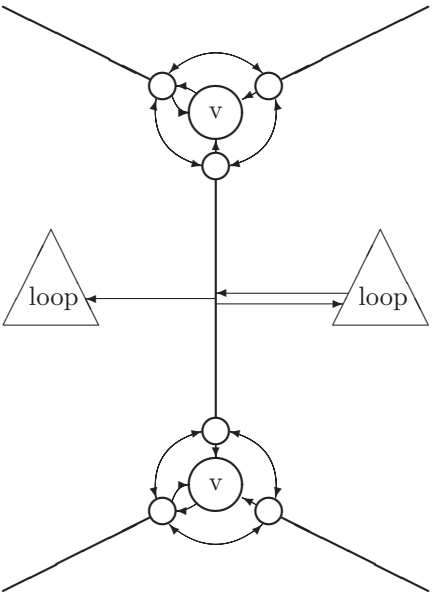


Figure 5.3: Vertex links in a non-manifold model

these, for example. The reasons for using a particular special representation, or for performing conversions, are heavily bound up with applications and personal taste and, hence, cannot be defined here. This chapter describes these representations as possibilities only.

For flat objects, all three representations are approximately equivalent in that conversions can be made between them, although with varying amounts of work. Similarly, non-manifold star and degenerate parts of volumetric models are also approximately equivalent. The conversions are outlined below.

Non-manifold star to degenerate: Radial edge structures can be converted to their degenerate model equivalents by inserting new edges, duplicating the single non-manifold edge, and splitting vertices, if appropriate. There should always be an even number of edge-loop links around the radial edge, so they can be grouped into adjacent pairs and assigned to new edges, and the end vertices of the original edge split if the new edges form two or more adjacent edge sets.

The problem is that there are two ways of grouping the links according to whether the edge is to represent a break in material or an infinitely thin section. A thin plate non-manifold model could be considered as a joined whole or as a set of separate parts, depending on the context, and the operation that is to be applied.

Non-manifold star to partial: Non-manifold models can be converted into partial models in a similar manner to the way that they are converted to

degenerate models, but only one pair of links (or possibly only a single link at model boundaries) is required in the final model.

The conversion processes are illustrated in figure 5.4. Two examples of non-manifold models are shown at the top of the figure, and possible degenerate and partial model equivalents are underneath. In the example on the left of the figure, the four non-manifold edges, edge 5, edge 11, edge 12, and edge 8, are copied, and the new edges are inserted into the model when converting to a degenerate model. However, as mentioned above, the non-manifold model is ambiguous, so there are two interpretations of the result, either as a connected model or as a set of four disconnected elements. For partial models, as shown at the bottom of the figure, there are three possible interpretations of the non-manifold model according to how edge-loop links at the edge are decomposed into pairs.

In the figure on the right, showing two cubes joined along a non-manifold edge, the degenerate representation can be created, as before, by duplicating the non-manifold edge and moving one pair of loop-edge links to the new edge. As before, the ambiguity of the non-manifold representation means that there are two interpretations of the resulting structure, either as one object with an infinitely thin 'neck', or as two touching objects. There is no equivalent representation for partial models.

Degenerate to star non-manifold: Converting degenerate models to non-manifold star models is more difficult than converting the other way because the edges on both sides of the degenerate model must be made to match (by imprinting or some such method) before the conversion can take place. Once the edge sets on both sides of the model do match, the conversion simply involves scanning through all interior edges, joining them with any other matching edges. A geometric test is needed if three or more matching edges exist to determine the correct ordering of the loop-edge links in the radial structure around the edge.

Degenerate to partial: In its simplest form, this involves 'losing' the edges, faces, and vertices from one side of the degenerate model. However, the complication is that it is not clear which side is to be removed and which should stay. If the sides of the degenerate model are different, then they can be coalesced so that the result model has all of the details of both sides, but some extra operational parameter must be supplied to determine which part of the degenerate model to remove.

The conversions are illustrated in figure 5.5.

Partial to non-manifold star: This conversion can be done by 'growing' faces on the undefined side of the partial model by creating matching faces to the faces on the defined side of the model. The matching edges can then be merged to create the final non-manifold model. See figure 5.6.

Partial to degenerate: As with the previous conversion, faces on the undefined side of the partial model need to be created, matching the faces on the defined side of the model. However, the final step, above, of merging matching edges is not needed.

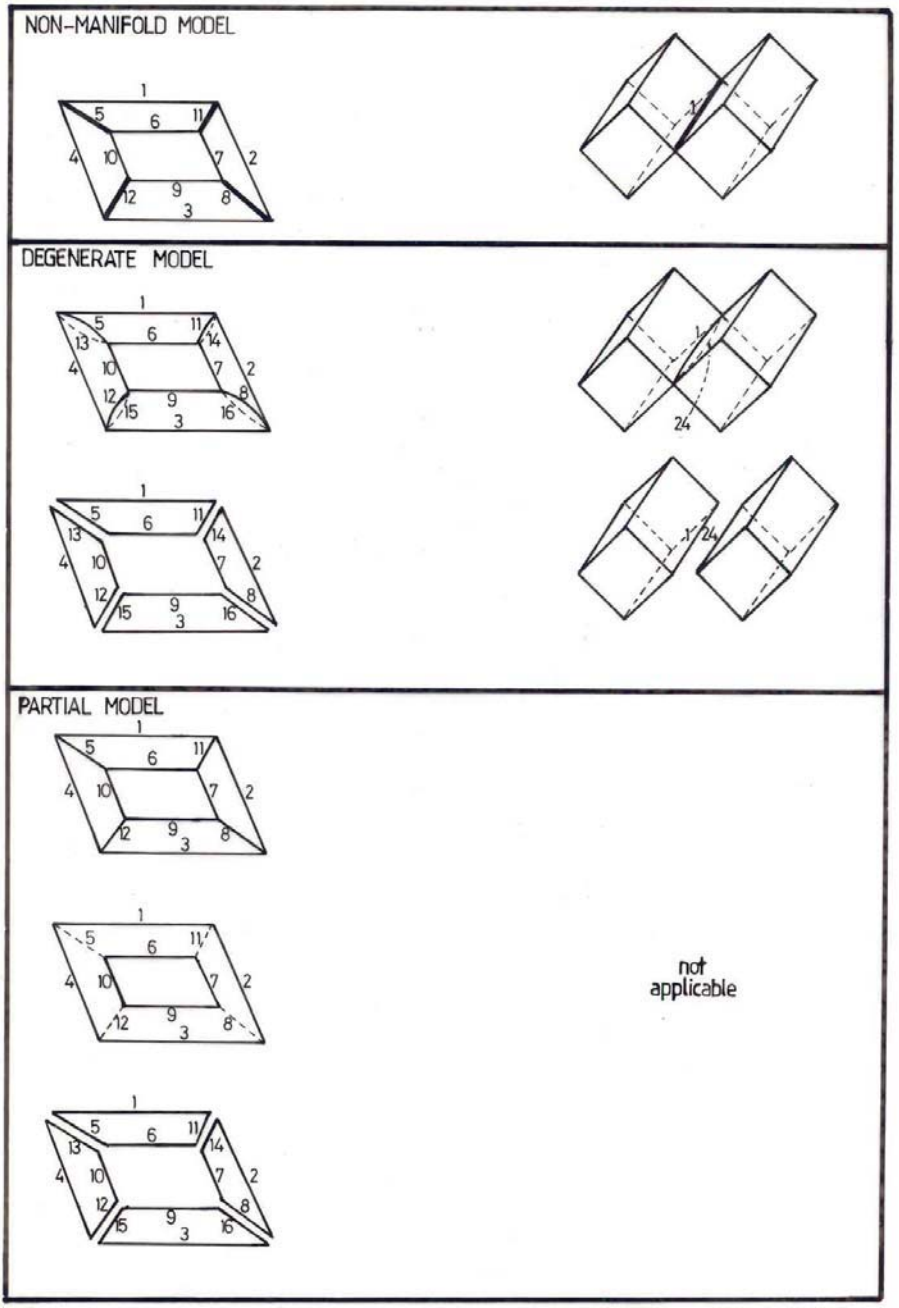


Figure 5.4: Non-manifold model conversion (from [125])

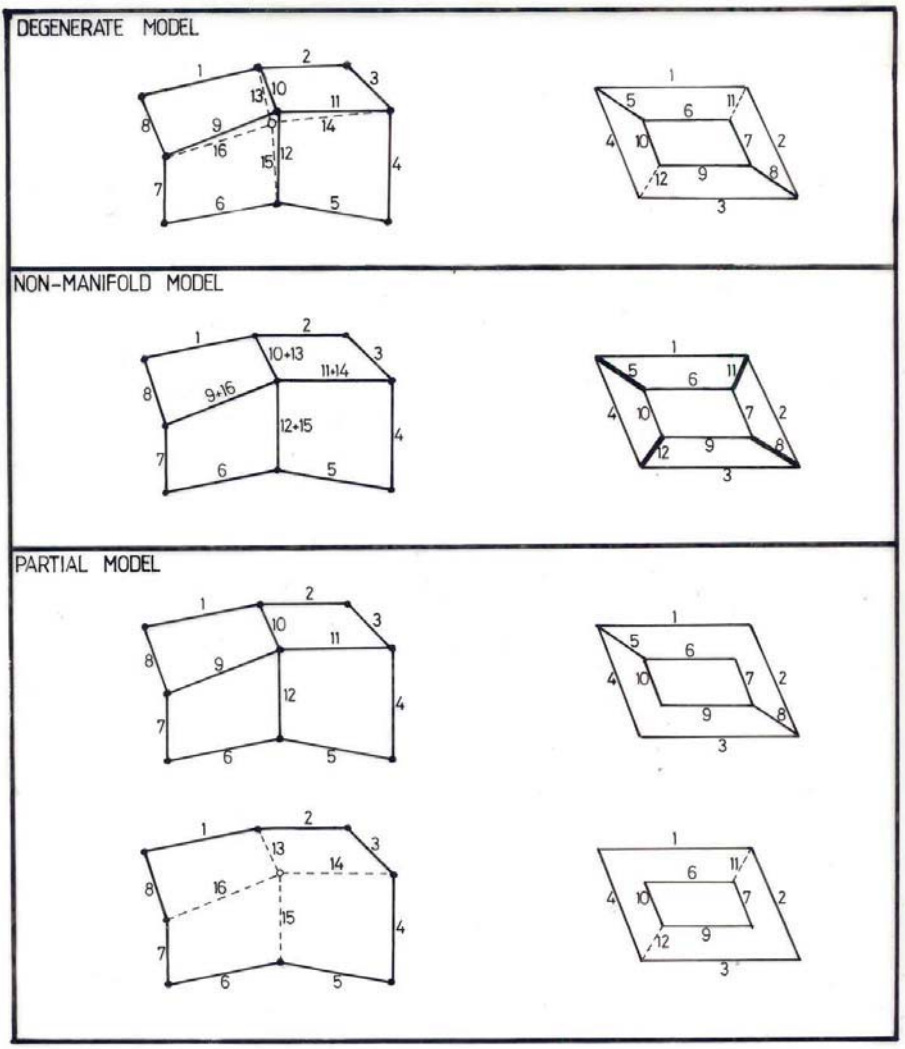


Figure 5.5: Degenerate model conversion (from [125])

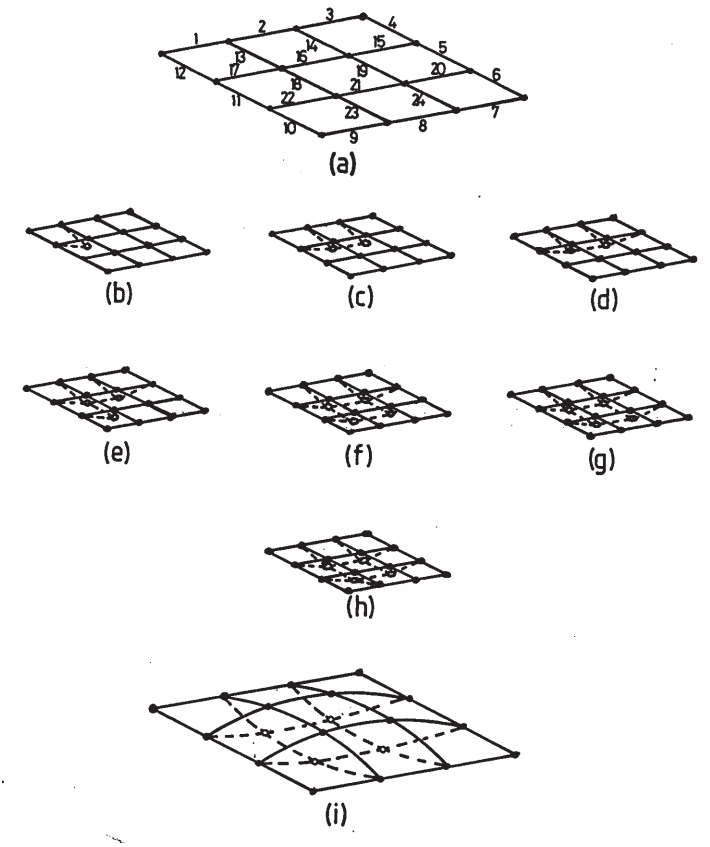


Figure 5.6: Partial model conversion (from [125])

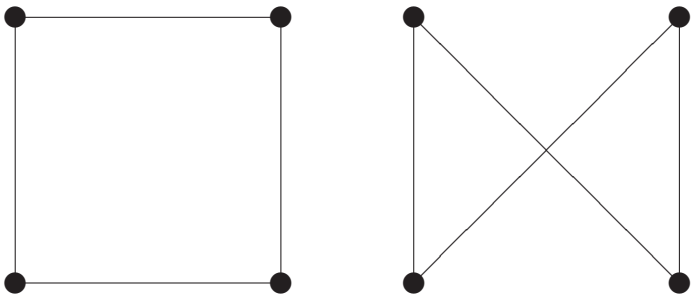


Figure 5.7: Simple wireframe objects

5.2 Modelling operations on special models

The descriptions of the effects of modelling operations on special model types is based on the algorithm descriptions in chapter 6.

5.2.1 Setting a wireframe model in a surface

If wireframe models are used to create two-dimensional shapes that are then swept, or extruded, to create volumes, then they have to have face structures added. Wireframe models consist only edges and vertices, and so the edges must be linked into loops.

The most convenient case is where there are only two edges at each vertex. An example is shown in figure 5.7a. However, although this makes things easier, it is not sufficient because edges may still intersect each other, as in figure 5.7b.

To detect the case in figure 5.7b, it is necessary, to check all edges against each other, that is, to intersect the edge curves and if there is an intersection point, to check whether it lies on the portion of the curve corresponding to the edge. If there is such a point that is not at a common vertex of the two edges, then there is a self-intersection. There are two options for handling such self-intersections. The first is simply to refuse to carry out the surfacing operation. The second option is to create a new vertex at the intersection point, to cut the edges at this point, and to relink the edges to the new vertex.

To make the connections, it is necessary to establish relations (direct pointers or links) between neighbouring edges. Consider figure 5.8. As there are only two edges at each vertex it is easy to establish that e1 has the next edge e2 in one loop (and the next edge e4 in the other loop). Note, as an aside, that loops are doubly linked. Establishing the next edge from e1 as e2 implies also that the previous edge around the loop from e2 is e1. Only the next relation is mentioned here for simplicity. The common vertex is v2 and at the other end of e2 from v2 is v3. The other edge around v3 is e3, so the next

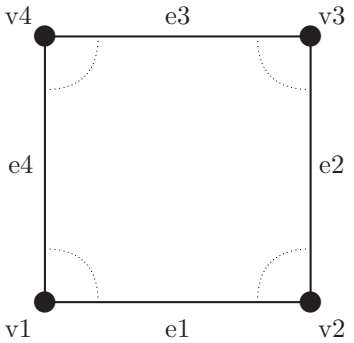


Figure 5.8: Linking wireframe edges

edge from e2 is set to be e3. At the opposite end of e3 from v3 is v4, with edge e4 as the other edge at v4. The next edge from e3 is set to be e4. As before, the opposite vertex along e4 from v4 is found, being v1, and the edge round this from e4 is e1, the starting edge, so the loop is complete. The edges sequence is e1; e2; e3; and e4. Establishing the second loop in the same way gives e1; e4; e3; and e2. The top and bottom loops are complete, and use of the utility to calculate the orientation of the loop (see E.9) is used to check the orientation against the surface. If no explicit surface is given, then only planar graphs should be handled, and the orientation utility gives the normal to the plane surfaces on either side of the graph.

Some special cases are shown in figure 5.9. These are more difficult to handle because where there are more than two edges at a vertex, the choice of the next edge is not obvious. On the left of figure 5.9 at v3, there is a choice of two edges as the next edge, e5 and e3. Here it is possible to use a trick: that the next edge counter-clockwise around the loop from e2 is the edge clockwise round v3 from e2, that is v5, using a surface normal to establish the clockwise relation. This gives a loop of e1; e2; and e5. Starting from e5, and using the same trick you get a second loop: e5; e3; and e4. Finally, the last loop is established as before as: e1; e4; e3; and e2 using the negated normal to find the back side of the shape. See Müller et al. [89] for details.

The figure on the right of figure 5.9 is more difficult. This might be flagged as an error because e5 and e6 intersect each other, but there is a valid interpretation of this case as well. Using the above trick, again, you would get a loop: e1; e6; and e4. Starting from e6 you would go to e2 and then e5. However, there is a problem establishing this link because the orientation edges around v1 for linking are already connected. Instead e3 needs to be used as the next edge, giving a second loop: e6; e2; and e3. On the back side there are again two loops: e1; e5; and e2 and e5; e4; and e3.

The problem with these two is that the assignment of loops is ambiguous

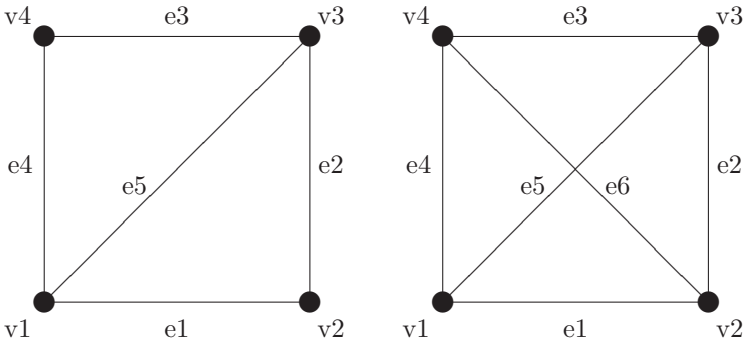


Figure 5.9: Complex wireframe objects

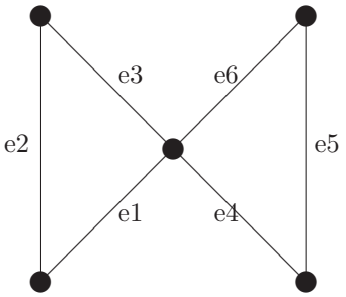


Figure 5.10: Special case wireframe object

and it is not clear what the user really would want. Also, compare this approach with the 'inking-in' procedure described in chapter 12 where Euler operators are used to establish the loops directly using help geometry as a guide to the geometry of the final figure.

Another type of complex figure is shown in figure 5.10. Here the result might be two sheet objects just touching, or one sheet object with a non-manifold vertex.

Finally, consider the cases shown in figure 5.11. Here there are two types of a multi-piece wireframe structure. In figure 5.11a, the two pieces are separate. The question is whether to return them as a single object with two shells or as two objects. It may be easier to make the result of the operation a single object with two shells, but then when the object is swept, or extruded, two separate objects should be created.

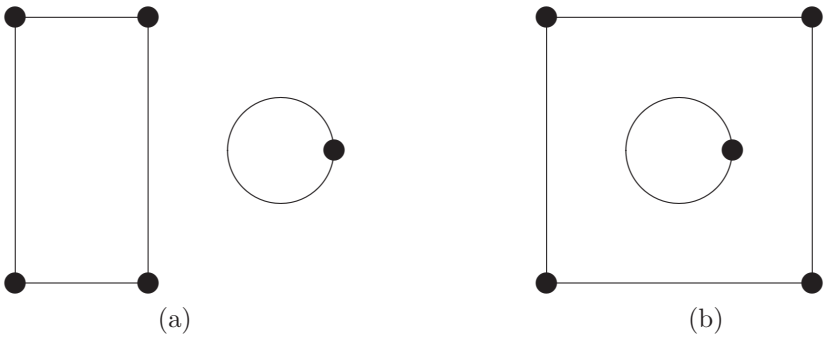


Figure 5.11: Multipiece wireframe objects

On the right, in figure 5.11b, the square piece surrounds the other, the circular piece. Here there are two options for combining the pieces. One option is that the circular piece appears on one side of the object. The other option is that it becomes a hole through the square piece. If the first option is chosen and then swept, effectively, the circular part is simply displaced or the faces have to be swept separately.

5.2.2 Blending and chamfering

Figure 5.12 illustrates the effect of chamfering a non manifold edge. There are two interpretations of the operation according to whether the edge is interpreted as a place where two parts just touch or whether they just miss each other. There is not the same problem with degenerate or partial models, because edges are not ambiguous, although there are different problems with these models.

Chamfering a similar edge in a degenerate model is not a problem because the interpretation is clear, although it is necessary to chamfer both matching edges, or edge sets to get a similar effect. However, chamfering an edge in a degenerate sheet model can cause different problems. Examples of the kind of problem that can occur are illustrated in figure 5.13. If the edge to be chamfered is convex (figure 5.13a), then the first chamfer will actually leave a negative volume as part of the sheet model (figure 5.13b). Chamfering a matching edge restores the sheet model, but care must be taken to avoid leaving degenerate faces bounded by coincident edges at the ends of the chamfer edge (figure 5.13c). If the edge to be chamfered is concave (figure 5.13d), then the intermediate model will have a positive volumetric sub-part (figure 5.13e) and the sheet model can be restored by chamfering the matching convex edge (figure 5.13f) and then merging the duplicate edges.

If the edge being chamfered has no single matching edge, as in figure 5.14, then the process of restoring the sheet model can be difficult. When cham-

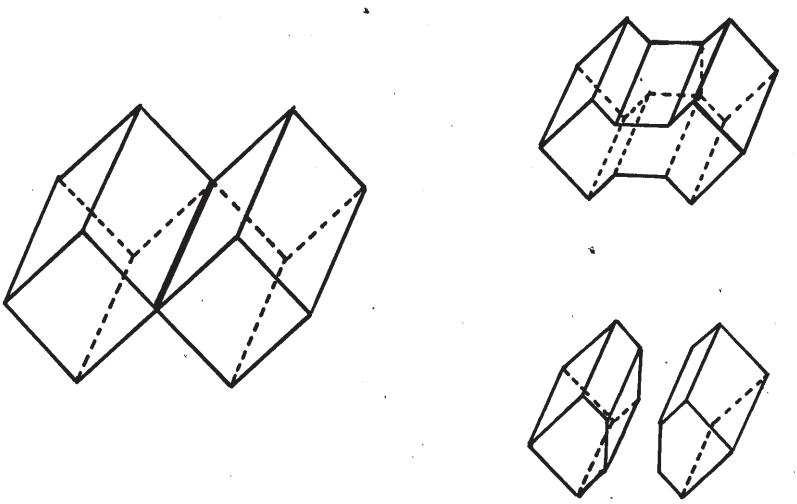


Figure 5.12: Chamfering a non-manifold edge (from [125])

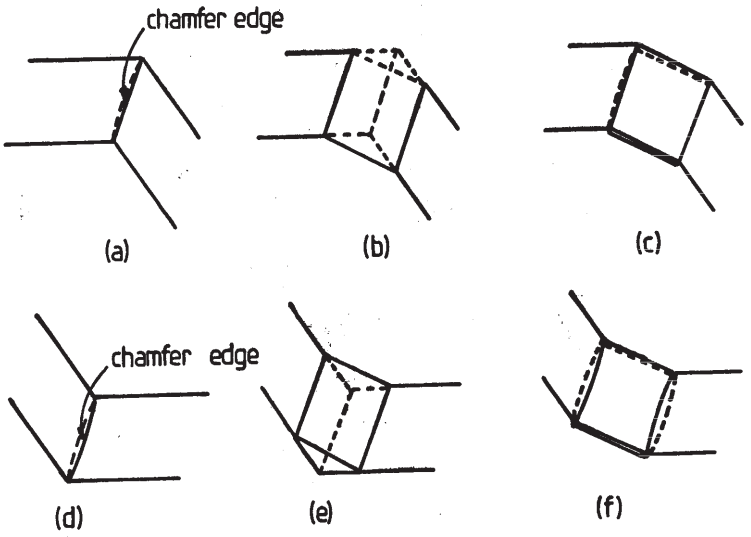


Figure 5.13: Chamfering an edge in a degenerate sheet model (from [125])

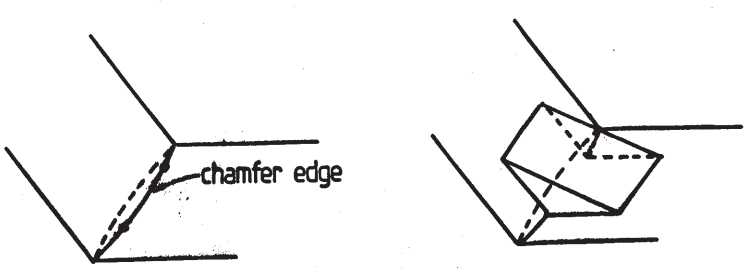


Figure 5.14: Chamfering an unmatched edge in a degenerate sheet model (from [125])

fering matching edges, as above, the process can be simulated by applying a suitably general chamfer operation to both parts, leaving the model as a degenerate sheet object. To avoid problems, therefore, it is preferable to perform a preprocessing step, and if the edge to be chamfered is not matched, then modify the object so that there is, and apply the chamfer operation to the new matching edge.

At the very least, applying this kind of modelling operation to specialised model types can cause a lot of extra 'housekeeping' work as volumetric subparts are created and removed. For idealised models, therefore, it is better to use special tools, possibly built from standard shape manipulation operations, but directed to maintain the model as an idealisation.

Chamfering edges in partial models is much easier, because they behave as edges in solids. However, some care may be needed if the vertices of the edge are adjacent to the boundary of the partial model if that boundary is to remain fixed, i.e., if it is a portion of a model that has been 'pulled off' and that is expected to be glued back subsequently.

For idealisations of thin plate objects, the side edges in non-manifold, degenerate, or partial models must be handled specially. In simple model terms, it does not make sense to chamfer these edges, but this can be interpreted logically as meaning that the degenerate faces corresponding to these edges are to be modified in some way. As described above, modelling with idealised models requires special consideration to maintain the validity of the idealisation. An example of a specialised application tool is the GPM assembled plate module (APM).

Chamfering is strongly related to blending, and hence, similar considerations exist for blending.

5.2.3 Sweeping

The effects of sweeping a face adjacent to a non-manifold edge and a face with a degenerate edge are illustrated in figure 5.15.

For a non-manifold model, the edges around the face can be preprocessed and converted into degenerate edges before sweeping. The potential ambiguity in the conversion can be resolved by interpreting the effect of the operation. In the example, the manifold edge is interpreted as concave if the face is to be swept upwards, and convex if the face is to be swept downwards. This means that when sweeping downwards, the object is first split into two separate objects, as shown.

In the equivalent degenerate model, there is no information that an edge adjacent to the face is special in any way. Sweeping upwards, therefore, produces the same result as with a non-manifold object, but sweeping downwards leads to a strange object with two volumetric parts connected by a sheet part.

Both of the above interpretations of the effect of the downward sweep are justifiable, perhaps somewhat surprisingly. The interpretation where the objects are separated has the advantage that the result is a valid set of volumetric models. The interpretation of the sweep as leaving a sheet connecting section is easier to repair than the interpretation where the result is returned as two separate objects. With only a little effort, it is possible to restore the original single object from the separate objects, should this be required, and it is also possible to annihilate the sheet section and change the degenerate interpretation into two separate volumetric parts. The problem is not that one particular interpretation is better, because different implementors are almost bound to have different ideas about what is 'better', but it concerns the fluency of use of this kind of special case. As either interpretation is possible, it means that the user has to be aware of what are the different edge representations, degenerate or non-manifold, and to be aware of how the modelling system will react, for any particular case.

For a partial model, face sweeping is the same as for face sweeping in volumetric models because partial models behave locally as volumetric models. The outermost edges of the partial model should be classified as 'static'; that is, they would be fixed.

Sweeping wires, that is, simple edge sequences with no attached faces, to create degenerate models is dealt with in [123] and [124] and in section 6.3. The algorithm creates a sheet object with back-to-back faces, duplicating edges on the front and back of the object. The sweep process for creating either non-manifold or partial models from wire models is similar. To create a non-manifold object, the matching edges can be created and then merged, or alternatively, instead of creating complete duplicate edges, extra pairs of edge-loop links can be created. The BUILD system [11] created partial models from open, non-branching wires (because closed wires automatically became lamina). However, it is not clear whether it is valid to generate partial objects in this way; it would seem better to create non-manifold or degenerate swept models from wires. In any case, the process can be simulated by sweeping each edge separately to produce simple sheet models and then joining them along coincident edges (corresponding to swept common vertices), as described in the next section, to produce the final model.

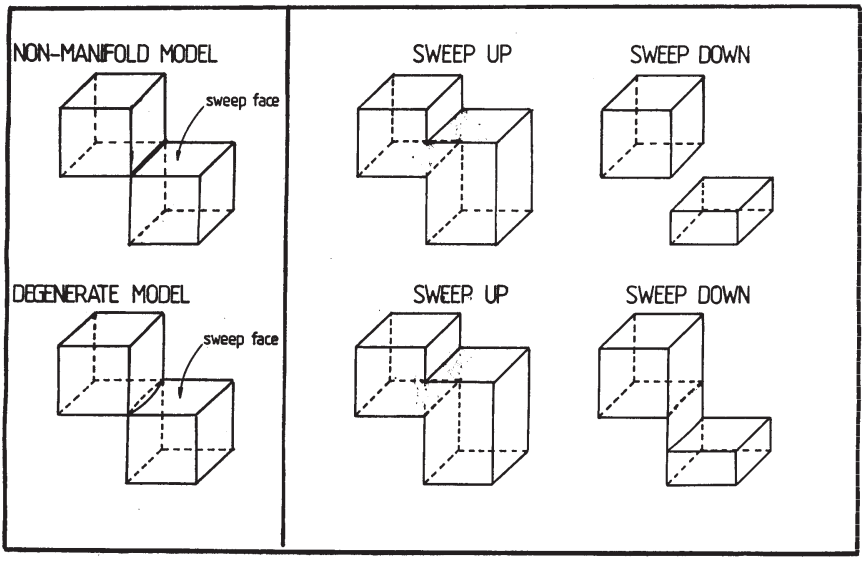


Figure 5.15: Sweeping faces with non-manifold and degenerate edges (from [125])

5.2.4 Joining and splitting along edges

The effects of joining edges in non-manifold, degenerate and partial models is summarised in figure 5.16.

Non-manifold models are joined by merging their start and end vertices and then transferring the loop-edge links from one edge to the other. The surviving edge will then have (at least) four loop-edge links. Note that if the loop-edge links are arranged in pairs, then there are two ways of joining edges: as places where edges just miss each other, and as places where the objects just touch, i.e., where there is an implicit gap and where there is an infinitely thin section of material. This suggests that there should really be two different types of joining operation to reflect the intention of the join.

Edge joining in degenerate models is described in detail in [124]. Basically the process consists of merging the start and end vertices of the edge and then modifying the loop pointers of the edges. For example, if the edges are oriented in the same direction, the right loop pointers of the edges are swapped, preserving both edges on opposite sides of the object.

Joining edges in a partial model is similar to the edge joining process for edges in degenerate models with an additional deletion of the edge on the undefined side of the object.

The inverse operation of splitting an object along an edge is illustrated in figure 5.17.

When splitting a non-manifold edge lying between four faces, i.e., with

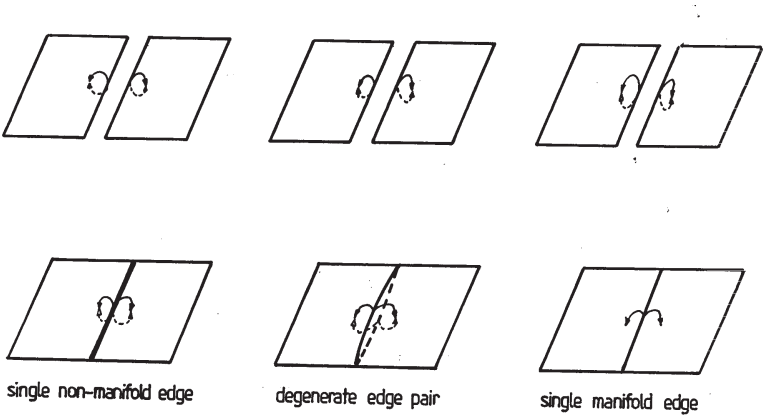


Figure 5.16: Joining edges in non-manifold, degenerate, and partial models (from [125])

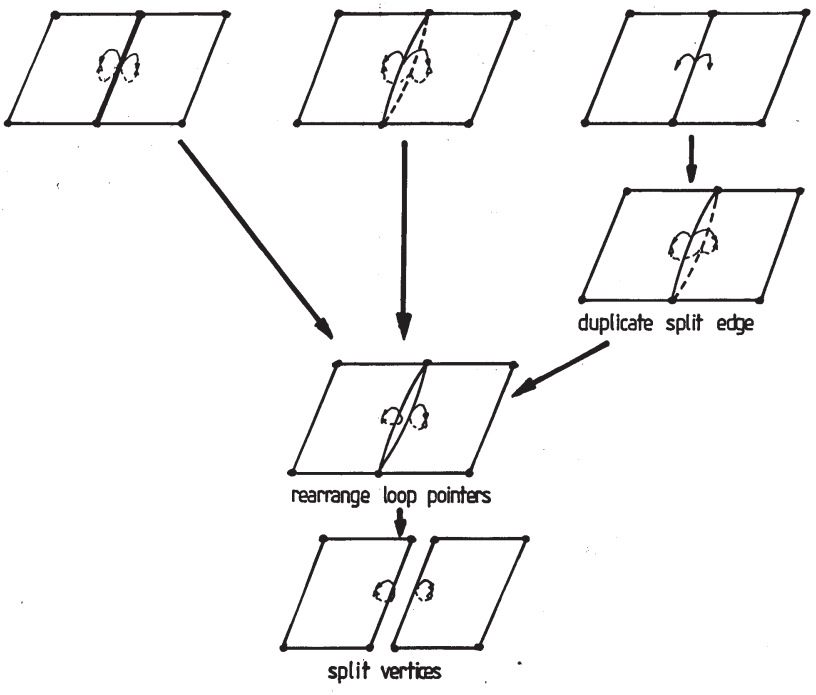


Figure 5.17: Splitting models along non-manifold, degenerate, and partial edges (from [125])

four loop-edge links, the process is similar to the task of converting the non-manifold edge into a degenerate edge and then splitting along this edge. Converting the non-manifold edge involves separating the links into appropriate pairs, duplicating the edge, and assigning one pair of links to the new edge. The vertices then have to be split so that separate edge sets belong to separate vertices.

If there are more than four loop-edge links at the edge, then there is some ambiguity about the operation. Without extra information, it is impossible to tell which piece of object is to be split off; hence, the meaning of the operation can be taken as separating the object into several pieces at the edge. In practice, this involves the same sort of process; the loop-edge links are separated into pairs and assigned to new edges. The start and end vertices are then split, and each separate edge set is assigned to a separate vertex.

Splitting along a degenerate edge involves, first, finding a matching edge or edges. Ideally there should be a single matching edge for the edge to be split, although this is not guaranteed. Assuming that a matching edge can be found, it is necessary to swap one of the loop pointers from the edges and to split the start and end vertices, if necessary. If there is no single matching edge, then the action taken is unclear; there is no guarantee that the chosen edge is matched by any edge. Where a degenerate representation is used as an idealisation of a flat-plate object, then an unmatched edge could be projected onto the other side of the sheet object before splitting.

In partial objects, the edge along which the object is to be split has to be projected onto the other, undefined side of the object. A similar technique can then be used to split the object. If the edge to be split is not adjacent to the partial object boundary, then this means, in effect, that a hole will be created through the object.

5.2.5 Planar sectioning

The description here is based on what can be termed a 'nose-following' section seam insertion algorithm. This process involves finding a face on the intersection boundary and then following the intersection seam around the body until it is completed. Creating the section seam for models with non-manifold and degenerate edges is illustrated in figure 5.13. Care must be taken when the section seam cuts through non-manifold or degenerate edges.

With non-manifold edges, the section seam splits into several pieces, corresponding to several section faces joined at a vertex. With the object shown in figure 5.18a, two cubes joined along a non-manifold edge, the section seam is as shown in figure 5.18c, with two faces joined at a common vertex.

With the equivalent object with a degenerate neck, i.e., with coincident edges on opposite sides of the model, as shown in figure 5.18b, the section seam bounds a single face, as shown in figure 5.18d, which has two geometrically coincident vertices. Obviously it is desirable to merge these two vertices, but in general it is difficult to know whether the section seam has such degenerate

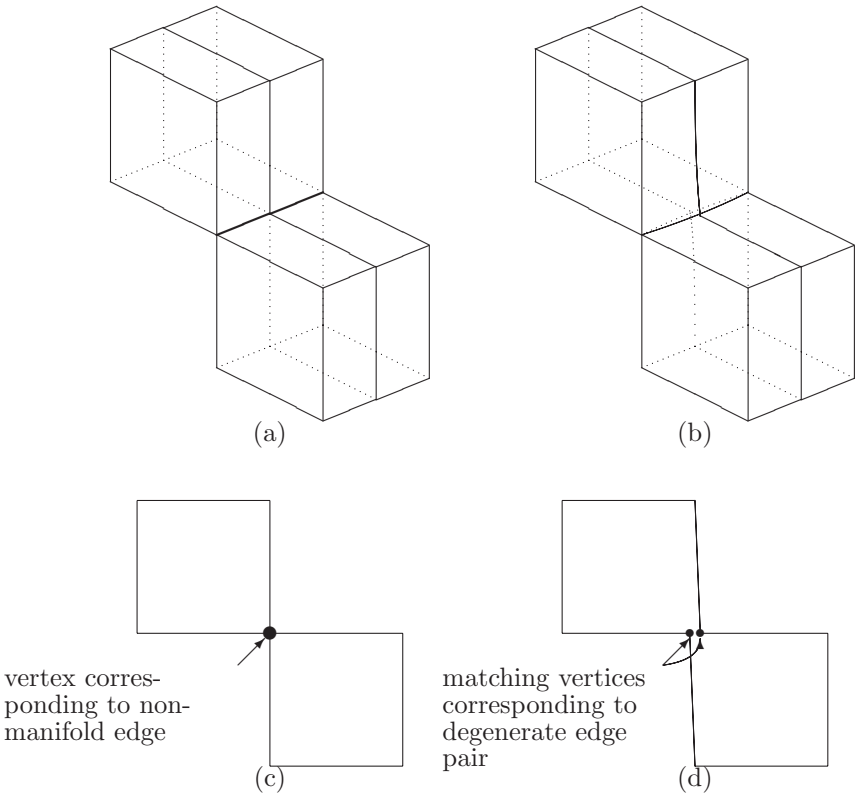


Figure 5.18: Planar sectioning of non-manifold and degenerate models (from [125])

parts without information that there are matching edges in the model.

When a model is a sheet model, then faces bounded by the section seam should be treated specially. When sectioning a non-manifold sheet model, each face in the section plane will be bounded by two coincident edges lying on opposite sides of the object and running between the same two vertices. These two edges can either be merged to produce a single non-manifold edge that can then be split or split using a similar technique to that for splitting along degenerate edge pairs.

With a degenerate sheet model, the face or faces in the section seam are bounded by sets of coincident edges. Once the section seam has been created, the object can be split using the edge splitting technique described in the previous section.

With a partial sheet model, the section seam will consist of a single set of edges on the defined side that have to be 'projected' onto the undefined side

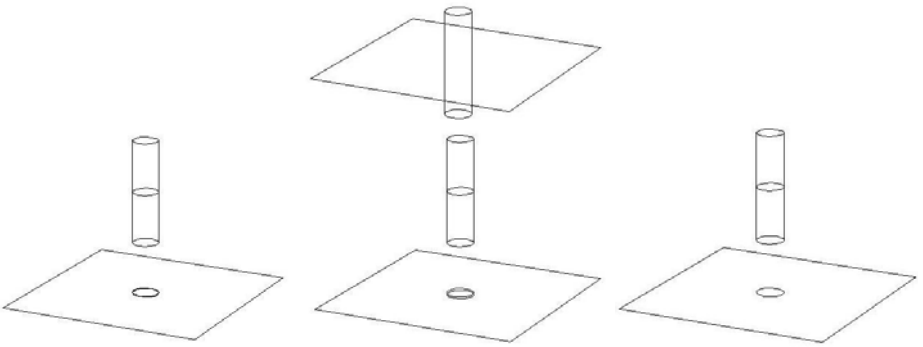


Figure 5.19: Cylinder-special sheet model Boolean operations (from [125])

to perform the split.

5.2.6 Boolean operations

Boolean operations between volumetric models with or without non-manifold edges or degenerate edge pairs involve similar considerations to those for sectioning. However, it is interesting to consider Booleans between volumetric models and special sheet representations.

Perhaps the most appropriate type of sheet model for Boolean operations is the partial model, providing the Boolean interaction boundary is limited to the interior. If it does, then the partial model behaves as though there were an infinite amount of material behind it.

To illustrate this, consider the Boolean operations illustrated in figure 5.19. This shows the result of creating the Boolean interaction boundaries between the cylinder and the three special representation types described in this paper. The Boolean interaction ring in the cylinder consists of a single set of simple edges in all three cases. For the non-manifold model, the intersection ring consists of a circular non-manifold edge (figure 5.19a). For the degenerate sheet model, the Boolean interaction ring consists of a pair of circular edges, one on each side of the model. With the partial sheet model, the Boolean interaction seam consists of a single simple circular edge.

Problems come when uniting the objects to produce the final result. With the degenerate and non-manifold models, the result is partially a sheet model and partially a volumetric model. With the partial model, the result is still a sheet model, because only one part of the cylinder is included in the final result. See figure 5.20.

As with chamfering, the inclusion of a volumetric part with a sheet model may lead to complications for other modelling operations, and needs to be considered thoroughly.

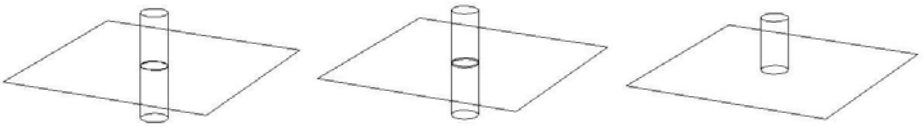


Figure 5.20: Adding a cylinder to the three special sheet model types (from [125])

5.2.7 Shelling objects

To create a shell object from a volumetric model is relatively easy. The volume is simply copied, negated, and put back as a new shell in the original object. This is illustrated in figure 5.21. The original object is shown in figure 5.21a. This is copied (figure 5.21b). The copy is negated (figure 5.21c) and the copy merged with the original object as a new shell (figure 5.21d).

After shelling it is possible to delete faces to create open structures. With the external face to be deleted identified, the matching internal face is found. Normally the faces should be in the same order in the shells, making matching simple, but a geometric check should be applied to check this. Figure 5.22a shows an external face and a matching internal face. The loop of the internal face is made into a hole loop in the external face (figure 5.22b) and the face is deleted. This is a basic Euler operator, as described in Appendix F. The first time this is done, the operation kills a face, makes a hole loop, and decreases the multiplicity. The final step is to merge matching edges in the hole-loop with the corresponding faces in the outer boundary of the external face, killing this face in the process (figure 5.22c).

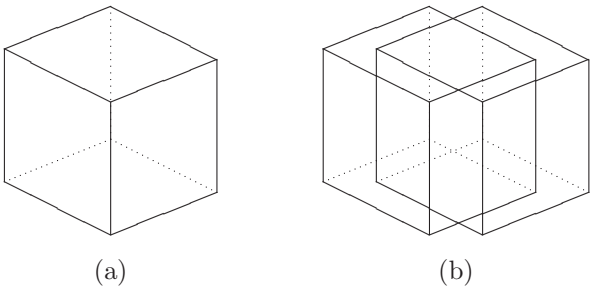
If the face to be killed has hole-loops, then the process is similar but each hole-loop in the internal face becomes an external loop of a new face and the corresponding hole-loop from the external face is moved into this new face. The merging process is then applied to the original external face which has now been separated from its hole-loops.

If one edge in the external face is adjacent to an internal face, then this edge has to be deleted. This happens when two neighbouring faces in a sheet object are deleted.

5.2.8 Unfolding models

This algorithm is for unfolding models. The topological part is not particularly difficult; it is the geometry that complicates things. Put simply, the process involves cutting the object along certain edges, flattening curved geometry and then rotating faces so that they lie in the same plane.

The algorithm uses the dual of the object to be unfolded as a support for cutting, so the first step is to create the dual of the object, as described in section 6.10. The dual is well known in graph theory. The dual of an object



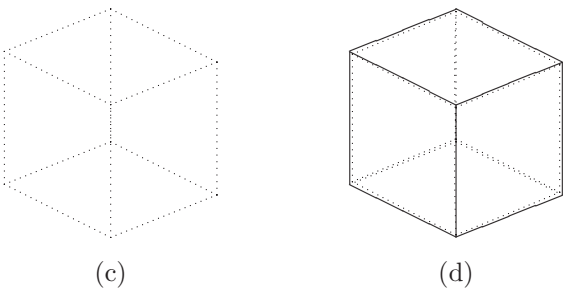


Figure 5.21: Cube and shelled cube

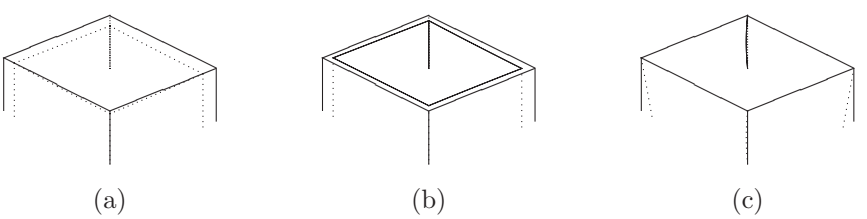


Figure 5.22: Deleting faces in a shell object

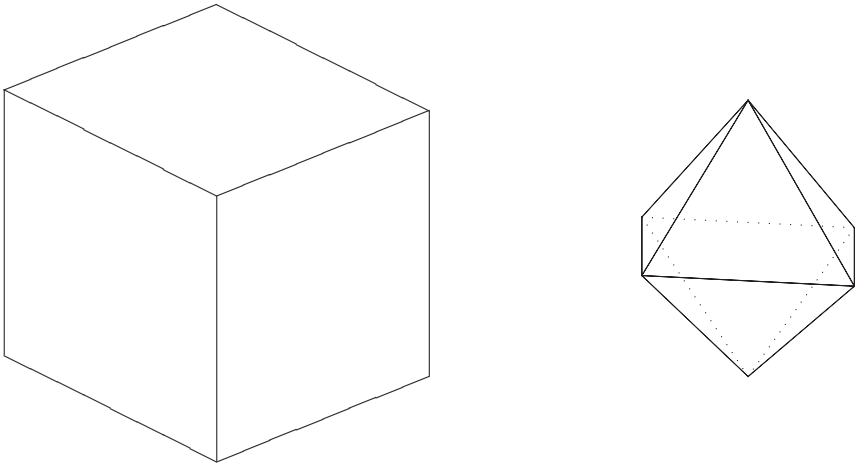


Figure 5.23: Cube and dual

model is another model where each face in the original model corresponds to a vertex in the dual and each vertex in the original model corresponds to a face in the dual object. There is an edge in the dual for every edge in the original model, but these are reconnected so an edge lying between two faces in the original model runs between the two vertices corresponding to those faces in the dual. An example is shown in figure 5.23

The dual representation is used as a guide for cutting edges before unfolding. If the object being unfolded is a sheet object, then the dual needs to be postprocessed. The 'sharp' edges in the model represent a boundary between the inside and the outside of the object. The dual needs to changed by deleting these edges. Once all of these edges have been deleted, the dual consists of two parts and one of these is simply deleted.

If the object is a solid, then after the dual has been created, it is converted to a sheet object by copying it, negating it, and inserting the shell of the copy as a new shell in the original object. This is the process described in section 5.2.7.

The object is then sliced along edges using the dual. The unfoldable graph is a so-called 'spanning graph' of the dual graph. The spanning graph is a graph in which all vertices are present but there are no loops. So, every non-wire edge (that is an edge with different left and right faces) is deleted from the dual and the original object is sliced along the corresponding edge. For the cube example in figure 5.23, the edges to be cut in the original object and deleted from the dual are shown bold in figure 5.24.

Once cut, the faces are then rotated so that they lie on a single plane. The faces are rotated in pairs so that they remain back-to-back. Figure 5.25.

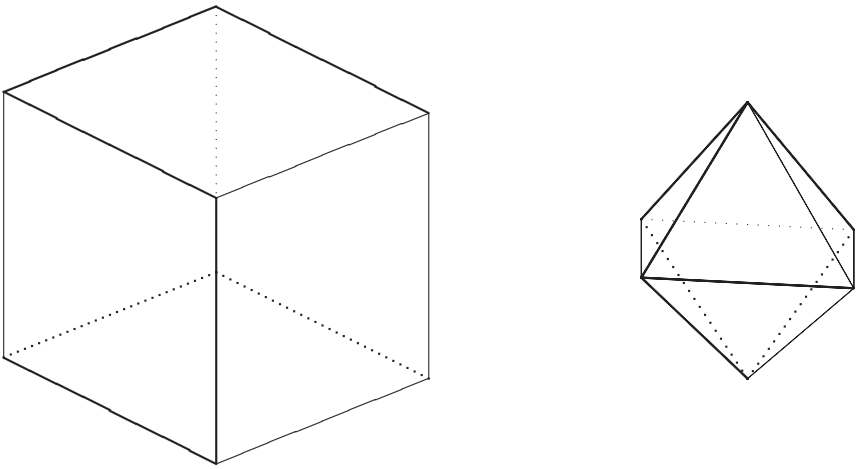


Figure 5.24: Cube and dual

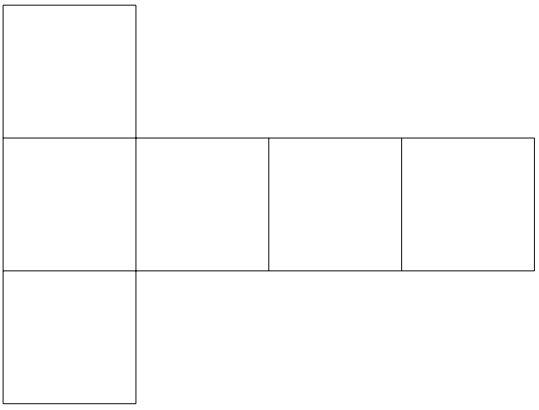


Figure 5.25: Flattened cube

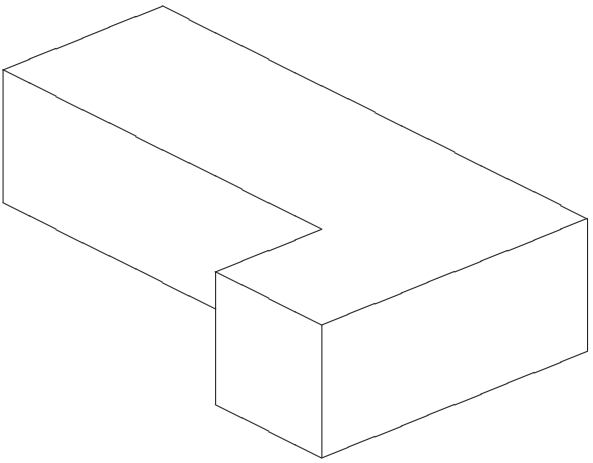


Figure 5.26: L-shaped object

There is an extra option, to note the concave edges being unfolded and to mark them to be shaped with a V-profile for bending.

There are obvious problems, though, because of the geometry of the object. With the object in figure 5.26, the concave edge means that there is a potential conflict and care must be taken about which edges to cut.

The are more serious problems with an object, such as that in figure 5.27.

This object cannot be made in one piece because the elements of the pocket will collide with each other and the surrounding objects. Curved elements are also a problem because they should have point contact rather than an edge. These problems can really only be solved by having an interactive system where the user controls where object parts are to be completely separated and where the cuts are to take place.

5.3 Compound models

It is possible to combine models to produce overlapping objects with distinct volumetric parts, or possibly even a volumetric model with partial feature descriptions. Compound modelling was suggested by Pratt [101] as a way of preserving features as explicit parts of a model so that they could be deleted or modified. Although it is not clear, from a modelling point of view, whether this is useful, the possibility exists to provide such models. Having to deal

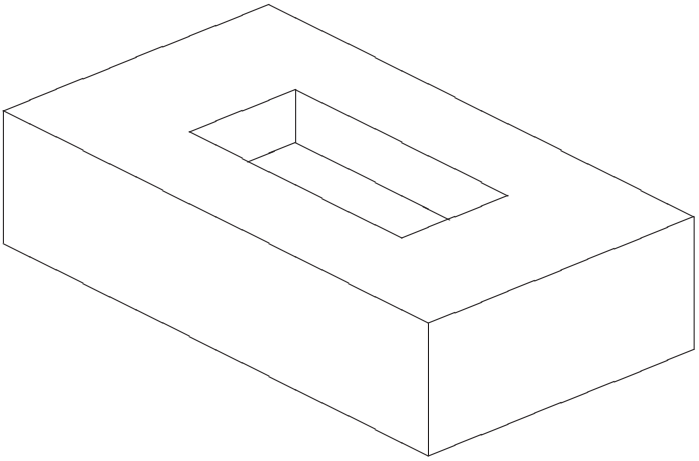


Figure 5.27: Object with pocket

with this type of model considerably complicates modelling algorithms, and it would require considerable effort to construct a fully functional, consistent compound modeller. Also, it is not clear whether this information cannot be provided in some other form, such as facegroups, for example. Feature recognition techniques (see chapter 9) perform local analysis of a body and try to construct some 'sensible' decomposition of the object, although this is a difficult problem in general.

One application where compound models may be quite useful is for process planning. One method for process planning involves 'building back' the final object to produce the blank. To illustrate this, consider the simple object in figure 5.28.

The original object is shown in figure 5.28a. With the feature removed and made into a volume and the object 'healed' (i.e., the gap left by removing the feature is covered over), two volumes are present, as shown in figure 5.28b. Recombining the healed object and the feature as a compound model involves making the original feature boundary edges into non-manifold edges and connecting them to their matching edges in the feature volume, as in figure 5.28c. Finally, if the feature is made into a partial model instead of a volume model, and recombined with the healed volume in the same way, the object is shown in figure 5.28d. The edge marked in figure 5.28c as non-manifold is a manifold edge in figure 5.28d because there is no matching edge in the partial model.

The possibility of creating compound models with partial models causes serious problems for a non-manifold representation using a simple ring of loop-edge links like that described above. Instead it is necessary to pair the non-manifold links, producing something like Luo and Lukacs' bundle [79] [78]. A possible extended non-manifold link is as follows:

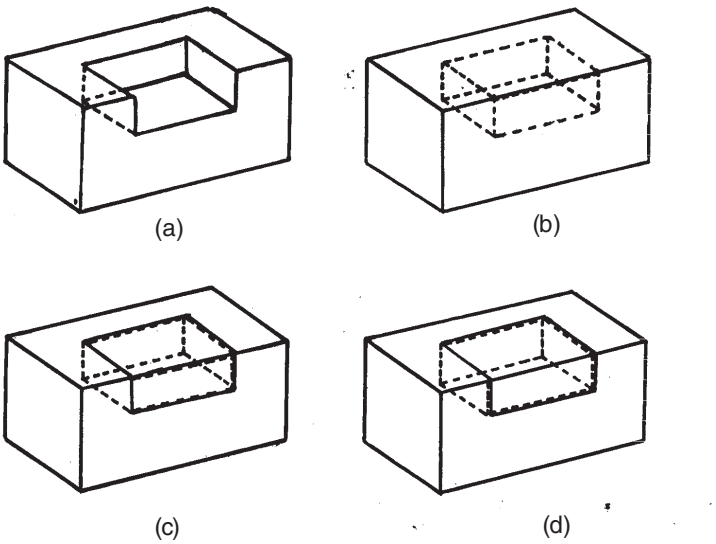


Figure 5.28: Feature, removed feature, and compound model

edge	EDGE ptr.	Pointer to the edge owning the link.
rloop	LOOP ptr.	Pointer to the 'right' loop owning the link.
lloop	LOOP ptr.	Pointer to the 'left' loop owning the link.
rnext	N-M LINK ptr.	Pointer to the next link around the right loop.
rprev	N-M LINK ptr.	Pointer to the previous link around the right loop.
lnext	N-M LINK ptr.	Pointer to the next link around the left loop.
lprev	N-M LINK ptr.	Pointer to the previous link around the left loop.
assoc	N-M LINK ptr.	Pointer to the next link around the edge.

For partial models, one of the rloop, lloop pointers (and the corresponding next/previous pointers) would be set to NULL.

With this scheme, though, the non-manifold link starts becoming complicated, almost like a standard manifold edge definition. For this reason, the datastructure definitions in Appendix A do not use the non-manifold scheme, but instead extra pointer fields are included in the vertex, edge, and face definitions for associating matching topology.

5.4 Conclusions

All three types of special representation have their uses in particular applications of B-rep, the problem is to be consistent about how they are used. As a general guideline the roles of the three special representations can be summarised as follows:

1. Partial models are suitable for representing surfaces and isolated object parts for a limited class of modelling operation.

2. Degenerate models are suitable as idealisations of thin plate models and for representing temporary stages in design.

3. Non-manifold models are suitable for applications, possibly for representing compound objects for process planning, or finite element models.

It should be noted that non-manifold models have strong connectivity information, but they are potentially ambiguous for modelling. Degenerate models are complete and unambiguous, but larger than the other two types. Partially complete forms are more efficient, representationally, than the non-manifold or degenerate forms for representing a limited class of models and with a limited set of operations. It would be wrong to make a definitive statement that any of the three is superior to the others; they all have strengths and weaknesses, and they all need special handling during modelling.

Other general conclusions are that the non-manifold datastructure is, perhaps, the most complete datastructure because it does not preclude the use of partial or degenerate models. However, one advantage of non-manifold representations, that is, that associativity is recorded explicitly, can be offset by adding an extra field to the standard B-rep edge and vertex structures to allow matching edges and vertices to be linked together into circular lists, as in the datastructures described in Appendix A. This would mean that the problem of the ambiguity of interpretation of non-manifold edges would be overcome,because degenerate models are unambiguous, while allowing special cases to be identified more easily. The ambiguity of simple non-manifold representations can be overcome by using the kind of 'wedge-based' structure suggested by Lukacs and Luo [78] and having radial edge links in pairs rather than having radial-edge links as a simple list, as described in the section on compound models. This also means that special models made up of a mixture of partial and other models, or simply partial models, can be represented properly.

Although not illustrated here, similar ambiguity problems occur with non-manifold vertices. There is a potential problem of traversing all edges (and, hence, faces) at a non-manifold vertex if the vertex refers to a non-manifold edge. For this reason, it is better if a non-manifold vertex refers to a non-manifold link, instead of to an edge. Lukacs and Luo also proposed a 'loop

equivalent' for vertices, called a bundle for associating separate edge sets at vertices that may be useful for avoiding the ambiguity. The degenerate model equivalent is to add links between related vertices to allow matching vertices to be identified easily, as in Appendix A.

Where the special model types are used to represent idealisations, there are problems about using standard modelling operations. In general it is better to have specific modelling operations, possibly built on top of standard modelling operations, which can tidy up after the operation to ensure that the special representation conditions are maintained.

It is helpful to be able to use automatic model verification techniques to check whether models contain invalid parts, such as volumetric parts in sheet models, or degenerate faces bounded by pairs of matching edges. Such a facility would indicate possible problems and allow a human user to concentrate on particular parts, modifying them if necessary.

Chapter 6

Stepwise modelling algorithms

The following descriptions are of several 'stepwise' algorithms that change a model gradually. There are two main reasons for developing stepwise algorithms: first, to preserve the validity of the model as far as possible, and second, to make the algorithms easier to comprehend.

Preserving the topological validity of the object during a modelling operation facilitates error recovery, should the operation fail. For example, the original Boolean operations in BUILD performed the intersections, without integrating the results into the models, and then pulled the objects apart, integrated the intersection results, and put them back together as one operation. This has the disadvantage that a great deal of work is done at one time, and when problems occur, it is difficult to examine the erroneous object because it can contain partially incomplete parts. The stepwise algorithms perform more gradual changes and, although sometimes less efficient than operations that perform major changes in one step, it is easier to monitor the changes, first to understand what is happening, and second to recover from errors. There is a technique, I know it from ACIS, that uses a so-called "Bulletin Board" to record changes to elements so that the original elements can be restored if necessary. A simple version of this is to copy objects before they are modified and to replace the original object with the replica if the operation fails.

Another topic to consider is the current practice of simplifying modelling operations by creating simple volumetric operations and combining these with the original object using Boolean operations. For example, instead of having a sweep face algorithm that adjusts surrounding faces, the sweep algorithm always sweeps a lamina to create a solid, and then this is added to or subtracted from the original object. This certainly solves some problems, such as the problem of self-intersecting objects, but has the disadvantage that information may be lost. Another approach is to use the more complicated algorithms

and then use a self-intersection operation to check and resolve such problems. This latter approach has the advantage that the self-intersection operation can also be applied to the results of operations that are not amenable to the compound approach using Boolean operations. The use of the compound operations allows the creation of operation patterns. The volumes created in the first step are simply copied and transformed before being combined using Boolean operations. This use, though, is less robust than, say, copying the input parameters and patterning these, so that the operation is applied anew under the transformed parameter set. In any case, the algorithms described here are in the more complicated form because the simpler form is a subset of it.

Appendix G contains information about known problems and implementation notes about the algorithms.

6.1 Boolean operations

The following Boolean evaluation uses basic Euler operators to modify the models so that during the process, the object remains topologically valid, as far as possible by integrating the results into the model. The algorithm described in sections 6.1.1 to 6.1.4 combines two single objects, either by subtracting, adding, or AND-ing them. Section 6.1.5 describes Boolean operations on assemblies, how the assemblies are traversed, and how the single objects therein are combined using the basic algorithm.

The basic algorithm can be divided into two main parts: finding the interactions between objects and sewing the objects into new shells. It relies on two main conditions:

1. That there are no completely new boundary portions
2. That the object is manifold

The condition that there are no completely new boundary portions is necessary because the algorithm builds the result object or objects from portions of the boundaries of the two original objects. There are no completely new parts, except for new edges where part of the boundary of one object cuts through the boundary of the other, the Boolean interaction elements. Once the edges where the objects cut each other have been inserted then the boundary portions delimited by these edges, suitably oriented or reoriented, can be sewn together to form the completed result.

The condition that the objects are manifold is included because there is some ambiguity connected with non-manifold modelling that introduces an element of uncertainty into the modelling algorithms, as described in chapter 5. With some extensions, the algorithm described here could be made to work for non-manifold models, but it is not possible to be definite about the results in all cases. For example, an infinitely thin portion of the boundary,

represented by two back-to-back faces, could either be retained or deleted. If two objects that just touch along an edge are added, then the edge could become a non-manifold edge or the objects could be left separate. Similarly with objects touching at a vertex.

6.1.1 Finding the Boolean interactions

As has been described already, boundary representation (B-rep) models are described in terms of their outer 'skin'. To find the interactions between the objects, each face in one object is compared with every face in the other object. The surfaces of suitable faces are intersected. If they produce a result, usually a curve, this is then intersected back with the faces to find the curve sections that lie in the faces. These curve sections are then compared to find common sections that are added to the faces as parts of the complete Boolean interaction; (figures 6.1a-d). This is done using the utility to intersect a curve with a face, which is described in section 3.2.5.

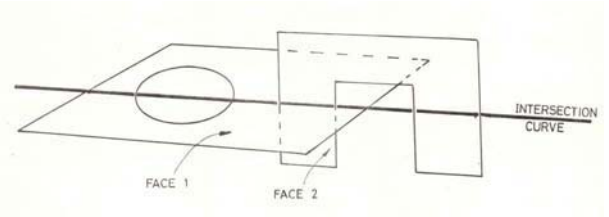
Comparing each face pair in this way is a time-consuming business. There are methods of improving the efficiency of this process, the so-called global tests. The global tests use simple measures of spatial occupancy to perform simple checks to exclude expensive tests between faces that 'obviously' do not match.

The tests are performed hierarchically and depend on where the spatial occupancy data are stored. It is certainly useful to have it in the single object entities (wireframe-, sheet-, and volume-objects), in shells, facegroups, faces, and possibly in edges. Assuming this, the test compares spatial occupancy at the object level, then the shell level, the facegroup level, and finally the face level before performing exact face-face comparisons. If any of these indicates that the compared entities do not occupy the same general portion of space, there is no need to proceed further. There are two common spatial occupancy measures, min-max boxes and spheres (bubbles), both of which are easy to compare and relatively easy to generate.

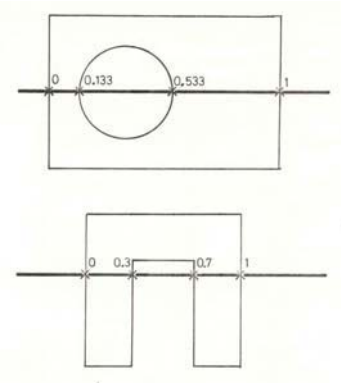
The bubble test works reasonably well for long, thin objects, but for more compact objects, or for badly structured objects, the bubbles do not exclude many faces, and so do not improve efficiency markedly. The orthogonal face-boxes are not uniform figures and so can provide a more exact fit to the face, although there are still examples where the facebox is not a good approximation. Obviously there are overheads in the creation and maintenance of these spatial occupancy descriptors, as well as in the comparison process, but they do increase the speed of the Boolean operations.

When intersecting the surfaces of two faces, in the exact face-face comparison, there are several possible cases that may occur.

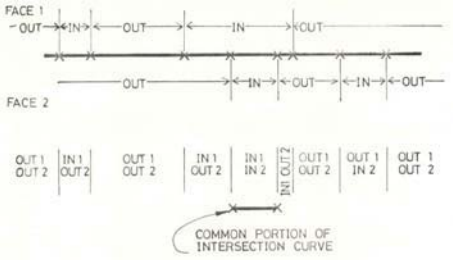
If the surfaces only touch at a point (figure 6.2), which lies inside or on the boundary of both faces, then the action taken is unclear. In the case of subtraction, the result becomes non-manifold if the object being subtracted lies inside the other, but there is no problem if the object being subtracted



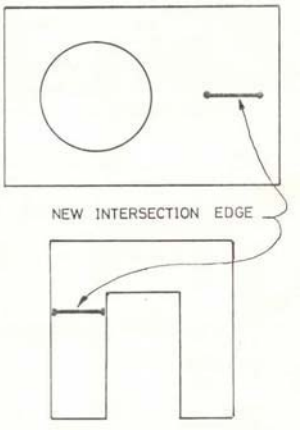
(a)



(b)



(c)



(d)

Figure 6.1: Finding common curve intervals

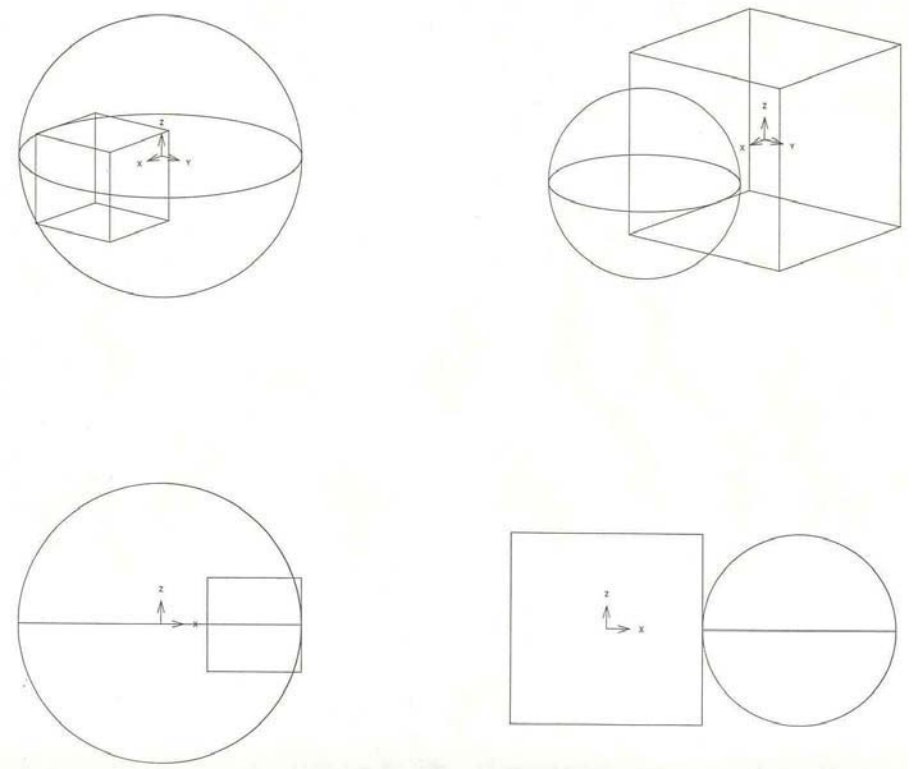


Figure 6.2: Surfaces touching only at a point

lies outside the other. For addition there is a problem if the objects touch externally at a point, but not if one is inside the other. When AND-ing the objects together, there is no problem about the result, but the point might cause problems if chosen to determine if one object is completely contained in the other.

If the surfaces coincide (figure 6.3), then the limits of interaction will be defined by the interactions between each face and the faces surrounding the other face, unless they, too, lie in the same surface as the current faces. The most complete way to handle the interactions is to imprint each face on the other to produce a set of completely matching part faces that can be joined later. Sometimes this is unnecessary because the matching part faces will lie inside the result object, and the interactions between the faces bounding the coincident faces will produce the necessary interaction elements for sewing up the objects. An example of where this is not enough, because the objects have several adjacent faces lying in the same surfaces, are spheres. If one sphere is subtracted from another, say, then it is necessary to produce the interactions using the curves bounding the faces, rather than the surfaces. These form

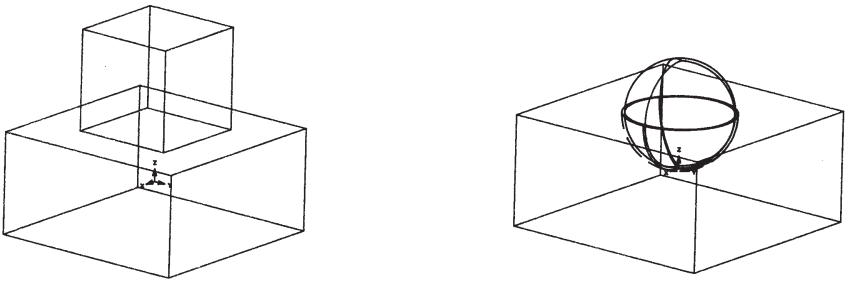


Figure 6.3: Coincident surfaces

what Mäntylä has termed the "A ON B", "B ON A" part of the objects.

As mentioned, where the surfaces intersect to produce a curve or set of curves, each curve is intersected with the two interacting faces to produce segments of the intersection boundary, which are then compared to produce common curve intervals. When intersecting the curves back with the faces, several cases have to be taken into consideration. For convenience, the BUILD terminology is used here. The face intersection point types are as follows:

- ENTERS – where an intersection curve enters a face.
- LEAVES – where the curve leaves a face.
- INOSCUL – where the curve is inside a face and touches an edge without leaving the face.
- OUTOSCUL – as for INOSCUL, except that the curve is outside the face and touches an edge in the face without entering.
- WIRE – the curve cuts a wire edge (edge with the same loop on both sides) in the face.
- ONEDGE – where the curve coincides with an edge bounding the face.

The classification of these crossing points allows the algorithm to determine when curve segments are inside both faces. Obviously also, unless there is, say, a tolerance problem, the next point type in a sequence is limited by the last event. So, after an ENTERS-type point, the next point will be of type LEAVES, INOSCUL, WIRE, or ONEDGE, not an ENTERS or OUTOSCUL type if the objects are correct. After a LEAVES-type point, the next point should be an ENTERS, OUTOSCUL, or ONEDGE, not a LEAVES, INOSCUL, or WIRE point. (The latter would imply dangling wires outside the face boundary.) After INOSCUL or WIRE points, the next result should be INOSCUL, WIRE, LEAVES, or ONEDGE, not ENTERS or OUTOSCUL (as for ENTERS). An ONEDGE result can be followed by anything. This

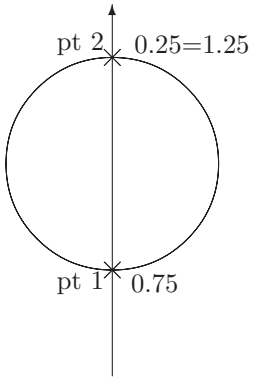


Figure 6.4: Intersection of line and closed curve

can help in classifying intersection sequences for objects where an intersection curve crosses two bounding edges at approximately the same point. In this case, if the curve is inside the face, then the order of the coincident points will be (1) LEAVES, (2) ENTERS, or if the curve is outside the face, then the sequence is (1) ENTERS, (2) LEAVES.

Special problems occur with closed intersection curves. Here, the parametrisation has to be performed with some care. For example, the intersection point where a curve segment enters a face may be parametrised with a value higher than the point where the curve leaves the face. This is a wrap-around effect, as illustrated in figure 6.4. The first intersection point has parameter value at 0.75, the second at 0.25, but the second intersection point should really have a parameter value of 1.25. Again, the intersection point classification can be used to help resolve problems. The sequence should start with a point where the curve ENTERS, if there are any such points.

ENTERS and LEAVES intersection point types are fairly straightforward to handle. They indicate the start or end of a new edge segment. As with all results, except OUTOSCUL points and ONEDGE results, unless the intersection curve cuts through an existing vertex an edge has to be split to produce a start or end vertex for an intersection segment.

Each INOSCUL or WIRE contact with the face boundary should produce a pair of intersection points. One member of the pair serves as the endpoint of a curve segment, and the other as the start point of the next segment. If there are only INOSCUL and/or wire points present, then the curve lies entirely within the face and the ordered sequence of intersection points should start with one of the pair, and finish with the other. The edge touched to produce an INOSCUL, or the wire edge, is split, if necessary, and each of the pair produces one new end vertex of a new intersection edge segment.

ONEDGE results are complex in that the intersection result can enter, leave, touch a face boundary on the inside, or touch it from the outside; thus,

they can imply ENTERS, LEAVES, INOSCUL, or OUTOSCUL cases. Here, the endpoints of the edge mark where any possible intersection segment starts or ends, and their parameter positions can be used for ordering. An extra test is necessary to see if the intersection curve is outside or inside the face before and after the ONEDGE result. The ordered set of results can help, or if, say, another ONEDGE result precedes or follows the current one, a point in face test could be used. A special case, relevant only for free-form curves, is where the intersection curve coincides only partially with an edge. Here the simplest solution would seem to be to split the edge twice, at the start and end of the coincident section, and treat the coincident edge portion as an ONEDGE result.

OUTOSCUL points indicate the intersection curve is outside the face. These might, although not necessarily, indicate potential error conditions where a non-manifold result might occur (see comments from earlier). However, these are ignored by the algorithm. If the result is part of a valid intersection, then an appropriate vertex will be, or has been, added by a different face-face comparison.

The main difference between the algorithm presented, which is in fact partly implemented in the GPM modeller, and that in BUILD is that the new intersections are added to the model directly during the face-face comparisons, not hidden as with BUILD. The new edges are added as wire edges, each of which consists of two new vertices, an edge, and a new inner loop. New vertices were also introduced into the boundary of the faces being compared where the new intersection curve segment cut or touched edges away from existing vertices. This means that the complexity of the model increases during the Boolean operations, which is not altogether desirable, but means that the model is kept correct. Having the new edges in the model does have one advantage in that intersections producing self-intersecting curve sets, such as where two perpendicular cylinders of equal radius are being intersected, no special checks are needed. Where the crossing point lies inside a face, the first intersection curve will produce a single new edge, the second intersection curve will also be intersected with this new edge, and creating two new edges and the edge from the first intersection is split at the appropriate place or places.

The new edges are added to the model as wire edges to avoid creating new faces while traversing the object face structure. However, this means that there has to be an extra, time-consuming step involving sewing up the new edges into the original object structure once the face-face comparison is complete. This can be speeded up by maintaining special additional structures to preserve correspondences between matching vertices to avoid searching.

6.1.2 Special cases in intersection boundary creation

One special case is where no intersection points are found, as illustrated in figure 6.5. Theoretically the curve lies entirely outside the face (figure 6.5

middle, left), or entirely inside the face (figure 6.5 middle, right), and a point-in-face test can be used to establish this. If the curve lies outside one or both faces, then there is no interaction, no further work is necessary, and the next pair of faces can be examined.

When the curve is contained entirely within both faces, then it is necessary to divide it artificially according to the rules chosen for the modeller. If 360 degree edges are allowed, then one division is necessary. If 180 degree edges are allowed, then two divisions are necessary. The number of divisions depends on the basic design of the modeller.

When a closed curve lies partly in one or both faces, there will be some points where the curve has to be broken, but it may still be necessary to perform an analysis on the intervals to see that they do not extend through too large an arc (figure 6.5 bottom).

Where two surfaces intersect at a single point, then that point can be tested to see whether it lies inside both faces. As mentioned, the existence of a single point contact might or might not indicate a problem. The implementation of the algorithm ignores these cases, even though a non-manifold result might occur. An alternative is to create a degenerate ring of edges in a circle of zero radius, and to treat this as a complete intersection boundary, using a post-processing step to remove it, if necessary. If the point lies on the boundary of one or both edges, then it might or might not be part of a valid intersection boundary, depending on the Boolean operation and whether the edge or vertex is convex or concave.

6.1.3 Creating the intersection boundary

Once all face pairs have been compared, the intersection boundary should be complete, but it has to be sewn into the rest of the object, and pre-processed, before joining shells.

Sewing the intersection boundary into each of the objects is done by working through the object face by face, merging the loops of intersection edges with the rest of the face boundary. The boundary creation process ensures that where there is an interaction between two edges, there is a pair of vertices at the common point, one for each edge. Sewing up the boundary consists of identifying the coincident vertices and merging them (see Appendix F, section F.15). This process creates the complete intersection boundaries that are joined to produce the result object. Note that this process changes the structure being traversed. If only one corresponding vertex is found, then only the current intersection loop is destroyed. However, if both end vertices of the edge in the loop match other vertices, then either another loop is removed, or a new face is created. Because of these changes, some care is needed in the traversal techniques.

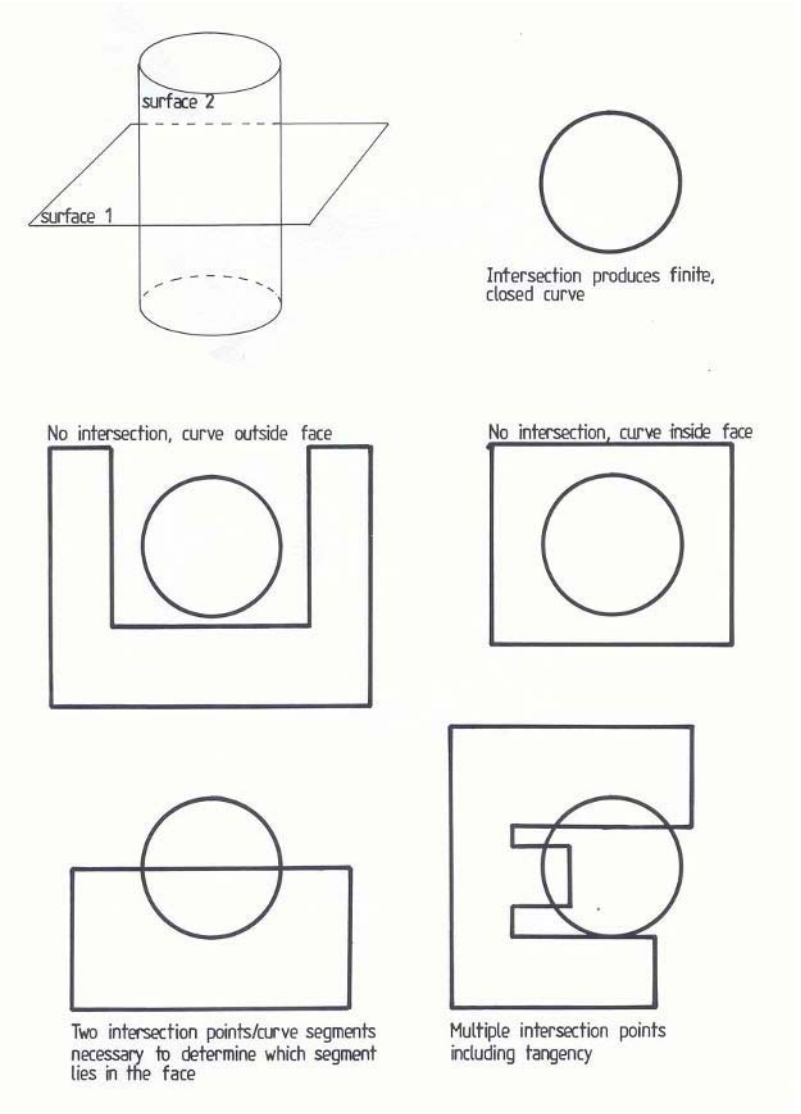


Figure 6.5: Special case where intersection curve does not cut face boundary

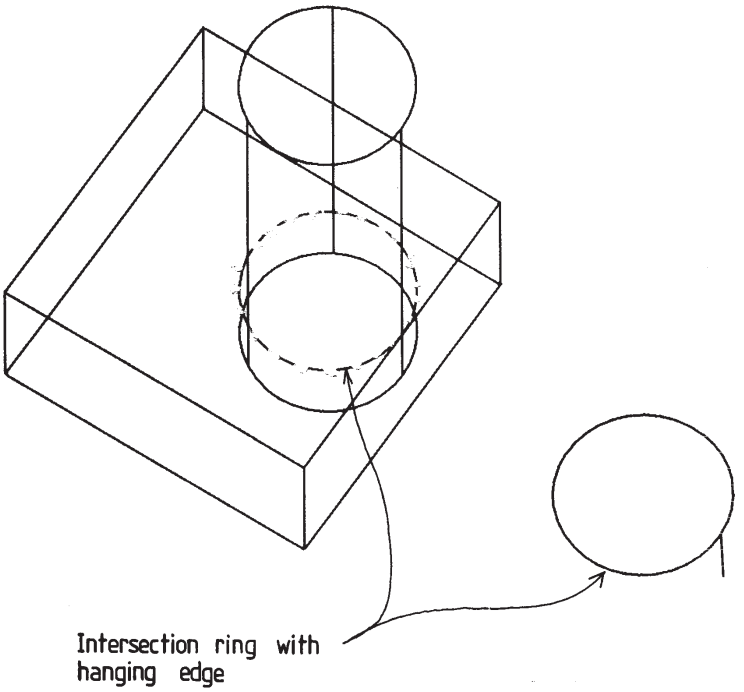


Figure 6.6: Intersection boundary with hanging edge

6.1.4 Merging the objects (creating new shells)

This phase has four main components:

1. Preprocessing the intersection edges to remove hanging edges
2. Negating one object, if subtraction is taking place
3. Merging the objects along the closed intersection boundaries
4. Separating the merged result into shells.

For each object, the edges created during the face–face comparisons are remembered and, as a first step, these edges are pre-processed to eliminate hanging edges. These hanging edges can arise from intersections such as that illustrated in figure 6.6. Here, one edge, from the intersection between the cylindrical surface and a planar face, is only connected at one end to a closed ring of intersection edges, and so it can be deleted. There will be two of these 'dangling' edges, one in each object, and both edges must be deleted. The pre-processing analysis checks each interaction boundary edge to see whether both end vertices of the edge have other intersection edges connected to the vertex. If not, the edge is removed, or marked as a non-intersection edge.

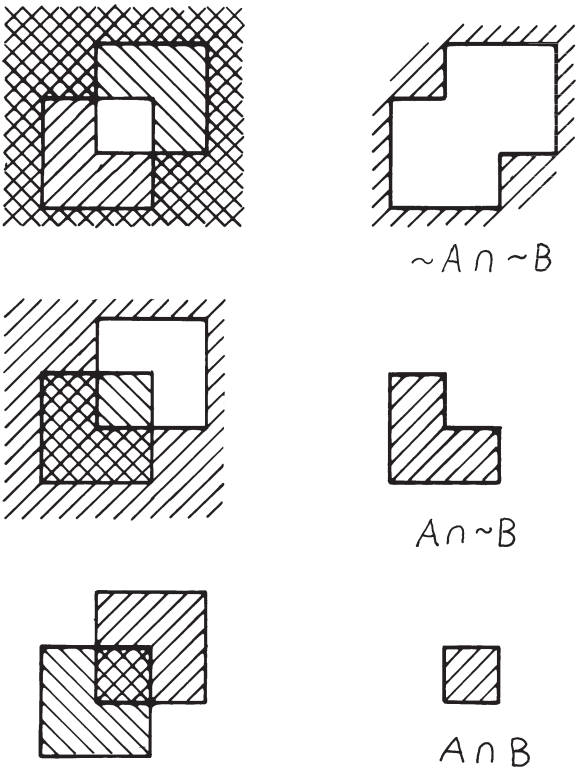


Figure 6.7: Comparison of Boolean results

Once only closed rings of intersection edges have been established, the objects are negated, if necessary, and joined. The negation step is strictly only necessary when one object is being subtracted from the other, it being more efficient to treat the joining processes for the different Boolean operations separately. However, Braid [10] identified that the set of Boolean operations (ADD, SUBTRACT, and INTERSECT) could be implemented using only a single operation, INTERSECT, on the basis that:

$$A + B = \sim (\sim A \& \sim B)$$
$$A - B = (A \& \sim B)$$

as illustrated in figure 6.7, with A denoting the main object and B the secondary object. This means that during the joining step, the result object is always the intersection of the two original objects, and so it uses a uniform joining method.

In the joining process, pairs of edges are joined. The initial part of the Boolean operations consists of creating matching intersection seams where the

objects cut or touch one another. The joining process unites the objects along these seams by joining matching edge pairs.

Mäntylä [82] has an elegant classification of the object parts as, for two objects A and B, as A IN B, A OUT B, A ON B, B IN A, B OUT A, and B ON A. These denote the part of A inside B, the part of A outside B, the part of A coincident with B, the part of B inside A, the art of B outside A, and the part of B coincident with A. For an addition, A+B, the result would be the union of A OUT B, A ON B, and B OUT A. For a subtraction, A-B, the union of A OUT B, and the negative of B IN A. For an intersection of A and B, the union of A IN B, A ON B, and B IN A. This is illustrated for two dimensions in figure 6.8.

Once the two objects have been joined, it is necessary to separate them into results and non-results. After the end of the joining process, all topology and geometry from the two objects is contained in a single object. During the object joining process, one edge is classified as 'result' and stored in the result group, and the other is classified as 'non-result' and stored in a different group. When pulling the objects apart, the process starts with an edge from the 'non-result' group and puts it and all connected edges into an object shell. If any edge in the 'non-result' group is still in the original object, then that, and connected topology and geometry, is moved to a different shell until all 'non-result' edges are in separate shells. Once all 'non-result' parts have been separated from the 'result' parts, the same process is carried out to separate the distinct result shells into new objects.

To move connected topology into a new shell, the utility for traversing all connected entities in a shell is used. When moving the non-result parts, the start point is an edge from the 'non-result' group. When checking the 'result' group, the procedure is slightly different. First, all edges in the result object are marked. Then, all edges in the same shell as the first edge in the 'result' group are unmarked using the "all edges in shell" procedure. Any marked edges still in the result list belong to a different shell and can be moved, and unmarked, again using the "all connected entities in shell" procedure. Once no edges in the 'result' list are found to be marked, the process of separating the edges into shells is complete.

The next phase involves checking the complete shells to see whether they form 'cavities' inside other shells. As the Boolean operation should have established the geometrical independence of the shells, it should only be necessary to use a "point-in-body" test to be able to check this. However, it should be noted that places where one shell just touches another without breaking through can cause problems, as mentioned earlier. Here, the shells are not independent, but no geometric intersection boundary segment results.

The "point-in-body" check should also be used if no intersections are found, providing the simple global checks described in section 6.1.1 indicate that there might be some interaction. If no intersections are found, then one object might be completely contained inside the other. The desired results for the operations when one object is contained by the other is as follows:

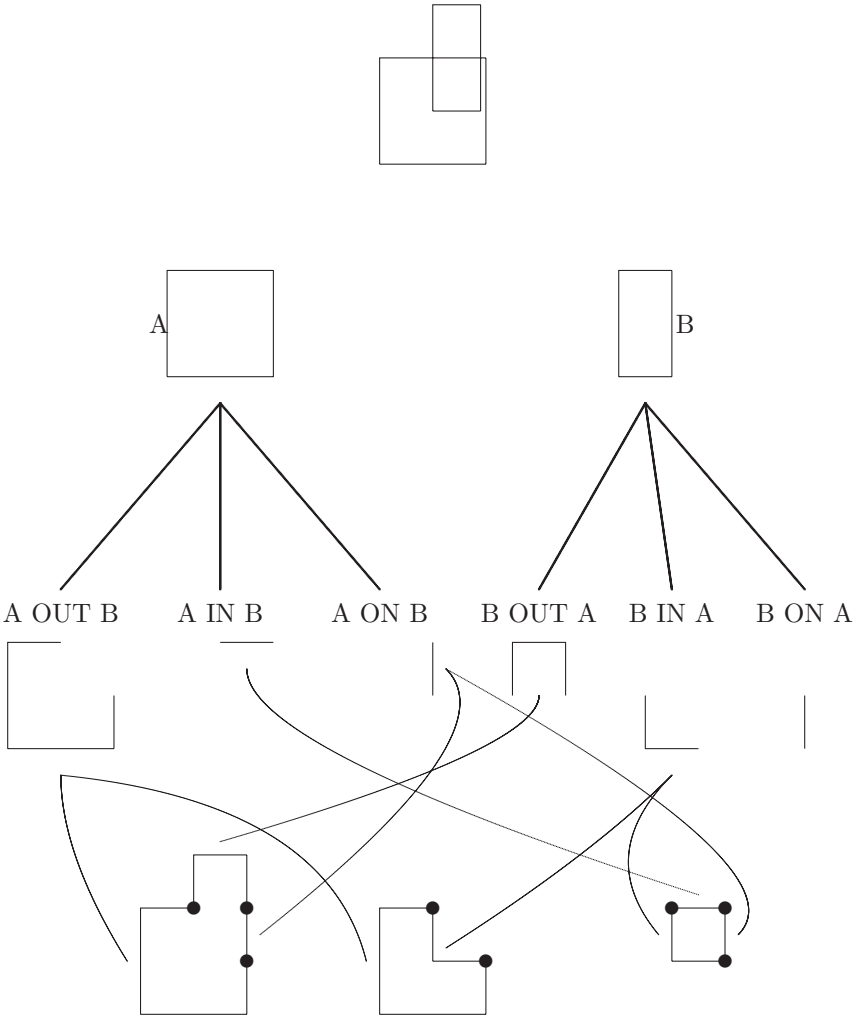


Figure 6.8: Decomposition and recomposition of Boolean results

	ADD	SUBTRACT	INTERSECT
A in B	B	nothing	A
B in A	A	cavity B inside A	B

There is no real reason for preserving the 'non-results' in new shells, as outlined above, in the standard Boolean operations. However, they provide a sort of 'dual' to the results, which may provide useful information. As they are a by-product of the joining process, it is easy to choose whether to preserve them or delete them according to user desire. If Braid's uniform method is used, with the extra negation steps, the results and non-results formed dual pairs; thus:

	Result	Non-result
$A+B$	$A+B$	$\sim (A\&B)$
$A-B$	$A-B$	$\sim (B-A)$
$A\&B$	$A\&B$	$A+B$

(The first 'non-result' entry in the table is negated because addition was handled as $\sim (\sim A\& \sim B)$.)

Handling the Boolean operations separately mean that these negation steps disappear leaving the result table as follows:

	Result	Non-result
$A+B$	$A+B$	$\sim A\&B$
$A-B$	$A-B$	$\sim (B-A)$
$A\&B$	$A\&B$	$A+B$

6.1.5 Boolean operations with assemblies

Sometimes it is desirable to apply Boolean operations to assemblies, for example, where two assemblies have to be compared to see whether they interfere.

As before, the two objects or assemblies to which the Boolean operators are to be applied can be called A and B, and the operations applied are A + B, A - B, or A & B (ADD, SUBTRACT, and INTERSECT, respectively). Assembly B is traversed first until a single object, say b1, is found, and then assembly A traversed until another single object, say a1, is found. Once two single objects have been found, they are combined using a single object combination method such as that described above. However, there are differences according to the operation being performed. If an ADD operation is being applied, then b1 acts as an 'accumulator' for any results, until the last single object from assembly A

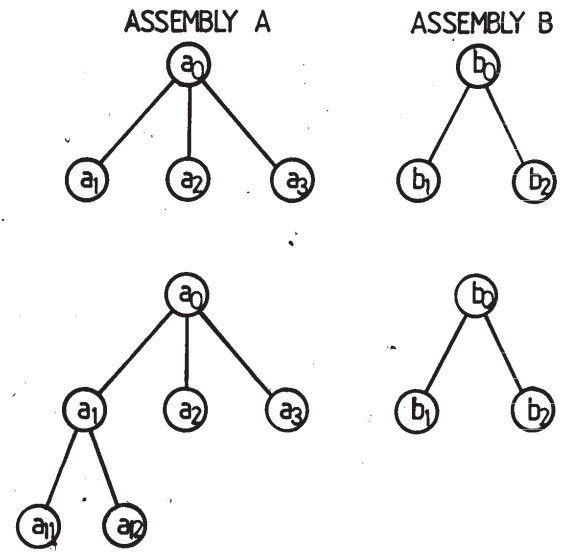


Figure 6.9: Example assembly structures

has been found, when it is replaced in assembly A. If a SUBTRACT operation is being performed, then the object or assembly to be subtracted is copied, and the copy is subtracted unless the object from assembly A is the last in the assembly. In an AND operation, the object from assembly A is also copied, and the copy AND-ed with the object from assembly B, unless the object from assembly B is the last object in the assembly; in which case, the original object is AND-ed. The union of the AND-ed results is returned as the result.

Take the assemblies shown in figure 6.9a as an example.

For ADDITION, A + B, the first object in assembly B, b1, is ADDed to assembly A. In A the first object, a1, is combined with b1. If there is no result, then a1 is left alone. If the two objects do interact, then a1 is removed from A and b1 becomes the combined result of a1 and b1. The next step combines b1 (either the original or the new combined result) with a2. Again, if there is no result, a2 is left in A; otherwise b1 becomes the combined result. The final step is to compare b1 and a3. As before, if there is no result, then a3 is left in A; otherwise, b1 becomes the combined result. As there are no more objects in A, b1 is inserted as the first object in A, and the next object, b2, is added to the new assembly A. Object b2 is first compared with b1. If there is a result, then b2 becomes the combined result and b1 is removed from A; otherwise, b1 is left alone. The process proceeds as before, b2 is compared with any of a1, a2, and a3, which did not interact with b1 in the first comparison. When the traversal is complete, b2 is inserted into A.

For SUBTRACTION, A-B, the first object in assembly B, b1, is subtracted

from assembly A. When the first single object in assembly A, a1, is found, it is not the last object in the assembly, so a copy of b1 is subtracted from a1. The result, either a single object or an assembly, is left in place of a1. Similarly for the second object, a2. A copy of b1 is subtracted from a2, and the result is inserted into A in place of a2. Finally, b1 is subtracted from a3. As a3 is the last object in the assembly, the original object b1 is subtracted from a3 rather than a copy. The same process is carried out for the second object in assembly B, b2. A copy of b2 is first subtracted from the result of a1-b1. Then, a copy of b2 is subtracted from the result of a2-b1. Finally, the original b2 is subtracted from the result of a3-b1.

Finally, for AND-ing, A & B, the process is a little more complicated because potentially parts of both assemblies may be copied. First, assembly A is compared with b1. As b1 is not the final object in assembly B, a copy of the whole assembly is AND-ed with b1, and the results are preserved in the copy. To AND the copied assembly A', with elements a1', a2', and a3', say, first b1 is compared with a1'. As a1' is not the last object in A', a1' is AND-ed with a copy of b1, and the results (if any) are left in A' in place of a1'. Next a2' is compared with b1. As a2' is not the last object in A', a2' is AND-ed with a new copy of b1, and, again, any results are left in place of a2'. Finally, a3' is AND-ed with the original b1, and any results are left in place of a3'. Once A' has been AND-ed with b1, the original assembly A is compared with b2. As b2 is the last object in assembly B, the original assembly A is compared with b2. A copy of b2 is AND-ed with a1, another copy AND-ed with a2, and finally the original b2 AND-ed with a3.

The process also extends to multi-level assemblies by applying the Boolean operations recursively to each level of the assembly. In figure 6.9b, object a1 from figure 6.9a has been replaced with an assembly of two objects, a11 and a12. For addition, the process is the same as though a11 and a12 were on the same level of the assembly as a2 and a3. For subtraction, when the copy of b1, b1', say, is subtracted from a1, the copy of b1 is passed down into the sub-assembly. A copy of b1' is subtracted from a11, and b1' is itself subtracted from a12. For the AND operation, the process is virtually the same as when AND-ing the previous structure, except that b1 is copied an extra time, and the copy AND-ed with a11.

6.1.6 Local Boolean operations

Finally, in this section, it is desirable to describe what might be termed 'local Boolean operations'. The Boolean operations described in the previous sections have dealt with the interactions between complete objects, but the same basic techniques can be used to compare limited portions of objects. The effect of this is illustrated in figure 6.10, which shows the effects of adding a sort of closed tube to an object with both global and local Booleans.

This technique can be very useful for producing special combinations of objects and adding partial models to volume models as possibly required for

adding features (see Chapter 9, section 9.3).

The basic techniques are the same; what differs is the model traversal techniques to produce the interaction elements. First, the operation can be defined by specifying two faces that definitely interact and by using the resulting interaction elements to define the next pair of faces to be compared instead of comparing every pair of faces as before. This kind of progressive creation of the Boolean interaction is used elsewhere and is described later in the descriptions of the bending and planar sectioning algorithms. The method here is slightly more general, but it is very similar in effect.

Having created the first Boolean interaction or interaction edges, the ends of the edge are used to determine the next pair of faces to be compared. Ignoring places where an interaction edge just touches a face boundary, the end of an interaction edge will be where it cuts the boundary edge of one or both faces. The face on the opposite side of the cut edge from the original face then replaces it in the pair of faces being compared, and the process is repeated until the interaction boundary reaches the first edge again.

Once the local interaction boundary or boundaries are complete, the objects are merged along these in the manner described previously to produce the final object.

6.2 Sweeping/Swinging

Here, 'sweeping', means extruding a body or a part of a body in a straight line. 'Swinging' is used to denote where the body or part of the body is extruded in a circular arc or complete circle. As they are strongly related, the geometry and special cases differing slightly, the term 'sweeping' is used here to cover both straight sweeping and circular sweeping or swinging. However, when swinging a flat shape in a complete circle, there is a final step to join matching faces to create the final solid.

Note, also, that there are other variants of sweeping. One is to add a vector offset for circular sweeping to be able to create a spiral shape. Another is to sweep a shape along a curve. Sweeping along a curve requires curve offsetting and lofting type techniques to create the surfaces, as described in chapter 13.

In its simplest form, sweeping (and similarly for swinging) involves extending each vertex of the face and adding edges between neighbouring extended vertices. This is illustrated in figure 6.11. The face to be swept is shown in figure 6.11, top. A start edge is selected, which is the edge referred to by the loop of the face, here edge 1. An edge and vertex is added with the vertex clockwise from the edge, here vertex 1, as base (figure 6.11, row 2 left). Then, another edge and vertex are added to the vertex at the other end of the edge, vertex 2 (figure 6.11, row 2 middle), and a closing edge across the top (figure 6.11, row 2 right). Then the sweep operation works around the face, here counter-clockwise, adding a new edge and face to each edge around the face

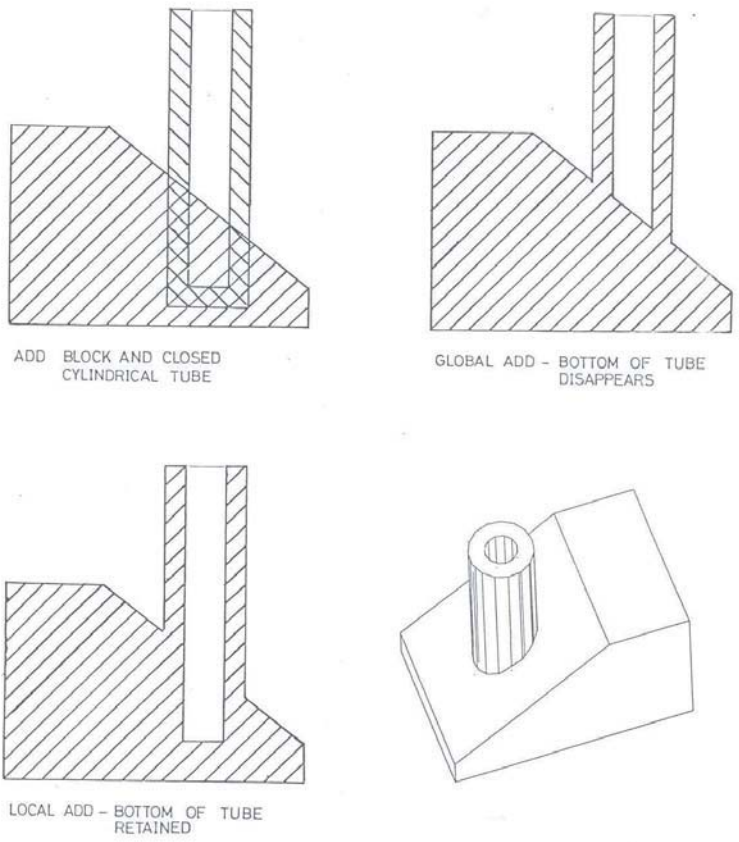


Figure 6.10: Illustration of global and local Boolean operations

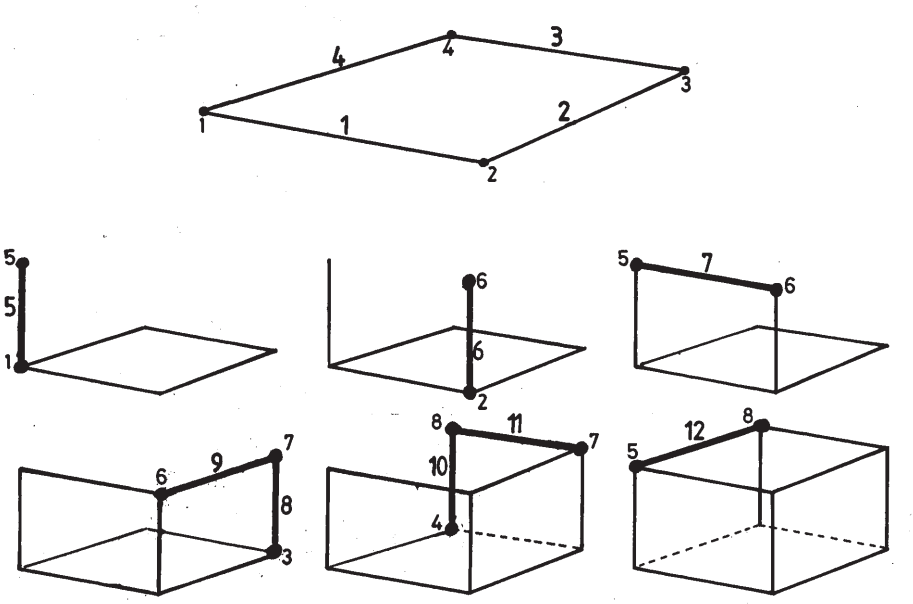


Figure 6.11: Sweeping a four-sided face

(edge 2, figure 6.11, row 2 left, and edge 3, figure 6.11, row 2 middle) until the last edge is reached. The final step is to add an edge between the last extension vertex (vertex 8) and the first extension vertex (vertex 5 figure 6.11, bottom right). This can be termed 'vertex-based' sweeping, because it ignores the edges to be swept.

This sweep algorithm can basically be summed up as a sequence of Euler operators:

MEV – First extension vertex

MEV, MFE – First extension face

MEV, MFE – Second extension face

MEV, MFE – Third extension face MFE

and so on, for each edge around the face until the final edge is found.

MFE – Final extension face

6.2.1 Improved sweep/swing algorithm

The above-mentioned algorithm is sufficient for sweeping laminae, but it does not cope with the case where an edge should extend or contract a neighbouring face. The difference between the earlier, vertex-based sweep algorithm and the improved one is that each edge in the face is examined to see whether it can be moved, as well as examining the end vertices; i.e., it is 'edge-based', which means that a larger class of individual faces in volumes can also be swept. The algorithm is still 'stepwise' in the sense that the modifications involve simple Euler operators, but the choice of which to apply and where is more complex.

The sweep algorithm described here sweeps a face. If a flat shape, or lamina, is to be swept, then there is a pre-processing step to discover which face (or set of faces) is to be swept. If the side of the lamina that is to be extended is divided into several individual faces then these are all swept separately; the face sweep algorithm handling face adjacencies.

The face being swept is first isolated in its own facegroup. This is partly done to ensure that the surface referred to by the face (which might be in a facegroup) is only referred to from one place, and partly to allow some sort of meaning to be preserved in the facegroup structure (although it is not clear whether this is a good idea). A transformation matrix is established that is used to modify the geometry (vertex positions, curves, and the surface of the face being swept).

The algorithm sweeps each loop in the face separately, proceeding around each loop, in a clockwise direction, for example, sweeping each edge separately. The same transformation, established in the first step, is valid for each loop.

If the loop is associated with an isolated vertex in the face, then the position of the vertex is changed (using the transformation matrix), and no new edges are established, unless it is the only vertex in the object.

If the loop is a ring of edges, then the edge referred to by the loop is taken as the first edge to be swept. The edge counter-clockwise around the loop from this edge is found, because this will be the last edge swept and is treated as a special case as well as being the termination point of the sweep. If the edge referred to by the loop is a wire edge, or the counter-clockwise edge from this edge is a wire edge, then care must be taken. There are four options to avoid this problem as follows:

1. To scan around the loop looking for a non-wire starting edge.

2. To pre-process the face and remove wire edges altogether, possibly creating internal loops (boundaries) inside the face.

3. To put all edges in the loop into a separate list before modifying the structure and to process the list.

4. To adapt the algorithm to handle wire edges.

The final option is the most general, so the algorithm described here will assume that wire edges may exist in the face, and that the start edge and the adjacent edge may be wires. Sweeping faces composed only of wire edges is a special case, and it is discussed in Appendix G, section C.2. Whichever of the above options is chosen is a secondary consideration; the algorithm works in terms of sweeping/swinging an individual edge.

When allowing for wire edges, it is necessary to distinguish between the first time they are encountered and the second time, and the trailing vertex used for orientation may be used for this. Another alternative is to use the marker bits of the edge to distinguish between these two events. Marker bits are also useful for another purpose, to distinguish between the original edges in the face and the new edges added when sweeping a loop. Normally the new side edges are hidden because the next clockwise edge around the loop is found before the clockwise swept vertex is obtained. However, when wire edges are present in the face, the side edges added when sweeping them for the first time will be encountered when the traversal comes to the wire edge a second time. For this reason, all edges in the loop should be marked before the sweep and then unmarked as they are swept. If an edge is found that is not marked, then it should not be swept. In fact, if, when the next edge clockwise around the loop is not marked, it is necessary to get the next marked edge (unless the last edge to be swept is found), in case the unmarked edges are side edges that will disappear. Wire edges will thus be swept the first time they are encountered, and the side edges will be tidied when the adjacent edges are swept. Figure 6.12 illustrates this process. The original face is shown in figure 6.12, top left. The first edge to be swept is a wire edge, creating extra wire edges (figure 6.12, top right). When the last edge in the inner triangular edge sequence is to be swept (figure 6.12, bottom left), one end vertex is attached to the wire edge created when sweeping the first edge. This disappears in the sweep (because the edge is movable), as illustrated in figure 6.12, bottom right. The other end is tidied in a similar manner.

So, having marked the edges in the loop, the next step is to obtain the last edge to be swept so that the sweep algorithm will know when to terminate. This should normally be the edge counter-clockwise from the first edge being swept. If the first edge is a wire edge, then clockwise and counter-clockwise are ambiguous, so either the start or the end vertices can be chosen to give the edge an arbitrary orientation. Assuming the start vertex of the edge is chosen, the last, or terminator, edge is the edge clockwise around this vertex from the first edge, and the second edge to be swept is the edge counter-clockwise from the first edge around the end vertex of the first edge.

Next, the swept vertex for the vertex lying between the first edge to be swept and the terminator edge is obtained. The process of obtaining a swept vertex will be described in section 6.2.1. This first swept vertex will also be used when sweeping the terminator edge, so a pointer to it is preserved until the complete loop has been swept.

Now the basic sweep edge pattern can be used. Basically, each edge, until

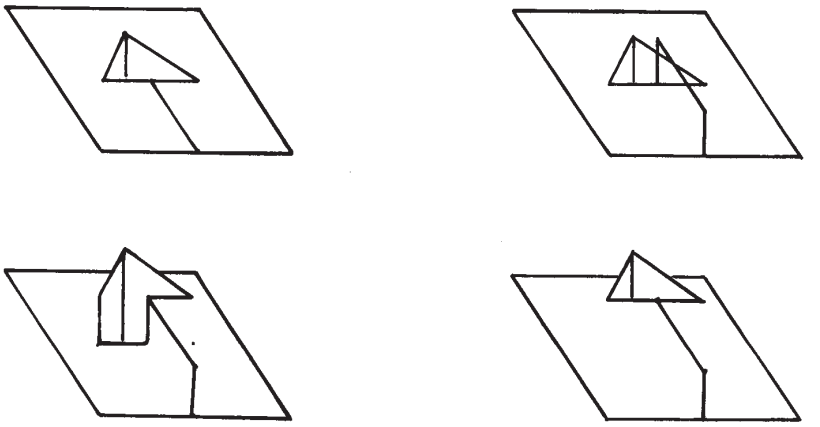


Figure 6.12: Sweeping a wire edge

the last, is swept by producing a swept vertex corresponding to the clockwise vertex, and to join this and the previous swept vertex with an edge. The terminator edge is swept by joining the last swept vertex to the first swept vertex from the loop. Several special cases for sweeping an edge have to be handled, and these are described below.

The algorithm allows faces to be swept up to the level of one or more surrounding faces (figure 6.13, top) or past them (figure 6.13, bottom). Obviously, if a face is swept up to the level of a neighbouring face, then one or more side faces and two or more edges will disappear. If swept past a neighbouring face, a side face will survive, but the normal direction will be reversed.

One further complication, mentioned above, is when a face is swept up to the level of a neighbouring face. Here it may be necessary to delete the side edge; there should already be an edge connecting the new end vertices, and a side face will disappear.

Sweeping an edge

Two basic conditions govern sweeping an edge: 1) whether it is on the axis of rotation for circular sweeping (swinging), and 2) whether it is movable. If the edge is on the axis, then no action need be taken other than setting up the next edge to be swept. If not, then the edge is examined to see whether it is movable, and the new end vertices are established.

The criterion for an edge to be movable is that it 'slides' in the surface of the face opposite (across the edge) from that being swept. For example, take the edge in figure 6.14, lying between faces f1 and f2. If face f1 is being swept,

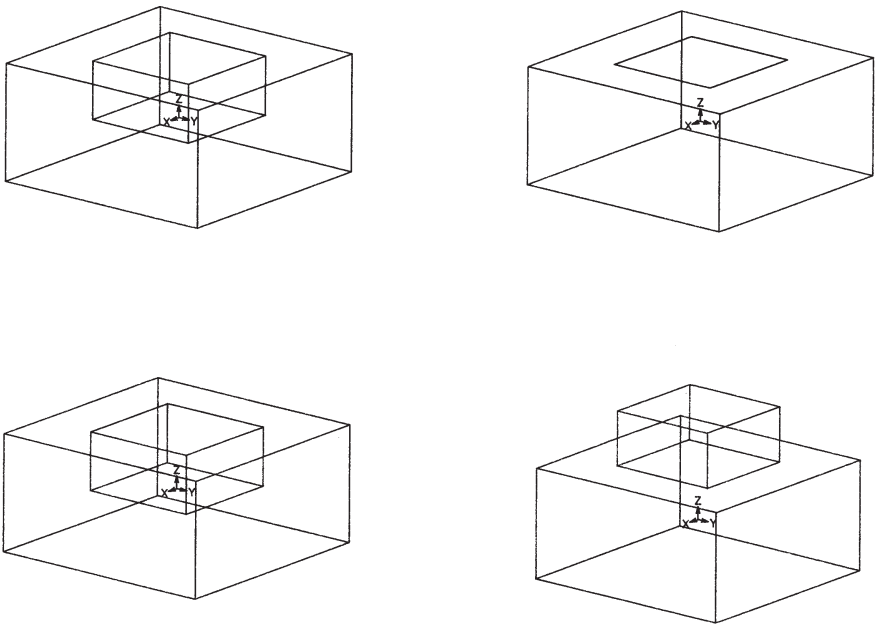


Figure 6.13: Sweeping faces up to or past neighbouring faces

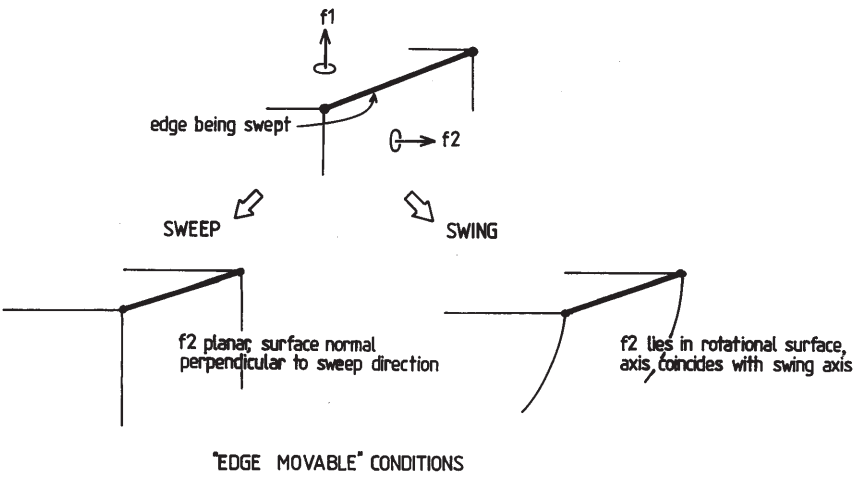


Figure 6.14: Conditions for an edge to be movable

then the edge is movable for a straight sweep if face f2 has surface normal perpendicular to the sweep direction. The edge is movable for a circular sweep if the surface of f2 is a rotational surface with axis coincident with the swing axis, for example, if the surface is cylindrical with cylinder axis on the swing axis. In addition, the edge is movable if the surface of f2 is a plane with normal parallel to the axis. If 'fake' edges (see section 3.1.1 and 3.1.4), then there are additional criteria governing edge mobility. Note that logically wire edges are always movable, whatever their shape, because they are contained in the face being swept.

If the edge is static, a new edge is always added between the two swept vertices, even if they are already joined. This is because if the edge is static, then a new face will be created; even if the edges are already joined (as illustrated in figure 6.15), a new face is needed. The positions for the swept vertices are determined by transforming the positions of the end vertices of the edge. Similarly, the curve of the new edge is obtained by transforming a copy of the curve of the original edge. The surface of the face is more difficult to generate, but this is treated here as a 'black box' function by the sweep algorithm. See section 3.2.4.

If the edge is movable, then it is reattached to the two swept vertices, if it is not already attached to one or both, or if there is not already an edge linking the two vertices. The edge can be attached to both vertices if both were movable, and only their positions were altered. Another possibility is that both vertices lie on the axis, although the edge does not coincide with it. The edge can be attached to one vertex either if the vertex was movable or if it lies on the axis. If the edge is not attached to both then the swept vertices are checked to see whether they are already connected. The connecting edge has also to have a face in common with the edge being swept; otherwise, the swept edge and the existing connecting edge are both needed, and the object becomes non-manifold.

Reattaching the edge can necessitate some rearrangement of the loop pointers of the side edges. If the side edges are not part of the loop being swept, then one (right or left) loop pointer will be changed. If not in the face being swept, the side edges will be in the face on the opposite side of the edge being swept from the face being swept; otherwise, they will not have been considered as candidate side edges. So, if not already in the face being swept, the loop reference to a loop of the face opposite across the edge is replaced by a reference to the loop of the face being swept. If, after reattaching the edge, the former end vertex becomes a spur vertex, then this vertex and the attached spur edge must be deleted.

Obtaining a swept vertex

The end vertices of the edge can also be classified as movable, static or fixed. The criteria for a vertex to be movable are that at most three edges meet at the vertex, the one being swept, which must be movable, and at most two

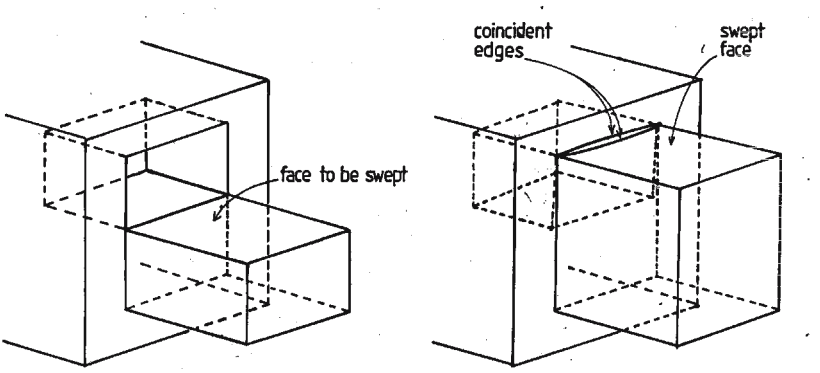


Figure 6.15: Swept vertices already connected

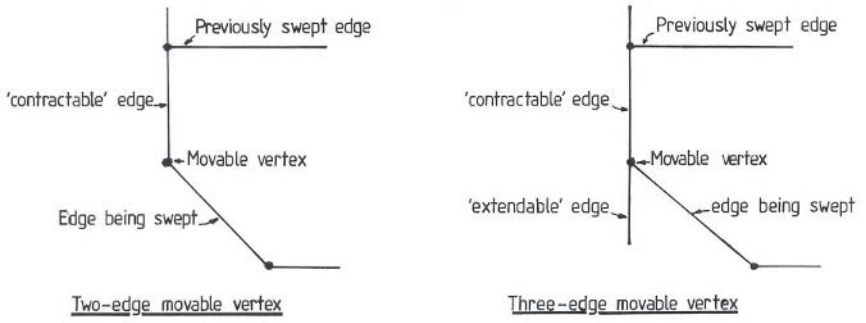


Figure 6.16: Two- and three-edge movable vertices

other edges along the curves of which the vertex can slide. If sweeping in a straight line, the other edge or edges must be straight and parallel to the sweep direction. If sweeping in a circular arc, they must be circular, with the circle centre on the axis, and the normal of the plane of the curve must be parallel with the circular sweep axis (essentially moving with the face being swept and sliding in the surfaces of the other faces which meet at the vertex). Figure 6.16 illustrates two- and three-edge movable vertices. For circular sweeping, there can also be restrictions about the size of the angle subtended by the edge at its centre.

If the vertex is movable, then its position may be changed, altering the curve of the attached edge(s) not being swept if necessary. This is similar to the case of splitting an edge by inserting a vertex (see Appendix F, section F.2).

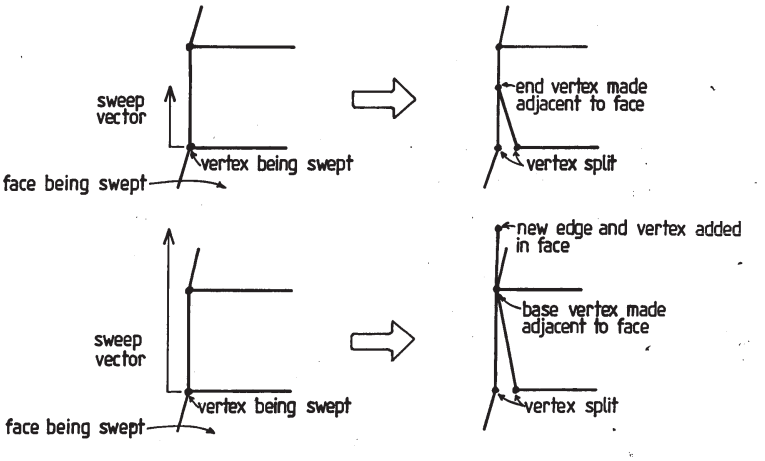


Figure 6.17: Slicing edges to create swept vertices

When the vertex is not allowed to move (because the conditions specified above are not fulfilled), then a new side edge has to be found or created. In the simplest form of sweeping, a new edge and vertex are added, but this is not possible if at least one edge attached to the vertex is movable during the sweep. In this case, there may be edges attached to the vertex that (for straight sweeping) lie in the direction of the sweep or (for circular sweeping) have centre of curvature on the swing axis. However, these edges may not be adjacent to the face being swept, and so some manipulation may be necessary before they can be used. Figure 6.17 illustrates this. If the side edges are longer than needed for the sweep, then they are split by inserting a new vertex at the right position, and then sliced up to this vertex, as on the left-hand side of figure 6.17. If the side edges are not long enough then they are sliced and a new edge and vertex are added to the far end vertex, as illustrated on the right-hand side of figure 6.17.

Finally, if rotational sweeping is being carried out, and one end of the edge is on the axis, then that vertex is returned; no new vertex is needed.

6.2.2 Sweeping along a path

The BUILD system had an operation to sweep a face along a path composed of a sequence of edges. In effect the edges were only straight or circular, so the operation consisted of a series of straight and circular sweeps with parameters determined from the edge geometries. Straight edges give rise to vectors; the centres of the circles and the normals to their planes define the axes for circular sweeps, or swings. See figure 6.18, top left.

One thing to note is that, if the start point of the path does not lie on the

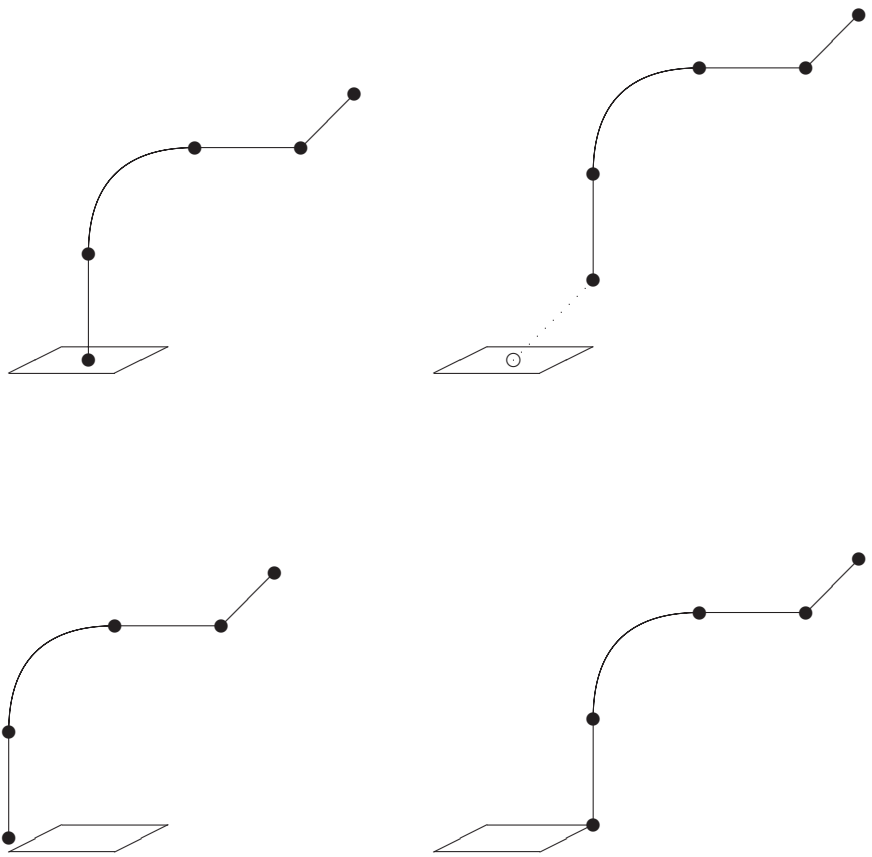


Figure 6.18: Sweeping along a path

face, then the path geometry is relative, not absolute. The difference between the path point on the face and the first point of the path defines an offset vector, as shown in figure 6.18, top right, with the offset vector drawn dotted. This means that the centres of the circles need to be translated by the offset vector and not just taken as absolute. For straight sweeping the path point has little significance, but for circular sweeping, the relative position of the face to the swing axis is important. It is easiest to insist that the path starts at a point on the face. Otherwise the centre of the box surrounding the face could be used as the face path point. Figure 6.18, bottom left, shows a path that may cause the swing operations to fail; figure 6.18, bottom right, shows an arbitrary alternative. Obviously this depends on the path geometry, so it is probably best not to apply an automatic calculation and leave it to a user to get the path right.

Note, also, that there is an increased risk for creating self-intersecting objects with a compound operation such as this one. It is useful to apply a check and possible correction of such objects after the operation.

6.3 Sweeping wireframe models

In this context, wireframe models are non-Eulerian models, consisting only of edges and vertices. They can contain multiple components; that is, not all parts need be attached to each other. Sweeping one of these wireframe models generates a 'sheet model', a flattened shape, or connected set of flattened elements, like the models described in chapter 5. The sheet-models generated from each separate piece of the graph are Eulerian, with faces, edges, and vertices, so there are problems because of the conversion from non-Eulerian to Eulerian objects.

The sheet models generated by sweeping wireframe models are of the 'degenerate model' type; that is they are 'locally manifold' with a set of faces on each side as illustrated in figure 6.19. Two algorithms are presented. The first, which has been implemented, can be used to generate degenerate models or, with some modification, non-manifold or partial models. The second is more appropriate for creating degenerate type models, but it can use the same type of face sweeping as described in section 6.2. A convention followed here is that the faces on each side of the sheet object should be in separate facegroups. This has practical implications for some operations, such as giving sheet objects thickness (section 6.7), and for splitting the sheet object along existing edges.

If there is no restriction on the wireframe models that are being swept, some sort of pre-processing step should be applied to ensure that the wireframe model is sensible. For example, if the wireframe model has vertices with more than two edges attached, then ambiguity problems can occur. In the simple case illustrated in figure 6.20, the actual object structure can vary if facegroups are used to represent sides. The branched wire object gives rise to a branched

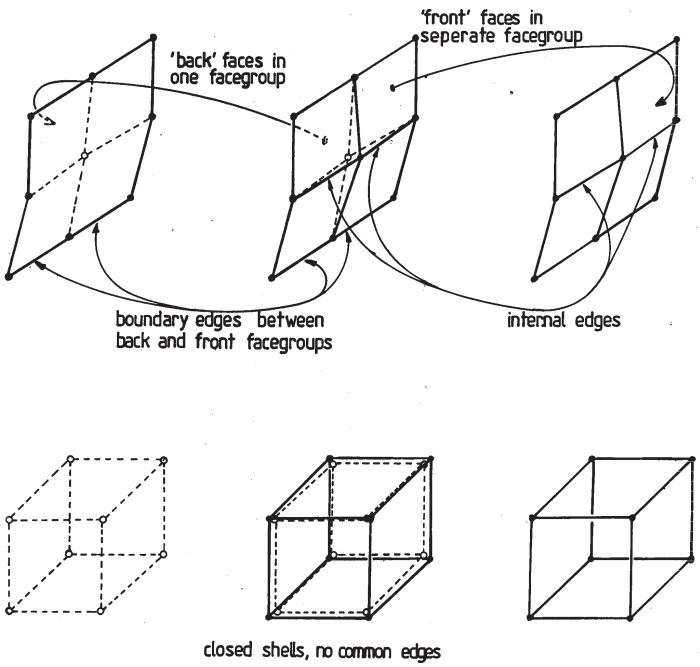


Figure 6.19: Degenerate sheet objects

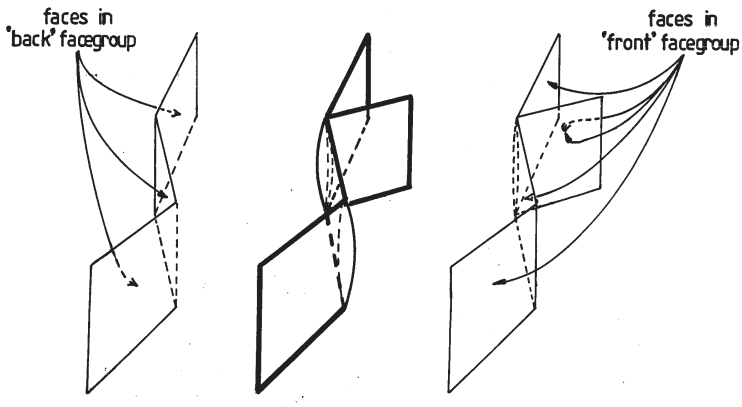


Figure 6.20: Branching wire giving rise to branching sheet object (from [123])

sheet object, but the simple use of one facegroup for the faces at the back of the object, and another for the front of the object, makes the interpretation of front and back ambiguous in this case.

6.3.1 First algorithm

The first step is to see whether the graph contains multiple components. If it does, then these are separated into separate objects and swept separately, the whole collection of sheet objects being returned as the result of the sweep, collected into a group-of-objects. For each separate component, the algorithm works through the edge list in the object sweeping each edge separately. As with face sweeping, the structure being traversed changes during the traversal, and marker bits are again used here to mark the original objects in the model before the traversal. For convenience the original vertices in the model are also marked, and they are unmarked when they are extended the first time, so that it is easy to check whether an edge adjacent to the edge being swept has already been swept.

For straight sweeping, only the non-Eulerian sweep step is needed. For circular sweeping, non-Eulerian sweeping is used for the first step, and thereafter Eulerian sweeping if the model is swept in several steps. If the graph is being swept in a complete circle, then the final edges are joined with their matching original edges using an edge–edge glue analogous to the exact face–face glue. Each edge is swept on its own rather than sweeping all edges in the object with the non-Eulerian sweep, and then continuing with the Eulerian edge sweep. It means that it is easier to keep track of the first and last edges that will be joined after sweeping in a complete circle, but it has drawbacks for the edge joining procedure, which may have to cope with both Eulerian and non-Eulerian edges.

6.3.2 Non-Eulerian edge sweeping

The first extension of the wireframe model is awkward because the object has to be converted to an Eulerian object. If the edge is again extended, then a different sweep operation is performed.

For each edge being swept, the end vertices are examined to see whether they are still marked. If a vertex is still marked, then no neighbouring edge has been swept, so there will be no existing extension edges at the vertex, so a new extension edge and vertex are added. If the vertex is not marked, then there is at least one existing extension edge, and the edges at the vertex are checked to find it. Because of the way that laminae and sheet objects are represented, it is necessary to find a swept vertex at the end of an existing extension edge. If there is only one extension edge at the vertex, as in figure 6.21a, then this is easy. If there is more than one extension edge at the vertex, as in figure 6.21b, then it is necessary to determine on which side of the existing faces the new

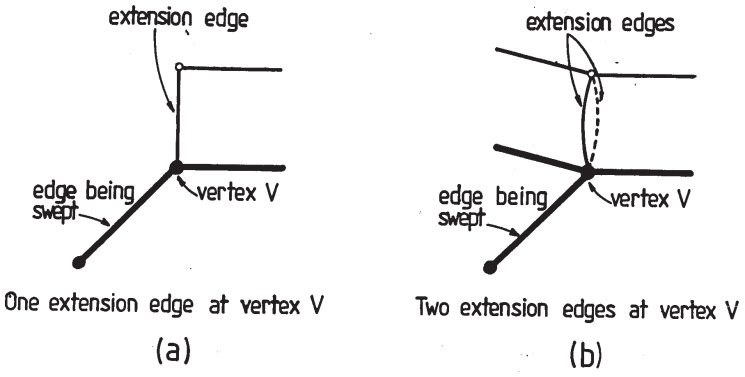


Figure 6.21: Adding faces at branch points

face will lie. Once the swept vertex has been found, then a new edge is added between the base vertex and this swept vertex.

Once both ends of the edge have been dealt with, the two swept vertices are linked by a new edge, two new faces are created, and all new edges are linked in as their boundaries. Figure 6.22 shows three cases in which there are no extension edges, one extension edge, and two extension edges. In figure 6.22a, the edge marked as 1 is swept, producing extension edges 2 and 3, and the extension vertices are connected with edge 4. Here, one face will have the edges in the order: 1 -> 2 -> 4 -> 3 and the other in the order: 1 -> 3 -> 4 -> 2. In figure 6.22b, there is one existing extension edge, edge 2. Two more extension edges are added, edge 3 and edge 4, with the final edge, edge 5, connecting the two extension vertices. Here, one face perimeter will have the edges: 1 -> 3 -> 5 -> 2 and the other the order: 1 -> 4 -> 5 -> 3. In figure 6.22c, there are two existing extension edges, edge 2 and edge 3. Two more extension edges are added, edge 4 and edge 5, and the extension vertices are connected by edge 6. One face will have the perimeter: 1 -> 4 -> 6 -> 3; and the other the perimeter: 1 -> 5 -> 6 -> 2.

One final comment to be made concerns the cases in which a wire model is being swept in a circular arc (or swung), and one or both ends of the edge lie on the axis. In this case, it is unnecessary to add the extra wire extension edges, the vertex or vertices on the axis are used when closing the face, and the faces created will have two- or three-edge perimeters. However, it should be noted that the case in which both end vertices of the edge lie on the axis is only allowable if the edge is curved, i.e., that it does not coincide with the axis.

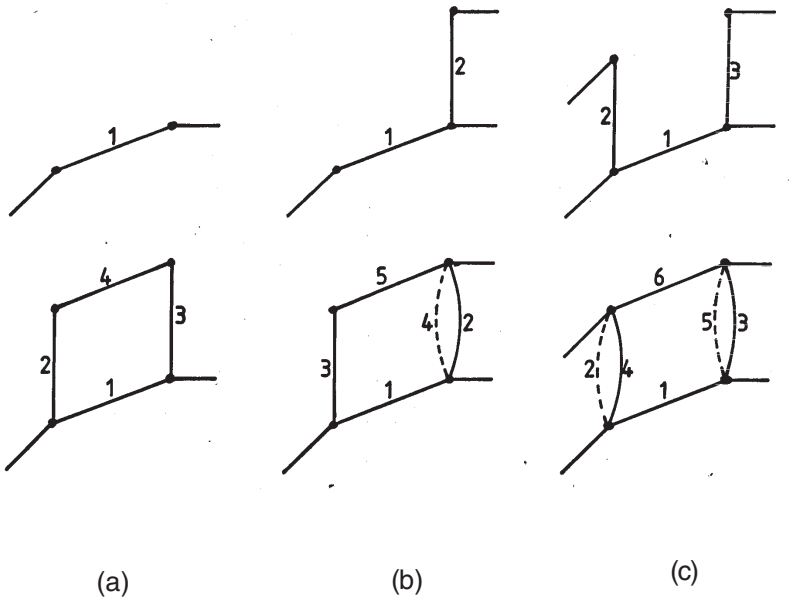


Figure 6.22: Generating faces with neighbouring faces

6.3.3 Eulerian edge sweeping

For circular sweeping, if the wireframe model cannot be swept in a single step because of geometric restrictions, it is swept in steps. Once the first sweep extension of an edge has been made, the edge being swept is locally Eulerian and can be swept further using the Euler operators, so when swinging graphs, the first step uses the non-Eulerian sweeping method, and any subsequent step or steps use an Eulerian sweep technique.

For Eulerian edge sweeping, the basic step when sweeping an edge is to slice it lengthwise, creating a face with a two-edge perimeter; then the two extension edges are added using the MEV Euler operator, and the extension vertices are connected using the MFE operator. See figure 6.23a.

As with the first, non-Eulerian sweep, if the end vertices already have extension edges, then they have to be treated specially. Figure 6.23b illustrates the case in which one vertex has extension edges, and figure 6.23c illustrates the case in which both ends have extension edges and vertices. In these cases, the existing extension edges are split, together with the end vertex. In figure 6.23b, the existing extension edge is split, together with the relevant end vertex, creating a face with four edges; the second extension edge is added using MEV, and the face is added with MFE. In figure 6.23c both end vertices of the edge being swept have extension edges that are split, creating a large face with six perimeter edges. The final edge splits this face into two four-sided

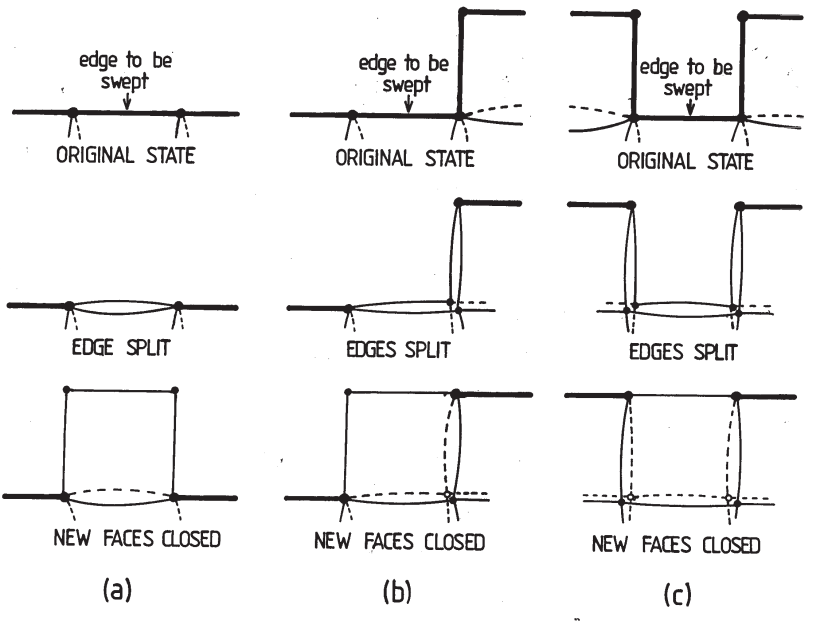


Figure 6.23: Eulerian edge sweeping for wireframe models

faces.

The cases in which one or both end vertices of an edge being swung are on the swing axis are handled in a similar way, except that instead of using an extension vertex, the vertex itself is used as an end vertex of the edge closing the face.

6.3.4 Second algorithm

The second algorithm, which has not been implemented, uses the face sweeping code described in section 6.2 with an initial pre-processing and a final post-processing step. The pre-processing step involves slicing all edges in the graph lengthwise, and splitting all internal vertices, to produce a sheet model with degenerate faces. This degenerate model is then swept with the face sweeping procedure to produce the sheet model. The post-processing step involves collapsing any degenerate faces back into wires. This method has the advantage that existing code is used to generate the new topology and geometry. However, the associativity between matching edges on both sides of the sheet model is lost, making the algorithm only appropriate for generating degenerate type models.

Figure 6.24 shows two examples of wire objects sliced to become degenerate laminae. The simplest way of slicing the graph is to work through the edge list

of the object, slicing each edge separately, and slicing the end vertices when there is at least one other edge at the vertex that has already been sliced. This will, potentially, keep creating and merging faces, but without recursion, it is less complicated than trying to keep track of which edge-branches are being modified and which have been traversed. Again, marker bits can be used to determine which are original edges and which have been split already. The first stage is, therefore, to mark all edges and vertices in the graph with some chosen marker bit. Then, the edges in the graph are again traversed, slicing the marked ones.

Slicing an edge essentially creates a degenerate face with two edges. When a neighbouring edge has already been split, then instead of creating a new two-edge face, the face belonging to the neighbouring edge is extended to include the two halves of the newly sliced edge. Marking the original vertices in the object provides a simple test as to whether a neighbouring edge has already been split. As each edge is split, the chosen marker bit of both end vertices is cleared. If an edge is encountered where either just one, or neither, end vertex is marked, then a neighbouring edge must have been split, so the corresponding end vertex must be split. If neither end vertex of the edge is marked, then it is necessary to coalesce two disjoint faces or create an internal loop.

Because the faces created during this process are degenerate, geometry cannot be used to sort out the object and it is necessary to take care when creating them. Figure 6.25a shows an edge being split when no neighbouring edges have been split. The original edge is marked as 1, and the other half of the sliced edge is marked as 2 (figure 6.25d). The order of the edges in the faces is the same for both faces (but the relative edge directions are different).

When a neighbouring edge has already been split, as in figure 6.25b, then the vertex is split, and the old degenerate face is extended by two edges. If the edges attached to the vertex that are part of the old degenerate face are labelled 3 and 4, then the order of the edges will be:

... -> 3 -> 2 -> 1 -> 4 -> ...

proceeding clockwise round one of the faces (figure 6.25e).

When both end vertices of the edge are adjacent to edges that have been split, figure 6.25c, then both end vertices of the edge have to be split. If the edges of one existing degenerate face are labelled 3 and 4, and the edges at the other end are labelled 5 and 6, the order of edges in the combined face will be:

... -> 5 -> 1 -> 4 -> ... -> 3 -> 2 -> 6 -> ...

proceeding clockwise round one of the faces. If the edge is part of what was a closed loop of edges in the original graph object, then at one stage the edges marked 3, 4, 5, and 6 will, in fact, belong to the same loop and so the edges will be divided into two sets, one of which will be an internal loop. There

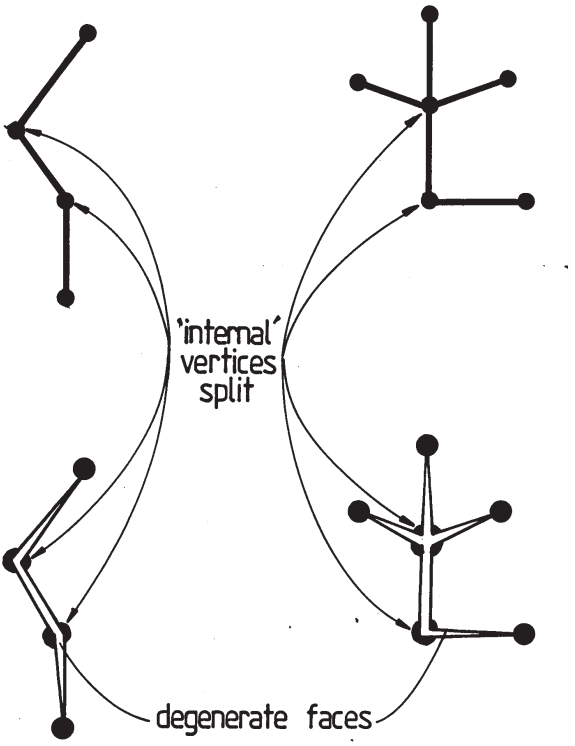


Figure 6.24: Slicing edges in wire objects to produce degenerate laminae (from [125])

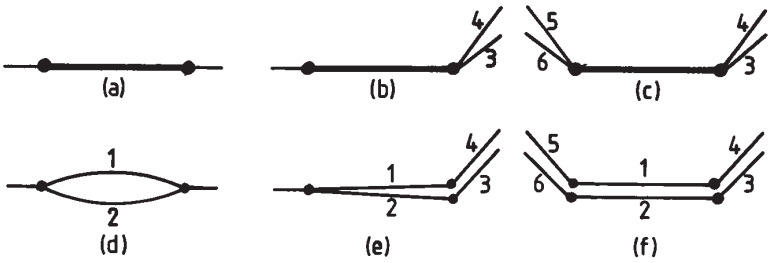


Figure 6.25: The three possible conditions when slicing an edge

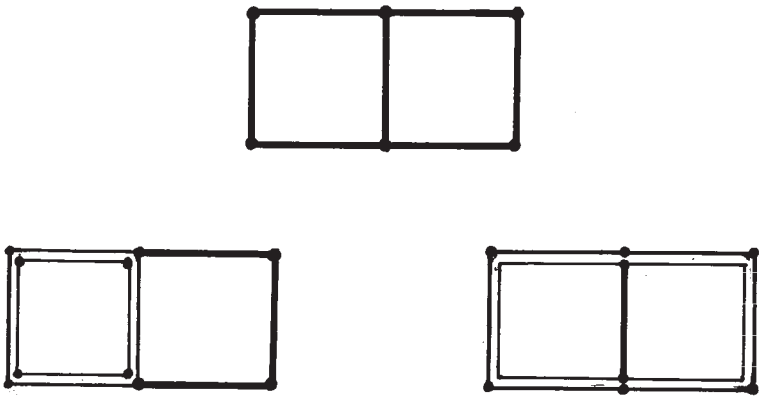


Figure 6.26: Slicing closed loops of edges

can be problems if, say, the wire model was in the form of a rectangle with a wire across the middle (figure 6.26, top). If one of the small faces has been completed, it is necessary to have the unfinished branch as the outer loop (figure 6.26, bottom left). If the large loop has been completed, then it is necessary to have the unfinished loop as the inner loop (figure 6.26, bottom left). Although it is possible to distinguish between some cases, it would seem preferable to place restrictions on the way wireframe models are used.

Splitting a vertex requires geometric checking for vertices where there were more than two edges in the original wireframe model. This is because the edges at the vertex, apart from the current edge being sliced, and the previously sliced edge, have to be re-allocated to one or other of the split vertices. Taking the edge being sliced as a kind of base edge, and the previously sliced edge as a reference, the angles at which the other edges at the vertex enter can be used to partition them. Figure 6.27 shows two examples of vertices being split.

Once all edges in the object have been sliced, and the degenerate faces produced, one face is swept, in a straight line, circular arc, or complete circle, using the face sweeping procedure described in section 6.2. After the sweep, the two degenerate end faces are collapsed back into a set of edges and, if necessary, joined using the edge joining process described in section 6.4.2.

Collapsing back the faces involves finding matching edges and coalescing them. If the original graph model component contained a simple open sequence of edges, i.e., with two end vertices with only a single edge attached and the rest with just two edges attached, then the process is fairly easy. If there were branches in the wire, then it will be necessary to close each branch separately. If there were closed loops of edges in the original model, then matching edges will involve matching edges in a boundary ring of edges with

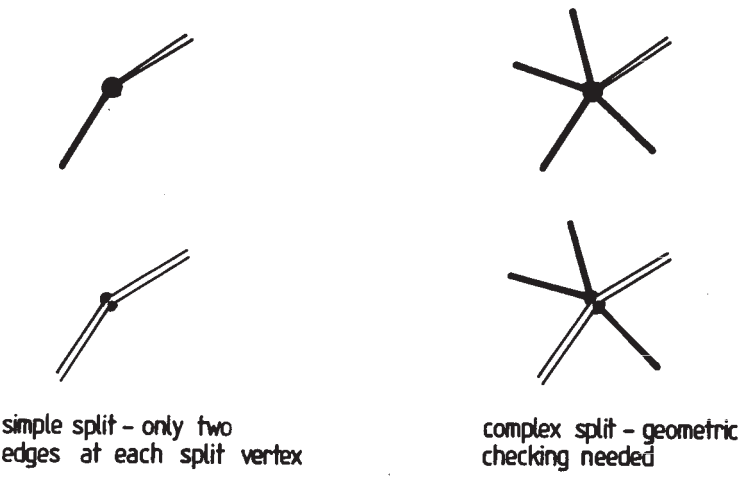


Figure 6.27: Sorting out edges when slicing vertices

edges in an internal loop.

To begin with, the edges in the face, say, F0 with outer loop L0, are scanned, looking for a pair of adjacent edges that match each other. If none is found, then, if the face really is degenerate, the edges in the outer loop must match edges in an inner loop or loops. This case is similar, but it will be dealt with separately later. Once two matching edges have been found, say E1 counter-clockwise and E2 clockwise around the loop, the common vertex, V1, is identified (figure 6.28a) and the edges are coalesced. To coalesce an edge, the vertices at the opposite ends of E1 and E2 from V1, say V2 and V3, and the loops on the opposite sides of E1 and E2 from L0 are found, say L1 and L2 (figure 6.28b). If V2 and V3 are, in fact, the same vertex, then E1 and E2 are the last edges in the loop, and after they are coalesced, the degenerate face has been collapsed. If they are not, then all edges meeting at V3 are moved to refer to V2 (figure 6.28c). The edge counter-clockwise around the loop L0 from E1, say E3, is made to point at the edge clockwise around the loop from E2, say E4, and E4 is changed to point at E3 (figure 6.28d). The loop pointer of E1, which refers to L0, is changed to refer to L2 (figure 6.28e). The edges clockwise and counter-clockwise around L2 from E2 are made to point at E1 rather than at E2, and E1 is changed to point at these edges (figure 6.28f). E2 and V3 are then killed (figure 6.28g). The process then continues with E3 and E4 as the edges being coalesced. If the next pair of edges, E3 and E4, do not, in fact, match each other, then E1 and E2 were the last edges in a branch. In this case, the process starts from the beginning, scanning the loop looking for a matching pair of edges. Eventually all sub-branches will disappear leaving one main chain of matching edges.

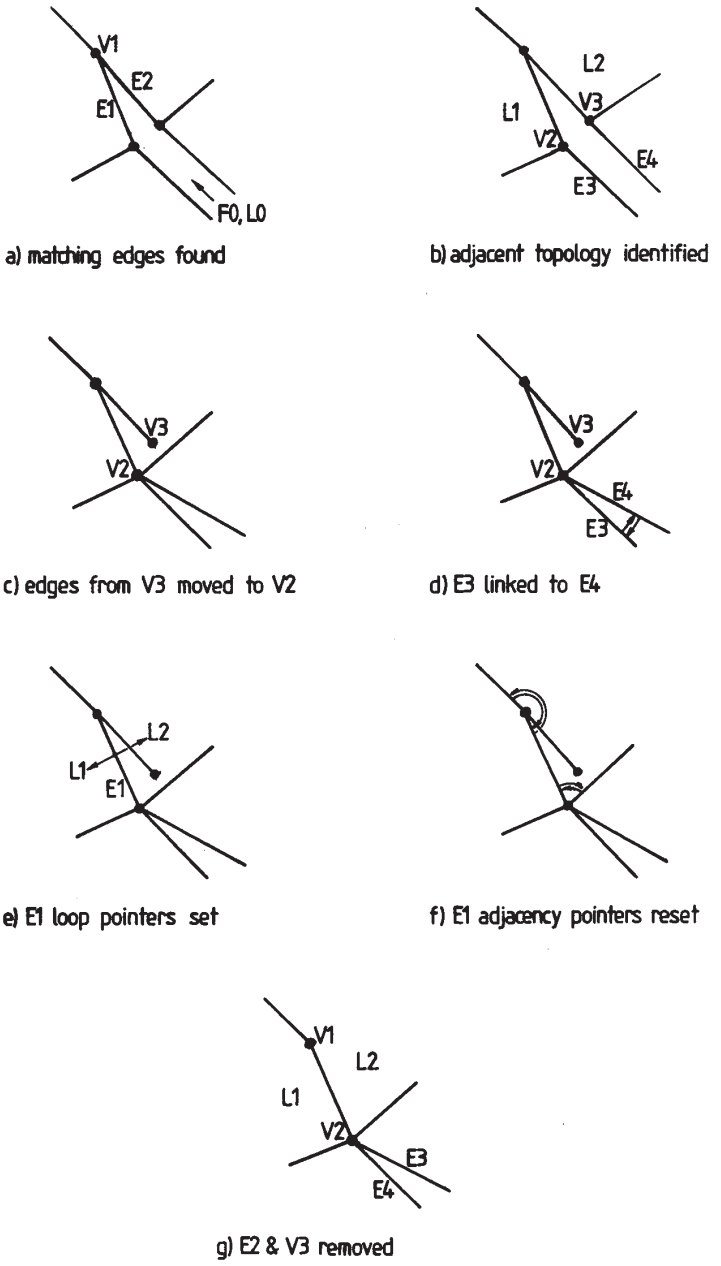


Figure 6.28: Coalescing matching edges in a degenerate face

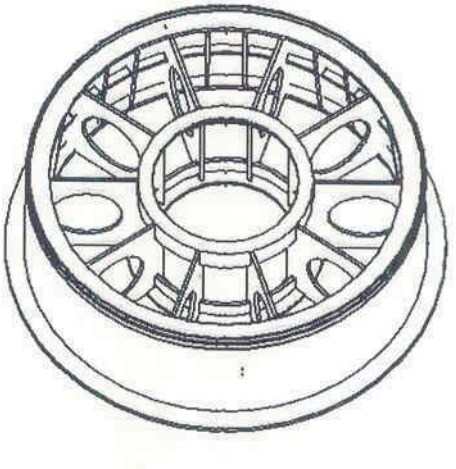


Figure 6.29: The Contraves object

If no matching pair of edges is found in the degenerate face, then this implies that the edges in the boundary match edges in an inner loop. To collapse this kind of face, one edge in the outer loop is chosen and all edges in the inner loops are scanned until a matching edge is found. When a matching pair of edges has been found, these and both pairs of end vertices are coalesced. This creates the conditions described above, when pairs of matching edges have one vertex in common. Once the first pair of edges have been coalesced, therefore, the process proceeds as described above.

6.3.5 Swinging flat

To end with, this is a short description of an application for creating flat shapes from wire objects swept, or swung, around an axis. This application was developed using GPM as a demonstration for a CAM-I meeting in 1983 and is described by Kjellberg et al. [71]. The CAM-I meeting was the second to allow different modelling groups to present the results of modelling three benchmark objects. The first object, which was the subject of the first CAM-I comparison meeting, was the MBB Gehause. The second object was the Cranfield 'thing', an idealisation of an oil rig part with interesting blends. The third part was an object designed by Contraves, an object made from thin plates.

The Contraves object, shown in figure 6.29, was modelled as a three-dimensional model using the GPM sheet objects. Another demonstration was prepared with the flattened shapes to be bent to create the shapes to be cut from metal sheeting, figure 6.30, which shows a collection of all the flat objects

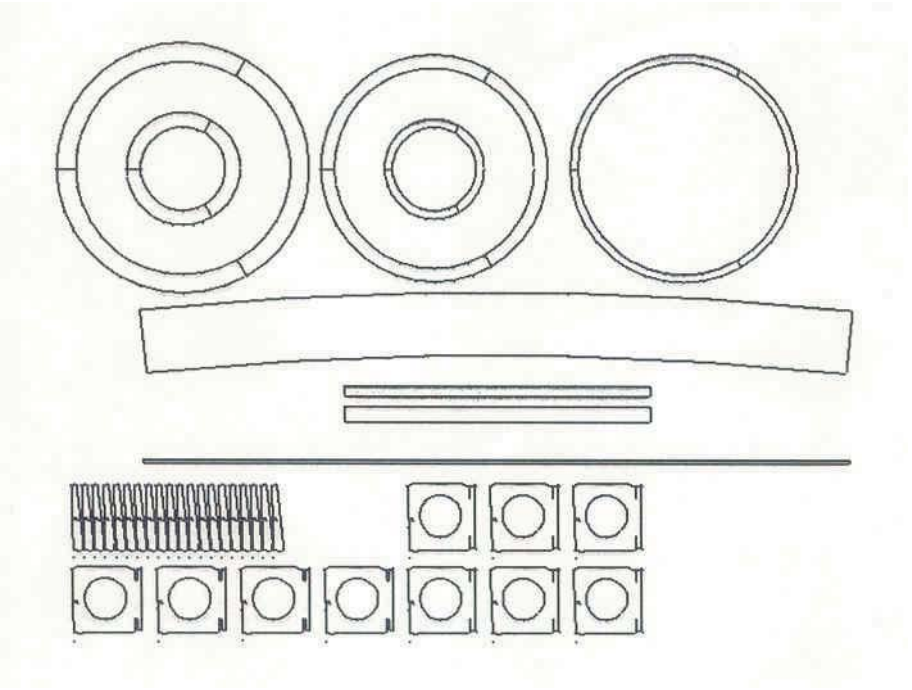


Figure 6.30: Flat model defining plate shapes for the Contraves object

including the ribs and stiffeners. The idea was to add a sweep option so, with the same parameters used to create the three-dimensional solid but with the option 'flat' the two dimensional objects were created. Note that the objects shown in figure 6.30 are not 'nested', that is, they are not placed to minimise material. The operation is just to create the flat shapes, not to nest them which would be necessary for a full-scale implementation.

The operation worked on a wireframe shape composed of straight lines, such as that shown in figure 6.31. Each edge in the wire was separated and analysed with respect to the axis.

If an edge is perpendicular to the axis, such as edge e1 in figure 6.31, then it will give rise to a circular disc. If neither end vertex is on the axis, then the resulting shape will be a disc with a hole in it. In the figure the horizontal edge gives rise to a circular disc radius $d1 + e_{len}$ with a circular hole radius $d1$.

If the edge is parallel to the axis, then it will generate a cylindrical sheet object. This is converted into a rectangular shape. In figure 6.31, edge e2 gives rise to such a shape; the length is $2/pid2$, and the width is the same as the length of edge e2.

If the edge is inclined with respect to the axis, as edge e3 in figure 6.31,

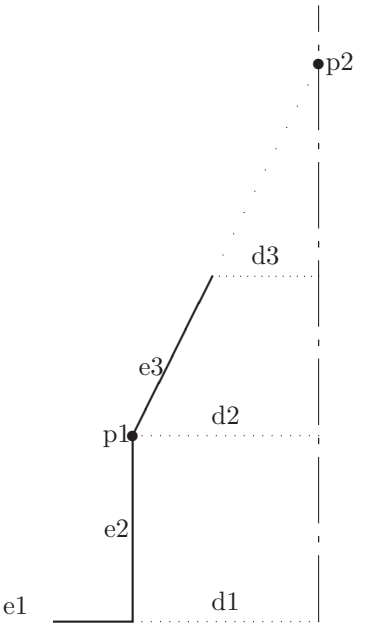


Figure 6.31: Flat swinging

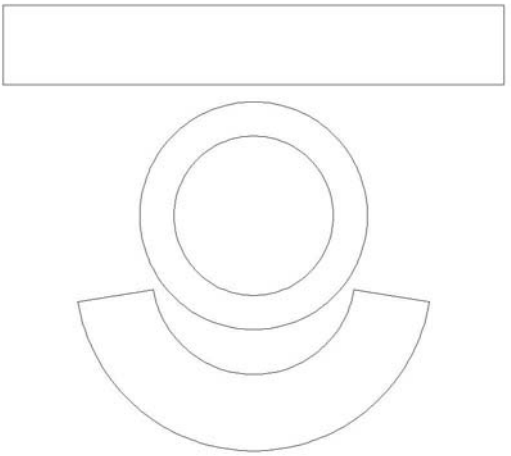


Figure 6.32: Flat model of simple object

then it gives rise to a portion of a disc. The radius of this disc is the distance of p1 to p2. The length of the portion of this disc is given by $2\pi d3$. This gives the objects shown in figure 6.32

This is an alternative to the process of unfolding objects described in chapter 5. It is slightly easier to arrange in two dimensions than in three dimensions.

6.4 Gluing coincident identical topology

Gluing identical, coincident topological entities, i.e., faces, edges, and vertices, is essential for operations such as swinging and reflect, and it is also useful as a basic manipulation tool for users. This section describes three algorithms for joining such topological items.

6.4.1 Gluing identical faces

The algorithm described here joins two faces, either belonging to the same object or to different objects. If the faces are in different objects, then the stepwise joining process might be as shown in figure 6.33. If the faces already belong to the same object, then the process might be as shown in figure 6.34 or figure 6.35, where there is a common edge between the faces being joined. The changes in the Euler–Poincaré formula for each step are also indicated in the figures.

If there are two separate objects, as illustrated in figure 6.33, the first step is to join the two objects into one and join two matching edges in the face,

coincidentally removing two vertices and an object. This sets up the basic conditions for the sequential process of joining matching edge pairs around the face. One of the edges joined becomes an edge between the two matching faces, and it is useful, but not crucial, to remove this first, unifying the faces as in Figure 6.33. The edge merging procedure then proceeds around the face, merging matching edges until the final pair are merged. At each stage, one edge is removed from the face being sewn up and the other moves to the outside of the object. There is an option to check the edge on the outside to see whether it is necessary. Here, an edge is regarded as unnecessary if it is a 'smooth' edge, i.e., lies between two co-planar faces.

Whenever a smooth edge is removed, then the start and end vertices of the edge should also be checked to see whether they can be removed, here the condition being that the vertex lies between two suitably oriented collinear edges. After the first step shown in figure 6.33, both vertices have four edges attached and so cannot be removed. The next step is to take the pair of matching, adjacent edges and 'zip' them together. This operation produces a dangling spur edge in the face being removed by rearranging existing topology, but it does not actually create any new entities in the model. The spur edge and the attached spur vertex are then removed. The remaining edge is also examined to see whether it is smooth, and in this case, it is removed. This now leaves one end vertex with only two collinear edges attached, and that vertex and one edge can also be removed. The 'zipping' step is repeated, at each stage producing a spur edge that is killed, and tidying up the remaining edges and vertices. Finally, only two edges in the face remain. One of these is deleted, along with the face, adjusting the topology appropriately. The remaining edge is, as before, checked to see whether it is smooth. In the example shown in figure 6.33, it is smooth and so it is removed, together with the start and end vertices that are also unnecessary.

The case shown in figure 6.34 can be solved in much the same way. Here the faces are in the same object, so it is unnecessary to merge the objects as in the first example. However, the first step is again to merge two edges, actually removing two vertices and increasing the genus of the object, because the faces already lie in the same object. The edge that is now between the faces being joined is then removed, and the other edge checked to see whether it is smooth. In the example given, the outside edge is smooth; i.e., it lies between co-planar faces. The next steps are the same as outlined above, using the sort of 'zipping' operation that combines two edges, thus producing a spur edge, that is deleted, and an edge which is checked to see whether it is smooth. In this example, when the smooth edges are removed, the actual operation removes a wire edge and creates a hole loop. The edges between the curved faces may or may not be removable depending on the extent of the resulting face and the basic modeller restrictions. The final step is, again, to kill an edge and the face being removed (rearranging the topology as necessary), and then to check the remaining edge to see whether it is smooth.

In the example shown in figure 6.34, edges between curved faces are re-

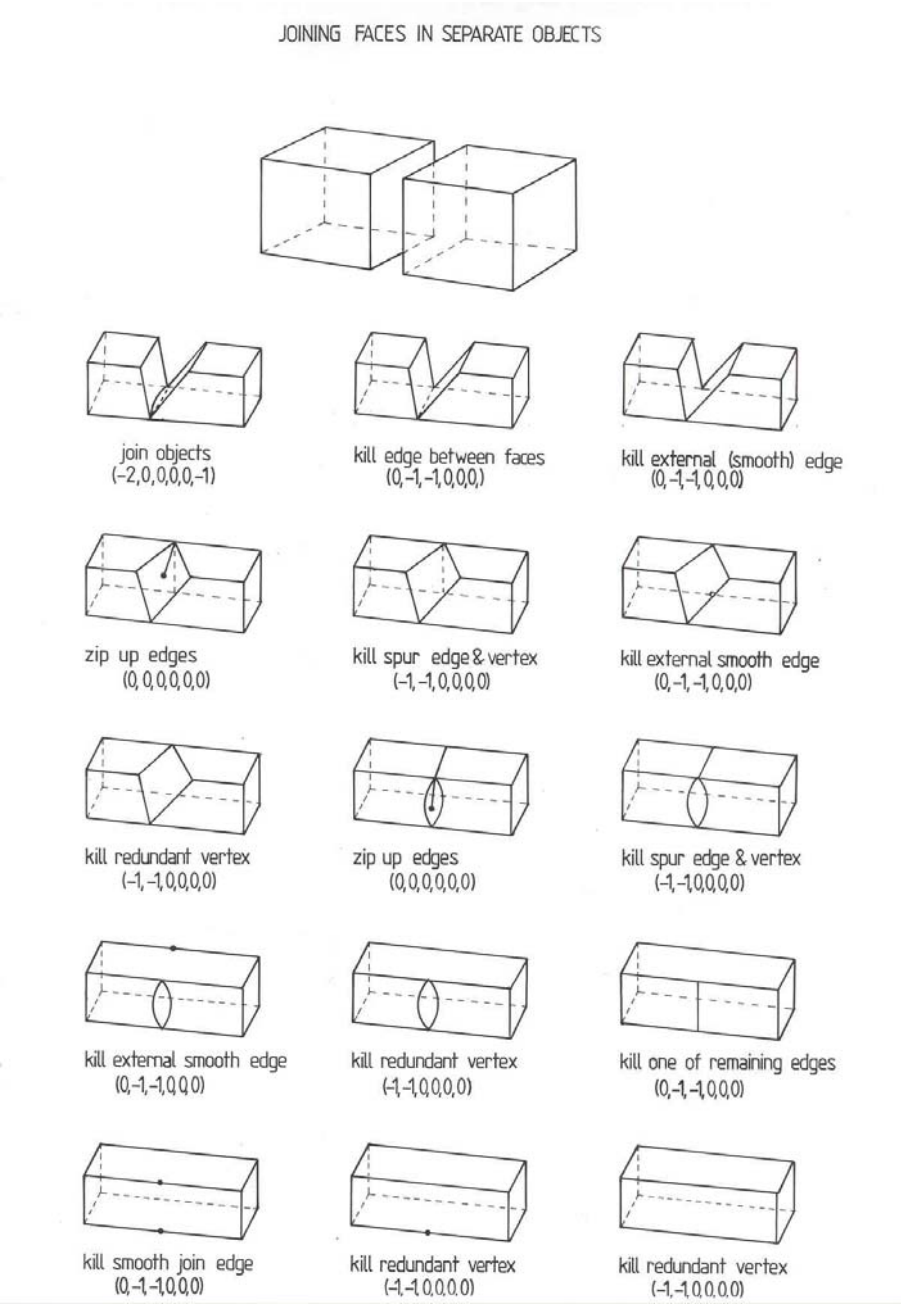


Figure 6.33: Joining faces in different objects

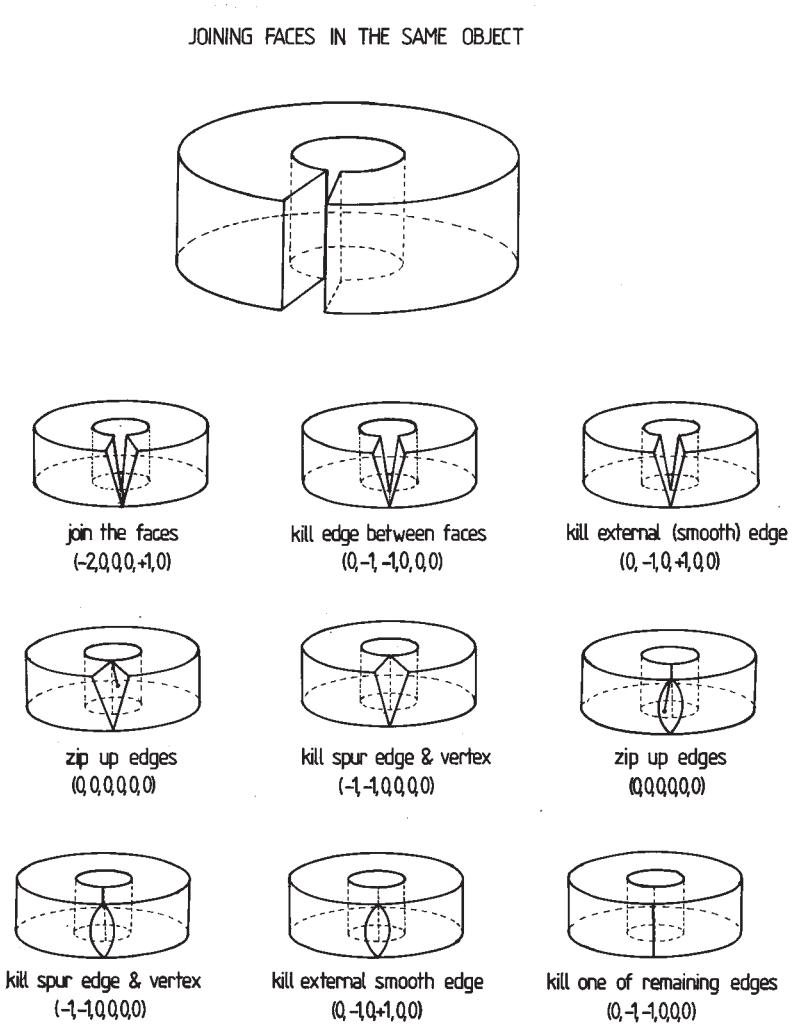


Figure 6.34: Joining disjoint faces in the same object

tained, so the final edge is not deleted. Note that because the edges between the curved faces are retained, none of the end vertices of the smooth edges are deletable.

The final example, illustrated in figure 6.35, is very similar to the other two examples, except that the faces being joined already have a common edge, so the first step for the previous two examples is unnecessary. If the joining process starts on this common edge, then the first step is simplified, and otherwise everything proceeds as before. The first step is simply to delete the common edge, because it already lies between the two faces being joined. The remaining edges are zipped together, and the final edges are removed exactly as described above, except that no hole-loops are produced.

If, however, the joining process starts on an edge other than the common one, then the common edge becomes a wire edge. For two four-sided faces, the first step might be to remove two vertices and increase the genus and then join from there (figure 6.36a), or it might only involve a rearrangement of topology (figure 6.36b). Because the implementation starts on any edge, then the repeated step, the 'zipping', has to check that the edge that is normally left (which is checked for smoothness) is, in fact, a wire; in which case, it is deleted. It is also necessary to check whether the edges being joined are the same edge (in fact the common edge) or different. The reason for not checking all edges in one of the faces to see whether it lies between the two faces is that it is perfectly possible that there are several such edges. The algorithm would, therefore, have to cope with wire edges anyway, so it is always necessary to check edges to see whether they are wire edges.

Note that these operations do not use only Euler operations. As mentioned in chapter 4, some complex operations perform Euler changes directly. For Euler purists, another version is shown in figure 6.37.

The first operation is to make the loop of the bottom face of the top cube into a hole-loop of the top face of the bottom cube, an MHKFB operation. The loops are actually coincident but are shown offset for clarity. The next Euler operation joins the hole-loop with the outer loop, an MEKH operation. The next operation joins another vertex of the original hole-loop with the corresponding vertex of the outer loop, an MFE operation. The next two operations remove these edges, with two KEV operations. The final operation in the figure is a KFE operation to remove the face added in step 3. The process is repeated, joining vertices with MFE operations, removing the new edges with a KEV operation, and removing the two-sided faces with a KFE operation until the original top face of the bottom cube is removed.

Note that instead of using the common MEKH, MFE, and KEV operations, it is possible to use KVH and MFKV operations. This is another illustration of what was said in chapter 4, that more operations are useful than the basic set of five necessary.

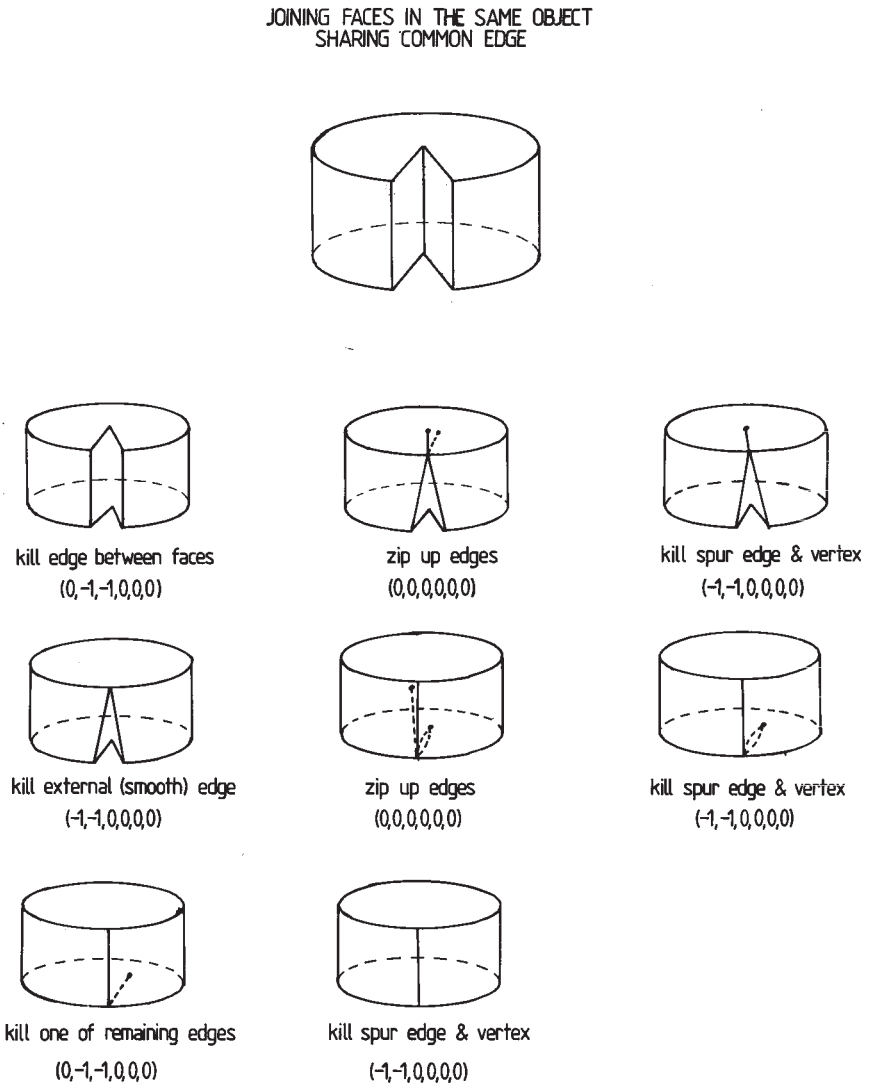


Figure 6.35: Joining adjacent faces in the same object

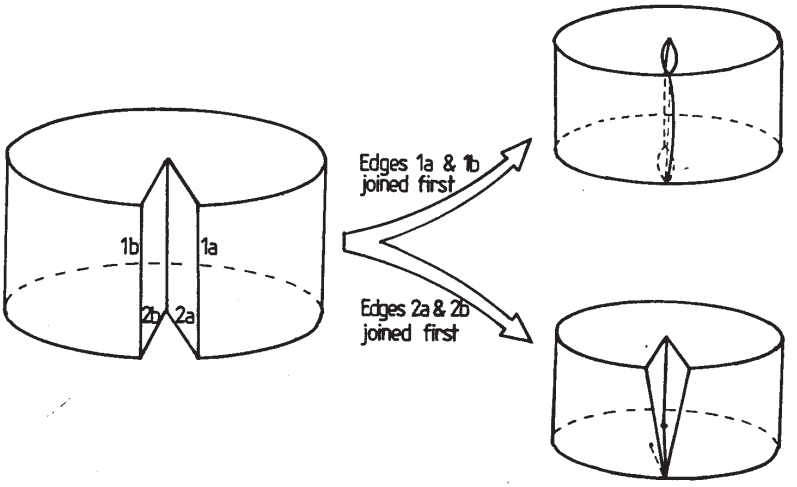


Figure 6.36: Initial edge joining for adjacent faces

6.4.2 Gluing coincident identical edges

With the kind of degenerate type of model considered in this book, when edges are joined, both are retained and lie on opposite sides of the object as though there were an infinitely thin sliver of material between them. This actually applies whether the edges being joined belong to sheet models or to volumes. Gluing edges is most useful for gluing matching edges after a swing in a complete circle, or after reflect, for example, but the process is Eulerian, so it can be applied to any Eulerian objects. Depending on whether the edges share common vertices the vertices are duplicated or merged, as shown in figure 6.38.

Joining edges with no common vertices

To join the edges E1 and E2 with no common vertex, as shown in figure 6.39a, with vertices V11, V12, V21, and V22, the first step is to transfer all edges meeting at V21 to V11, and all edges from V22 to V12 (figure 6.39b). The loop pointers and adjacency pointers are then adjusted. The reference from edge E1 to loop L12 is replaced with a reference to loop L21. The reference from edge E2 to loop L21 is replaced with a reference to loop L12 (figure 6.39c). The references to E1CW and E1CC, the edges clockwise and counter-clockwise around loop L12 from E1 before the join, are replaced with references to edges E2CW and E2CC, which were originally clockwise and counter-clockwise around loop L21 from edge E2. Similarly, the references to E2CW and E2CC from edge E2 are replaced with references to E1CW and

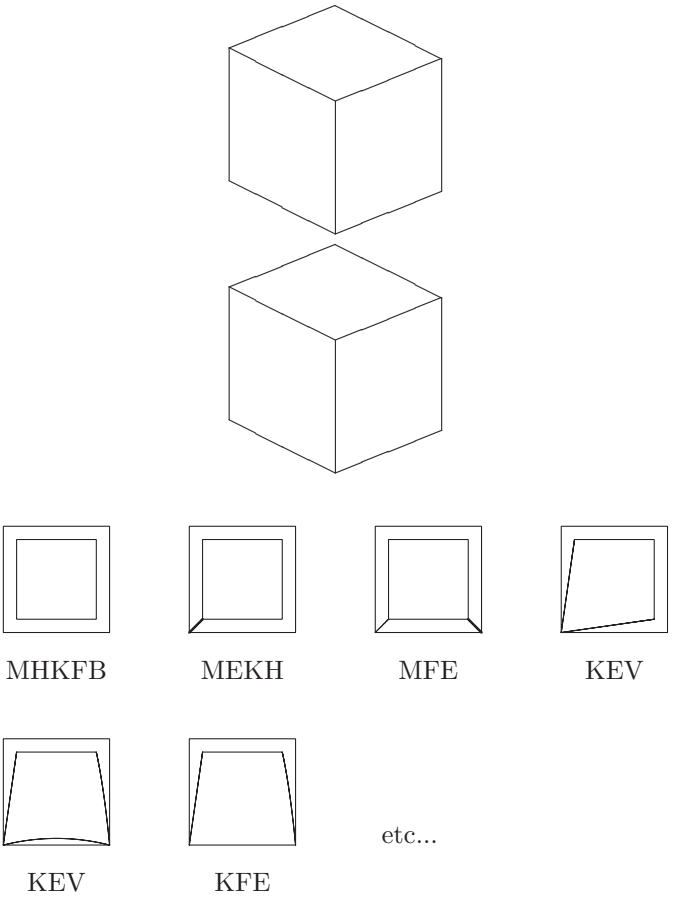


Figure 6.37: Eulerian face–face glue

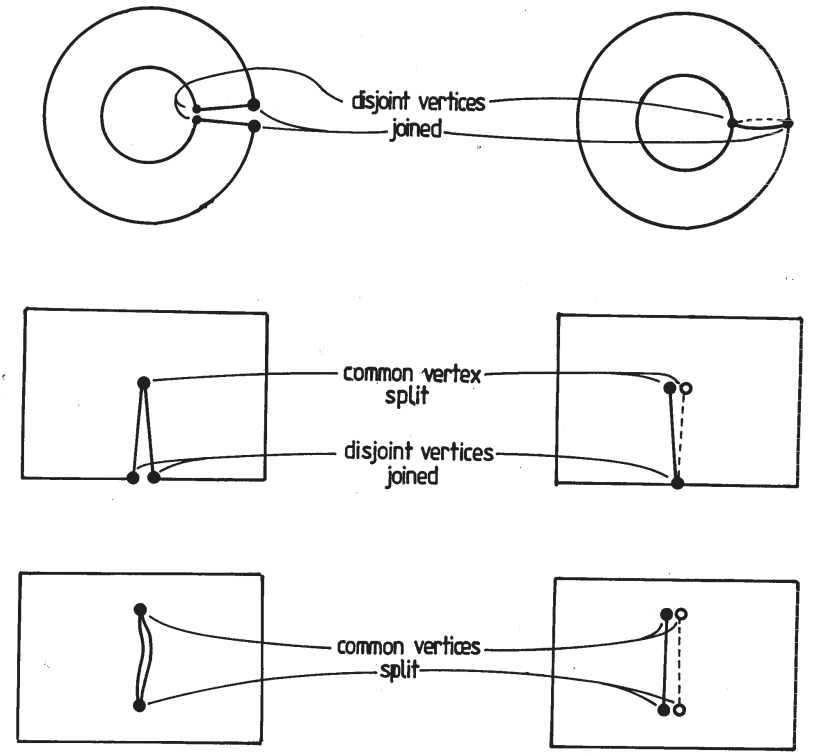


Figure 6.38: Edge joining with zero, one, or two common vertices (from [123])

E1CC (figure 6.39d). The superfluous vertices, V21 and V22, are then killed.

Joining edges with one common vertex

If the two edges being joined have a vertex in common, then two cases have to be considered, as follows:

1. The edges lie between faces on the same side of the object; i.e., they lie in the interior of the object.

2. The edges lie between faces on different sides of the object; i.e., they lie on the boundary of the object.

If the edges being joined separate faces on the same side of the object, then the operation is related to coalescing edges in degenerate faces, which is described in section 6.3. In Euler operator terms, the two non-common vertices can be joined with a new zero-length edge; if they are not already

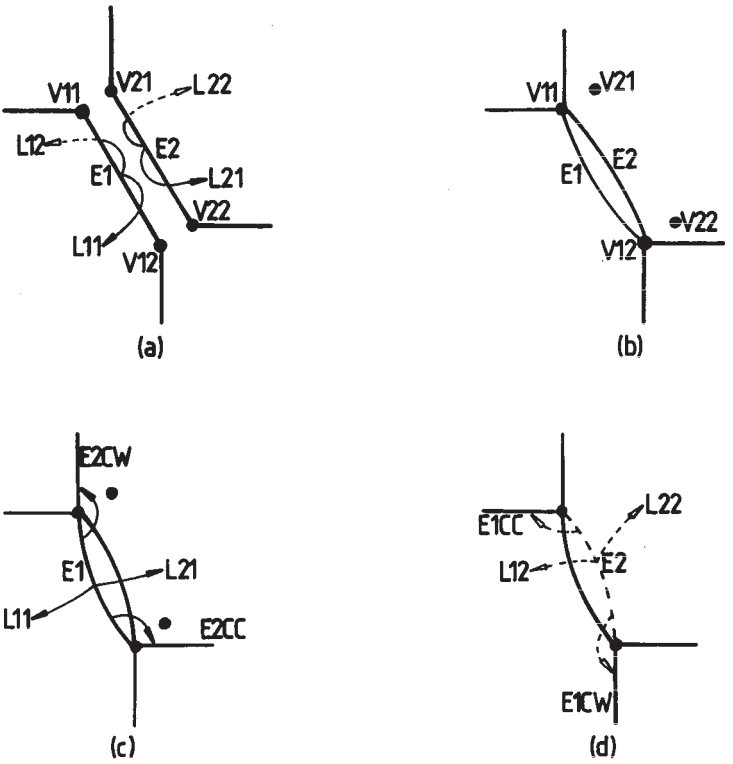


Figure 6.39: Joining edges with no common vertex

joined, then one of the edges and the degenerate face are killed. Finally the zero length edge and one of the coincident vertices can be removed.

If the edges separate faces on opposite sides of the object, then they lie on the boundary of the object, and so the common vertex has to be split during the join. This is because the common vertex, which before the join lies on some boundary, becomes, after the join, part of the interior. The edges being joined partition the edges meeting at the vertex into two sets. After the join, one of these sets lies on one side of the object and the other on the other side, so the vertex has to be duplicated, and one set of edges changed to meet at the new vertex. Starting with one of the edges, therefore, and proceeding clockwise around the vertex, all edges until the second edge is reached are moved to the new copy of the vertex. The last edge moved is also adjusted so that it points to the first edge as an adjacent edge around the vertex. Similarly, the edge counter-clockwise around the vertex from the first vertex has to be adjusted to refer to the second edge as the edge clockwise from it around the vertex. The loop pointers adjacent to the two edges have to be readjusted to be consistent. The rest of the operation is similar to that described above, where two edges without common vertices are joined. The non-common vertices are merged, and the loop and adjacent edge pointers are adjusted.

Joining edges with two common vertices

If the two edges have both vertices in common, then there are again the same two cases to consider; i.e., the edges lie between faces on the same side of the object or between faces on different sides of the object.

If they separate faces on the same side of the object, then the operation is really a "Kill Face and Edge" operation, killing one of the edges and the degenerate common face.

If the edges separate faces on opposite sides of the object, then the operation can be interpreted as equivalent to 'sewing up' a slit in the interior of a sheet object. To perform the joining operation, the sets of edges meeting at both vertices are partitioned into two pairs of edge sets and one of each pair attached to a new vertex, as described in the previous section, adjusting the loop and winged-edge pointers appropriately.

6.4.3 Gluing coincident identical vertices

The description of this operation concerns joining vertices in wireframe objects. There is an Euler operation for merging coincident vertices, described in Appendix F, section F.15. The two cases are shown in figure 6.40. The first joins two vertices in separate bodies, thereby killing a face and a body in the process. The second operation joins two vertices in the same object, thereby creating a new face as well.

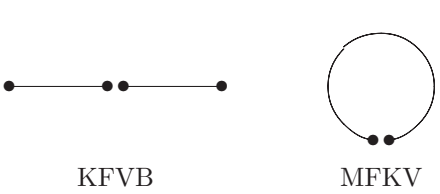


Figure 6.40: Eulerian vertex-vertex glues

In wireframe models, gluing the vertices consists solely of transferring the edges from one vertex to the other vertex and deleting the first vertex. There are no adjacency pointers to adjust, and the order of the edges around the vertex is not important. If the vertices belong to different objects, then it will be necessary to merge the object chains; otherwise, no more needs to be done.

6.5 Reflect

Reflecting an object is a fairly simple process in modelling terms, involving copying the object being reflected, transforming it using a reflection matrix, and joining any coincident topological elements with the tools described in section 6.4. The effect of operation on laminae and solids is shown in figure 6.41 (from Jared and Stroud [65]). An L-shaped lamina is defined, as shown in figure 6.41, top left. This is reflected about an edge at the end of one of the arms to produce a U-shaped lamina; (figure 6.41, top right). This is swept to produce a solid, as shown in figure 6.41, bottom left, and the solid is reflected about one of the 'prongs' to produce the final object, as shown in figure 6.41, bottom right.

The first step is to obtain the planar reflection surface that is used for checking and to calculate the reflection matrix. The surface may either be given explicitly or calculated from a topological entity (i.e., planar face, straight edge, or spur vertex). The object is then copied and transformed with the reflection matrix using standard utilities.

There are various options about how a reflect operation should function. In its simplest form, the reflected copy of the original object can be returned as the result. Another option is to join the original object and the reflected copy at coincident faces, edges, or vertices. If the original and reflected objects are to be joined, then it is necessary to check both looking for the elements lying in the reflection surface and joining these with the methods described in section 6.4. If the objects are to be joined, then it is useful to perform a simple check to see that the combined result object will not be self-intersecting, that is, that the original object lies on one side of the reflection surface only. If none of the faces lies in the reflect plane, then two objects are returned. Yet another option, used in some commercial systems, is to join the original and reflected objects with a Boolean operation, thereby removing the restriction

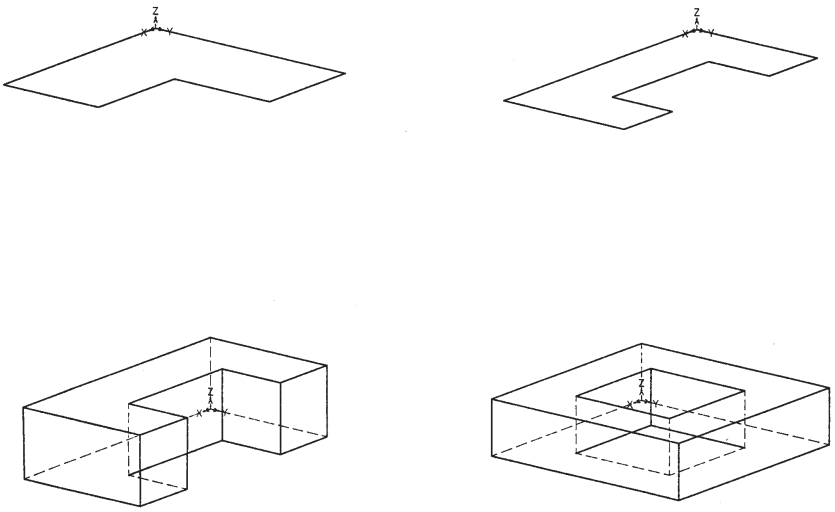


Figure 6.41: Reflecting about straight edges and planar faces (from [65])

that the reflect plane should not cut the object.

6.6 Bend

The bend operation is interesting as an illustration of a specialised process that is difficult to duplicate in, say, a CSG system. The operation bends a portion of an object about a given axis, as illustrated in figure 6.42.

The operation is specified by defining a swing axis, a (planar) face in the portion of the object to be bent, a face in the half of the object to be moved, and the angle of the bend. The operation is equivalent to adding a cross-sectional 'wedge' in the object and rotating one part of the object to make the model consistent. The cross-section where the wedge is inserted is termed the "bend seam".

6.6.1 The basic algorithm

The bend seam is inserted as a sequence of edges by stepping around adjacent faces in the object, starting with the face given as the part of the object where the object is to be bent. The edges are found by intersecting the bend plane with each face in turn. In the same manner as when checking a surface against a face in the Boolean operations, the bend plane is intersected with a face to produce a curve, which is intersected back with the face to produce

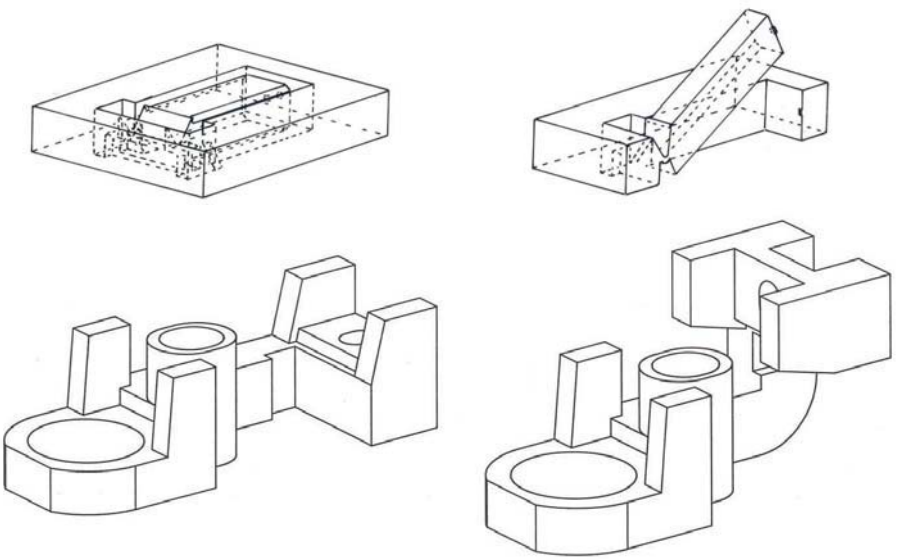


Figure 6.42: Bent objects (from [65])

a set of curve segments corresponding to edges of the bend seam. The next face to be intersected is found from the places where the curve cuts the face boundary. Where the curve cuts an edge, the face on the opposite side of the edge defines the next face to be checked. If the curve cuts through a vertex, then the faces meeting at the vertex have to be analysed to see which one is cut by the plane.

This process is illustrated in figure 6.43. Figure 6.43a shows the original part to be bent. In the figure, the first face to be intersected is face 1; the face is given explicitly as part of the operation specification. The curve is calculated by intersecting the bend plane with the face surface. This curve intersects edge 2, which is split by inserting vertex 9, and edge 3, which is split by inserting vertex 10, and a new edge is inserted between these two new vertices (figure 6.43b). The face on the other side of the original edge 3, face 6, is chosen as the next face (arbitrary choice). Face 6 is now intersected with the bend plane. One place where the plane cuts the face boundary is vertex 10, a new vertex inserted in the previous step. The other place is on edge 12, which is split by inserting vertex 11, and a new edge is inserted joining vertex 10 and vertex 11 (figure 6.43c). The face on the opposite side of the original edge 12, face 2, becomes the next face to be intersected by the plane. Face 2 is intersected in the same way. One starting vertex, vertex 11, exists; the other vertex, vertex 12, splits edge 9; and a new edge is inserted between vertex 11 and vertex 12 (figure 6.43d). The final face to be split is face 4. Here, however, both end vertices where the bend plane cuts the face boundary

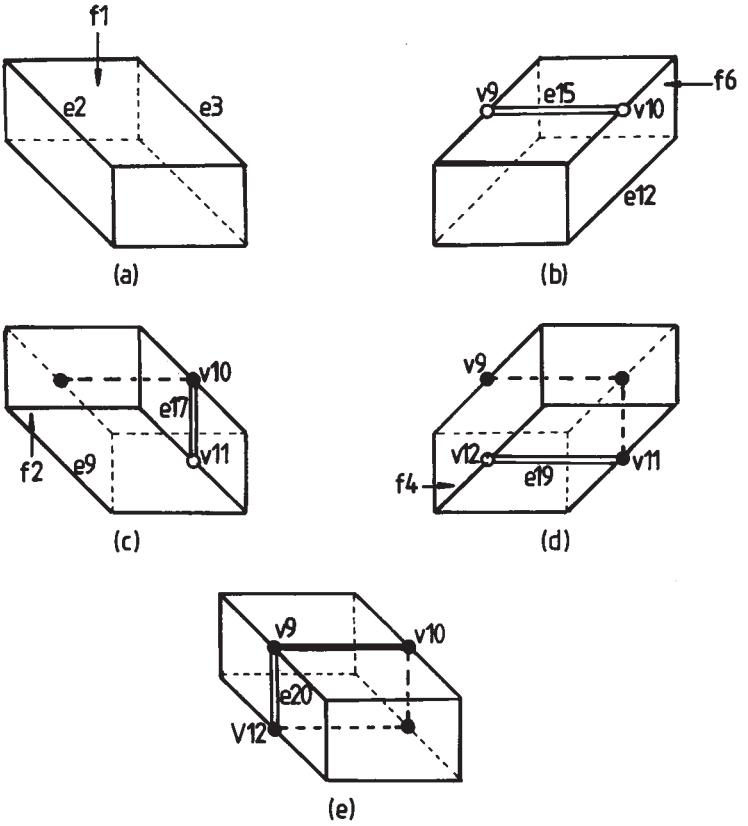


Figure 6.43: Creating the bend seam

exist already, vertex 12 and vertex 9. An edge is inserted between them, and, because vertex 9 was the starting point, the bend seam is complete and the process terminates.

This kind of 'nose following' sequence, where the intersections with one face are used to indicate the next face to be intersected, is also used in the algorithm used for splitting objects with a planar surface, described in section 6.8.

Once the bend seam has been inserted, another adjacent set of edges and vertices is inserted. These two edge rings, the original bend seam and the adjacent set, divide the object into three parts, as shown in figure 6.44. One part is the part of the object that moves; the second part is not changed; and the third part, which is between the original bend seam and the adjacent one, is where the bent part of the object appears. The adjacent seam is inserted

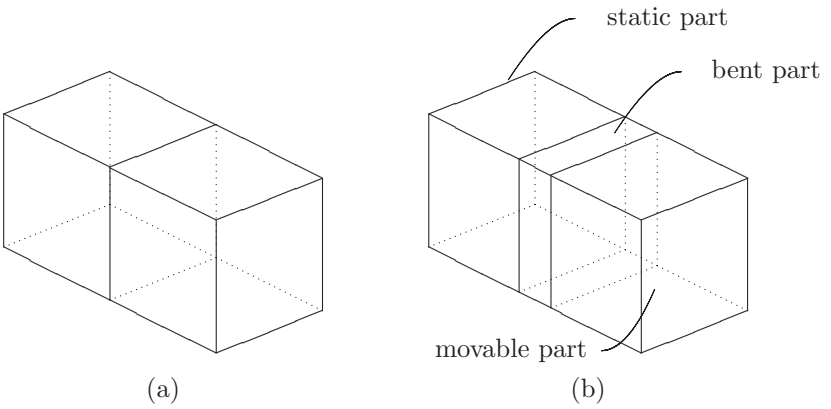


Figure 6.44: Doubled bend seam

as a sort of 'offset' from the first bend seam, touching it if any of the edges coincide with or touch the bend axis. Where an edge in the bend seam lies on the bend axis, no new edge is needed. If one end vertex of an edge in the bend seam lies on the bend axis, then that vertex is used as an end vertex of one of the edges in the adjacent seam, and a three-sided face is made. Otherwise, an edge is inserted that is logically parallel to the bend seam edge, but geometrically coincident, enclosing a four-sided face.

The bend seams are the only changes to the object topology. The geometry of the part of the object to be moved is changed, starting at a face specified as input and working back to the bend seam. The geometry of this part of the object has to be rotated about the bend axis by the bend angle (specified as input to the operation). Each face is modified separately, transforming the surface of the face, the edges bounding the face, and finally the end vertices of the edge. Obviously, each edge, except wire edges, will be encountered twice, and the end vertices will be encountered several times, depending on the number of edges at each. To avoid modifying them more than once, marker bits can be used. Before starting, all faces, edges, and vertices are marked. The first face, the one specified in the command as being in the part of the object to be bent, is then processed. The marker bit of the face is cleared, and the surface is modified. For each edge bordering the face, if it is marked, the marker bit is cleared and the curve is modified. If the edge was marked, then for each end vertex, if marked, the marker bit is cleared and the position is modified. Also, for marked edges, the face on the opposite side is found, and if not marked, this same transformation process is applied to the face. As described above, the edges in the bend seam are marked so that they can easily be recognised during the traversal. If an edge in the bend seam is encountered, its curve and end vertices are modified, as before, but the faces

on the opposite side are not processed.

The faces between the edges in the two bend seams are assigned geometry in a separate step. The appropriate surfaces are rotational surfaces (e.g., cones, cylinders, spheres, toroids, planes, and rotational free-form surfaces) and can be defined in the same way as the surfaces generated when swinging a face.

6.7 Giving sheet objects thickness

This operation is for converting the degenerate type of model described in chapter 5 into a volume model. It performs a sort of offset process, going over the model offsetting the internal faces and creating small side faces at the 'sharp' edges that lie between the 'sides' of the model. However, it is not a general offsetting process because it is not intended to cope with the removal of faces where the process offsets past them. It is intended to create thin-plate volumetric models from degenerate representations. An example of the process is shown in figure 6.45. The original object is shown in figure 6.45, left, and the expanded object is shown in figure 6.45, right.

6.7.1 The basic algorithm

The algorithm first changes the topology of the model and then modifies the geometry to produce the final object.

The first step in the algorithm is to examine all edges in the object, marking the sharp edges with one marker bit and the internal edges with another bit. This is done because the algorithm deals with each edge separately, and the original faces, edges, and vertices in the object have to be marked to distinguish them from new elements added as part of the process. The original vertices in the object are also marked so that they can be checked easily to see whether at least one edge at the vertex has been split.

If the edge is an internal edge, then no topological changes are made and the edge is left alone, for the moment. If the edge is a sharp edge, then it represents a degenerate face and has to be 'sliced' to produce an actual face in the expanded volumetric model.

The expansion process for converting a sharp edge into a face involves creating a degenerate face with two edges. However, when one or both of the end vertices of the edge have edges that have already been sliced, then these vertices are also split. Slicing the edge is done as a special case of MFE (making a face and edge), as described in Appendix F, section F.5. The relevant marker bits of the end vertices of the edge are also cleared to indicate that an edge at the vertex has been sliced. The geometry of each new degenerate face is added when the face is created. Arbitrarily, here, the surface is defined to be a ruled surface generated by sweeping a straight line along the curve of the edge. (The direction of the straight line is defined by

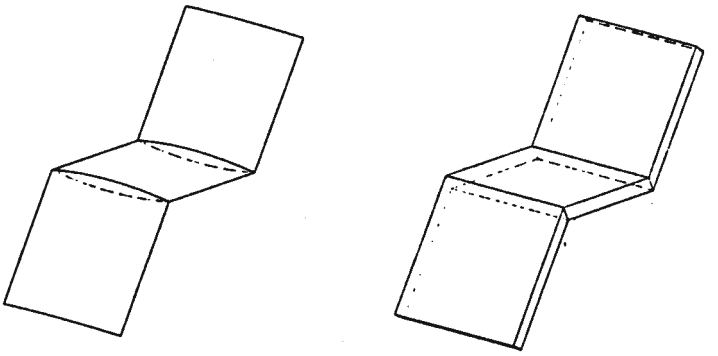


Figure 6.45: Offsetting a degenerate model

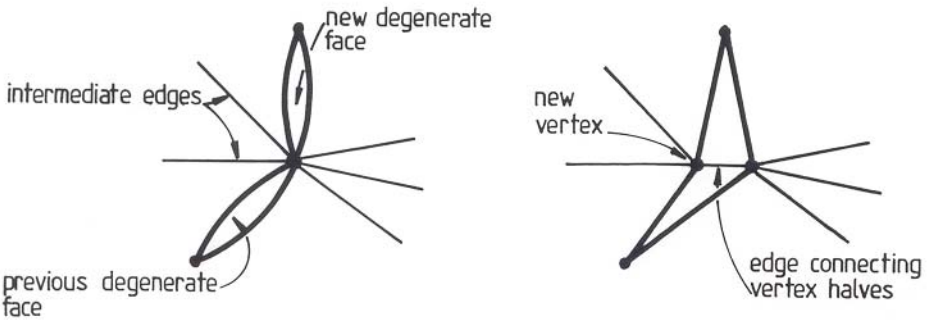


Figure 6.46: Splitting a vertex

the normal vectors to the surfaces meeting at the edge.) The new edge is assigned a copy of the curve of the original edge, the curves being modified with the rest of the geometry in the second step.

If an end vertex of the edge is adjacent to at least one other edge that has been sliced, indicated by an unset marker bit, then the vertex must be split. Vertex splitting involves creating a second vertex and edge linking the two vertices, and then transferring some edges from the first vertex to the new vertex. There are several equivalent ways of doing this, depending on which edge is used as a start edge and how the edges at the original vertex are traversed. What is described here is one way of doing this. If the vertex being split is the start vertex of the edge, then the new edge is taken as the start edge; otherwise, the original edge is used. The start edge and all edges counter-clockwise from it around the vertex up to and including one half of a sliced edge are then moved to the new vertex. The right-loop pointer of

the edge connecting the two parts of the split vertex is set to be the loop of the newly created degenerate face. The left-loop pointer is set to the loop of degenerate face bounded by the last edge moved. See figure 6.46.

Once all 'sharp' edges have been processed, the geometry is changed, starting with the vertex positions, then the curves of the edges, and finally the surfaces of the original faces. The new vertex positions are calculated using the normals of the surfaces meeting at the vertex with the real offset value given as a parameter for the process. However, if the vertex is adjacent to newly created faces, then the surface normals of these are ignored. Once the new vertex positions have been calculated, the curves of the edges in the object are modified. The new curve is defined to be a curve offset from the original curve passing through the newly calculated endpoints of the edge. Finally, the surfaces of all faces, except the newly created degenerate faces, are offset, calculating the new surface as being offset from the current surface and passing through the point of one vertex adjacent to the face.

As a final step, it may be necessary to check for self-intersections. Here the faces in what was one 'side' of the object are checked only against each other, not against offset faces in the other side of the object. For example, if the sheet object was originally closed, then the faces on the inside of the object are checked against the other faces on the inside, and the faces on the outside of the object are checked against other faces on the outside of the object.

6.8 Planar sectioning

The operation sections a given volume or sheet object with a plane and returns both parts of the object as a group of objects. An example of an object sectioned twice is shown in figure 6.47.

6.8.1 The basic algorithm

The process involves traversing the object, creating each section seam separately. The section seams are produced using a simple 'nose following' technique similar to that used when creating bend seams (see section 6.6). This has the advantage that degenerate models, as described in chapter 5, can be treated as 'locally manifold' while creating the section seams, and the degenerate faces collapsed, rather than handling this type of model with a completely separate operation. Once all section seams have been created, the resulting faces are checked to see whether any are to become hole-loops in others, and finally the objects are pulled apart into separate shells.

The section seams are created as a double seam, gradually 'unzipping' the object as the seam is created. Once the final edges in the section seam are sliced, the object has been separated into two unconnected pieces that are pulled apart later by traversing connected shells, moving the elements into

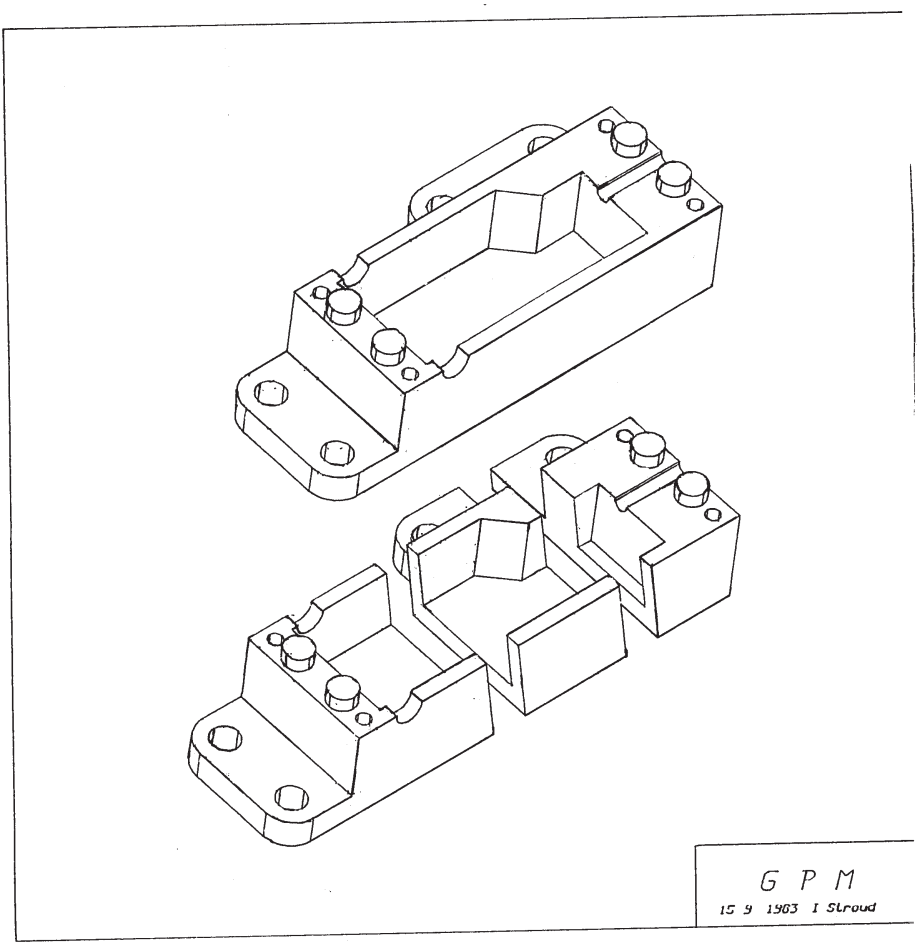


Figure 6.47: Planar sectioning

new objects.

Creating the section seams

To start with, all faces in the object to be split are marked. One reason that this is done is because as each part of a section seam is found, it is followed until a complete double section seam separating parts of the object has been created. Therefore, to avoid processing faces twice, the original faces are marked, and unmarked faces found during the section seam creation are ignored. Another reason for marking the original faces is because new faces are added to the datastructure during the sectioning process, either because a face has been split or by completing the section seams.

Next, all marked faces in the object are traversed, checking to see whether they are sectioned by the given section plane. The actual process of sectioning a face with a plane is very similar to that of comparing faces in the Boolean operations. The section surface is intersected with the surface of the face to produce a curve (usually).

If the face lies in the section surface, then it is ignored because the faces surrounding it will add the relevant edges, if necessary, to the section seam. If the face is bounded only by convex edges and/or smooth edges (figure 6.48, top left), then it does not cut through the surface, a sort of modelling equivalent to a local maximum or minimum. If the face is bounded only by concave edges and/or smooth edges (figure 6.48, top right), then the perimeter of the face is, in effect, a complete section boundary. If it is bounded by a mixture of convex and concave edges (figure 6.48, bottom), then the concave edges will be on a section seam, but these will be found from other faces, and the convex edges will be ignored. If the face just touches the surface at a point, then it is ignored.

The concavity and convexity of edges depends on the relative orientation of the surfaces of the faces adjacent to the edge with respect to the edge direction. (See also section 3.2.5, number 8.) For a concave edge, the gap between the right face and the left face of the edge is less than 180 degrees. For a convex edge, the gap is greater than 180 degrees.

As with Boolean operations and bending, when the surface–surface intersection produces a curve, the curve is intersected back with the face to produce a series of intersection points. If there are no intersection points and the curve is closed, then it has to be checked to see whether it lies completely within the face. If it does so, then it forms a complete section boundary and can be dealt with at once. The object is then split along this boundary creating two section faces.

The usual case, however, is that the curve cuts the face boundary at several points or along existing edges. These are sorted into order using parameter values, new vertices are added where the curve cuts an edge at an internal point, and new edges are added between vertices adjacent in the face. All new edges are added to the face when it is processed. If there are several segments,

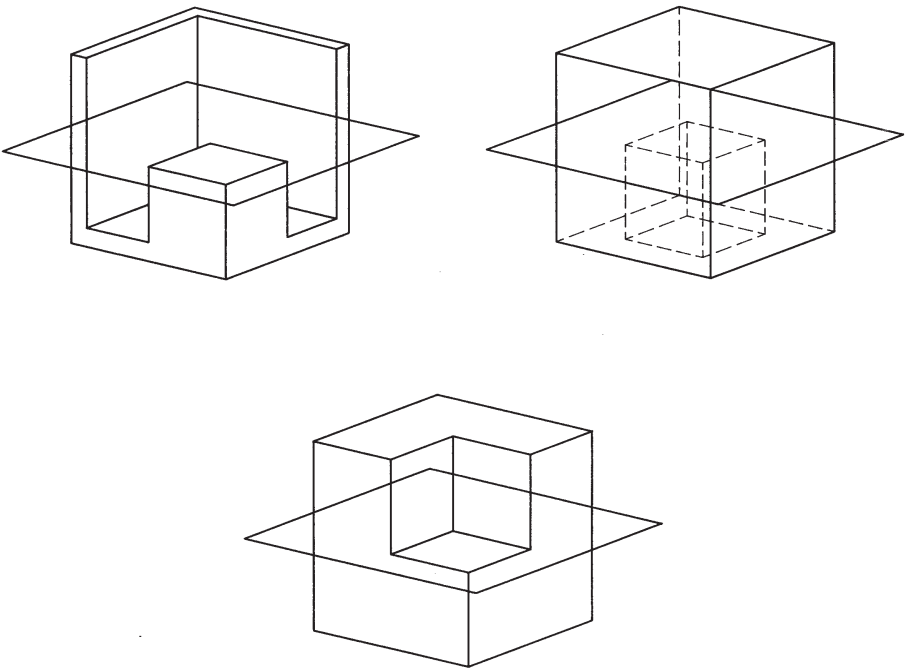


Figure 6.48: Convex-, concave-, and mixed-faces in the section plane

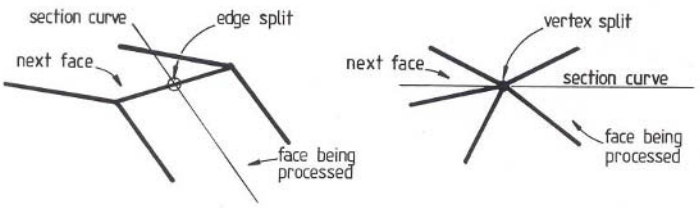


Figure 6.49: Finding the next face when creating the split seam

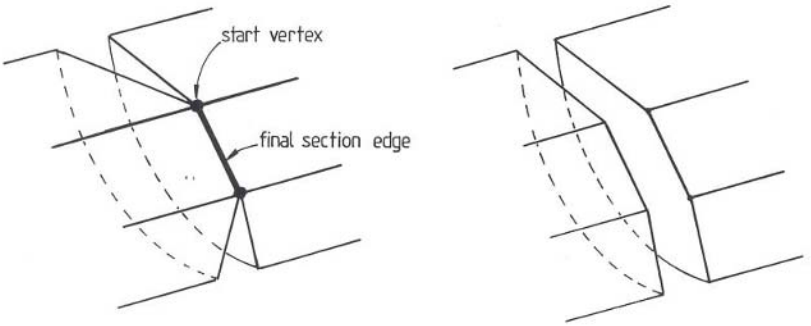


Figure 6.50: Slicing the last edge in a section seam

then these are processed as they are found. The first edge, between the two vertices with lowest parameter positions, is taken as the initial edge in the ring. The end vertex of this edge is used to find the next face to be processed. If the vertex was created by splitting an edge, then the face on the opposite side of either edge meeting at the vertex will be the next face to be processed (figure 6.49, left). If the vertex was an original vertex in the object, then it is necessary to search for the next face for processing using a geometric test (figure 6.49, right).

The searching process involves starting with the one of the edges at the vertex that is adjacent to the current face, and then working around, clockwise or counter-clockwise, until the next face to be split is found. For any face, if the two adjacent edges of the face are on opposite sides of the section plane, then that face will be the next to be processed. If an edge coincides with the section surface, then it is only interesting if the faces it separates lie on opposite sides of the section plane. If this is so, then the object cuts through the section surface along the edge.

The edges in the seam are sliced to produce two coincident seams, one on one side of the section surface and the other on the other side. When an edge

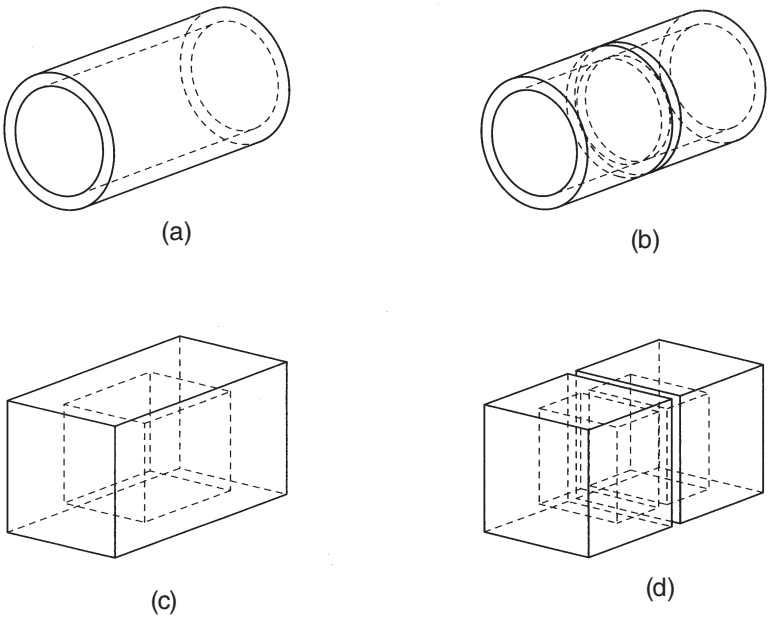


Figure 6.51: Sectioning tubes and objects with cavities

in the section boundary has been created or identified, it is sliced. For the first edge in the seam, the edge is sliced (as described in Appendix F, section F.5), producing a degenerate face bounded by two edges. If the edge is not the first edge in a section boundary, then one end vertex is also split, and the previously created face is extended to include the new seam edges. For the final edge, the vertex at the other end is also split, creating two complete, separate loops of edges (figure 6.50). These two separate loops bound two new section faces, which are recorded for later processing.

Once a section seam has been completed, processing of all faces in the object continues. If several new edge segments are added to the face, then these should be found later, as parts of the same section seam or of different section seams.

Reorganising the section faces

Once all faces have been processed, there is a set of separate section faces. It is possible that some of these should be hole-loops in other faces instead of actual faces. Figure 6.51 illustrates two such cases. The object shown in figure 6.51a is a sort of tube. When split, it produces two tubes, as shown in figure 6.51b. The object shown in figure 6.51c has an internal cavity, or void. When split, it produces the two objects shown in figure 6.51d, each with part of the void.

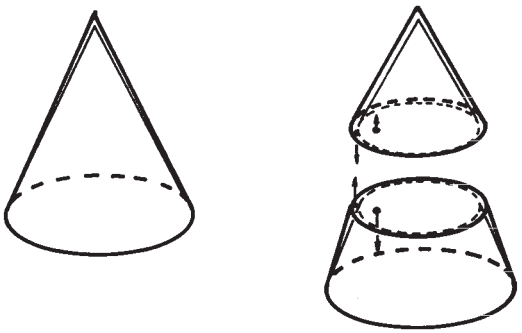


Figure 6.52: Sectioning through a conical degenerate model

Rearranging the loops involves, simply, going through the list of section faces, checking to see whether the loop of one face is contained in another. However, this is only relevant for solids. If the original object was a degenerate model, then the section faces will often have no area, and should be collapsed back to edge sequences. Also, obviously, the two faces produced from a single section seam should not be compared, only faces produced in different steps. If the object was not self-intersecting, then all faces should be either completely contained in another or completely outside. The faces should be compared and ordered on the basis of their area, so that the largest faces are checked first.

If the object sectioned was a degenerate model, then straightforward geometric tests, such as the loop-in-face test used for volumes, cannot be used. The presupposed representation of degenerate models (see chapter 5) uses different main facegroups for each side of the degenerate model. If a section seam is only adjacent to faces on one side of the degenerate model, then a matching seam adjacent to faces on the other side is required to match the first seam. If a section seam is only adjacent to faces on one side of the degenerate model, then the seam should enclose a face with area greater than zero. The matching seam will enclose a face of the same area but oriented in the opposite direction. See figure 6.52. Also, as with the object shown in figure 6.52, the outer loop of edges will be convex, and the inner loop will be concave. The loop containing the concave edges is transferred to the matching face, and the face it surrounded is killed. Once any matching faces have been coalesced, the section faces have to be collapsed into edge sequences, as described in section 6.3.

Separating the sectioned object into new shells

Once the faces have been sorted out and the loops assigned correctly, the separate object shells have to be pulled apart into new objects. This is the same basic process as that for the Boolean operations, described in section 6.1.

First, for example, all original vertices in the object are marked with some marker bit. Then, one vertex is chosen as a starting point, and all connected vertices are unmarked using the topology traversal method described in section 3.2.1. Any vertices still marked after the traversal belong to a separate shell and are moved into a different object using the same traversal procedure with a marked vertex as a starting point. The process is repeated until no marked vertices are left in the original object. As the process depends only on topology, it is the same for both degenerate models and for volumes.

Again, as for the Boolean operations, it is possible that internal cavities, not split by the operation, have to be handled. These will be separated out into separate objects by the above procedure, and so they have to be remerged with the correct object part. The simplest way of doing this is to perform a 'point in body' test, choosing a point on each body and comparing it with likely candidate objects to see whether it lies inside. The test only need be carried out if the original object contained multiple shells, i.e., cavities; otherwise, it is not necessary.

6.9 Imprinting

The imprinting algorithm is for imprinting the projection of a shape onto a face as illustrated in figure 6.53. Basically, imprinting involves defining the intersection between a face and a 'virtual volume', implicitly defined by a two-dimensional shape and a sweep direction. The extent of the sweep is, at least, the greatest distance between the face and the shape, although a larger distance is advisable.

6.9.1 The basic algorithm

The algorithm imprints each edge in the lamina separately, so it could be used for imprinting wireframe models or solids as well, in terms of their individual edges.

The basic process imprints an edge using four surfaces. One surface is the surface generated by the edge as it is swept in the direction specified for the imprint. The second surface is the surface onto which the edge is to be imprinted. The other two surfaces are generated to delimit the extent of the imprint. They are defined as planar surfaces through the endpoints of the edge. The plane normal is defined using the tangent to the curve of the edge at the endpoint and the projection direction. If P is the projection direction

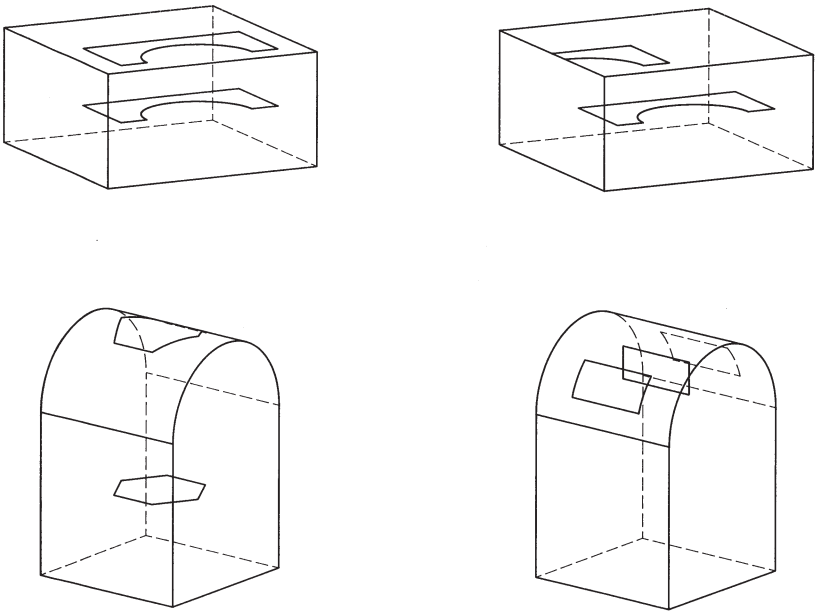


Figure 6.53: Examples of imprinting on faces

and T is the tangent direction at an endpoint, then the plane normal is defined as:

$$(P \times T) \times P$$

Figure 6.54 illustrates the surfaces generated from the edge to be imprinted.

The steps involved in imprinting the edge are similar to those for the face–face comparison in Booleans operations. The first step is to intersect the swept surface with the surface of the face to be imprinted, producing, normally, a curve. The two bounding surfaces from the edge define which portion or portions of this curve correspond to the edge. The curve is intersected with the face being imprinted to find intervals that are inside the face. These two sets of intervals are then compared in the same way as when comparing curve intervals in two faces in Booleans. The common intervals are then inset in the face as new edges.

Several cases can occur when intersecting the surface generated by the edge and the surface of the face. If there is no intersection between the swept surface and the face surface, then the edge does not produce a corresponding imprint in the face, and no more work is necessary. If the surfaces intersect in a point, then the intersection is also ignored, as is the case where the surfaces are coincident.

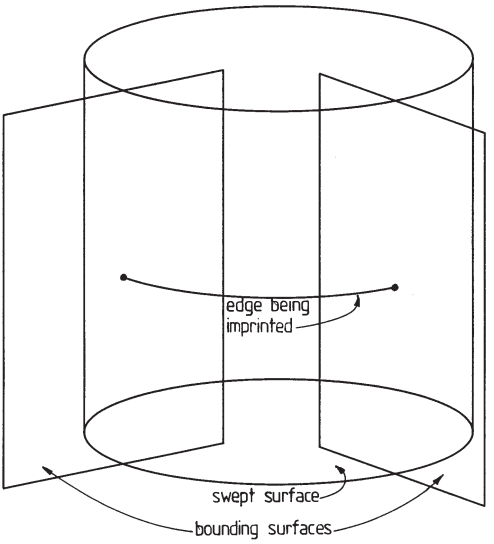


Figure 6.54: Surfaces used for imprinting

When there is an intersection curve, the curve is intersected with the bounding surfaces, generated from the edge being imprinted, to find which portions are of interest. The curve can cut neither, one, or both surfaces. If it cuts neither surface, then it can lie between the two surfaces, i.e., within the 'virtual face' swept out by the edge, or outside it. One case where the curve cuts neither surface, but lies between them, is illustrated in figure 6.55a. A case where the curve cuts only one surface is illustrated in figure 6.55b, and a case where it cuts both surfaces is illustrated in figure 6.55c. Figure 6.55d illustrates a case where there are two significant curve intervals for comparison with the face. The crossing points are ordered into a set of pairs where the curve enters the space between the surfaces and where it leaves, inserting artificial upper and lower bounds where necessary.

The curve is then intersected with the face in exactly the same way as described in section 6.1. The face–curve intersection process produces a set of results where the curve enters, leaves, touches from the inside, touches from the outside, cuts a wire, or coincides with an edge. These are then compared back with the first set of intervals (found by intersecting the curve with the bounding surfaces) to produce a set of one or more edges where the edge is imprinted on the face. Each result denotes the start or end of an interval. If the result is of type 'ONEDGE', where the curve coincides with an edge in the face, it might both end and begin an interval. Likewise, if the curve cuts a wire or touches an edge in the face, the same intersection vertex might be the end of one interval and the start of the next.

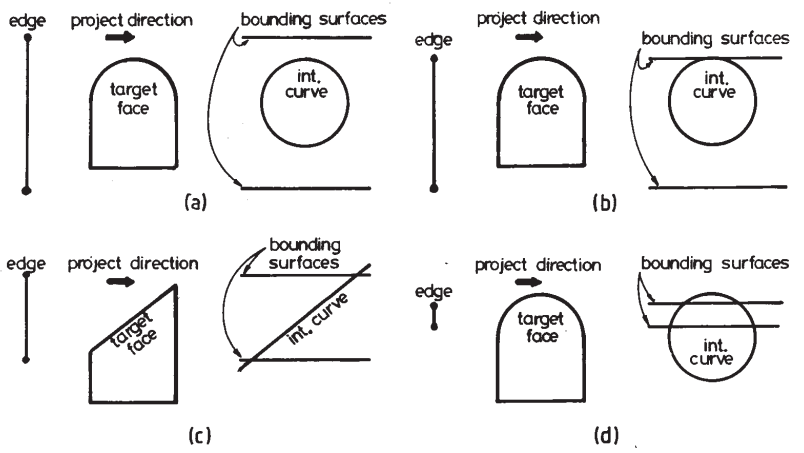


Figure 6.55: Various cases for imprinted curves

The edges are added as Eulerian wire edges (with the same loop on the left and right sides) or as edges that complete faces. This differs from the algorithm for Booleans, where the edges in the intersection boundary were always added as unconnected wire edges, and then joined together in a separate step. For Booleans, adding new faces while creating the intersection boundary is complicated because the face structure is used to traverse the datastructure, and adding new faces during the traversal can complicate this traversal process. For imprinting, only one face is being modified, but it is necessary to 'remember' the new faces so that they are imprinted as well as the original face. This is because the basic operations do not guarantee which portion of the original face is the new one and which one is the old one. Also, without performing an unnecessary and complicated geometric check, it is not possible to know which of the sub-faces will be imprinted by subsequent edges. Therefore, as new faces are added, they are chained together, and the imprinting process is applied to each new face. The exact method of keeping track of the new faces is unimportant, but some method is necessary unless all new edges are added as wire edges and connected up after all edges have been imprinted.

Once all edges in the original object have been imprinted, the process is complete.

6.10 Creating the dual of an object

Creating the dual of a solid model is an operation that takes a solid model and creates a new model with vertices corresponding to faces in the original model

and faces corresponding to the original vertices. It is interesting because it illustrates how the original model is used as a 'map' to guide processing. However, the operation is rarely useful directly in practice because it is a very specialised transformation. Also, as described later, assigning geometry to the transformed model is problematical.

6.10.1 Constructing the topology of the dual

The creation of the new topology is similar in many ways to how disc files are written and read in BUILD, using entity numbers to find and link topology. The entity numbers are used to access array elements containing the new dual model elements. The first step is to count the numbers of faces, edges, and vertices in the original object. Integer arrays of these sizes are then set up containing the original numbers of the entities so they can be reset later. The faces, edges, and vertices of the original object are then renumbered. Next, the topological entities for the dual model are created and put into the appropriately sized arrays. There is one vertex for every loop of every face in the original object, one edge for each edge of the original, and one face for each vertex. Finally, the edges in the original object are traversed, and the new topological entities are linked together using the original topology to find the new topology. This is, perhaps, better explained using an example.

To create the dual of a cube, the vertices are first renumbered consecutively 1 to 8, the edges are renumbered to be 1 to 12, and the faces and loops are numbered 1 to 6. Figure 6.56, left, shows the original cube and figure 6.56, right, the dual (with numbers corresponding to array elements). Arrays containing the new topology are then created, containing 6 vertices, 12 edges, and 8 faces. The first edge to be traversed is edge 1. Edge 1 has vertex 1 as start vertex and vertex 2 as end vertex, with face 1 as left face and face 3 as right face. The new edge, in the first element of the new edge array, is connected so that it has the vertex in vertex array element 3 as start vertex and the vertex in vertex array element 1 as end vertex. The face in face array element 1 is set as the left face of the new edge, and the face in the second array element is set as the right face. The winged edge pointers are also set in a similar manner. The edge clockwise from edge 1 around the perimeter of face 1 is edge 2, so the edge clockwise around dual vertex 1 from the first dual edge is set to be new dual edge 2. The edge counter-clockwise from original edge 1 around the perimeter of face 1 is original edge 3, so the edge counter-clockwise from dual edge 1 around dual vertex 1 is set to be dual edge 3. Similarly, dual edge 5 is set as the edge clockwise from dual edge 1 around dual vertex 3, and dual edge 6 set to be the edge counter-clockwise from dual edge 1 around dual vertex 3. See figure 6.57. The process is repeated twelve times until all edges in the original object have been set up. Once the relationships between the edges have been set up, the final connections between loops and edges, and between vertices and edges, can be made easily.

Duals of objects with hole-loops can also be handled in the same way,

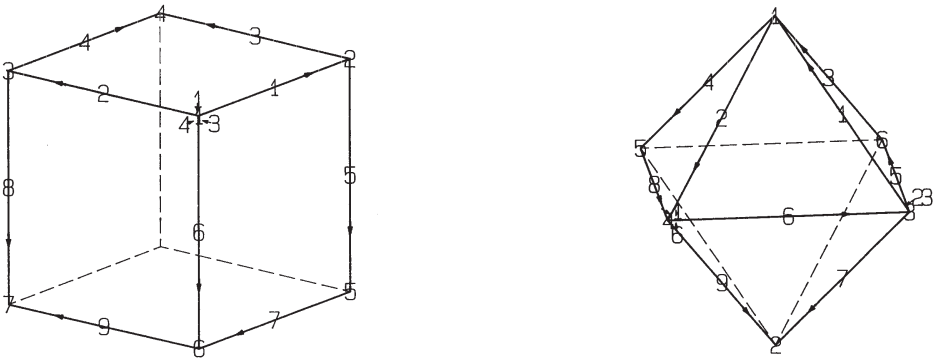


Figure 6.56: Cube and dual

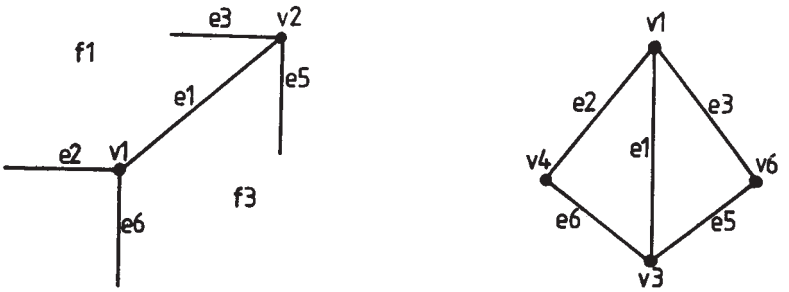


Figure 6.57: Dual of 'edge 1' in a cube

although the meaning of duals of non-Platonic solids in modelling terms is not entirely clear. With a standard B-rep datastructure, hole-loops are dealt with by creating a new vertex for the hole-loop and connecting the duals of the edges surrounding the loop to this vertex. However, a problem with this is that it is impossible to tell, on purely topological grounds, between the dual of an object with a hole and the dual of an object with a cavity. A better solution is possible if a datastructure that allows multipli-connected vertices is used, a multipli-connected vertex can then be used as the dual of a multipli-connected face. Each loop in an original face corresponds to a separate edge set in the dual vertex, rather than to a separate vertex.

Wire edges and spur vertices, where there is only one edge at a vertex, also cause problems. Spur vertices imply dual faces with only one edge. Wire edges imply edges with the same start and end vertex. It is debatable whether the algorithm should cope with these. From one point of view, it is better to avoid finding duals of objects containing extraneous elements such as wire- and spur-edges. From another point of view, it may be necessary to be able to apply the dual twice and restore the original topological structure. The duals of some objects may contain wires or spurs, and so it may be necessary to cope with these if the operation is to be self-inverse. However, the problem is also connected to the original design of the modelling system. For example, if edges with the same start and end vertices are not catered for by the system, duals of objects with wires or spurs may cause problems for other parts of the system.

Similarly, faces surrounded by one or two edges give rise to vertices with two or only one edge in the dual. Again, the possibility of these types of faces is connected with the original design of the system. As a specific example, there are several ways to represent cylinders in modelling systems. One type of representation has three faces; the two planar end faces are bounded by single circular edges. The object and dual are shown in figure 6.58, top. Another representation has a single wire edge connecting the two vertices. This and the resulting dual are shown in figure 6.58, middle. A representation with three curved side-faces separated by 'fake' edges and its dual are shown in figure 6.58, bottom.

Once the traversal of the edges is complete, and the dual structure has been built, the original numbers of entities in the model can be restored from the arrays in which they were stored so that the object is unchanged after the operation.

6.10.2 Creating the geometry of the dual

The dual of an object is mainly of interest from a topological viewpoint. However, when creating Platonic solids and duals of other objects with regular topology and geometry, it is possible to assign some geometry to the dual model. In general, if the object is convex, all faces have only one loop; there are only straight edges and trihedral vertices; then the geometry of the dual

can be made so that the dual is also geometrically sound.

The centres of the faces can be used as the positions of the corresponding dual vertices. The surfaces of the dual faces can be calculated from the original vertex positions. If the vertices in the original object are all trihedral, then the faces of the dual object are all three sided, and planar surfaces for these can be calculated. If the original object contains vertices with more than three edges, then the corresponding dual faces will have more than three sides, which may not all lie in the same plane. If there are vertices that have fewer than two edges then the resulting faces will be degenerate.

If the original faces are not convex, then the geometry of the dual is harder to calculate so that it has meaning. The centre of the face is not really appropriate as a vertex position for the vertex dual of the face.

If the faces have more than one loop, then the dual will be non-manifold, which, if separate vertices are used as the dual of each loop, may mean that there are two vertices at the same position, and may mean that the result is self-intersecting. If a dual representation using non-manifold vertices is used, then care has to be taken when arranging the geometry of the edges in each edge set. Even if all edges in the different edge sets are made straight edges, then there is still a risk that the resulting object is self- intersecting.

If there are curved faces and/or edges in the original object then the topology and geometry of the dual may be strange (figure 6.58). The dual operation was developed in BUILD to perform the very limited function of creating one Platonic solid, the dodecahedron, which was easier to create in this way. As such, no great attention was paid to what should be the geometry of the dual object. If there are wire edges in the original object, then the resulting dual edges have the same start and end vertices, so it should logically be assigned closed curves. The duals of the cylinders in figure 6.58, middle and bottom, can be represented as two back-to-back cones if different geometry is assigned to the dual model.

6.11 Setting a surface in a face (SETSURF)

Setting a new surface in a face was an operation developed by Graham Jared for the BUILD system and is described in Jared and Stroud [65]. It was used both for making local changes (tweaks) by modifying an existing surface and for adding free-form geometry to a model (see chapter 13) or other geometry replacement. These uses are illustrated in figure 6.59.

6.11.1 The basic algorithm

SETSURF works by recalculating the curve equations of the edges surrounding the face and the positions of the vertices adjacent to the face. In the basic algorithm, there is also the restriction that there should be no topological changes to the model. Extensions to the basic algorithm are discussed in

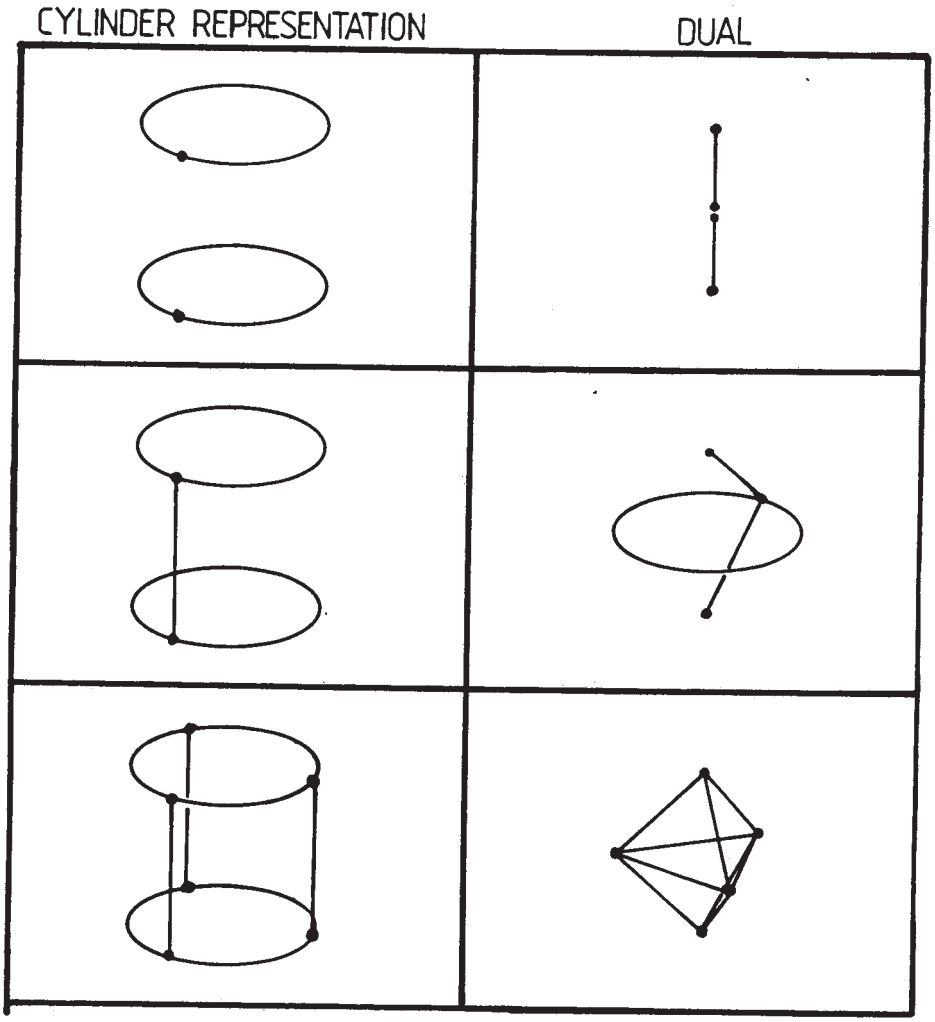


Figure 6.58: Cylinder representations and the corresponding duals

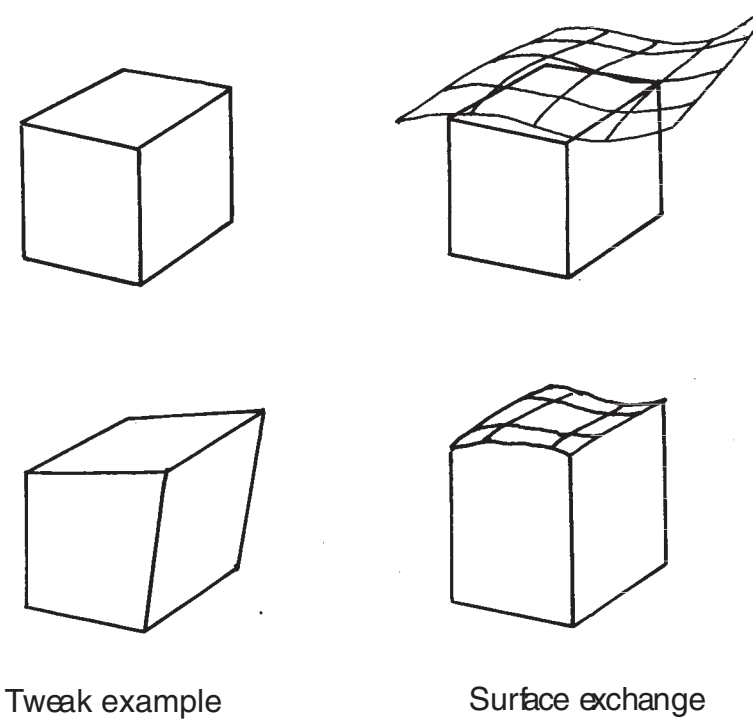


Figure 6.59: Examples of the use of SETSURF

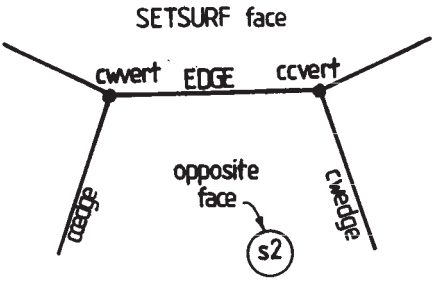


Figure 6.60: Topological arrangement of an edge

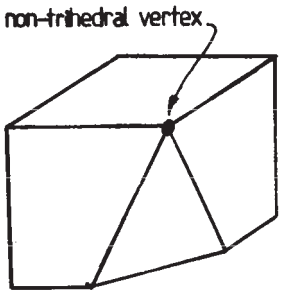


Figure 6.61: Non-trihedral vertex

section 6.11.3.

The curves are recalculated edge by edge for all loops bounding the face. There are two variants: 1) the new curves are stored in a general group, or 2) the new curves are calculated and assigned to the edges, while the old curves are inserted into a general group. The old curves are needed in case an error is detected while recalculating the curves. If the surfaces do not intersect, then the whole operation fails.

Figure 6.60 illustrates the topological arrangement around an edge. The new curves are simply the intersection between the new surface and the surface of the face opposite the current loop across the edge, surface s2. Similarly vertex positions are recalculated by intersecting the curves of the side edges, ccedge and cwedge, with the new surface. Note that if the vertices around the face have more than three edges attached, then topological changes will be necessary. Figure 6.61 illustrates a simple example where the SETSURF operation would require such topological modification, so it is excluded from

consideration here. It is also necessary to check that the new vertex position lies on, or extends, the side edge. The edge is not allowed to become of zero length because the new vertex position and the position of the vertex at the opposite end of the side edge coincide.

If everything is successful, the old geometry is deleted and the new geometry is associated with the edges and vertices around the face, and the face is modified to refer to the given surface. If no new curve equation can be calculated for an edge, or no new vertex position can be calculated for a vertex, or there is a topological change, then the operation fails and the new geometry is deleted, leaving the face unchanged.

6.11.2 Special cases

Care must be taken with some types of geometry. If, for example, a side intersects the new surface in several distinct curves, or the side edges (ccedge and cwedge in figure 6.60) intersect the new surface in several places, then it is necessary to perform some calculations to select the correct replacement geometry. Consider the simple example of replacing a planar surface with a cylindrical surface, as illustrated in figure 6.62.

There are two possible results of the operation, depending on which set of geometry is used; hence, some way must be found of selecting which result is desired. To do this the surface normals of the cylinder and face can be compared. If the cylinder is normal, then the result is as on the left. If the cylinder is negated, then the result is as on the right.

A more difficult problem occurs when the surface being set into the face is a free-form surface that bends back on itself two or more times (figure 6.63). In this case, there can be more than one surface portion where the surface normals and the face normals approximately agree. Surfaces like this should be broken up in some way into simpler portions before trying to set them in a face.

6.11.3 Extensions to the algorithm

Some modificatons can be made to allow the SETSURF operation to make limited topological modifications.

The simplest topological modification is where edges or faces disappear, as in figure 6.64.

Edges disappear when the start and end vertex positions coincide (more complicated cases will be considered later). Disappearing edges are removed by merging the coincident vertices, a variant of the “Kill Edge and Vertex” Euler operator that is described in Appendix F section F.11.

Once the zero length edges have been removed, degenerate faces can also be removed. In the simplest case, these faces will be bounded by two edges lying in the same curve, but in general, it is necessary to be able to handle more complex cases in which a face is bounded by several coincident edges.

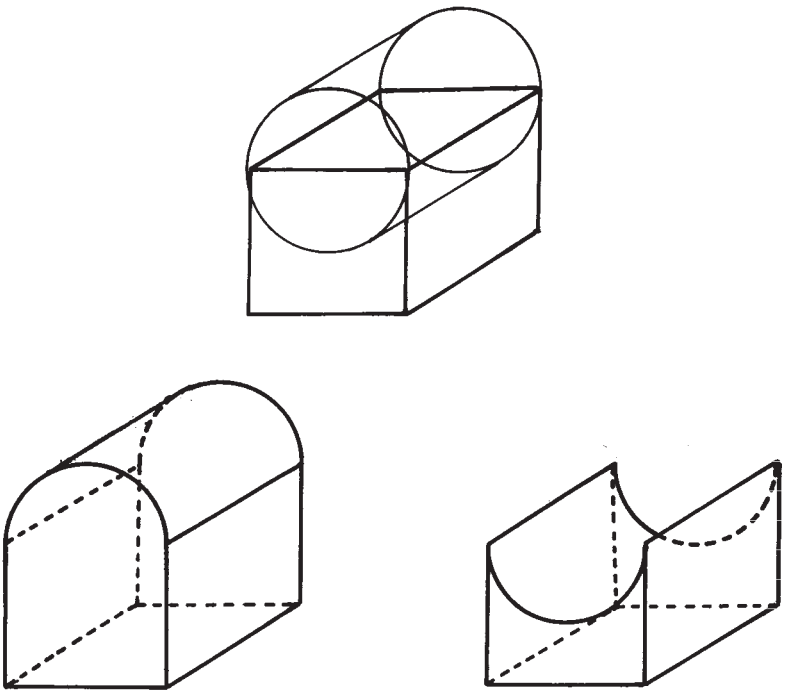


Figure 6.62: Replacing a planar surface with a cylindrical surface

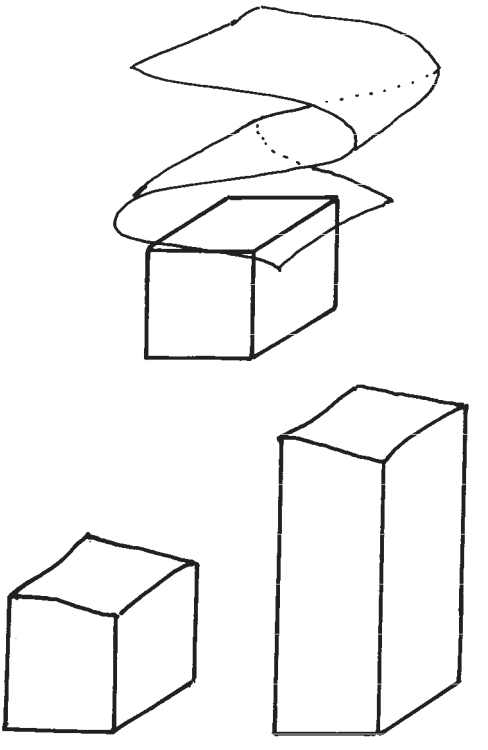


Figure 6.63: Setting a complicated surface into a face

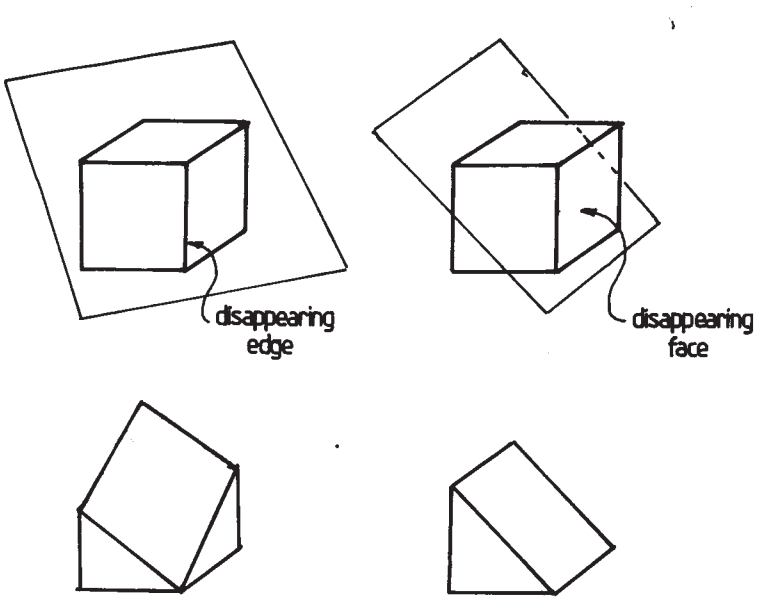


Figure 6.64: Disappearing edges and faces

It is also possible to extend the operation so that a local sectioning is performed and non-trihedral vertices are extended.

Sectioning is needed where an edge is undercut, that is, where the new vertex position lies below the vertex at the opposite end of an edge. The sectioning process would then extend the face boundary until the original face boundary can again be modified. However, this is complicated, and from the point of view of stability, it is probably best to avoid this kind of complication. Another important consideration is that of inversion. This means that the previous object can be recovered if the result of the operation is not what the user required. It is not impossible to make an invertible local sectioning operation, but it is not clear that this is a logical result of the operation.

Handling non-movable vertices is relatively easy, but the strategies for extending such vertices upwards is different from where the new surface interscts all edges at the vertex.

For upward extension, the changes are illustrated in figure 6.65. The surfaces (os1 and os2) of the faces adjacent to the face that is to be modified are intersected to get the new side curve. The non-trihedral vertex is then split, with the edges bounding the face being modified attached to one vertex and the other edges attached to the other. The edge between the new and the old vertices (which for the moment is zero-length) is assigned the new curve, and the new vertex is handled as a normal trihedral vertex.

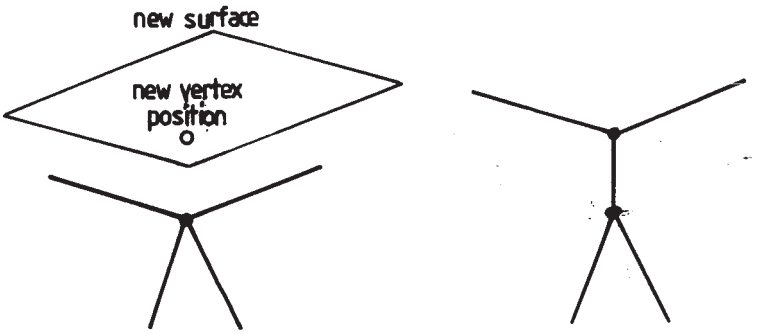


Figure 6.65: Extending a non-trihedral vertex

Figure 6.66: Undercutting a non-trihedral vertex

Figure 6.66 illustrates the case in which the new vertex position lies 'below' the current non-trihedral vertex position. The process is similar in effect in that trihedral vertices are created from the non-trihedral vertex by splitting, and then these are handled as in the basic algorithm. The vertex is always split between edges not adjacent to the face being modified. The curves of these new edges are irrelevant, because they will be assigned new curves in the modification process if successful, or will be left as zero-length edges that should be removed by merging their end vertices.

For all of these topological modifications care has to be taken to remove the extra topology if the operation fails at some point.

6.12 Drafting

Drafting is an example of a strange operation. Possibly the first implementation was in BUILD as an demonstration of the flexibility of B-rep compared with CSG. It is strange because the operation is linked closely with manufacturing, specifically mould making, rather than design.

The original drafting algorithm in BUILD worked on planar polyhedral objects. The effect of the operation is illustrated in figure 6.67, from [13].

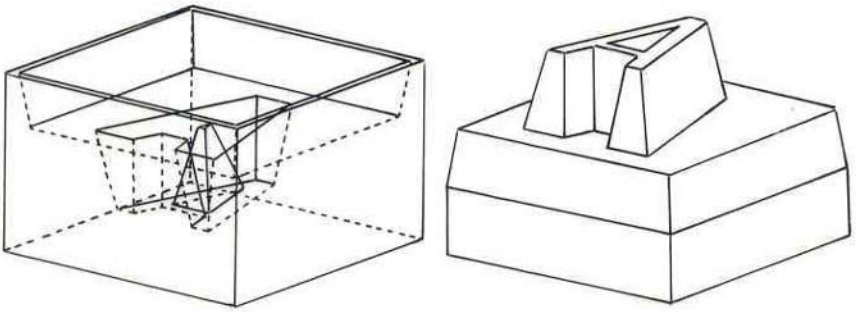


Figure 6.67: Object with draft angle

The input to the operation were a face and the draft angle. The basic procedure is as shown in figure 6.68. Each edge around the face was used as a rotation axis for rotating the surface on the opposite side of the edge from the given face. The surfaces were then put back into the face, and the new geometry was calculated.

Although this worked reasonably well for planar objects, there are some extra considerations for objects with curved surfaces. Figure 6.69, top, shows what happens when a draft angle is added to an object with blends in some commercial systems. The blends become conical. The bottom part of the figure shows what happens if the object has a draft angle added first and is then blended.

In the solution at the top of the figure, the blends change from constant radius to variable radius. While the draft angle is actually small, maybe half a degree, this is not really noticeable. However, the 'normal' solution, of changing cylinders to cones, seems inappropriate. It would seem more reasonable to rotate the cylindrical surfaces. However, doing so does mean that there is an intermediate step where the object will be in a funny state, where the rotated cylinder is tangential to one surface but not to the other. This may cause problems for the SETSURF command; hence, care needs to be taken when redefining the geometry of the side surfaces.

6.13 Gluing non-matching faces

This operation is an example of a 'local' specialised Boolean operation. The two faces being joined are intersected with each other to produce a set of sub-faces that either match completely or lie outside one or other of the input faces. The matching sub-faces are then joined using the process described in section 6.4.

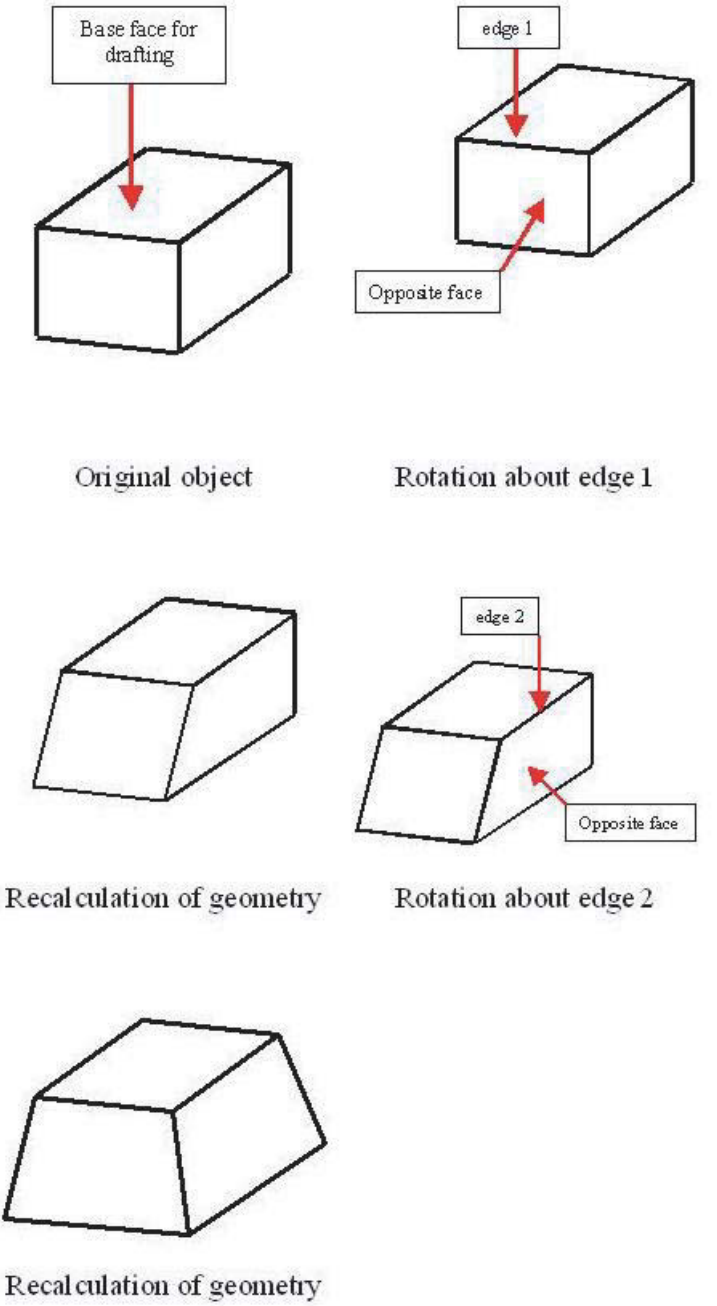


Figure 6.68: Adding a draft angle to a block

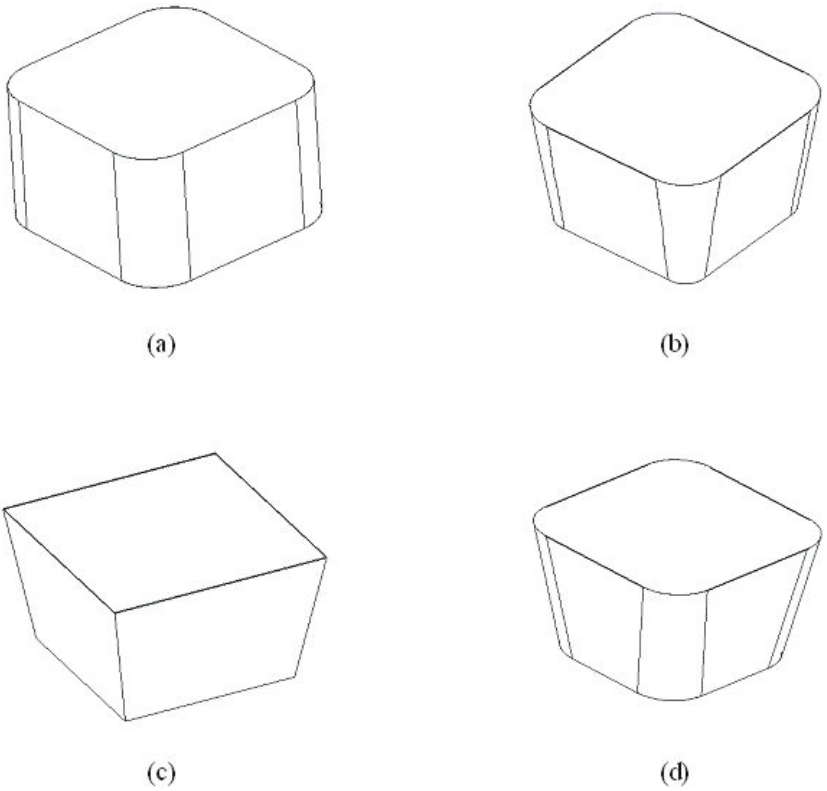


Figure 6.69: Adding a draft angle and blends

6.13.1 The basic algorithm

The complication with the operation is that the faces are modified as the operation proceeds. This means that it is necessary to use some trick to avoid a lot of unnecessary processing. This can be done by marker bits, but instead the operation preserves a representation one face. Once this is done, the other face is imprinted onto the first face, and finally, the preserved representation is imprinted onto the second face. The new faces generated during this procedure are preserved in some way and then matched and joined.

The first step, then, is to create a list of curves in the first face together with the start and end vertex positions, or start and end parameter values. At the same time, it is possible to isolate the faces into their own facegroups, which can be used as parents for all new faces created.

To imprint the second face onto the first face, all edges in the face are traversed, intersecting their curves with the first face. The same basic utility used in Booleans, for intersecting a curve with a face, can also be used here. The results from this operation can be post-processed, throwing away those intersections that do not lie on the edge. The others are analysed, and portions of the curve that lie within the first face are made into edges. When these new edges are added, it is likely that they will create new sub-faces from the original face. These new faces must also be intersected with all subsequent curves. Note that if the start or end vertex position of the second face edge lies inside the first face, then that position corresponds to a new vertex.

Once the first imprint is complete, the face copy, the set of curves from the first face, is imprinted onto the second face. This is done in the same way as for the first imprint; each curve in the curve set is intersected with the second face, intersections not lying on the relevant portion, which is determined from the start and end positions or parameter values; is removed; and new edges and faces are made. Note, also, that if start and end parameter positions from the first face are recorded, then care must be taken not to reparametrise the corresponding curves during the first imprinting. If reparametrising takes place, then the recorded parameter positions will be invalid; hence, it may be better to record the start and end vertex positions and recalculate the corresponding parameter values when they are needed.

Figure 6.70 illustrates the process of imprinting for two faces.

When both imprints are complete, the two faces should have been divided into a set of matching coincident portions and non-matching, non-overlapping portions. The final part of the operation is, therefore, to match the faces and join them. Matching the faces is done relatively simply by using face boxes, because the sub-face portions are distinct. Joining them is done using the method for joining matching faces, as described in section 6.4.1.

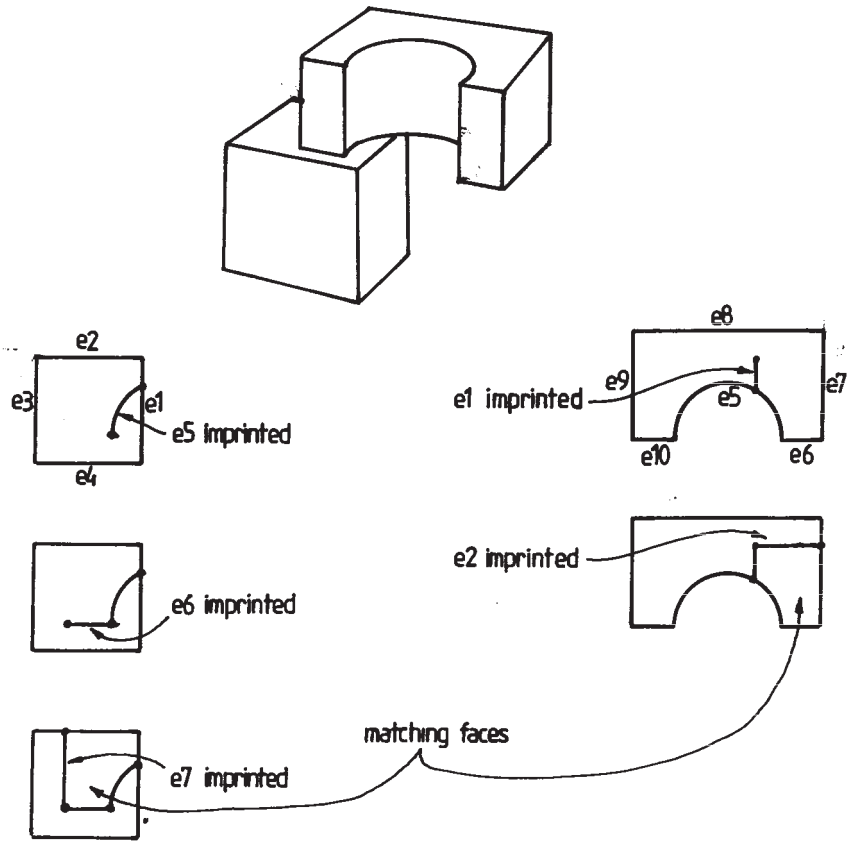


Figure 6.70: Imprinting a face pair

6.13.2 Special cases

One special case is where the boundaries of the two faces do not interact. In this case one of face is totally enclosed in the other, or completely outside it. It is possible to check for interaction while imprinting. If no interaction is found, then points on the outer loop of each face are chosen and tested to see whether they lie inside the other face. If one face is enclosed by the other then they can be joined very simply by blocking any inner loops and then making the outer loop a hole-loop inside the larger face.

Where one or both faces have hole-loops, it is necessary to check whether these hole-loops become blocked by the other face. This can be done in the same way as described above. If a loop is found that does not interact with any edge in the other face, then a point on that loop is chosen and tested to see whether it lies inside or outside the other face. If it lies inside the other face, then it is 'blocked off' with the "Make Face Kill Hole" Euler operation described in Appendix F, section F.13.

Trying to glue self-intersecting faces will cause problems. The imprinting method described above will create a non-self-intersecting imprint that will not match the corresponding self-intersecting face portion.

Chapter 7

Modelling operator definition

Chapter 6 outlined several modelling algorithms to illustrate the use of the modelling tools described in chapters 3 and 4 and general modelling methods. This chapter extracts some principles used for algorithm development. Chapter 8 explains how the addition of information can be done and some problems that occur when modelling with it.

One justification for trying to extract methods for developing modelling tools is for automating the process by providing the user with a sort of 'modelling toolbox'. Current modelling packages usually offer some kind of user interface, either as a subroutine package or through some kind of macro-language, but it is difficult to communicate the modelling techniques necessary to use these to near-maximum extent. If it is possible to provide the user with some kind of software help, then this should facilitate the process of integrating modelling systems into existing company software. In general, it is difficult for the modeller developers to foresee all possible uses for the modelling software and to provide tools for all functions. The people who know how the modeller is to be used are in the firms applying the software, but often they lack knowledge of how to do the things they want to do with the modeller they have. The special representations described in chapter 5 and the modelling tools described in chapter 6 illustrate the flexibility of modelling techniques. It is possible to do a great deal, but it is difficult to know what to do.

The basic steps involved in developing a modelling tool are as follows: definition of the requirements for the tool; decomposition of the task of the tool into smaller sub-components; and deciding how to handle special cases and error conditions. These are dealt with in the following three sections, and two examples of this process are given in the fourth section.

7.1 Definition of requirements

The first step is to define what the operation is to do. For local operations, this is, in general, to take one part of a model and transform it into something else. For example, the chamfering operation, implemented by Jared in BUILD [11] and by Bosser in the GPM volume module [15], takes an edge or vertex and replaces it with a face.

It is difficult to see how to automate the definition of requirements, or to make any kind of standard definition method that will cope with the general modelling tool definition. One method would be to take 'before' and 'after' models and compare them. However, this seems an unnecessarily clumsy way of defining local changes. Also, with operations such as the reflect operation (see section 4.5), the comparison would involve complex geometrical analysis to determine that the new object had a plane of symmetry. Both the chamfer operation and the reflect operation are local operators in that they use a limited portion of an object to specify a modelling operation. The effect of the reflect operator is somewhat larger than that of a chamfering operation, but they are both quicker and less complex than, say, Boolean operations. To start off with, therefore, the modelling operations will be divided into groups according to general criteria. The first class can be defined as operations that modify the shape of an isolated part of a model and affect only that and immediately surrounding entities. Another class consists of operations that transform degenerate models, or parts of models, into more complete models. The final class is a sort of 'catch-all' consisting of operations not covered in the first two categories.

One general point that can be made is that the information specifying the command should be kept to a minimum and should be as independent as possible. This is in line with other information processing problems (see the design axioms of Suh [131]). From a modelling point of view, the information should be kept to a minimum to avoid information conflicts due to user errors. Keeping the information as independent as possible is, again, to avoid user errors made by giving conflicting information.

As a simple example of the interplay of parameters, take, say, the command to split an edge.

Perhaps the simplest form of the command takes two parameters: the edge to be split and a parameter value along the edge. The only check that needs to be made is that the parameter value is greater than or equal to zero and less than or equal to one (assuming that the curve is parametrised with the start vertex of the edge at parameter value 0 and the end vertex at parameter value 1). A parameter value of 0 or 1 implies a split at an existing vertex, which may be justifiable under certain circumstances. Parameter values less than 0 or greater than 1 imply that the split is to be made off the given edge, so they can be treated as error conditions.

If it is required that the new edge is to be inserted at a particular end of the edge, then it is necessary to specify a vertex attached to the edge as well.

However, giving a vertex directly increases the potential for error and requires a check that the given vertex is either the start or the end vertex of the given edge as well as the parameter value check. To reduce the potential for error, a Boolean variable can be given to specify either the start or the end vertex, although this might require an extra comparison step for the calling function.

If, instead of a parameter value, it is necessary to specify the position of the new vertex, then there is additional potential for problems in that the given position has to be checked to see if it really does lie on the edge.

Any of these ways of specifying the operation can be justified, depending on the way the operation is to be used, but the information conflict in the over-specified operations should be noted. Similar considerations exist when specifying information for more complex modelling operations.

7.1.1 Finishing and manufacturing operations

A useful class of modelling tools in the first category, defined above, is that which performs manufacturing- or assembly-related operations on a model. The chamfer operation (simulating the effect of shaving off an edge or vertex) is one example, as is the related function, blending (to round off an edge or vertex). These operations are useful in that they perform a modification corresponding to a physical operation, which offers the possibility of retaining information about the shape element, or feature, created. In some cases, it may also provide a logical tool that is more natural to use than, say, producing the same result with Booleans.

As an aside, it has to be pointed out, strongly, that mentioning the possibility of adding information to a model through the use of specific modelling tools should not be seen as an endorsement for the technique of 'design by features'. Creating information in this way has both advantages and drawbacks. A simple illustration is that of the "make boss" operation to create a simple extrusion on a face. It may be natural, when creating a model, to make a boss that will be used for assembly, say. However, for manufacturing, if the boss is to be created by milling, it is not the boss itself that is interesting, but the absence of material surrounding it. Feature modelling is too complex a topic to be dealt with at any length here. However, some aspects of modelling with models containing such information will be discussed in chapter 8. Feature modelling is described further in chapter 9.

When defining a modelling operation, the first step is to decide how the operation is to be specified. With these local operations, the effect is a combination of parameters and local model conditions. If the local operation corresponds to a finishing operation, then it is likely that only a minimum of extra information is needed; the operation will mostly be defined by the existing model. If the operation creates a new feature in the model, say, then it is likely that there is more flexibility in its use, and more information will have to be provided by the user. If the operation is too general to be specified in a single command, then it may be necessary to perform the operations in

stages, using subsidiary models. For example, a "make boss" operation to make circular pin extrusions might take real values for the radius and height, a vector position, and a face to specify the changes to be made. If a boss with a more general shape is to be created, then it may be necessary to provide a separately defined contour to be imprinted on the face to specify the shape.

There is also the problem of over specifying the operation, which was mentioned as a general problem of keeping information to a minimum and as independent as possible immediately before this section. Take as an example an operation, mentioned above, to create locating pins or holes on planar faces. If the operation is specified using a position on the face, a radius, and a height, then it can use the face normal at the supplied point to specify the extrusion/intrusion direction. If, on the other hand, the operation is specified using a centre-line for the holes/pins, the face, a radius, and a height, then there is a possible information conflict between the direction of the line and the normal to the face. It may be desirable to use a common centre-line to specify the command so that the matching features on the two separate objects share a common definition. However, it is necessary to be clear about what to do with the extra information. The face normal and centre-line direction can be checked and an error can be flagged if they do not agree. Another alternative is to always take the face normal direction. The final alternative is always to use the centre-line direction. From a modelling point of view, it is only possible to make an arbitrary choice between these alternatives; they are all equally valid, and it becomes a matter of engineering or design practice as to which to choose.

7.1.2 Converting degenerate models and parts of models

Using special model types for particular purposes means that it is possible to define a conversion process to 'interpret' simple models as their full equivalents. For example, in the GPM volume module, there were sheet objects, for representing thin plates models by their centres that could be converted to volumetric models (see figure 7.2). It is also possible to interpret other model types, such as wireframe models, provided that they have a specific interpretation.

An early experiment with the use of wire-frame models in the GPM project was to model the centre-lines of pipes in a valve block (see Kjellberg [69]). However, wire-frame models were also seen as a general mechanism for representing initial design sketches. As such they were no longer confined to having some uniform associated cross-section, such as a circle, but could theoretically be assigned any shape. In general, as mentioned, individual edges in wireframe models could represent general extruded shapes, but there is a definite need for care in how they are used. It seems unreasonable to attempt to make a general mechanism for expanding wire-frame models into general shapes. A more appropriate strategy would seem to allow only simple cross-sections and to use Booleans and separate modelling methods to perform the other cases.

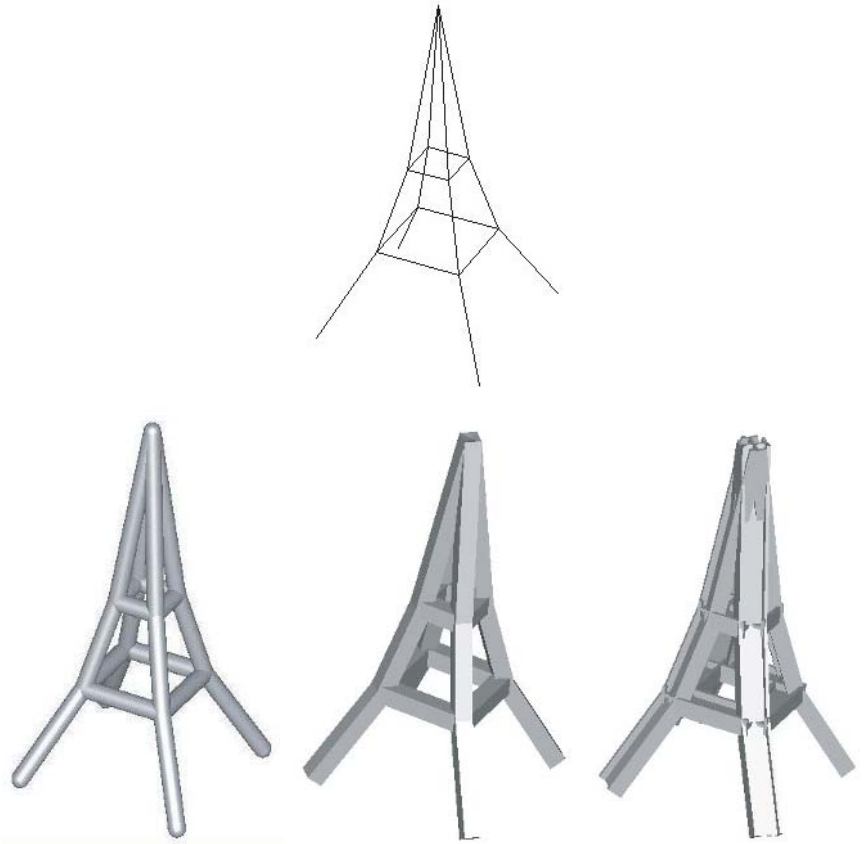


Figure 7.1: Converting wireframe models to volumetric models

Some examples of a wireframe model and its conversion to volume objects are shown in figure 7.1.

For degenerate models such as these, it is best to have a clear, unique interpretation of the model so that operations can be specified that conform to this interpretation.

If the whole model is to be converted, then there are two basic strategies: specifying completion information (plate thickness or cylinder radius) as a parameter; or attaching it to each object part separately. If the information is attached to individual model parts, then, for example, one wire-frame model could be used to model the centre lines of all holes in an object. It could also mean that a single sheet object could be used to represent a model composed of plates of different thicknesses. Again, caution needs to be used. It seems more advisable to think about how the models are to be used rather than trying to develop some general solution. If there are several object parts

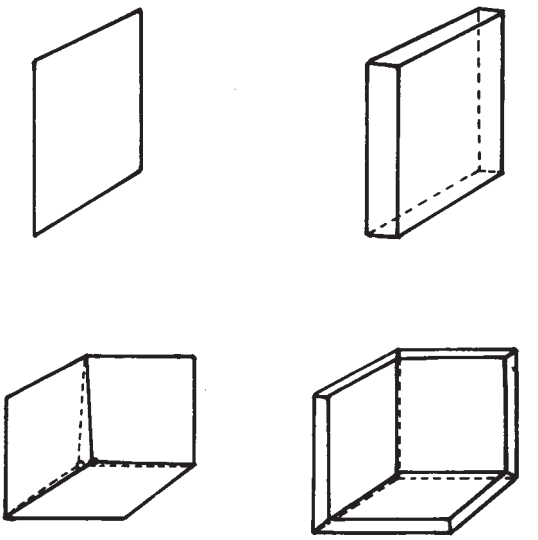


Figure 7.2: Converting sheet models to volumetric models

with different properties, then it seems more reasonable to represent them as separate objects that are grouped together rather than as one object. From a practical point of view, if there are several holes of different radii, for example, then these are maybe produced by different tools, and it seems reasonable to have one tool for one model. For sheet objects, plates of different thicknesses may imply that plates of different thicknesses are to be welded together. If the user really wants to create a thin-walled object with plates of different thicknesses, then he or she is not precluded from doing so using separate expansions and Boolean operations. What the user is forced to do is to think about what is required.

Converting isolated parts of a model, say, offsetting one face in a sheet model, can, theoretically, be done using a different selection criteria for the parts to be converted. For sheet objects, all edges in the object are examined when converting a sheet object to a volume. If offsetting a face, it would be necessary to look at all edges adjacent to the face. For wire objects, a single edge would be converted rather than all edges in the object. However, the object resulting from such a partial conversion will be a mixed-type object that may invalidate the criteria for other operations. It would be necessary, for example, to mark the converted object parts in some way and treat them differently from the other parts of the object. For wire-frame models, it would imply that part of the object was Eulerian and part of it non-Eulerian. The question then arises: Is it worth converting just part of a model? The answer would seem to be no. If such a partial evaluation strategy is necessary, then

it would seem more sensible to isolate the part of the object to be converted into a separate object. Convert it, and assemble the result with the remaining parts of the original object.

Similar considerations apply when sweeping wireframe or sheet models. Normally sweeping is applied to a complete model and is specified by a vector offset or by an axis and an angle for circular sweeping. As mentioned, sweeping just part of a model is possible, but undesirable.

To sum up, then, when dealing with general conversion of objects, it seems advisable to deal only with complete objects, and to keep the extra information needed for the operation simple. This simplifies the work of the modeller and forces the user to be clear about what is wanted. Making general tools may also cause the final model to invalidate the original criteria for the designer. On the whole, though, this kind of complex operation is probably something that needs to be added as a basic part of an original modeller with a clear, uniform philosophy, rather than as an update added in later.

7.1.3 Other operations

The matter of operations not covered in the previous two categories is more difficult to describe, because they perform more than just 'finishing touches' to a model, but they do not apply a uniform transformation to the whole object. The information requirements and effects vary widely so they tend to form part of the basic modeller, rather than enhancements for particular applications. Examining the existing modelling operations, some of which are described in chapter 6, the information needed to apply them is:

- BEND Two faces, an axis (two vectors), and a real value
- REFLECT A face (alternatively surface and object).
- SWEEP OBJECT Object and vector
- SWING OBJECT Object, axis (two vectors), and a real value
- DRAFT (sets a slight angle on a faceset for moulded parts so that they can be extracted from the mould) A face and an angle
- DUAL The object whose dual is to be created

What seems clear is that for complicated operations, the task should be defined in terms of simple steps. For sweeping or swinging a body, say, the task can be reduced to one of sweeping all appropriate faces in the object. For bend, the task can be broken down into the process of inserting the extra topology of the bend seam and modifying part of the object. For reflection, the task can be broken down into the steps: copy the object, transform the copy, and join the coincident faces (or edges for laminae) if any. The drafting operation involves examining all faces surrounding the specified face and setting the given draft angle on each.

This way of defining complex operators as a sequence of simpler operations may well produce less efficient algorithms than if the operation is defined as a single change, but it can lead to more comprehensible algorithms. Moreover the operational requirements for a complex operation can be broken down into a set of requirements for the individual operations. However, this has drawbacks for the simpler operations. If they are to be used for general operations, it implies that there are more complicated preconditions for their use and implies that it is necessary to add more tests to differentiate between cases. There are added complications if Euler-type operations are to be applied to transition models that are not completely Eulerian. Other complications occur if a sub-operation includes geometric tests that cannot be used on degenerate models.

Another basic decision that has to be made is whether the operations are to be destructive, i.e., change the model to which the operation is applied, or whether they copy the object and use the original only for guidance. For complicated operations, it may be desirable to preserve the original object in case the operation fails so that a user can modify the command. It is possible, also, that the operation performs a complex change, such as creating the dual of an object, as described in section 4.10. This operation needs to traverse the original object while building the new object, so it needs the original object unchanged while constructing the dual. The finishing type operations described in section 7.1.1 should normally change the model, because they usually provide 'finishing touches' to a model. Similarly, the conversion type changes described in section 7.1.2 perform major changes that alter the basic nature of the original model. Whether the original, degenerate model is copied, the result of the conversion operation is essentially a new model. However, with the other operations covered in this section, the decision is not clear.

7.2 Task decomposition

Task decomposition can again be split into groups according to the extent of the change made to a model. For finishing operations, the change is usually a limited change, and the operation can be described as a series of basic geometric and topological changes. For the model conversion algorithms described in chapter 5, the changes are also made in terms of a series of simple changes, but applied globally to the model. The final group of operations seem to be best described in terms of a set of higher level operations, such as sweep, faceglue, copy model. These higher level operations are themselves broken down into sequences of simpler operations in a similar manner to the operation decomposition described in section 7.2.1.

7.2.1 Finishing operation task decomposition

As described in chapter 4 on Euler operators, one object can be modified in a sequence of elementary steps until it has the required topology. The number and type of these steps can be determined from the changes in the number of vertices, edges, faces, hole-loops, the number of shells, and the genus. The total change can be made using a spanning set of five simple operators and their inverses.

However, when changing one object into another, this sequence of operations is usually only an approximation. For example, there are at least three separate interpretations of the change (1,1,0,0,0,0), or Make and Edge and a Vertex as described in Appendix F, sections F.1, F.2 and F.3, and illustrated in figure F.4. The operation described in section F.1 adds a spur edge and vertex to an existing vertex (figure F.4a,b,c). The operation described in section F.2 splits an edge by inserting a new vertex (figure F.4d). The operation described in section F.3 splits a vertex and inserts an edge joining the two part vertices (figure F.4e). A further example, illustrating different Euler operator application sequences, can be seen from figures 4.1 to 4.5, which show five different ways to build a cube. The number and type of operations are: one MBFV (Make Vertex, Face and increase the Multiplicity) operation, seven MEV (Make Edge and Vertex) operations, and five MFE (Make Face and Edge) operations. It is possible to choose different strategies to determine how to apply these. Figure 4.1 shows the sequence if trying to maximise the number of four-sided faces. Figure 4.2 shows the sequence if trying to maximise the number of trihedral vertices. Figure 4.3 shows the sequence when applying all MEV operators before the MFE operators. Figure 4.4 shows the sequence when the five MFE operators are applied before the MEV operators. Figure 4.5 shows the actual sequence used when building a cube using sweeps in BUILD.

Another problem is that although it is possible to perform the topological changes using combinations of only five operators (and their inverses), it is not always the most logical way to work. Also, as illustrated in the sequences of operations for performing exact face–face glue (see figures 6.33, 6.34 and 6.35 in section 6.4) there is an operator that causes no change in the numbers of elements, but rearranges the topology. This null operator can be simulated, in this case, using an MFE operation, an MEV operation, a KFE operation; and a KEV operation; however, this is less efficient than the simple rearrangement performed by the 'null operator'. The list of operators that can be applied can be expanded from the basic five to include more appropriate operators for a particular application sequence, because a spanning set can be chosen in several ways, but the problem then becomes one of knowing which set of operators is appropriate. This was described in section 4.2.

Ignoring the information handling problems, which are outlined for the Euler operators in Appendix F, and which will be discussed further in chapter 8, a further important consideration concerns how the geometry is to be

changed. The purely topological changes indicated by the Euler operators ignore the geometry, but it is necessary to change both parts of the model to be consistent, as outlined in Appendix F. Also, operations such as the SET-SURF operation for inserting a new surface into a face, described in section 6.11, and the TWEAK operation (for modifying the geometry of a face, edge, or vertex by a specific transformation) may not change the topology at all, only the geometry of a model. See [11] for details of tweaking.

Returning to basics, the datastructure in boundary representation (B-rep) modellers can be divided into two parts: 1) the structure or topology and 2) the shape or geometry (figure 7.3). These two parts have corresponding hierarchies, for topology: 1) faces, 2) edges, and 3) vertices. For geometry the hierarchy is 1) surfaces, 2) curves, and 3) points. These hierarchies imply that, for example, the shape of an edge cannot be changed without changing the position of the surface, unless the new curve also lies in surfaces of the faces adjacent to the edge (usually only true for edges between faces lying in the same surface). Similarly, the position of vertices cannot be changed unless they lie on the curves of all edges meeting at the vertex (usually only true for vertices between two edges on the same curve), and hence that the vertex position lies on the surfaces of all faces meeting at the vertex. A model is represented as a patchwork of faces that are portions of surfaces. Edges between faces lie in portions of the intersection curves of the surfaces of the faces. Vertices lie at the intersection points of, usually, three or more faces. Therefore, if geometry is to be changed, it is necessary to start with a surface and a corresponding face, and change the other entities to be consistent with that. If an edge is to be moved, then it implies that the surface equations of one or both neighbouring faces must be changed. If a vertex is to be moved, then it implies that the surface equations of all faces meeting at the vertex will be changed, and the curve equations of the edges meeting at the vertex will also have to be changed in accordance with the new surface equations.

The kind of finishing operation covered in this section can usually be described in terms of a sequence: set up; repeated basic operation; and final operation. The most important of these is the repeated basic operation. This is related to the entity being transformed in the finishing operation. Thus, when sweeping a face, the basic process is to sweep an edge. This is translated in the simplest form of sweeping, as creating a side edge, and an edge connecting the end vertex of the previous side edge and the end vertex of the new side edge closing the new face. Thus, the initial step is to create the first side edge, and the final step is to close the last face without creating a new side edge.

So, if the basic entity to be modified is a face, then the edges and vertices surrounding the face will be modified. If the operation is specified by an edge, then the adjacent faces and the end vertices will be changed. If the operation is specified by a vertex, then the edges and faces meeting at the vertex are affected.

A first step in defining the way the operation is to work, therefore, is to

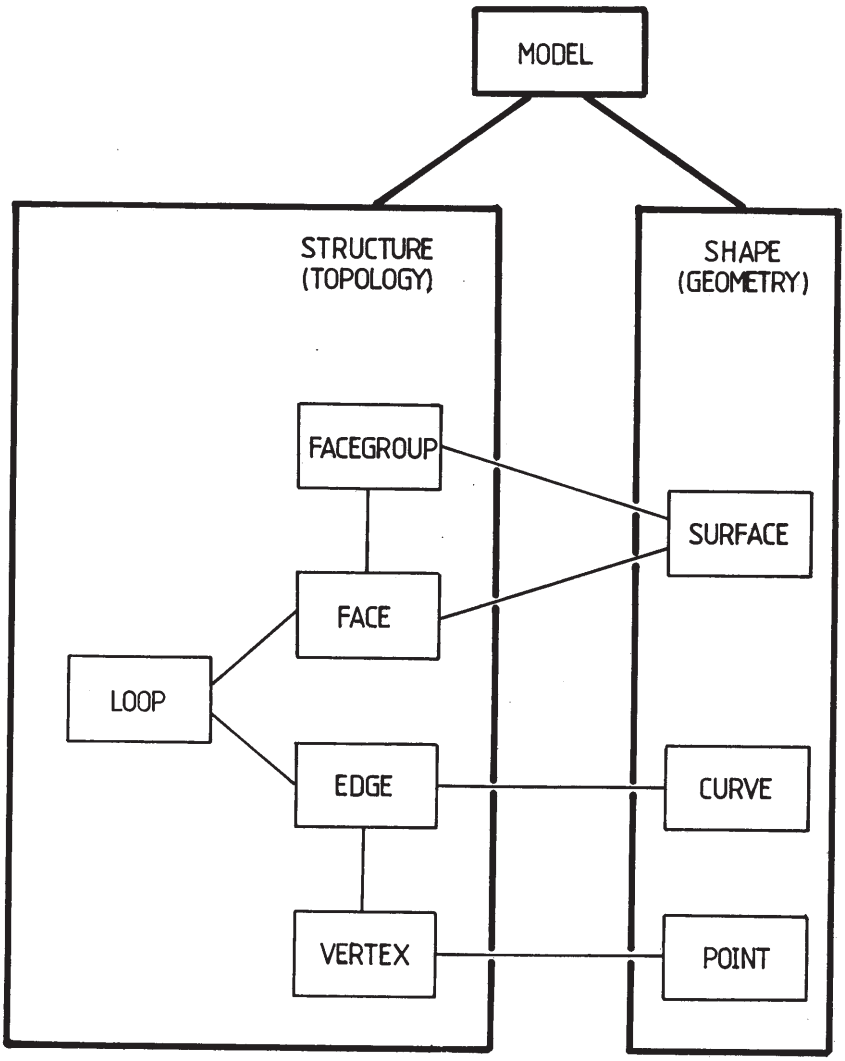


Figure 7.3: Structure and shape elements of the B-rep datastructure

identify which is the main element that will be changed. Subsequently it is possible to identify the model elements that surround this main element and that may be affected by the operation. The functioning of a simple case can be analysed to identify the final result of the operation. This result may help in identifying the topological changes needed using the Euler decomposition method described in chapter 4. The possible decomposition is tallied with the types of elements surrounding the main element to identify the repeated step. Once the repeated step has been identified, it is possible to see what preconditions are needed for the first of the repeated steps, and set these up as an initial phase of the operation. The final steps need to complete the topological and geometrical changes and ensure the consistency of the operation. The repeated step forms a sub-sequence of simple changes within the complete sequence; the remaining changes are the set up and final steps.

When decomposing operations into steps, the topological and geometrical utilities and operations identified in section 3.3 and chapter 4 have proved useful for previous modelling operations. The way that the Euler operations are applied, and the necessary changes to the topological elements, is indicated by the nature of the repeated step. As described in Appendix F, it is sometimes also useful to include geometric information with the Euler operators, so the new faces, edges, and vertices are assigned surfaces, curves, and positions, respectively. However, if these operators are organised as a 'layered set', then the finishing operations have more flexibility in how they are called.

7.2.2 Model conversion task decomposition

Model conversion operations differ from finishing-type operations in that they affect the whole of a model. As the model to be converted is a degenerate representation, some or all of the model elements represent degenerate parts and these have to be identified, while any others have to be modified to be consistent with these. The degenerate parts can be identified by having a clear understanding of what the degenerate parts of the model represent.

For example, the sheet objects in the GPM volume module represent thin plate models, with the narrow side faces represented by edges. So, when expanding sheet objects by giving them thickness, the degenerate parts are the sharp edges, and the non-sharp edges have to be offset to be consistent with these.

If wire-frame models represent solid structures, with circular cross-sections, and have to be expanded to volumes, each edge in the wire-frame model has to be expanded into a volume. The edges represent cylindrical shapes and the vertices the joints between these.

As with any modelling operation, it may be necessary to distinguish between new and original parts of the model. With model conversion, when the whole datastructure of a model is likely to be changed, this is very important because it is likely that the structure being traversed to perform the operation is changed during the operation. Distinguishing between the old and the new

parts can be done using the marker bit mechanism, using separate lists or by making parallel datastructures in a similar way to that described in sections 3.3.1 and 6.10. Either way is possible; there is no definite advantage to either that makes it an obvious choice, so it becomes a matter of the general modelling strategy that is chosen. If modelling operations are generally destructive of the original model, that is, modify the original model, then marker bits or separate lists should be used. Otherwise separate datastructures can be used, setting up the expanded model parts in arrays and using the original model to indicate how they should be linked. Having separate datastructures is, possibly, facilitated if the model datastructure includes 'half-edges', when the separate datastructures are bounded by half edges rather than complete edges, which can simply be united when joining up the final structure.

As an aside, note that performing the expansion operation by traversing the object and modifying the parts appropriately is closer to the philosophy of 'stepwise' modification outlined in Braid et al. [13]. Otherwise the operation becomes equivalent to the operations to produce primitive solids, where a complete object is the end result. A stepwise approach is, perhaps, easier to monitor to see what the operation is doing.

In the algorithm for expanding sheet objects to volumes, described in section 6.6, the topological changes are made first, followed by the geometrical changes, and the original model is destructively changed into the expanded model. If the expansion is done in a separate datastructure, then the new object parts can be set up complete with geometry as partially complete elements referred to, in some way, from the original object structure. During the final phase, the original object is used to identify adjacent elements that are then joined together.

7.2.3 Complex operation task decomposition

As described in section 7.1.3, complex operations are difficult to classify and decompose into simple steps in their original form, and they are easier to comprehend in terms of logical sub-elements. For example, operations such as REFLECT and BEND can be broken down into steps. For REFLECT:

- Copy object
- Transform body with reflection transformation
- Join coincident faces or edges

For BEND:

- Produce local intersection seam
- Create wedge adjacent to seam
- Modify geometry of part of the object

Each of these separate partial operations forms a more limited change, and can, itself, be decomposed into elementary steps, as described in section 7.2.1. Operations such as copying and transforming objects form part of the basic utilities in modelling systems, as mentioned in section 3.3.3. Likewise, joining coincident faces, described in terms of its elementary steps in section 6.4, is also useful for several operations, and so it is likely to be present. In contrast, the basic steps for BENDing objects are more specialised. Producing the local intersection seam is common to other operations, but it is usually hidden. The other operations are special to the way that the bend operation works. However, the same principles apply, that the complex operation is broken down into simpler operations.

How this is done is hard to define. For example, the BEND operation could be defined as the two partial operations:

- Create wedge
- Modify geometry of part of the object

where the 'create wedge' operation performs the intersection and inserts extra elements for each face around the part of the body where the bend is to take place. Producing the bend seam first and then creating the wedge may be preferable because the step to produce the seam is similar to the same step in other operations.

The difference between the two decomposition sequences has implications for the repeated steps mentioned in section 7.2.1. In the first sequence, there are three sets of repeated steps. The first, to produce the local intersection seam, involves intersecting the bend plane with the surface of the current face to produce a curve, intersecting this curve back with the face, and then inserting new edges between the edges cut by the curve. The second step, to produce the wedge, involves splitting these edges and one end vertex successively until the complete seam has been traversed. The third repeated step is to modify the geometry and is common to both decompositions. In the second sequence, as the edges in the bend seam are inserted in a face, they are sliced and the end vertices split. Although the second sequence may seem more efficient, avoiding traversing the object twice, if, in the first sequence, the edges in the bend are recorded, then slicing them can be done relatively easily. The choice between the two becomes largely a matter of personal preference, the first emphasising the two elements in creating the bend wedge, the second combining them.

The operation for setting a draft angle on a face is much simpler in terms of decomposing the operation. It simply traverses all non-wire edges bounding a face, modifying the surface of the face on the other side of the edge and changing the geometry of the elements surrounding that face to be consistent with the new surface. The difficulty is in calculating the new surface for the face.

The following general operations can be identified as useful for decomposing complex operations:

- Copy an object
- Modify an object or other sub-element with a transformation
- Give a face a new surface (SETSURF)
- Join two coincident equal faces
- Join two coincident equal edges
- Sweep a face

Although this list will not cover all possibilities, such operations can be identified as useful from examples, such as reflect. Another example, the operation of sweeping a face through an object to another face, can be implemented in the following steps:

- Sweep a face
- Give the swept face a new surface (negated surface of the destination face)
- Kill the face, and make an interior loop in the destination face

The first two operations are included in the above list; the third is an Euler operation described in Appendix F, section F.9.

Like the operations described in section 7.2.1 these complex operations can be divided into three basic phases: 1) set up; 2) general modification step; 3) finishing. For reflection, the setup phase consists of copying and transforming the object. The general modification phase joins the coincident elements (if any). There is no finishing phase if the join operation removes 'smooth' edges; otherwise, the object can be post-processed to remove these. For bend, the setup phase involves inserting the bend seam, the general modification phase involves modifying the geometry of one part of the object, and the final phase removing superfluous topology in the bend seam. For the operation to sweep a face through an object the initial phase, setting up the face to be swept, is done before the operation. The general modification step involves sweeping the face and setting in the new geometry. The finishing phase is to make the inner loop in the target face and kill the swept face.

7.3 Pathological conditions

The final consideration concerns special cases and pathological conditions for the operations. Operations are usually defined for simple conditions, and the special cases are identified afterwards. There are two ways to handle

the special conditions: Either they can be identified and flagged as error conditions, or the algorithm can be modified to cope with them.

One way of identifying pathological conditions is related to the feature identification methods developed by Kyprianou [74], using the concavity/convexity of edges. Put very simply, faces bounded only by convex edges are extremes of the boundary. Faces bounded only by concave edges are extremes of intrusions. Similarly, vertices with only convex edges lie at extremes of the object and vertices where only concave edges meet lie at extremes of intrusions. Mixed concave/convex faces and vertices lie on feature boundaries. Operating on 'pure' elements, only convex or concave is easier than operating on mixed entities. By comparing the nature of the operation, that is adding or subtracting material, with the classification of the topological entity modified by an operation, it is possible to gain some insight into where problems might occur. Thus, if an edge 'changes character', i.e., changes from being convex to concave or from concave to convex, then there is a major change in the nature of the object part, a feature will change. This is connected with the nature of the operation in that an operation to add material to part of an extrusive feature will leave the feature as an extrusive feature, but an operation to remove material may change it.

Special consideration must also be given to edges that are neither concave nor convex, i.e., smooth edges or fake edges. Fake edges, which lie between faces lying in the same surface and whose role was described in section 3.1.3, should, as far as possible, be invisible to a user, because his/her role is artificial and connected with limitations in the modeller. Smooth edges are different in that they lie between faces lying in different surfaces, but with no change in normal direction along the edges. It is meaningless, say, to try and chamfer or blend such edges. They represent special cases.

Another problem concerns edges that become 'zero length' edges as the result of an operation. In effect, these edges disappear and should be removed by killing the edge and coalescing the vertices at either end. This is a special case of the KEV (Kill Edge and Vertex) operation described in Appendix F.

For geometry two general classes can be identified: infinite geometry or bounded geometry. Bounded geometry can also be subdivided into closed geometry, such as spherical surfaces and closed curves; and open geometry, such as B-spline or Bezier geometry. Bounded open geometry poses something of a problem for B-rep modelling because there are two sources of information about the model, from the topology and from the geometry, so there is potential ambiguity.

Also, for geometry, two special cases have to be identified: coincidence and tangency. If geometry generated by an operation is coincident with surrounding entities, then this may have to be merged with surrounding elements, although with the conditions on merging curved geometry mentioned in chapter 3. Tangential geometry is less of a problem, but it can be subject to numerical problems.

7.4 Examples

The following examples are some simplified examples of modelling operations to illustrate definition of algorithms. As opposed to the algorithm descriptions in chapter 6, which describe implemented algorithms, these are intended to illustrate the whole process of design and implementation of modelling algorithms. In terms of the categories into which modelling operations were divided, above, they are both examples of finishing type operations, making localised changes to a specific portion of the model.

7.4.1 Add a slot or pocket to a face

For the first example, take the definition of a local operator to add a slot or pocket to a face, which is related to the "make boss" and "make pocket" operations developed in BUILD by Charles Anderson. The operation may make either a slot or a pocket depending on the end conditions, but for simplicity, the term 'slot' is used to cover both.

There are various options for the operation. One option is to take an existing edge as the definition of the slot centre-line, convert this to a slot-shaped face, and sweep the new face downwards. Another option is to specify the operation using two coordinate positions as the endpoints of the slot, a width and a depth, the slot shape being imprinted on the face and the new face swept downwards. If the operation is to simulate a manufacturing operation, then it may also be necessary to change the surface of the bottom face of the slot appropriately, to a planar surface or other machinable surface. Note, though, that there is a difference between machining a pocket and machining a slot. For a pocket, it is necessary to enter the material from the top, where for a slot, it is possible to move the tool in from the outside. It is necessary to distinguish clearly between these two cases if the operation is intended to provide information for downstream applications. Yet another option is how the ends of the slot are shaped, whether they are square, rounded, square with rounded corners, and so on. The final choice about these options depends on how the operation is to be used. The operation should be logical and convenient for a particular application environment, and the options should be selected on this basis.

The various options also have implications for the choice of input parameters to specify the operation. If a slot is to be defined using an existing edge, then part of the information can be obtained from the edge. If the slot is defined without the edge, then extra information has to be supplied in the form of parameters. Creating a slot from an existing edge can allow complicated slots to be defined more easily by separating the parts of the creation process, but possibly at the cost of a fluency of use.

The first version, using an existing edge as a definition of the slot centre-line, can be simulated with the steps (illustrated in figure 7.4):

- Slice the edge to create the new slot base face.

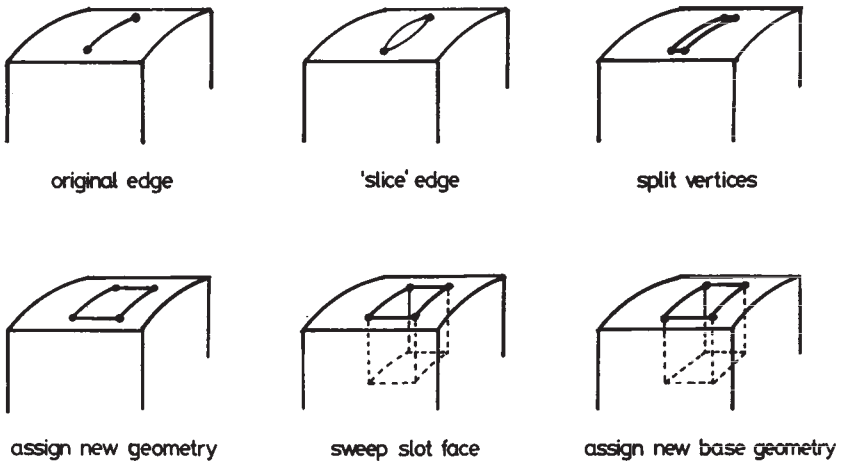


Figure 7.4: Creating a slot or pocket from an existing edge

- Split the end vertices of the original edge.
- Assign geometry to the two edge halves and the vertices.
- Sweep the new face downwards.
- Set a new surface in the bottom face of the slot (if appropriate).

The second version can be simulated using high-level operations with the following steps (illustrated in figure 7.5):

- Define the slot shape as a lamina.
- Imprint the lamina on the face.
- Sweep the new face (or faces) downwards.
- Set a new surface in the bottom faces (if desired).

Otherwise the operation can be implemented using approximately the same steps but performing the operations more directly:

- Define the geometry bounding the slot.
- Create the slot boundary edges by intersecting these surfaces with the face.
- Create the side faces by sweeping the slot face downwards.

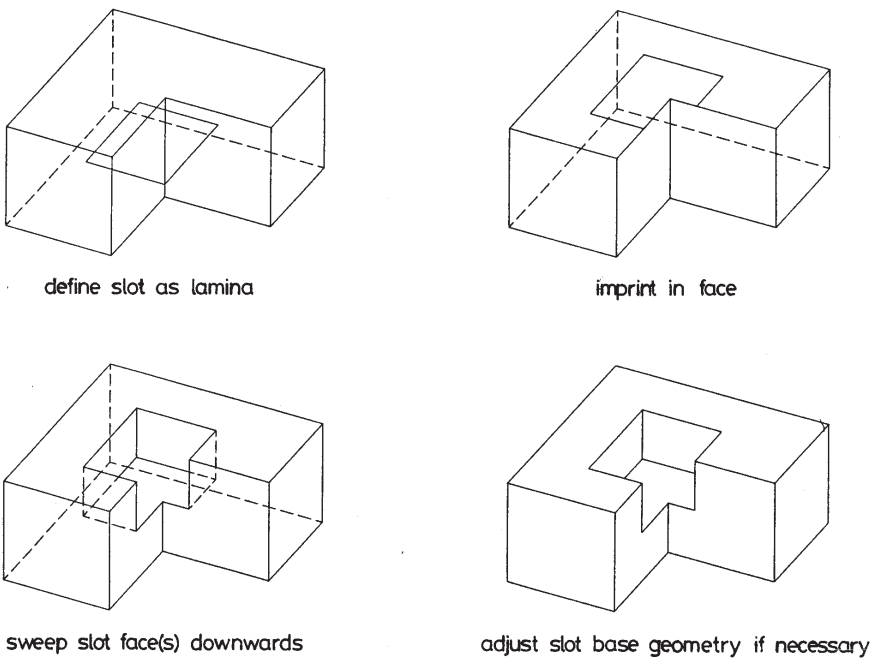


Figure 7.5: Creating a slot by imprinting and sweeping

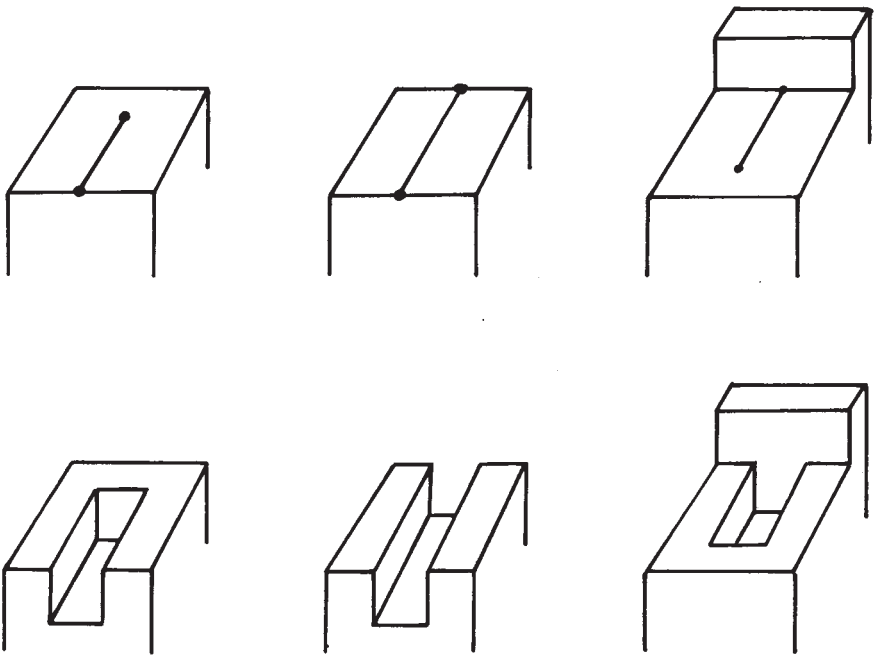


Figure 7.6: Conditions for the slot ends

- Set a new surface in the bottom face(s).

There are various special cases to consider. For the first version, there can only be one slot produced, because it converts a single edge. There are special requirements for the ends of the slot if the end vertices of the edge are not spur vertices. Some cases are illustrated in figure 7.6. On the left and in the middle a slot is created, whereas on the right, a pocket is created even though the edge touches a face boundary. For the second version, there are special cases where the slot shape might be imprinted in several pieces, where one or both sides might lie outside the face, thus producing a partially bounded slot or shaving off a face completely. If the operation is intended to simulate a manufacturing operation, then it is necessary to identify these special conditions.

Which of these versions to choose depends on how the operation is to be used. The first version is simple, whereas the second is more user-friendly, more accurate, uses well-defined modelling operations as basic elements, but takes longer. The third version is similar to the second version but more direct. For the purposes of the example, the logic behind the operation is that it will create a new slot or pocket in a face, rather than operating on an existing edge. It will also be implemented directly as a modelling tool, rather

than as application code, so it will use lower level operations than typical user available tools. For these reasons, the algorithm will be designed using the third version, above. Also, the algorithm will be developed as a design tool for producing an idealised shape, so it does not impose process planning-type restrictions on the design. If the ends of the slot have to be rounded-off later, then it is assumed that this will be done separately, by a separate 'blend' operator. For manufacturing modifications, it is also necessary to know how the slot is to be produced, whether it is to be machined or to come from a precision casting, say, before modifying the basic shape to conform to the manufacturing method. This is only known at the process planning stage. It is also assumed that the slot will be produced with a flat bottom as though the operation is to be produced by a simple planar tool motion; more general bottom shapes may be possible of the slot comes from a casting, but this simplifies the example.

Initialisation

The operation is specified with the face in which the slot is to be inserted, the endpoints of the slot, the width, and the depth of the slot. It is possible to define more complicated slot shapes using extra parameters, for example, an extra point between the two given endpoints to specify a circular path. For the purposes of the example, a straight slot will be created. The slot definition information is not entirely independent, because the endpoints of the slot are constrained to lie on the surface of the face. This is necessary because the surface normals are needed to help define the centre-plane of the slot. Two planes are used: one offset from the centre-plane by half the slot width in the direction of the centre-plane normal, and the other by half the slot width in the opposite direction to define the sides of the slot. The end planes of the slot are defined from the vector between the given start- and endpoint positions.

These latter four planes delimit the slot; intersecting them with the surface of a given face defines the borders of the slot. Once these borders have been defined the face is swept downwards to produce the slot. The surface of the bottom face is defined, from the slot centre-line, offset by the slot depth. The final step is to tidy up the slot ends.

Slot boundary creation (repeated middle step)

The process of creating the slot boundary is similar to the imprinting operation described in section 6.9. One plane is chosen, one of the long sides, say, and intersected with the surface of the face to produce a curve. This curve is then intersected back with the face to produce a series of intersection points where it cuts the edges bounding the face (using the general utility described in section 3.3.4). The curve is also intersected with the two neighbouring surfaces (for a long side surface the two end surfaces, for an end surface the two

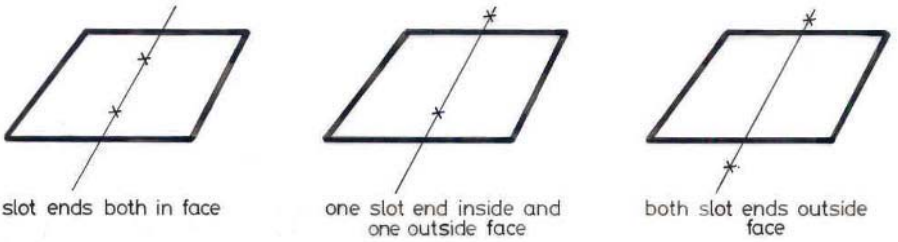


Figure 7.7: The three cases for the slot ends

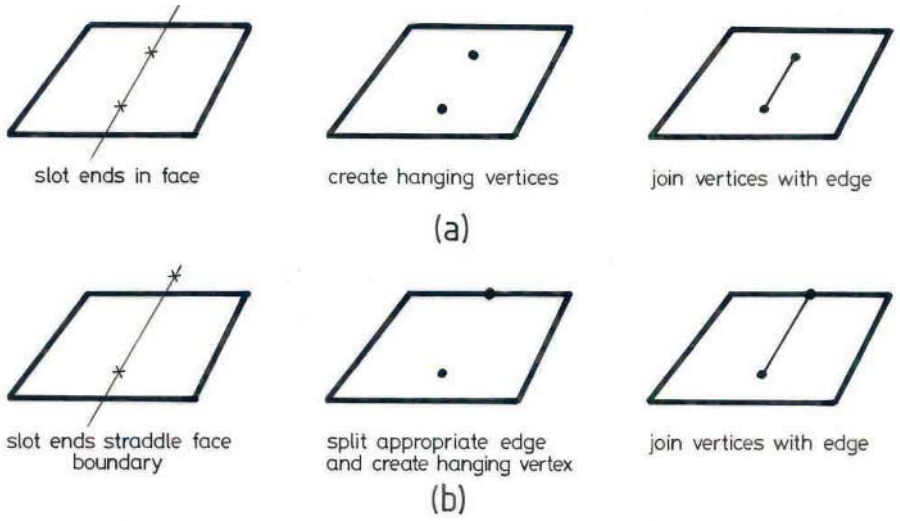


Figure 7.8: Creating slot end vertices

long side surfaces) to find the portion of the curve that is of interest and so which intersection points are relevant. New vertices are created at the intersection points on this curve portion, and then these are connected with new edges.

The first vertex will be added where the curve cuts the first limiting plane, if this lies in the face, or will be on the boundary of the face at an existing vertex or by splitting an edge, where the curve cuts through the middle of an edge. The intersection points are then taken in order until the final position, where the curve cuts the second bounding surface, is reached. Figure 7.7 illustrates the three cases for an end vertex, for a simple face shape. The case on the left produces a pocket. As described at the beginning, it is best to take a simple hypothetical example first and then generalise it.

The simplest case is where the start point and end point lie inside the face (figure 7.8, left), creating a pocket. In this case, two vertices are created as

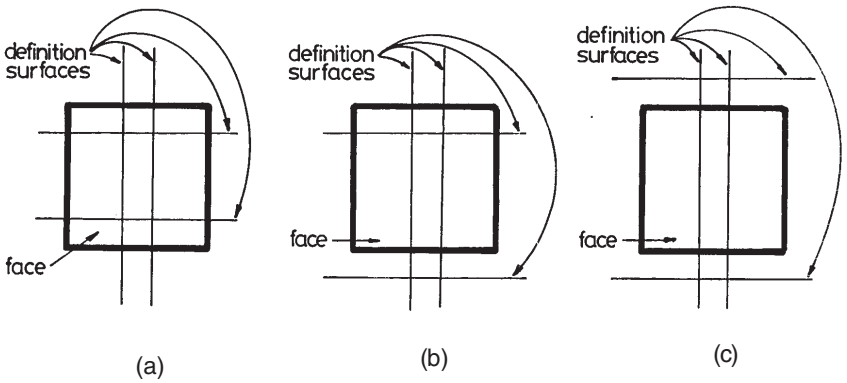


Figure 7.9: Surface-face intersection for the basic three slot cases

interior loops in the face and connected by an edge. If one or both vertices lie on the boundary of the face, then the edges cut by the curve are split at the appropriate places and the new vertices are used instead of the hanging vertices (figure 7.8, middle and right).

The next problem is to generalise this intersection to cope with multiple intersections between a curve and a face boundary. The intersection points can be handled as a sequence of interactions between the curve and the face. The first vertex is created, as in the simple case, as a hanging vertex or by splitting an edge. Once the first vertex has been created, other edges and vertices are added according to where the curve cuts the face boundary until the final point (where the curve cuts the second bounding plane) is reached, or until there are no more face intersection points. The vertices are added in pairs, the first in each pair as a 'hanging' vertex and the second closing the pair and connecting the hanging vertex and the second vertex with an edge. The types of interactions between a curve and a face are listed in section 6.1. It is necessary to be clear about what to expect, and how to handle the cases, which is determined by the exact nature of the modelling utility to intersect a face with a curve.

The other surfaces are intersected with the face in the same way as that described for the first curve. When the fourth surface has been intersected the slot shape should be complete. If the slot start- and endpoint are inside the face, then the slot will be as shown in figure 7.9a. If either the start- or endpoint of the slot lies on the face boundary or outside the face then only three surfaces are necessary to complete slot; the fourth either coincides with an existing edge or does not intersect the face in the appropriate region (figure 7.9b). If both start- and endpoint lie on the face boundary or outside the face then only two surfaces are necessary to define the slot (figure 7.9c).

Once this basic procedure has been decided on, it is necessary to consider special cases and where problems might occur. There are two obvious

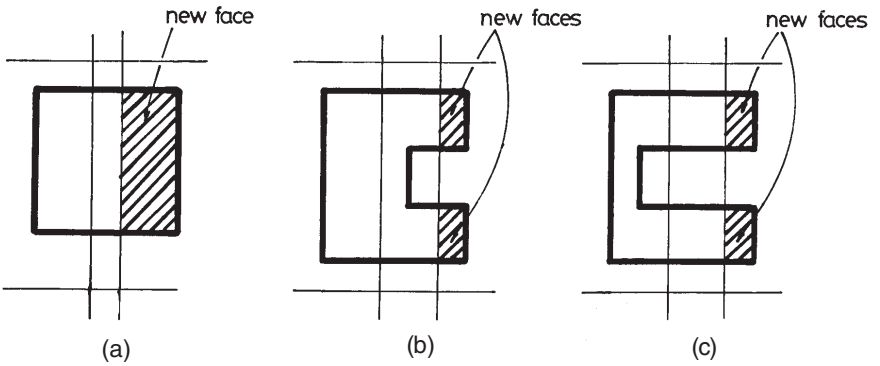


Figure 7.10: Creating new sub-faces for the slot

problems:

1. Adding the edges bounding the slot may produce extra faces and care must be taken to make sure that these do not cause confusion.

2. It is possible that none of the surfaces intersect the given face.

1. **Creating extra faces.** Figure 7.10 illustrates some examples of boundary edge creation where new faces result. In most cases, this problem can be avoided by ensuring that any new face is added away from the other slot definition surfaces, if the Make Face and Edge tool permits this. There are two ways this might be done: either by using knowledge of the implementation details of the Make Face and Edge operation, or by using an extra parameter (an edge or vertex, say) to specify which is to be the new face, if the operation allows this. However, if adding a slot as illustrated in figure 7.10c, where the slot completely exits from the face and then re-enters it, there can be problems. Another method is to preserve pointers to any new faces created, and to perform the intersection between the slot boundary surface and all extra faces. This is more expensive than the other options, but safer. Any of these is possible, although the first options imply restrictions on use of the slot, which might be perfectly acceptable in practice. It is necessary to consider how the operation is to be used, and choose one solution that best fits in with this use.

2. **There are no intersections between the slot definition surfaces and the face.** Figure 7.11 illustrates examples where the whole face is removed by the slot, and where the slot lies outside the face. If this problem occurs, then it is necessary to check

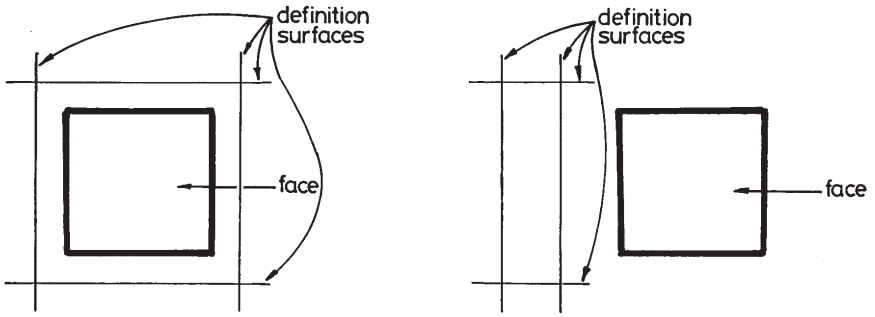


Figure 7.11: Cases where the slot boundary does not intersect the face

whether the whole face lies with the slot bounding surfaces or outside them. One point of the face can be checked against the surface normals to check this. These kinds of critical case are important if the modelling operation is supposed to provide information for downstream applications. In the first case, the slotting operation corresponds, in reality, to a manufacturing facing operation that would probably be done using a different tool from that needed for milling a slot. In the second case, the operation is unnecessary. In both cases, the user should be advised that a special condition has arisen in case this is not what was required.

Indenting the slot (final step)

Once all four slot boundary surfaces have been intersected with the original face, the new face or faces comprising the slot base must be moved downwards, creating the complete slot. The simplest way to do this is to sweep the face or faces downwards. This avoids performing the same sort of checking and special case handling already in sweep. However, again potential problems can be identified, to some extent from examples.

In the simplest case, where the start- and endpoints of the slot lie inside the face and the slot does not touch or cut the face boundary, this means simply adding four side edges, one from each corner of the slot, and connecting these by edges closing the side faces of the slot (figure 7.12a). The new side faces lie in the same surfaces as those used to create the slot shape in the given face.

If the slot boundaries touch the face boundaries, then some special cases will occur. If the slot boundary just touches part of the face boundary, but does not cut it (figure 7.12b), then an extra vertex will have been generated that may generate an extra edge and face in the slot. If the slot boundary intersects the face boundary, so that part of the original face boundary is included in the slot boundary, as in figure 7.13, then new geometry may be generated (figure 7.13a) or there may be holes in the sides or ends of the slot

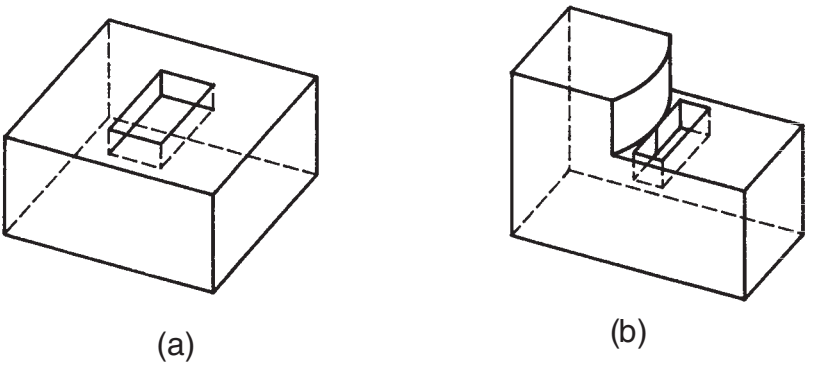


Figure 7.12: Slot creation examples

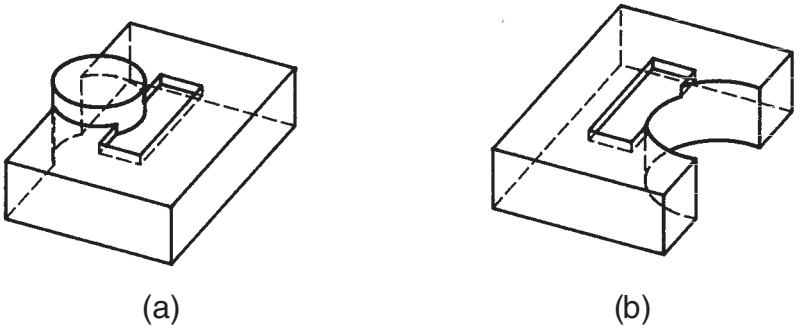


Figure 7.13: Slot interaction with face boundary

(figure 7.13b). This can be determined from the state of the edge, whether it is convex, concave, smooth, or sharp.

Where extra geometry is added, i.e., the edges added to the slot boundary from the face boundary are concave or smooth, this may imply restrictions on the width of the slot, and hence a need to adapt information tags associated with the slot. Alternatively, if the edge in the boundary is smooth, it may be a fake edge that should be treated as though it was not there. If this is so, then it may be desirable to extend the slot boundary creation process to the adjacent face on the other side of the edge. If an edge is concave, then it may indicate an error condition, because the desired slot cannot be created. At any rate, the analysis that such edges are present may be useful to a designer to allow him to examine that part of the model and possibly modify it accordingly.

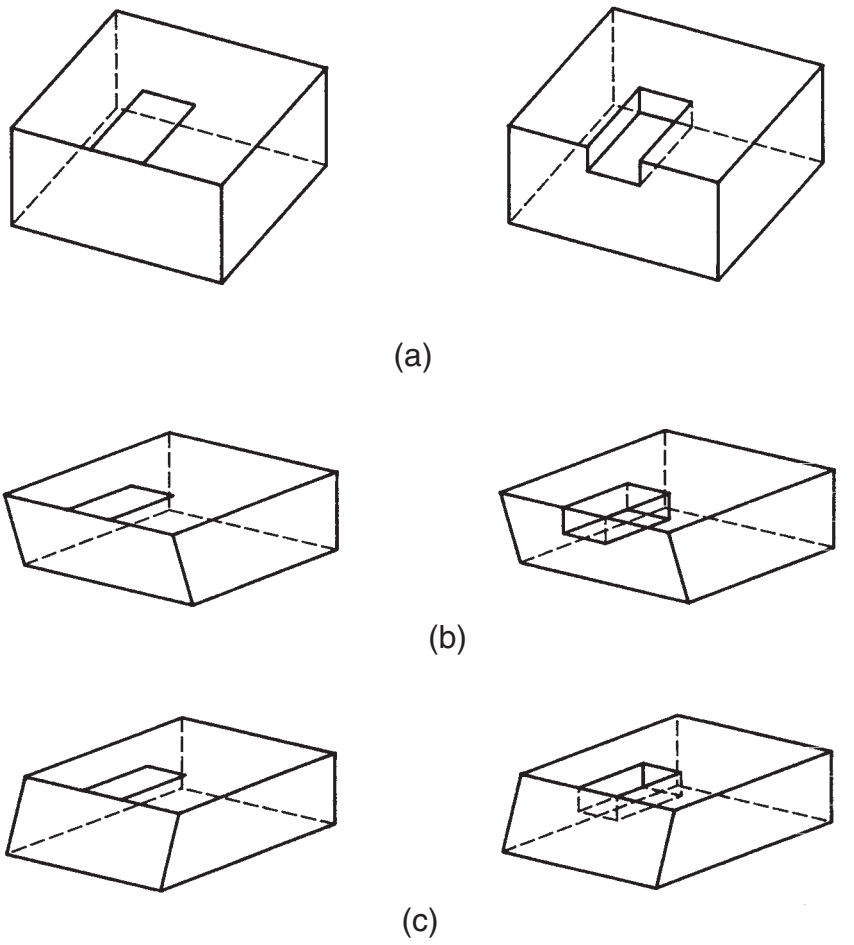


Figure 7.14: Interaction between slot and adjacent faces

Where holes in the sides or ends of the slot are indicated, because the included edges are convex or sharp, it may be necessary to perform extra steps to ensure that the holes are consistent with adjacent faces. If the face on the opposite side of the edge from the slot face is planar, and perpendicular to the slot (figure 7.14a), then the correct opening will be created by the sweep operation. If the face is not perpendicular (figure 7.14b,c), the sweep operation will not necessarily be enough. Figure 7.14b shows the case in which the slot operation creates a self-intersecting object, which should be repaired. In figure 7.14c, the slot may be defined as extending far enough beyond the face boundary top to create a full opening, but the slot boundary creation stops short at the face boundary. In both cases, the model can be 'repaired' by intersecting the slot side faces and the slot bottom face or relevant end surface with the neighbouring face to create the extra model parts, and adjusting the slot topology and geometry created by the sweep to fit with these. In effect, this turns the operation into a 'local Boolean' operation, with the slot being defined by a set of surfaces rather than a solid.

The final step is to clean up by deleting the temporary geometry created, the curves and surfaces.

Concluding remarks

This slot creation example shows the general steps outlined in sections 7.1–7.3 in practice. The first section describes how the operation parameters are defined and how these are used to set up the operation. The second section describes the repeated middle step, which is used to define the basic shape of the slot. The third section defines the final step producing the slot.

For each of these, error conditions and special cases are identified, as far as possible, through the use of specific examples to identify problem areas. In some cases, this can be done logically by examining the possible results from the tools used, such as the types of intersections returned from the tool that intersects a face with a curve, or the edge types: concave, convex, smooth, or sharp. For other possible special cases, it is necessary to use imagination to try and identify what might happen and plan around it. The example of how to handle new faces created while defining the slot boundary is an example of this. These special conditions can be flagged as error conditions that terminate the operation, or the code can be extended to handle them.

One topic not covered here is how to handle error conditions that terminate the operation. One possible method is to preserve pointers to the new entities so that they can be removed from the object datastructure. During the slot shape definition process, the extra edges and vertices created can be noted and deleted using Euler operators if problems arise. The sweep operation is treated as a self-contained unit. If an error occurs during the process, then the operation itself may tidy up, if so implemented. Otherwise, it should be possible to identify any new faces and edges added by examining the edges bounding the partially swept face. If problems occur when trying to tidy up

the ends of the slot, then it may be better to return the partially created swept slot, and only remove the partially patched ends, rather than the whole slot. Sweeping in a straight line, as described in section 6.2, is self-inverse, and so once the slot has been swept, it is possible for the user to complete repairs. In all cases, the stepwise approach, using Euler operators to preserve model integrity, facilitates the repair process.

Once the slot has been fully defined in terms of its shape, it is possible to consider what else to include. The slot is a 'feature', a logical sub-unit of the complete model shape. According to some points of view, it is desirable to preserve this kind of information in the model so that it can be found and reused for applications such as process-planning and automatic assembly planning. There are several problems with preserving this information in the model that will be addressed in chapter 8. However, it is worth noting that the use of these special local operators, as opposed to general operators such as the Boolean operators, allows this kind of dynamic information and feature structure creation.

Another consideration is that it is possible to investigate potential manufacturing problems while designing with this kind of manufacturing-related operation. If, for example, any of the edges bounding the slot are concave, and the angle between the face on the opposite side and the given face is less than 90 degrees, then fouling will occur if the slot is to be machined. This kind of potential problem can be noted by the modelling system, and the designer informed, if the modelling operation is matched to particular machining processes. Another possible example, already mentioned, is where part of the face boundary intrudes on the slot shape and creates an intrusive projection into the slot (figure 7.13a). This may place restrictions on the maximum size of the tool that can be used to produce a slot. It may also indicate a possible problem if the slot is necessary for assembly.

Note, as well, that modelling operations that are meant to represent application operations, such as creating manufacturing features, have a technological as well as a modelling aspect. This means that the decisions made about how the operation works should correspond to the technical requirements of the application. If this is not so, then the tool may be an elegant solution that is less useful because it creates misleading information. An example of this is how the operation to set draft angles works with blends, as mentioned in chapter 6. Another example is chamfers that create self-intersecting models or even cut through the part, separating material. Detaching material may be a correct solution geometrically, but if the user is not at least warned, then this may not be noticed until the manufacturing stage; by which time, it may have damaged a machine tool.

7.4.2 Rebate an edge

This operation is related to the chamfer and blend operations, producing a stepped face instead of a single face, as illustrated in figure 7.15. It is part

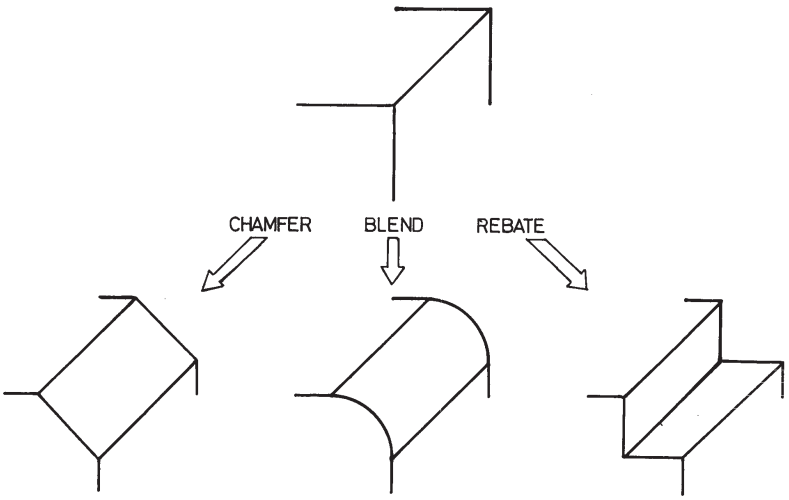


Figure 7.15: Chamfering, blending, and rebating an edge

of a general family of operations to produce a shaped-face or faceset from an edge. The face produced by the chamfer operation can be thought of as being defined by the motion of a straight edge along a path defined from the shape of the edge being chamfered (figure 7.17a). The face produced by the blend operation can be thought of as being defined by the motion of a circular arc along a path defined by the shape of the edge being blended (figure 7.17b). Similarly the faces created by the rebate operation can be thought of as defined by the motion of an L-shaped profile along a path defined from the shape of the edge being rebated (figure 7.17c). The operator definition and implementation described here draws heavily on the chamfer operation, illustrating how existing code and operations can be copied and adapted as a basis for new operations.

Some more exotic examples are shown in figure 7.16. These were produced for adding aesthetic types to edgings to an object. One of these is the simple rebate dealt with in this chapter, but the others also belong to this general family. Note, though, that the use profiles of these types of operation are different. Blends are traditionally added to many types of geometry and often run together. Chamfers are also applied to a wide range of geometry to avoid sharp corners and to help insertions. The aesthetic types of edging are probably more limited.

The top image in figure 7.16 is the original object. On the left of the second row is the object with a simple cutout of all edges. On the right of the second line, a rounded profile is included in the cutout. On the left of the bottom line, a square profile has been added, and on the right of the bottom

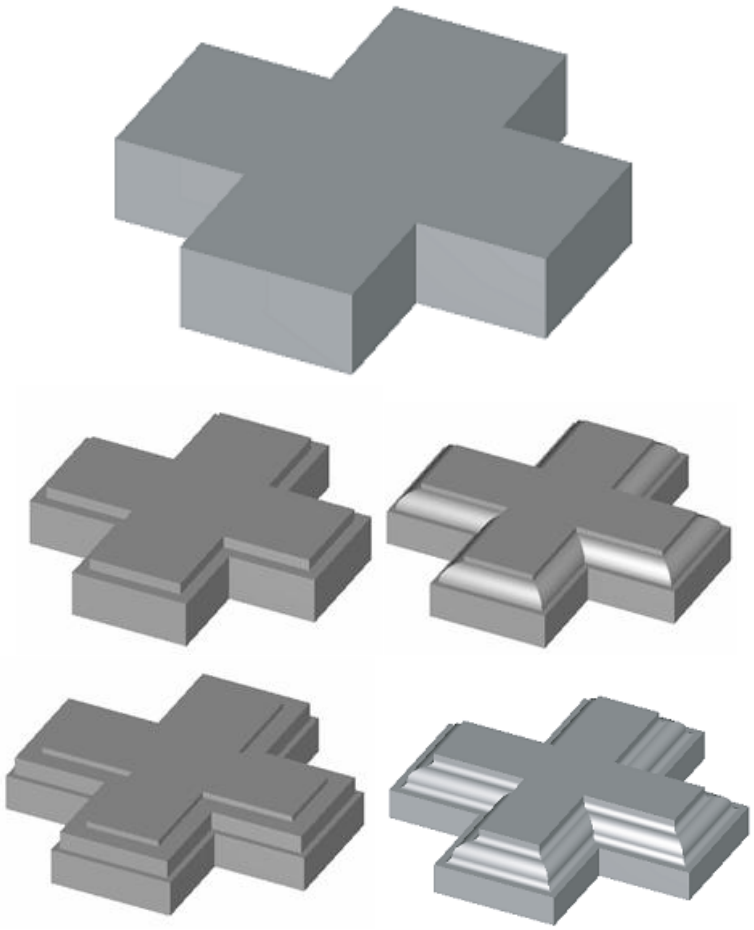


Figure 7.16: Aesthetic cutout examples (from [127])

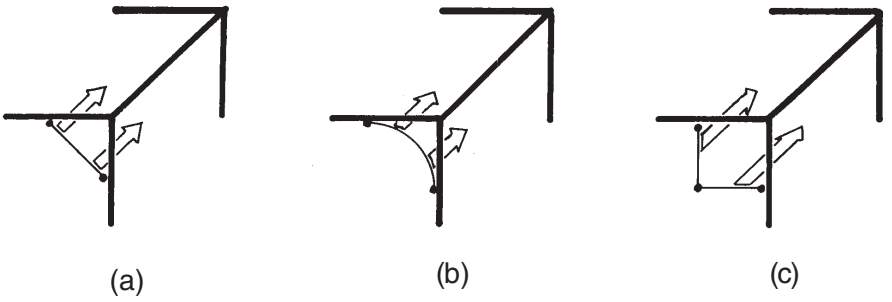


Figure 7.17: Different profiles for chamfering, blending, and rebating

line, a wiggly profile has been added.

The operation differs from that in the previous example because, this time, an existing part of the model is changed into something else. The operation can be considered to be a finishing type of operation, applied once a design is basically complete.

The operation is specified with an edge, and a real value, greater than zero, to specify the offsets of the rebated surfaces. For the purposes of the operation, only straight and circular edges will be rebated. Also, only convex edges will be rebated, the operation having little meaning for concave edges or smooth edges. Special edge types, wires, and spurs will not be handled, and it will cause a warning message to be generated.

The reason that only straight and circular edges are handled is because the operation works using offset geometry, and this simplifies the creation of the offset surfaces. Deciding this at the outset is a little strange, because normally it it is necessary to decide on the algorithm and then decide on its limitations. However, for this operation, it is possible to draw on experience with similar operations, as well as on general modelling experience. In general, with the type of modelling operation that simulates the action of a single manufacturing operation, it is best to adapt the modelling operation to the manufacturing operation. If the manufacturing operation can produce more complicated rebated edges, then the modelling operation should be extended.

The decomposition of the operation into basic steps involves:

- Creating the geometry
- Creating the side edges and rebated faces
- Adjusting the geometry and the ends of the edge

Initialisation

The first task is to check that the preconditions for the operation are fulfilled, i.e., that the given real value is correct, that the edge is convex (and lies

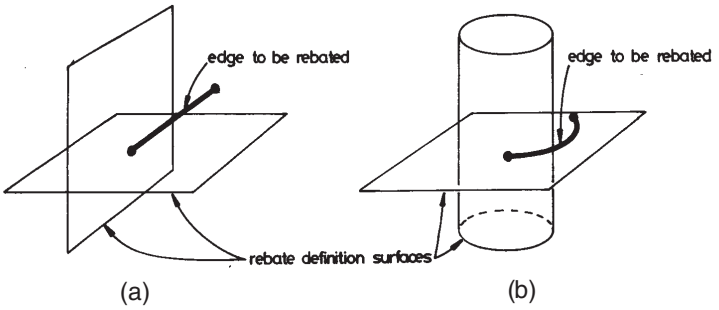


Figure 7.18: Offset geometry for the rebate operation

between different faces), and that it has the correct geometry. Assuming that these initial conditions are met, the surfaces used to define the rebate are created.

If the edge is straight the surfaces will be planar, with normals that are calculated from the average of the surface normals at the edge and the edge direction, (figure 7.18a). If the edge is circular, the surfaces will be a plane and a cylinder. The plane is parallel to the plane of the curve, offset as specified by the input parameters. The cylindrical surface has an axis passing through the centre of curvature of the curve, and axis parallel to the normal of the plane of the curve (figure 7.18b). Which side of the curve is used, and hence the radius, depends on where the material is with respect to the edge. In the figure, the material is towards the centre of the circular arc, and hence, the radius is smaller. Note that if the given offset value is greater than the radius of curvature of the edge, then an error should be signalled and the operation should be stopped.

Defining the rebated shape (the repeated middle step)

To define the boundaries of the new rebated edge, the surfaces are intersected with the surfaces of the relevant faces and the edges immediately adjacent to the supplied edge. If any of the adjacent edges do not cut the appropriate surfaces, then an error condition exists.

If the surfaces cut the adjacent edges at existing vertices, then the edges are unchanged; otherwise, they are split at the intersection point. The process starts, say, with the edge counter-clockwise around the right loop, intersecting this with one surface and producing a start vertex (figure 7.19a). Then the same process is applied to the edge clockwise around the right loop from the edge being rebated, producing a second vertex. These vertices are then

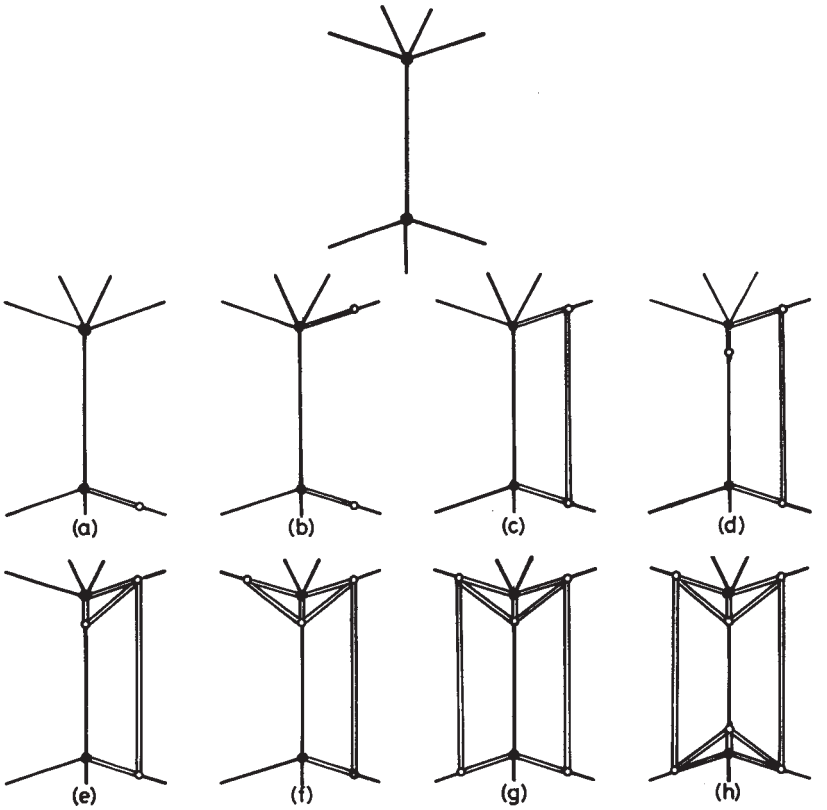


Figure 7.19: Adding the new rebate topology

connected, producing a small, four-sided face on the right of the edge (figure 7.19b).

The next step is to produce the topology for the end vertex of the edge. If the vertex has only three edges attached (the edge being rebated and the two adjacent edges around the vertex), then nothing is done and processing proceeds to the edge counter-clockwise around the left loop from the edge being rebated. If there are more than three edges (figure 7.19c) at the vertex, then it is necessary to create 'eaves' as in chamfer ([11]). To do this, the edge being rebated is split at the same position as the end vertex, the new vertex becoming the new end vertex of the edge being rebated, and the new edge being inserted between the new vertex and the previous end vertex (figure 7.19d). The previous vertex, produced by splitting the edge clockwise around the right loop from the edge being rebated, is then connected with this new vertex, producing a new, triangular face (figure 7.19e).

Next, the edge counter-clockwise around the left loop from the edge being rebated is processed. As before, it is intersected with the appropriate surface, and the edge split at the intersection point, if in the middle of the edge, or an end vertex used if the edge cuts the surface at one end. If eaves are being created, this vertex is connected with the new end vertex of the edge being rebated, creating a second triangular eave (figure 7.19f). The edge clockwise around the left loop from the edge being rebated is handled in the same manner. A vertex is found or created and connected to the previous vertex with a new edge (figure 7.19g).

Finally, the start vertex of the edge is handled in the same way as the end vertex, building eaves if there are more than three edges at the vertex, otherwise leaving it unchanged, as in figure 7.19h.

The repeated step is essentially to obtain (create or find) a vertex, and then to connect it to the previously obtained vertex. Obtaining the first vertex is included in this repeated step by making the previously obtained vertex a NIL pointer initially, and by handling it as a special case.

One error condition has been mentioned, where the edges adjacent to the edge being rebated do not cut the surfaces defining the rebate. It is convenient to terminate the operation if this happens, rather than to try and cope with it, in case the user has defined an unrealistic value for the edge to be rebated, or that the edge is not as the user thinks.

Another problem, not mentioned above, is if the start or end vertices of the edge have less than three edges attached. If there is one edge attached, then the edge is a spur edge, and this type of edge was specifically excluded from consideration as one precondition for the operation. However, it is possible for a vertex to have only two edges attached. This may be an artificial example, because normally vertices will have three or more edges, and vertices with just two are usually degenerate vertices; however, some cylinder representations may contain such vertices. It is important, though, to identify such gaps in the modelling operation algorithm, and to plug them, either by making them error conditions or by adapting the code to handle them. If such vertices are treated as error conditions, then the vertices at each end of the supplied edge can easily be checked as part of the initial checks applied to an edge. If the code is to be adapted to handle this type of vertex, then it is possible to create a plane, passing through the vertex and with normal identical to the edge tangent direction at the vertex, and use it to create two extra, artificial edges at the vertex.

Adjusting the geometry and the ends (the final step)

The middle step, the repeated step, essentially isolates the edge topologically so that the new geometry can be assigned to create a consistent object. The final step performs these ‘tidying-up’ steps.

The two small, four-sided faces on either side of the edge are set in the new surfaces set up originally to define the rebate. The new curve of the edge being

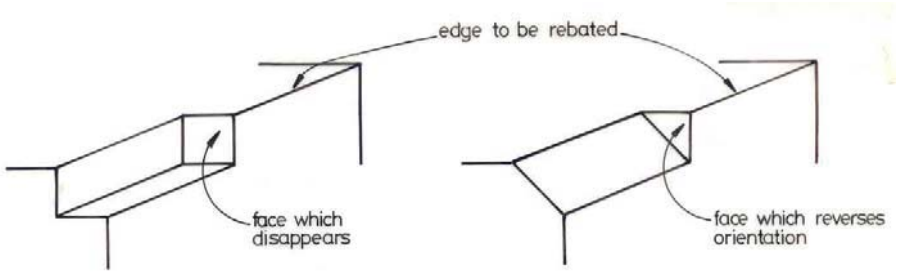


Figure 7.20: End face conditions for rebated edges

rebated is the intersection curve of these two surfaces. The positions of the end vertices can be calculated from where this curve cuts the end surfaces, if eaves were not built, or calculated if eaves were built. The curves of the edges adjacent to the edge being rebated also have to be adjusted to be consistent. The new curves can be calculated by intersecting the rebate definition surfaces with the surfaces of the faces on the opposite sides of the edges from the rebate faces. If eaves have been created, then it is also necessary to define the surfaces of these. As the eave faces are three-sided, it is possible to define a plane for the eaves passing through these points. The curves of the edges adjacent to the rebate edge will be straight, if the rebate definition surface is planar, or can be calculated by intersecting this plane with the non-planar rebate definition surface.

One extra problem is to adjust the ends of the rebate, if the rebated edge is adjacent to a previously rebated edge, as in figure 7.20, left, or adjacent to a small face, as in figure 7.20, right. In the case where the end faces were eaves built in a previous rebate operation, when the geometry of the ends is adjusted, the end faces become degenerate, with zero area. An extra check is necessary for this, to see if edges around an end face become coincident, and to remove both an edge and the degenerate face. If the end face is small, its orientation will be reversed with the geometry. This can be determined from the relative orientation of an edge adjacent to the rebated edge, and an edge adjacent to it in the end face before and after the geometry adjustment. If the cross product of the edge vectors is reversed, then the surface of the face should be negated.

Concluding remarks

As with most modelling operations, it is common to design and implement an algorithm for a modelling operation that copes with as many cases as can be identified by the developer, but it is usually only when other people start using it that many special cases become apparent. This is because different people see operations from a different perspective, and use them according to their perspective. This can be both useful and difficult. It can be difficult

because other people may expect an operation to behave differently to the way the developer intended in special cases. The developer then has the choice of modifying the algorithm, or making the original design decisions more apparent. Which to choose should be a matter for debate rather than dogma. Choosing what to do is easier for operations such as this one, to rebate an edge, which have a physical manufacturing interpretation rather than being very general shape creation operations, like the Boolean operations. If in doubt the operation should make a modification corresponding to the physical operation as closely as possible.

Generally, in the case of error, the operation should try to patch up the object and leave it unchanged, as with the previous example, and inform the user about what went wrong. This gives the user the chance of rethinking his changes, or making the same changes in another way, or saving a model until the operation can be modified to perform the way the user wants in a particular case.

Information creation is also facilitated in operations corresponding to finishing-type operations. The parameters concerning depth, and the original shape of the edge being rebated, may provide relevant information for a physical process. Also, if the operation corresponds to a finishing-type process, it may be applied after a design is more-or-less complete, so that there is little or no risk of information corruption.

Chapter 8

Product modelling

This chapter deals with the addition of information to models and its preservation during modelling. Addition of information to models is part of the development of product modelling. Using specialised modelling tools helps, to some extent, in the creation of this information by making it possible to record how parts of the model were created. Having more specialised tools for modelling should also mean that it is possible to determine when information clashes occur. However, the modelling tools described in chapter 6 are for changing shape only and need to be extended if they are to handle this extra dimension. Some possible techniques for doing this are described in this chapter.

The addition of extra datastructures and information to models has important implications for the modelling operations that have to handle models with this associated extra dimension. These extra model elements provide a high-level interpretation of part or all of the model that lies outside the realm normally handled by the modelling operations, which deal with the shape of the object in terms of simple boundary elements. Modifying a model independently of the extra information is analogous to modifying the topology of a model without modifying the geometry, leaving the model inconsistent. The problem is that traditional modelling techniques deal only with shape entities and ignore extra elements like information and feature datastructures.

The need for inclusion of information with a model has come about as the role of modelling systems has grown from stand-alone systems for producing drawings to communicating stations in an information-sharing production environment, so-called "Product Modelling".

Another problem for modelling is that modelling systems provide different tools from those traditionally available in manufacturing environments. The aims of commercial CAD systems have grown from the original ones of providing electronic drawing boards, into being solid modelling systems complete with applications. An aim for modelling development is to try and provide a flexible tool that can be adapted as new ways are found to use it. An example

of a possible change in practice concerns the use of dimensions and tolerances. If dimensions are used purely for communicating exact two-dimensional measurements between applications, they may not be necessary if each application has the same, exact computer model available for direct measurement. Also, if tolerances are used to define clearances between objects in an assembly, then the computer model of the assembly should be used to define the object segments where such tolerances are necessary, rather than forcing the user to set tolerances directly, when there is a risk of setting tight tolerances on model segments that do not contact other parts. Much work has been done on dimensioning and tolerancing already [57], [47], [67], but it seems unclear how this information should be communicated in a modelling environment. This is, admittedly, a naive view of dimensioning and tolerancing, based on modelling experience rather than design or manufacturing experience, but it represents something of the dilemma for the modelling software developer. Should methods be found for handling dimensioning and tolerancing as used at present? Or should effort be expended on developing equivalent tools, more suited to a modelling environment? The question is deliberately intended to be provocative to illustrate that former practices should be examined in the light of the changed environment offered by computer-based tools.

To try and avoid specific production examples, in the following, general classes of information are identified and strategies for handling them are outlined.

8.1 Product modelling

A full analysis of information creation and use in manufacturing is difficult except by examining particular examples, which is not the purpose of this chapter. However, to identify some general areas, consider the diagrammatic representation of a hypothetical manufacturing environment shown in figure 1.5. The complete description of the product used for communication is the so-called "Product Model". As this section is concerned with identifying the types of information associated with the solid model and how to handle them, this model is intended to be representative rather than definitive.

The product model contains more information than simply an annotated shape model. For example, the initial information in the product model is likely to be a product specification, as determined by a management group. The design, in the form of an annotated shape model, may be checked against the specifications, but these do not directly contribute to the solid model, so it can be ignored for the purposes of this discussion. Product price information and part numbers are also relevant to the product model, but not directly to the shape.

The main areas in the diagram that create the shape model and the associated information structure are as follows:

- DESIGN

- CUSTOMER-BASED ORDERS
- PRODUCTION PLANNING

8.1.1 Design information

There are two types of design result that should be examined separately. These are single part design and assembly design. They are treated separately because they are used in different places in the production model in the figure. For example, a single part will have to be manufactured in some way, and so be examined by production planners, while for an assembly, the final product or sub-systems will have to be put together in some way.

Before examining these two types, it is necessary to say something about features, which will be described further in chapter 9. The shape of the model may be unstructured or divided into features that are higher level shape units than the basic elements of the model (faces, edges and vertices). There has been much work done on features and feature structures, and it is possible that this information is a result of the design process. The philosophies concerning features, their use, and their origin are legion. One philosophy is that the designer starts with a blank or some basic shape and designs the final part by adding non-overlapping features, so-called "Design by features". This method seems to have only limited applicability, such as in the case where small modifications are to be made to an existing design. However, it is possible that some information can be added in this way through the use of operations that produce well-defined features. Another 'fact' about features is that the nature of what makes a feature a feature varies. What may be a feature for a designer may not be interesting as such for a process planner. A simple case of this is that of a 'boss' feature. A designer may create the boss in a single well-defined operation. For a process planner, however, the material to be removed to create the boss is of more interest. Another feature philosophy suggests that it is possible to transform one set of features into another, so that it is necessary only to retain a single set. Yet another philosophy is to retain only the basic shape of the model and to perform 'feature recognition' on objects to recover the relevant feature decomposition for a particular application. This can be wasteful in that some of the same features may be 'recognised' several times, but on the other hand, it avoids the problem of maintaining an extra parallel structure during modelling, and it may mean that feature elements that a designer has inadvertently created are identified. In general, it is reasonable to suppose that there will be several coexisting feature interpretations, one or more of which may be created by the designer; others are created in other applications.

Single part design

The areas where the single part model and its associated information structure might be used are as follows:

- ANALYSIS
- PRODUCTION PLANNING
- CATALOGUE PRODUCTION
- SUB-CONTRACTORS

For analysis purposes, the shape of the part, implicit shape transformations such as implicit blends or screw threads, and the material from which the part is made are important. For finite element analysis and weight and moment analysis, there is, possibly, no need for structured shape, the basic elements, faces, edges, and vertices are enough.

For production planning the shape of the part, implicit shape modifiers, the material from that which it is made, and additional notes about attributes of parts of the model, surface finish or colour, are important. Here, some feature information created during the design process may be useful. Design tools that correspond to manufacturing operations, such as chamfer or the slot-making operation, described in section 7.4.1, may directly produce structured shape descriptions that are useful for manufacturing. Similarly the shape modifiers may correspond to manufacturing operations that are complex or unnecessary to model exactly but which can be produced simply during manufacturing (such as screw threads). The attributes provide extra information or constraints that have to be considered.

For catalogue production, the unstructured shape of the parts is needed to produce the pictures for the catalogue, the shape modifiers, and any additional attributes relevant for production planning.

The same sort of information is needed for communicating with subcontractors as is needed for production planning and catalogue design.

Assembly design

Assemblies represent the finished products or sub-assemblies for the finished products. The design of such assemblies differs from the single part design, in modelling terms, because of the different information structures surrounding the design, as well as their complexity.

The areas where the assembled design model and its associated information structure might be used are as follows:

- DESIGN
- SIMULATION
- PRODUCT AND PRODUCTION STRATEGY
- ASSEMBLY
- CATALOGUE PRODUCTION

- CUSTOMER-BASED ORDERS

The information supplied back to the design process concerns the initial sketch models of an assembled design and the geometric framework surrounding a single part. The initial design sketches provide information about the organisation of a product into separate pieces that are then designed piecewise. Information about which parts are connected or adjacent provides a geometric framework within which a part may be designed [69]). This might be recorded as free-standing geometric entities, or as portions of neighbouring parts ([145]). Another approach was taken by Csabai [25] who developed methods for defining a layout with kinematic connections between simple shapes (rectangular blocks and cylinders). The kinematic connections also provide a geometric framework for design.

Similarly, information about the functionality of a design, how it is connected, and information about these mechanisms allows simulation of the product to check whether it functions as expected.

The finished product will, presumably, be evaluated and approved, or returned for modifications, by a management group responsible for product and production strategy. For consumer products, one part of this evaluation will probably be visual, in terms of advanced graphics. This requires the shape of the outside of the product, together with colour and material information.

For assembly, the basic linkage information from the design sketches is useful. Feature information is also useful to match linking features, bosses, and pockets or slots, say. As before, this information may be supplied directly as ready partitioned feature structures, if suitable, or recognised anew.

For catalogue production the basic shape of the part is needed, together with any attributes, like colour, material, and so on.

Customer-based orders require information about what can be changed in a product design. The simplest information of this kind is the attribute information such as colour, but design parameters may need to be changed as well. Also required is information about the fixed and variable parts of the design, either sub-parts of an assembly or individual features in a part.

8.1.2 Customer-based order information

Once the initial basic design has been decided, it may be desirable to allow customers to specify variants. As indicated, basic information about colour and visual attributes may be supplied according to individual customer requirements.

Changing the shape of the product to suit individual requirements is more complicated. It may be possible to transform the product design by including new parts in an assembly; in which case, the linkage information from the design is used to establish new positions for individual parts.

It may also be that individual parts in the product may be changed, either by modifying specific features or parameters, if the part is defined paramet-

rically [55]. The information supplied from this process will, presumably, be an exact shape model, suitably annotated, for manufacturing and assembly.

8.1.3 Production planning information

For production planning, there may be several interrelated parts, such as process planning, factory planning, and warehouse management, for example. Factory planning, warehouse management, and all the others involve many interesting and complex problems, such as creating factory models for the manufacturing facilities needed for the product. However, although these topics are interesting in their own right, they will be ignored here. Only process planning will be considered.

For process planning, when examining a part to be manufactured, the first question to be considered is whether to buy it, manufacture it, or subcontract the manufacture [66]. Buying a part is suitable if it is a standard part, such as a bolt. This requires no modification and no information handling in the model. Similarly, subcontracting the manufacture to another company may be a complex enough task, involving communication of the product model of a part, but it does not require information handling for the company modelling environment. Only if the part is manufactured by the company is there some need for looking at the manipulation of information in the model.

In this simplified production model, it is assumed that the model shape and associated information structure represents the final part shape. This may then be modified, if necessary, in conjunction with the original designer to facilitate manufacturing. Once the shape has been finally fixed, the manufacturing plan has to be produced. To identify the information handling processes, it is assumed, here, that this involves 'un-making' the final part shape until a suitable 'blank' has been identified. This is obviously an over-simplification, because process planning is a complex process, but this section is concerned with information and modelling, not with process planning as such. Tasks such as jig and fixture design, and operation sequencing, for example, are too complex to consider in specific terms.

The structured shape description produced in design may be only partially useful for manufacturing. If the design was created using manufacturing-related modelling operations, then this information may have been recorded in the model and can be retrieved for use in process planning. However, operations for creating extrusions, which may be useful and reasonable operations for a designer, create structures that cannot be used directly but have to be transformed or the model re-recognised to retrieve the desired information. At the same time, features such as bosses may well be useful for later processing, for assembly, say. Other 'features', such as 'clamping features', necessary for analysing the manufacturing sequence, are unlikely to have been directly modelled as such, and so they are unlikely to be included in the part design as a specific features. This supports the view that there should be multiple coexisting feature interpretations associated with the model, one or more of

which are created for process-planning purposes.

Some parts of the model may well have associated information, such as surface finish, which are important for the functioning of the product and which are related to manufacturing. In the process planning task description, above, the original part model is successively modified back, removing such information and certain features, through a series of intermediate models until a suitable initial form has been reached. This might be a block of metal of standard shape, or a part that can be produced by casting. This means that the information in the series of models is gradually transformed. For example, part of a model with a particular surface finish might be offset and assigned a new, less exact surface finish, simulating the inverse of a finishing operation. This in turn may be offset, and the surface finish removed, or even a complete feature removed, simulating the inverse of a roughing operation.

This process of modifying the finished model involves translating information into shape and removing structural elements, features. Although this is, in many ways, almost the inverse of the design operation, it still requires that the information in the model be handled properly so as not to corrupt information in the non-affected parts. The types of information affected are basically those introduced by the design process, and the extra manufacturing type features that might be produced by applying manufacturing-oriented feature analysis.

8.2 Information classes

From the above, the following information categories can be identified:

1. SHAPE
2. SHAPE MODIFIERS
3. FEATURES
4. ATTRIBUTES
5. CONSTRAINTS
6. GEOMETRIC FRAMEWORKS
7. CONNECTIONS

8.2.1 Category 1 – shape

This concerns the basic shape of the model, and it is the information category that is mainly dealt with by conventional modelling operations.

8.2.2 Category 2 – shape modifiers

This includes implicit shape modifiers such as implicit blends and screw threads. These modifiers are used to describe specific modifications to the basic shape. The common element is that it is not necessary to model these exactly, either because they do not contribute a great deal or because they are too complex to model, but relatively simple to manufacture. An example of the first type is the implicit blending tool developed by Braid for BUILD [11]. Braid's philosophy was that these blends were added by a mould maker by running his thumb along parts of the mould. A later implementation in the Romulus system was reported to have included a 'thumb-weight' parameter for shape control [112]. Slightly different are implicit shapes such as screw threads. These are complex to model and important parts of the basic shape, but they, too, may not need to be modelled exactly. Adding screw threads is a straightforward manufacturing operation and retaining the shape in a high-level form may actually be preferable to modelling the exact shape and having to rediscover that it represents a thread.

8.2.3 Category 3 – features

This information concerns features, which have a higher level information content than the basic shape elements and build shape sub-structures in a model. Much has already been said about these; they were originally investigated in BUILD by Kyprianou [74] for the purpose of shape classification, and later included with the model as extra datastructures for process-planning [66].

8.2.4 Category 4 – attributes

Category 4, or attributes, can be classified as 'passive information', such as colour. This information does not affect the shape, as that in category 2, but it provides an additional, 'additive' element to the design.

8.2.5 Category 5 – constraints

This category concerns constraints on the shape, such as surface finish. This type of information does not modify the basic shape, as category 2 elements. Nor is the information passive, as with category 4 elements. The most obvious example is that of surface finish; another example might be tags indicating the material from which a part is to be made. These two examples are important for applications, and they are essentially passive when building up a model. A different type of constraint information, which is important when creating a basic shape, is where parts of the same model are connected with a specific type of relationship, such as being at a fixed distance, or at a specific angle to one another.

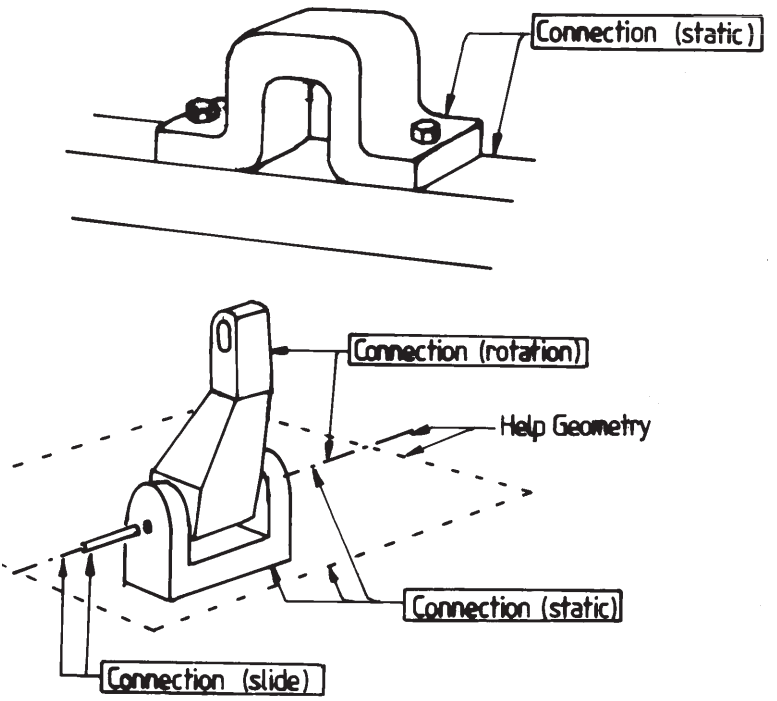


Figure 8.1: Illustration of assembly constraints

8.2.6 Category 6 – free-standing geometry

This concerns free-standing geometric frameworks that might be set up as a guide for defining the limits, or the shape of parts of a design or even for centre lines and movement guides.

8.2.7 Category 7 – connections

Finally, category 7 is concerned with the connection information that can be used to define linkages between the separate parts of a product design. These linkages may be 'static', defining that two objects are to be fixed together, for example, welded, glued, or bolted, or the linkages may represent some kind of movable relationship. Examples of these are shown in figure 8.1. These differ from the constraints mentioned in category 5 because they refer to whole objects. Category 5 constraints are concerned with limitations on the shape of single parts. This category is concerned with constraints between separate objects in an assembly.

8.3 Handling information during modelling

As far as modelling is concerned, the role of a computer shape model, be it wire-frame, surface, or solid, is as an information carrier in a larger product model. Unfortunately this places heavy demands on modelling techniques because of the increased complexity of the model. Modelling methods have been developed mainly for handling the shape of objects (category 1 of the seven mentioned), with a few ad hoc methods for handling some other types of information. To use the model successfully as an element of a product model, the basic modelling techniques have to be extended to cope with extra information categories associated directly with the model.

One method for coping with these extra entities, used in the Romulus and ACIS systems, for example, is to provide a 'bulletin board' where the entities modified during a modelling operation are recorded. Using this facility code developed for an application can attempt to keep track of the attributes and modify them as required. The ACIS system also allows users to define a set of basic attribute handling routines that are called by the modeller when appropriate. However, both of these solutions may be confusing for a user or for the application programmer who is forced to understand the basic functioning of the modeller.

Otherwise there are a few empirical methods for handling information. For example, in BUILD, when an edge that has been marked as having an implicit blend is split, then the blend information is copied. In GPM attributes that belong to faces are copied during the Boolean operation processing, although regardless of their meaning.

The stepwise method of developing modelling operations means that information handling can also be applied in a stepwise manner, for the information categories linked to the basic elements of the model. Also, by making each complete operation a series of well-defined sub-operations, the model can be made locally correct, and thus be properly interrogated during the course of the operation. This means that it is possible to perform some simple types of 'geometric reasoning' (Jared [64]) while an operation is being performed.

As far as the information is concerned, the initial requirement is that all classes of information in the model can be distinguished in some way. How this is done for model shape elements and, by extension, the geometric framework has been covered in chapters 3 and 4; the other categories need to have some sort of class identifier so that they can be distinguished. Information handling techniques should be developed to avoid the need for applying particular solutions for particular information types wherever possible, which is why the information was divided into categories, above, so that general strategies could be outlined. However, features can be divided into two general classes: obtrusive and intrusive (Jared [64]). Similarly, various types of mechanism can be differentiated. Generally, identifiers are important for determining when information clashes take place, and so the identifier can be used to check whether two pieces of information are of the same type even

though their actual meaning is not needed.

The information handling methods proposed here are described category by category.

8.3.1 Category 1 – The basic shape of the model

This is dealt with by the modelling operations, so it is not described here.

8.3.2 Category 2 – Shape modifiers

Shape modifiers imply that part of a model has a different shape to that specified by the faces, edges, and vertices, with their associated geometry. It is likely that this information is added after the basic shape of a part has been defined, so it is unnecessary to have a complex information handling strategy. However, it is easy to envisage circumstances in which this information may be simply modified, say if a part is sectioned for viewing purposes.

The simplest strategy is to delete the modifiers as soon as the entity to which they are attached is involved in a modelling operation. The user would then have to replace the information. However, this is unnecessarily restrictive and causes a lot of extra work for the user.

Another strategy is to have an 'evaluator' for each implicit shape, and to evaluate the model parts affected in any operation. This strategy has the advantage that it provides a simple basic strategy; however, there are several disadvantages. It may be complicated to implement in practice, especially if threads have to be evaluated. It also means extra work in terms of creating the evaluators that may, to some extent, run counter to the advantages of using them as implicit modifiers.

A better strategy is to be more selective, delete the modifiers when the geometry type of the entity to which they are attached is changed, copy them if the entity is split, and avoid evaluating them.

The condition that the modifier is deleted when the geometry type of an entity is changed is, for example, to avoid problems of having elliptical holes marked as to be threaded. The reasoning behind it is that, because the modifier is concerned with shape, if the shape of the underlying entity is changed, so will the modified shape. If the entity is split, or extended, then the basic shape is not changed, and the modifier is copied to the new part or remains.

A different problem results when merging two shape entities, for example, merging two edges by removing the common vertex, or merging two faces by removing the edge between them. Here, types of information in the various categories should be compared to check whether each entity has the same type of information attached. If two information items of the same type and with the same content are found, then there is no problem. If the content is different, one of the two should be chosen and a warning should be generated.

8.3.3 Category 3 – Features

Feature datastructures can be added as the result of applying feature-oriented modelling operations, which create new features, or as the result of recognising a part, either the final shape or a partially finished shape.

The problem is that the feature datastructures are separate from the basic shape of the part, but they contain the same sort of information. This can, of course, result in information conflicts if consistency is not maintained. If the basic shape of the object is always reanalysed to extract the feature structures, then there is no problem because it will always be based on the current shape of the object and the application's needs. If the feature structures are added during modelling, and the feature information structure is meant to survive during modelling, then the structures have to be maintained because the feature information is useful only if it can be relied on to be correct.

One possible strategy for dealing with features is to have a set of modelling operations for each feature type to create, modify (MOVE, CHANGE DIMENSIONS, etc.), and delete them. These operations would then take care of preserving the integrity of the feature datastructure and any associated information. However, although this may provide a user with many useful and logical tools, this strategy precludes the use of general operations like the Boolean operations and general sweep operations. This forces the user to limit, and possibly warp his or her thinking to make changes with a limited set of modelling tools, an aim inconsistent with the purpose of providing features in the first place, as high-level shape sub-parts that are more logical to use. The aim should be to adapt the modeller to cope with the information, not to adapt a user.

The basic feature handling strategy, proposed here, is that specific feature handling operations, as mentioned, should be provided for use, where possible, to create and modify feature structures, and to perform re-recognition on features possibly corrupted by general modelling operations. Having identified features as explicit datastructures facilitates the process of re-recognition, or validation, and re-evaluation of feature data. This allows both the modeller, as part of an automatic process, or a user to ask explicitly whether a particular feature is of the same type as that created. This also requires that feature definitions are provided in syntactic form so that the feature recogniser is 'programmable', as proposed by Jared [63].

There are two possible approaches for analysing the effect of modelling operations on existing features. One is to use information about the nature of the modelling operation being applied, and to perform high-level reasoning about how a feature is affected by the operation. The other is to perform low-level geometric reasoning about the new or altered model elements. Using the nature of the operation for geometric reasoning is possible for the 'local operations' because of their specific nature; however, this is not possible for general operations where the nature of the changes to the shape cannot be determined so easily. For this reason, it is sensible to use a two-level approach,

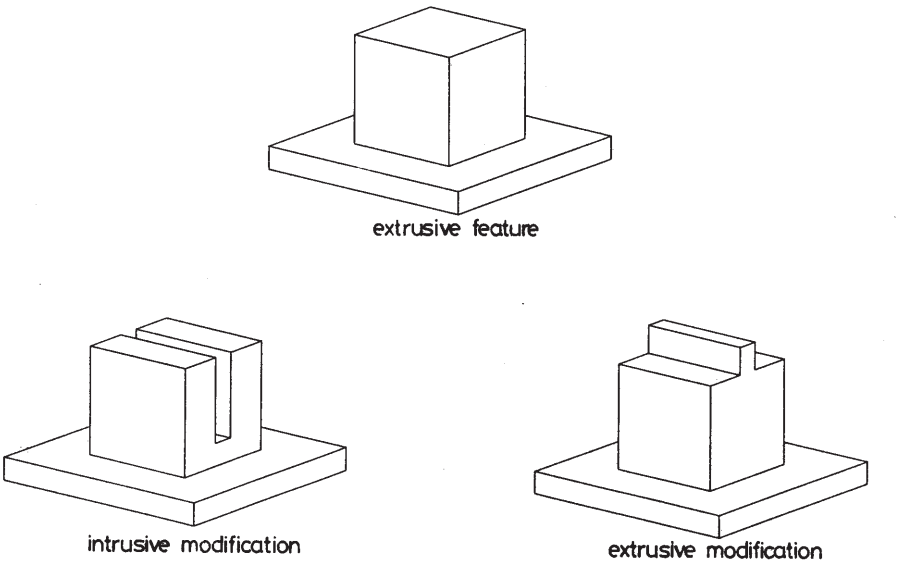


Figure 8.2: Extrusive feature and modifications

using the information about the operation where this is available, and using the general strategy for other cases.

The general, low-level feature checking approach uses the same kind of analysis techniques as applied by Kyprianou and Anderson to recognise objects during modelling. Many features can be classified as 'obtrusive' or 'intrusive', and the model elements affected by an operation can be examined to see whether they are consistent with the basic nature of the feature. Figure 8.2 illustrates an obtrusive feature, a boss, and intrusive and obtrusive modifications. Figure 8.3 shows an intrusive feature, a slot, and intrusive and obtrusive modifications.

For Boolean operations, this means checking the edges in the interaction boundaries to see whether they are parts of features, and analysing the edges in the merged interaction boundary for concavity, convexity, or smoothness. If edges are analysed as being smooth and lying between faces in the same surface, then normally they would be removed (providing they are not 'fake' edges). However, if these edges lie between faces in different features, or between a feature and the root node, as illustrated in figure 8.4, then there are extra considerations. One strategy is not to delete such edges, leaving the features separate. This is not a particularly good strategy; this type of edge is very artificial and leaving them adds complexity to the model. Also,

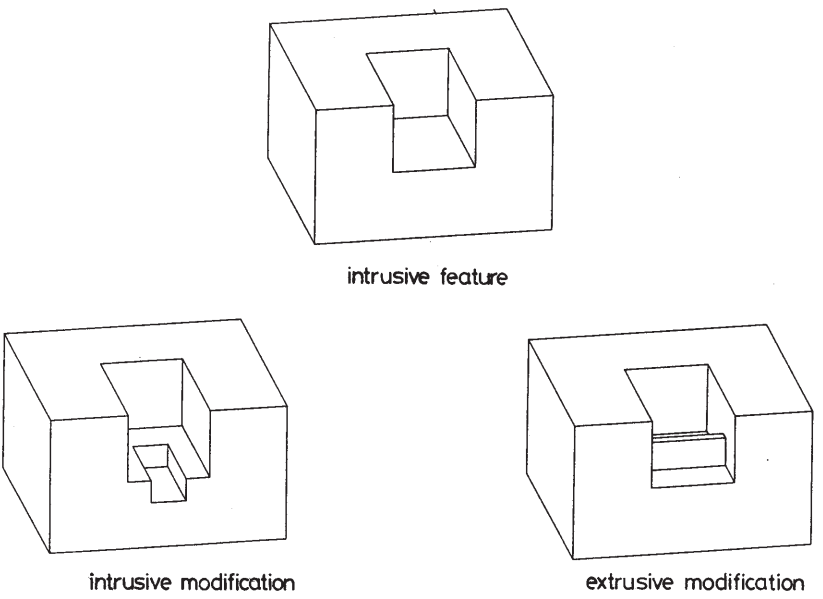


Figure 8.3: Intrusive feature and modifications

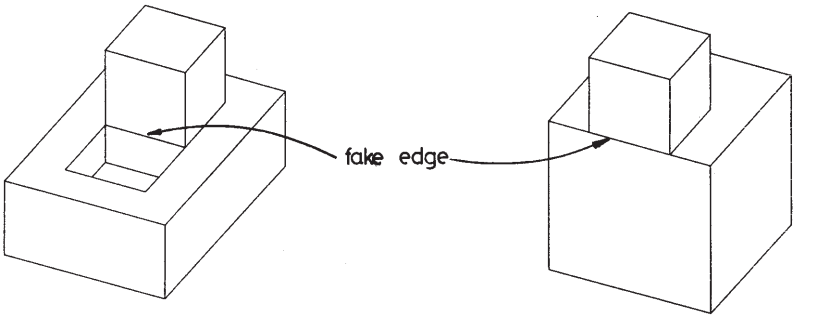


Figure 8.4: Fake edges in the model

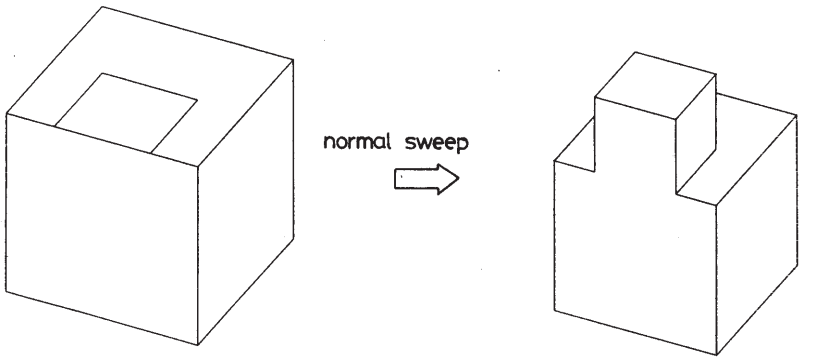


Figure 8.5: Boss made by sweeping

as illustrated in figure 8.5 where the result object has a boss feature with one face in common with the root, creating a feature by sweeping may mean that such edges are never part of the datastructure. If the feature recogniser has to cope with this case anyway, then removing such smooth edges remains a natural choice. However, when removing such an edge, it is necessary to note that the faces lie in different subparts, and to replace the reference to the deleted face in one feature with a reference to the new united face.

In sweeping, as described in section 4.2, edges are classified as movable if they can 'slide' in faces adjacent to a face being swept. However, if an edge is movable in geometric terms, but is adjacent to a feature, then it is necessary to decide whether it should be left fixed instead. Figure 8.6 shows two examples of such edges. If an edge adjacent to a feature is moved, then this means that the feature face will be expanded or contracted, possibly becoming a shared face with another feature.

A related problem concerns maintaining information about feature dimensions, if this is stored as part of the feature structure. It is conceivable that

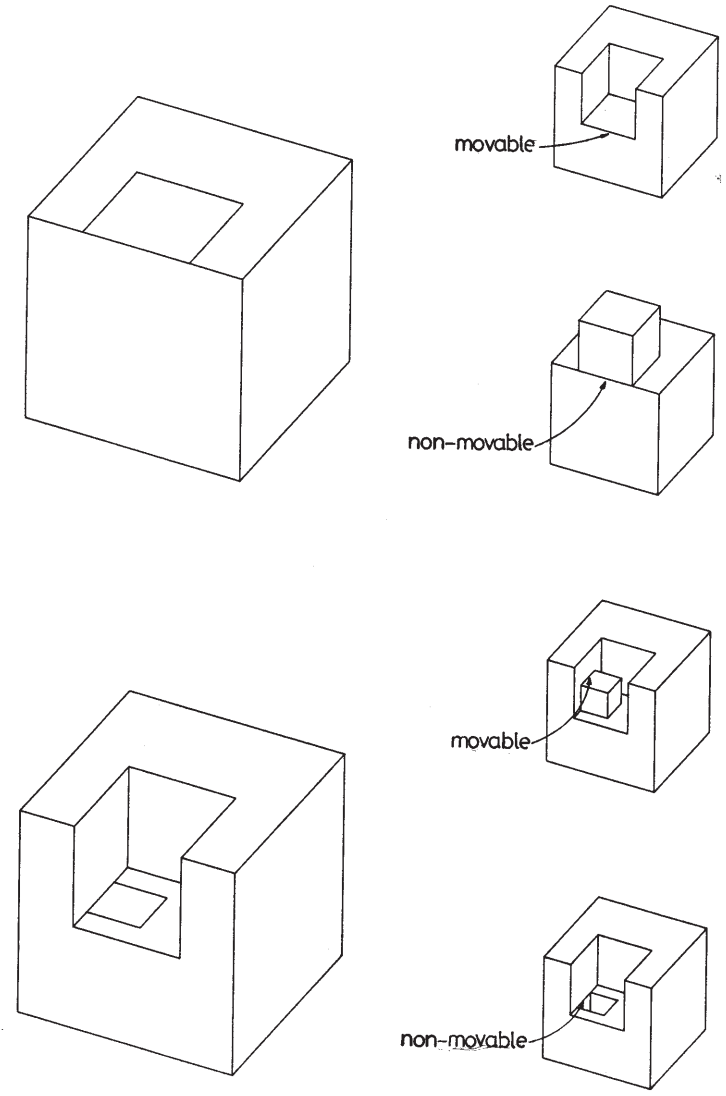


Figure 8.6: Movable and non-movable edges when modelling

modelling operations that create specific features could record this kind of information from the parameters given to specify the operation. The problem for modelling operations is to recognise this information, if stored, and to be able to tell how a feature has been modified to be able to change the information. This, again, can be handled by reanalysing the model and extracting the dimension information from the geometry of the model.

8.3.4 Category 4 – Passive information

This kind of information has, specifically, no effect on the shape of the model, and is not affected by it. The task of handling it during modelling involves copying information to the new part when dividing a model entity, and checking for the same information types and information equality when merging entities. If entities being merged have information of the same type, but different content, then a simple strategy is to give a warning to the user, and to preserve both pieces of information so that one can be removed explicitly. This implies that it is necessary for applications to allow for information conflicts, and to query if these conflicts cannot be resolved. Where two entities are being merged, then the information set belonging to the deleted entity should also be merged with the surviving entity.

8.3.5 Category 5 – Constraints

It is necessary to make a distinction between constraints used for applications and shape constraints.

For application constraints, as above, the basic method of handling this information is to copy information to the new part when dividing a model entity, and to check for the same information types and information equality when merging entities. However, different action should be taken when handling conflicts where entities being merged have information of the same type but different content. A simple strategy is to refuse to unite entities where this type of conflict occurs. Another strategy is to try and handle this information according to its content, identifying common examples as special cases. For surface finish, say, the tighter constraint should be taken as the constraint on the merged entity.

Handling shape constraints is a more complex task for modelling operations, and it has more far reaching effects. The simplest strategy is simply to refuse to operate on constrained entities. For local operations, the entities affected by the operation can be checked and the operation refused if any of these are constrained. For Boolean operations, checks can be performed while creating the intersection boundaries, and the operation can be halted if a constrained element is modified. Once the boundaries are complete, then it is also necessary to check if constraints ‘straddle’ the boundary; i.e., constrained elements lie inside and outside the parts of the object to be kept.

8.3.6 Category 6 – Geometric frameworks

These can be of two types. The geometric framework can either provide assistance for the definition of a part shape or other purposes or represent limitations on the shape of a part.

Handling these depends on the entities with which they are associated. If they are only associated with complete objects, then only operations that affect whole objects need to handle the geometric frameworks. If they are also associated with features, faces, or other parts, then operations that affect these, the local operations, must take account of them and handle them as well. Handling the extra geometry, though, is straightforward because the geometric entities are essentially passive elements, mainly under user control.

When splitting or copying model elements with attached subsidiary geometry, this geometry should be copied to the new element. When joining elements, attached geometry sets should be merged, removing common elements. The common elements can most easily be determined if supplementary geometry elements with the same geometry refer to the same underlying geometry, not to separate copies, unless of course the geometry is transformed. This means that comparing geometry for equality means comparing pointers, not performing geometric analyses.

If an object is to be transformed, scaled, translated, or rotated, then it is necessary to distiguish between the two types of supplementary geometry. Geometry representing limitations on the object should not be modified automatically, whereas the geometry used for defining shapes or for, say, hole centre-lines, should be modified.

8.3.7 Category 7 – Mechanisms and linkages

Because these are concerned with relating separate objects or instances, they affect how these can be modified. If they refer only to objects and supplementary geometric frameworks, then they can be handled at the object level, by operations that affect only complete objects, such as the Booleans, copying or modifying. However, if they contain references to parts of the instanced objects, they should be altered as the model is altered. Their use could be limited so that they are only allowed to refer to instances and supplementary geometry, not to individual parts. This might not be exactly what a user may want to do, but it avoids the problem.

There are two readily apparent problems with these: what to do about transformations and how to avoid circular linkage definitions.

The set of linked objects should, really, be treated as a unit, unless the object is free to move in the way specified by the transformation. If they are not treated as a unit, then there is no point in linking them in the first place. The problem is how to transform the linked objects to maintain consistency. For modelling, the simplest strategy is to refuse to modify objects that are linked together and provide special mechanism modification functions to

handle linked groups.

The requirement that circular links are not established is necessary to ensure that there is no feedback if one object in a linked group is modified.

Note that if a sub-assembly is linked together, then the links apply for every instance of that sub-assembly, even if the sub-assemblies should move independently.

Chapter 9

Feature modelling

There is a great deal of debate about features and solid modelling, and the area has become a verbal battleground with many views and many well-entrenched positions so it has become difficult to make sense of the subject. Even the nature of features, what makes a feature a feature, is disputed.

The purpose of this chapter is not to provide a survey of feature work but instead to outline some methods of feature modelling. It is impossible to provide an adequate description of all methods in the space available. Providing such a description is a major undertaking in itself so the interested reader would do well to refer to specific texts. A, somewhat, classic survey was done by Shah et al. [121]. There is a more up-to-date survey by Parry-Barwick and Bowyer [97]. Descriptions about particular methods can be found in Kyprianou [74], Choi et al. [21], Henderson [56], Ansaldi et al. [1], Wingård [145], Corney and Clark [23], Ranta et al. [104], and Laakko and Mäntylä [75] [76].

So, why features? What are they?

The definition followed here is that a feature is an identifiable collection of model elements composing a subset of the complete model. This means that, here, a feature is considered to be a set of faces, usually, although not necessarily, a connected set. Some people also allow features to be edges or vertices, which is certainly understandable if they have associated shape modifiers such as chamfers, but it is simpler, here to assume that they will be facesets.

There are three main methods of acquiring feature information:

1. Manual acquisition
2. Feature recognition
3. Design by features

It is important to note that these are not mutually exclusive and that it can be beneficial to have more than one method coexisting in a modeller.

Manual acquisition is tedious but can be used to correct mistakes or to define 'subjective' features that are difficult or impossible to define by rules. Design-by-features can provide a useful method by which a designer can introduce logical units into a model, either as basic shape elements in their own right or to match shape elements of a neighbouring part in an assembly. Feature recognition can be used to interpret the shape of a part logically, regardless of how it was built, based only on the current shape of the model. Feature recognition techniques can also be used to verify that the interpretation of a feature introduced explicitly is correct, because the shape of the feature might have been changed during modelling.

Obviously these methods have their disadvantages as well; the potential tedium of manual acquisition has already been mentioned. Design-by-features fixes the features as unchangeable, unless supplemented by some kind of modification or verification process. As pointed out in chapter 8, if a feature is added, then the simple shape-changing modelling algorithms described in chapter 6 will not take account of this and can be used to change parts of the feature in isolation, invalidating the information stored with the feature and removing the value of introducing the feature explicitly in the first place. For feature recognition, the main disadvantage is the amount of work that has to be done. The shape of the part is treated as unstructured, in feature terms, and the whole part is examined anew each time feature information is required.

Another important aspect to note about features is that there are likely to be multiple feature interpretations associated with a part. Different application areas are likely to need their own interpretations of parts of a model. A simple illustration is that of an object with a boss (figure 9.1).

For a designer and someone looking at automatic assembly, the boss is interesting as an extrusive element (figure 9.1 bottom left). For a process planner, however, the material to be removed is interesting, and the boss is (almost) incidental; it is the material that is left after removing the surrounding slot(s) (figure 9.1 bottom right). For the 'design-by-features' purists, this problem is overcome by the use of so-called 'feature transformations', whereby existing feature structures are transformed into new feature sets using some rules. From a personal point-of-view, it seems pedantic to refuse to accept feature recognition in any form, hence, the suggestion above that the most sensible course seems to be to allow both methods to coexist and to use the more appropriate one for any task. Kyprianou also pointed out this duality of protrusive/intrusive features in his dissertation [74] indicating that interpretations can coexist and be used for different purposes. To summarise, design-by-features introduces the appropriate feature for the designer, whereas feature recognition finds the appropriate features for an application.

Historically, feature recognition was developed by Lyc Kyprianou at Cambridge University in the BUILD system at the end of the 1970s. He was principally concerned with shape classification for large part-database handling, although he pointed out other uses for features. Other methods also

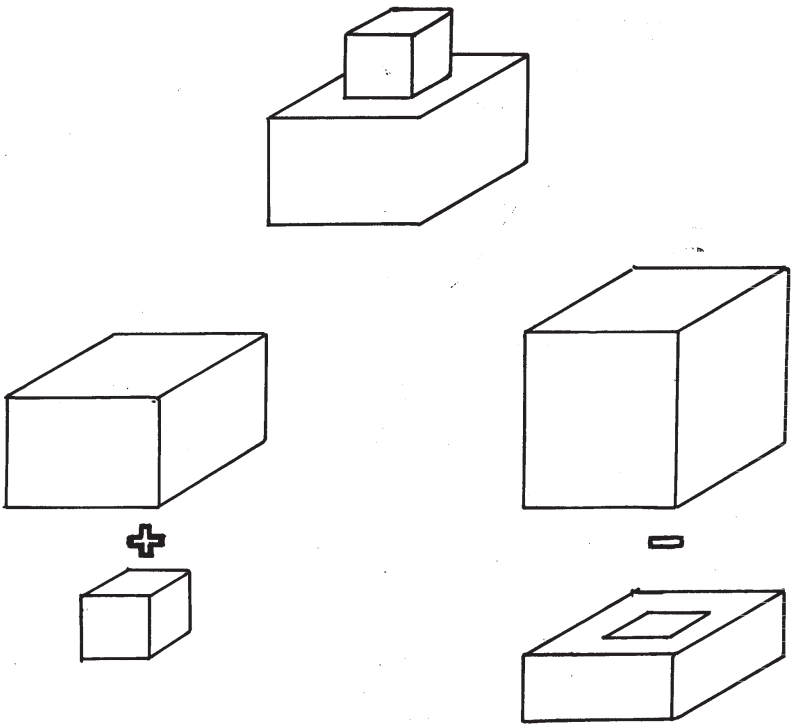


Figure 9.1: Object with boss

appeared about this time [21], [147]; see Shah et al.[121]. Design-by-features may have been developed to introduce explicit feature interpretations for application areas. It is certainly useful to be able to explicitly define feature structures to avoid working on the two difficult problem areas of feature recognition/recovery and feature-based reasoning at the same time. Both of these techniques, however, soon grew in importance and have become widely used in other areas than originally intended.

'Design by features' is a term used to describe the explicit inclusion of features into a design. Features can be included directly as modifications to a base part or existing design. They can also be introduced directly as isolated shape elements. These two methods are described in sections 9.3.1 and 9.3.2, respectively. Introducing features explicitly means that information about the associativity of shape elements in the model can be recorded directly instead of having to rediscover the information. However, the features that are logically introduced for design purposes may not be logical for manufacturing or assembly or some other purpose. To get around this problem, the notion of a 'feature transformation' has been introduced. Feature transformations take one feature and regroup the feature elements and adjacent elements into a new feature or features.

Another related geometric reasoning problem is described by Tate et al. [134] and concerns symmetry analysis. This is more difficult than the traditional feature detection methods because it needs global rather than local methods.

9.1 Feature datastructures

First, although a digression, it may be useful to define what a feature is, in modelling terms. From the above definition, a feature datastructure consists, basically, of a set of topological elements, possibly with relationships between them. Also as stated features will be assumed here to be collections of FACEs, rather than any topological elements. There is no absolute definition of a feature; it could consist of one or more edges or a vertices, or a mixture of elements, depending on the application definition. Here two types of feature datastructure are considered: 1) simple face sets and 2) feature frames. Another method that has been proposed for representing features is to use compound volumes, with features represented as volumes loosely attached to the main volume. The idea was put forward by Pratt [101], but it will not be described further. The method presupposes that features are introduced as volumetric elements, or at least are simple enough to make into volumes automatically. Luo [79] describes methods for creating such compound volumetric structures.

As mentioned, a basic requirement is that it should be possible to include faces in several different features so that multiple feature decompositions can be represented. It is also necessary to be able to find feature structures from

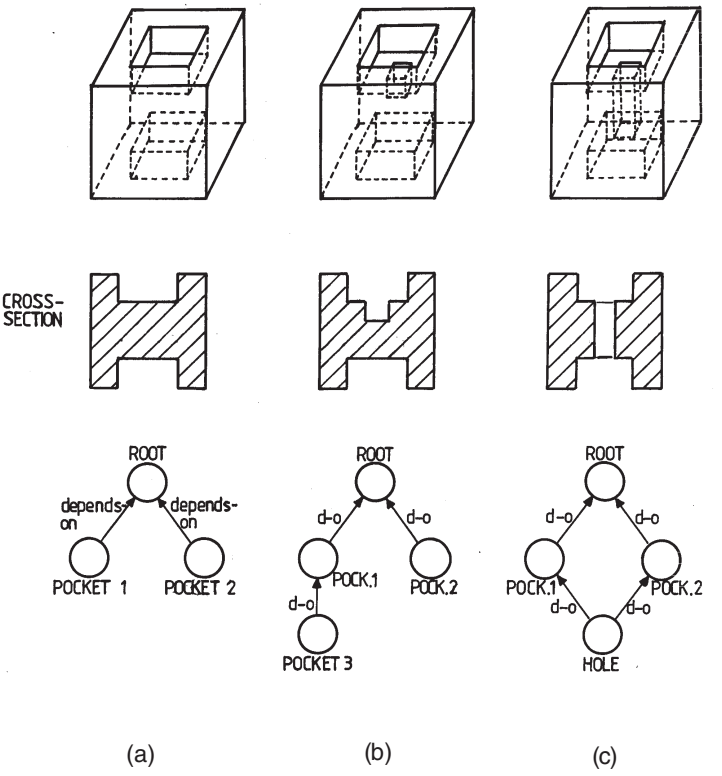


Figure 9.2: Features and feature dependencies

the body 'root node', possibly to be able to identify features and to attach attributes to them. It is also desirable to be able to represent dependencies between features.

9.1.1 Feature facesets

Feature facesets provide a way of grouping faces together independently of the face-grouping mechanism described in chapter 3, which is really for structuring objects in a different way. The feature faceset is really a simple face list, but with the added requirement, as mentioned, that faces can belong to several features. The simplest way to do this is with a 'feature-face link'. A face points to a list of links to features in which it participates and a feature points to a list of links to faces that it contains.

The feature link definition is given in Appendix A, section A.1.21. Features need to be linked in a tree structure from the body. There is also a requirement to be able to identify dependencies between features. For example, consider the object shown in figure 9.2a. The object is basically a block with two

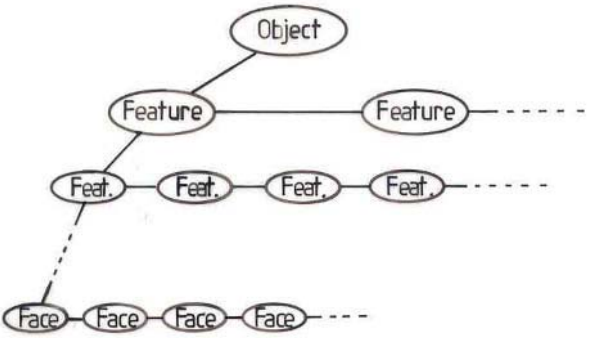


Figure 9.3: Feature partitions

pockets. The block is the root node and the two pockets are dependent features. In figure 9.2b, one pocket has another pocket in its base face, which is thus dependent on the first pocket. Contrast this with the object shown in figure 9.2c, which has a hole between the two pockets. The hole can thus be regarded as dependent on both pockets because its boundaries lie within both pockets. The dependency graphs of the feature structures in figures 9.2a and 9.2b are tree structures, whereas in figure 9.2c, it becomes a graph with a loop.

The general requirements are outlined in figure 9.3 where it is desirable to be able to represent multiple partitions of an object into feature sets. The need for multiple sets is either where there are alternatives for an application or where several applications share the same model.

The implementation of this can be done using link entities to establish the link between features and faces, as shown in figure 9.3. Figure 9.4 shows how an object with a boss can be decomposed in two different ways.

9.1.2 Feature frames

Another possibility is to represent features using 'frames' (Minsky [86]). A frame can be thought of as a structure with several slots to be filled. Each feature type needs to have a separate frame with slots corresponding to the various parts of the feature. An example of feature frames is given in Laakko and Mäntylä [75], [76]. The frames described there also contain rules that the feature sub-parts obey. Such a construction is important for feature verification to check whether the feature has not been invalidated during some modelling operation (see chapter 8).

Representing features with frames also allows feature elements to be identified specifically so that, for example, the floor of a slot may be identified as a separate face set from the sides. With the simple face-list feature representation described in section 9.1.1, the information about the interpretation of

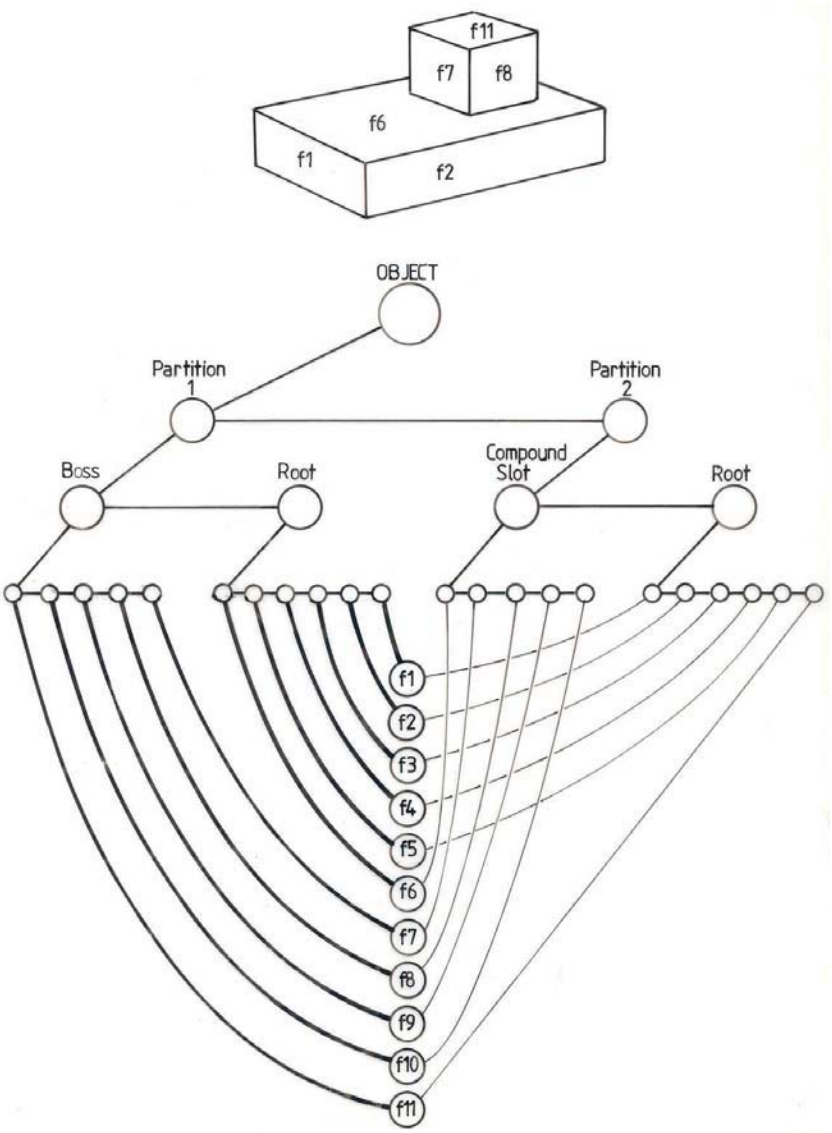


Figure 9.4: Feature datastructures for a cube with a boss

the various parts is lost. The disadvantage of feature frames is that they have to be defined separately for each feature type. Feature facesets can be set up dynamically as and when required, even under user control. On the whole, though, it seems preferable to use frames rather than simple facesets.

9.2 Manual feature acquisition

Manual feature acquisition is the simplest feature method to implement, relying on user judgement to sort out what constitutes the feature.

The sorts of operations needed for this are:

- DEFINE FEATURE
- DELETE FEATURE
- ADD FACE TO FEATURE
- REMOVE FACE FROM FEATURE
- DEFINE DEPENDENCY
- DELETE DEPENDENCY

with the commands being made explicit or hidden under some interface.

If the feature is being defined as an unstructured faceset, then establishing the feature involves giving the feature some sort of identifier and then picking a series of faces to be included in it. As each face is picked, it is added to the face list referred to from the feature. Face picking is, perhaps, most conveniently done using a graphics picking method, allowing a user to pick faces with a mouse or pen. However, this is not really necessary; any method of selecting faces can be used. Extra faces can be added afterwards, or faces can be removed if erroneously added.

If the feature is defined as a frame, or some other structured entity, then it is possible to set up the sub-structures separately in the same way. A slot feature, for example, may have the sub-structures: Base, Side-1, Side-2, and End. When the feature is defined, it is possible to define each of these in turn.

The advantages and disadvantages of this method of feature definition are fairly obvious. Implementation simplicity makes it an obvious basic tool for feature definition, and it can be used to supplement other methods for correcting mistakes and defining complex features. However, with complex objects, the method can be tedious and it can possibly be difficult to select the correct faces. It can be seen as (hopefully) a temporary stop–gap method while other methods become more stable.

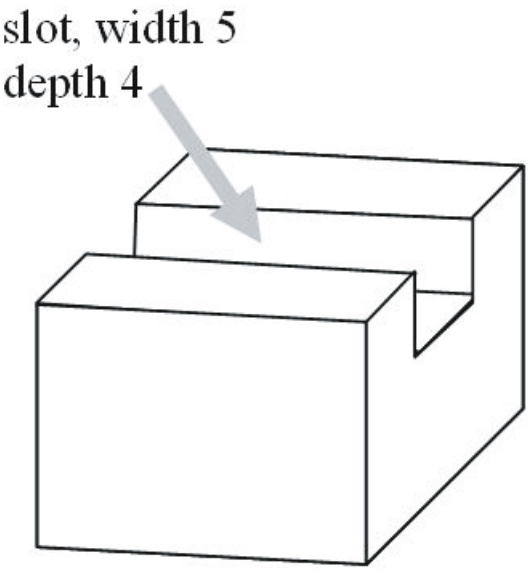


Figure 9.5: Slot with operation parameters

9.3 Design by features

The important point to note about design by features is that features are explicitly added to a model, either directly as shape sub-structures or by using feature creating operations. Note that it is useful to supplement design-by-features methods with feature verification, as outlined in section 9.4.2.

9.3.1 Adding features to a shape

Two possible methods for introducing features into a shape are feature-producing local operations and introducing features from a feature library or some other source.

Feature-producing local operations, such as the slot-producing operation or the rebate operation described in chapter 7, can produce feature datastructures as a by-process, recording their parameters as part of the datastructure. Figure 9.5 shows a slot labelled with parameters that might have been used to create it.

Features in a feature library, or from another source, for example, from another object, can be represented as 'partial objects' (see chapter 5). The interior shape of the feature is completely defined; it is necessary to define the boundary of the feature with respect to the object in which it is to be

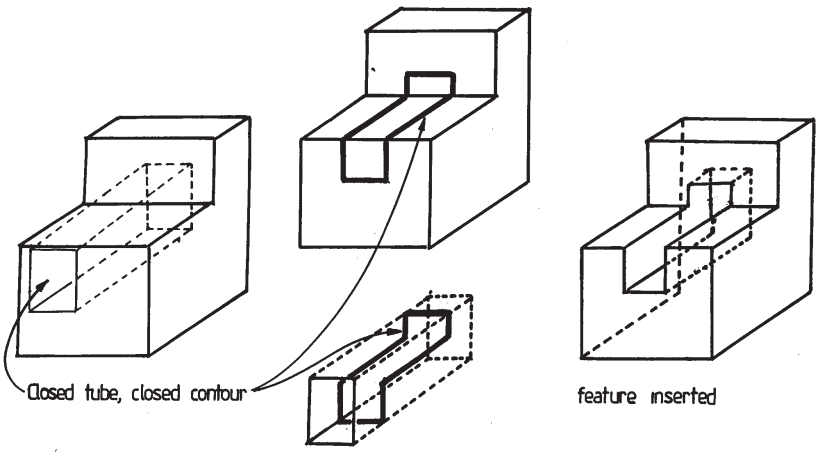


Figure 9.6: Inserting a feature into an object

introduced. To do this it is possible to use the same kind of technique as that for Boolean operations described in section 6.1. The faces of the partially defined feature are intersected with one or more faces of the destination object to produce a closed intersection ring. The feature is then trimmed back to the intersection ring and joined into the object, as illustrated in figure 9.6.

Obviously there are problems if the intersection process does not produce a closed intersection ring, as illustrated in figure 9.7. This condition has to be recognised and an error signalled if it occurs because the whole object would be left in a strange invalid state if the feature and object are joined along the partial intersection ring. If the feature is closed by the addition of an extra face, changing the feature from a slot to a tube, then a complete intersection ring is formed and the tube can be added (figure 9.6). Note that failure to add a feature as specified can be useful information, indicating that the model is not as the designer envisaged. As an aside, note, however, a related example, shown in figure 9.8. Here a tube is added through the object, but the result can be interpreted as a coincident hole and slot instead of merely the hole feature added. The hole/slot combination can be useful in manufacturing if the slot is produced, say, in one setup (from direction 2) and the hole in another setup. Although this may seem like extra work, it can be realistic if, for example, the hole is too long to be produced in one step. This kind of problem occurs again and again in feature research. Different people may see features in different ways: merging, splitting or reinterpreting feature elements.

Returning to the main problem, inserting features, the standard Booleans (section 6.1) can be used for creating the interaction rings in the feature and

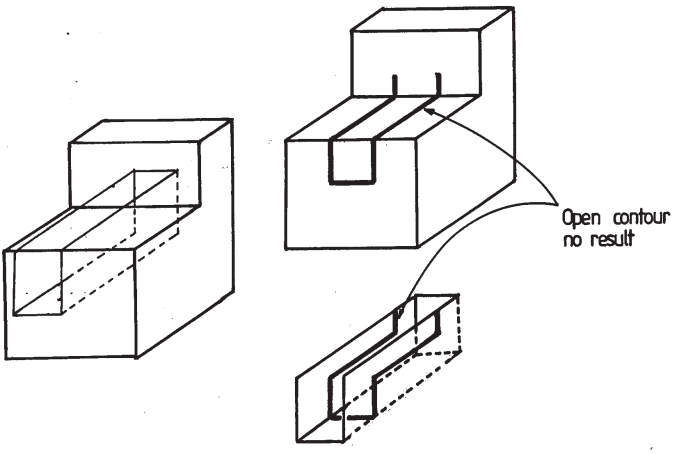


Figure 9.7: Invalid feature insertion

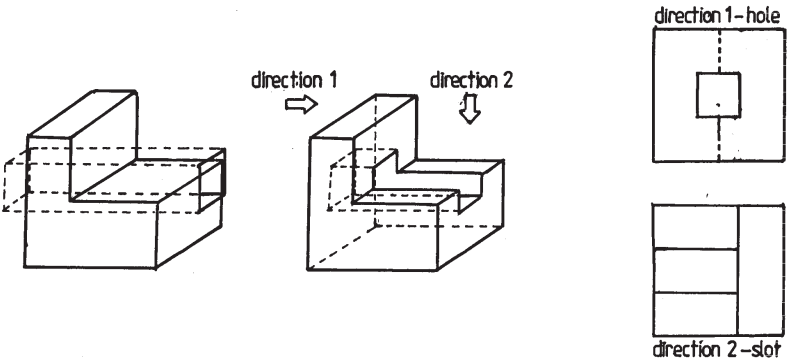


Figure 9.8: Hole, hole/slot insertion

target object, but these then should be preprocessed to check that they are closed.

Ranta et al. [104] describe a method of adding features that allows the borders of a feature to be dynamic and extensible until they intersect the object in which they are being inserted. This is an attractive idea and can produce some desirable results. With this method the boundary faces of the feature are defined to be dynamic, which means that they are infinite in extent. The surfaces of the dynamic side faces (the faces containing the partial boundary edges) are intersected with the target object, and the resulting curves are assigned to the partial edges. The start and end vertex positions of the edge must also be recalculated appropriately using the curves of the edges adjacent to the partially defined edge. As with SETSURF, care must be taken that the new vertex positions do not make these adjacent edges of zero or negative length. There is also a possible danger that the extended faces of the feature can intersect each other before they intersect the object in which the feature is being inserted. This condition should be checked for automatically or visually after feature insertion because it is relatively easy to repair the object by reinserting the portion cut out to make room for the feature. Trying to rebuild an object later after this object portion has been deleted is much more difficult, in some cases, impossible.

Feature libraries provide a set of predefined features. It can be useful to be able to define features dynamically to match portions of other objects. This can be done simply by allowing the user to pick a faceset to define the feature and then copying the faces in this faceset as the feature. It is more problematical for the user to set up rules to define this feature, though, if such rules are included, for example, if the feature is defined as a frame. It is also difficult to verify such user-defined features; hence, such a facility needs to be used with care. It can be useful to use the same kind of techniques to introduce general shape elements into a design, but if these shape elements are large, then it is better not to preserve them as 'features'.

9.3.2 Building objects from isolated features

Another technique for introducing features into an object is to pick off shape substructures from objects and fill in the gaps between these, as described by Wingård [145]. Such a design method is particularly useful for designing parts of an assembly. When some parts of the assembly have been designed, these will constrain connecting parts. For example, consider a shaft that is to run between two previously defined blocks. If the holes through which the shaft is to pass have been designed, then there is no need to redesign the parts of the shaft that are to pass through the holes; the shape is defined by the holes. It is relatively simple to pick the hole sub-shapes from the two blocks, negate them, and use them as parts of the new design.

This technique is related to the idea of design sketches, described by Kjellberg [69] in his dissertation. There is obviously a danger with allowing partial

designs or idealisations with modelling, but as described in chapter 5, standard modelling operations can be extended to cope with these. It is also desirable to have an automatic 'body checker' (see chapter 14) to provide a way of finding potential deficiencies in a design so that these can be corrected before the design is finalised.

Note that this is a possible use for the partial models described in chapter 5. If the known model parts defined as features are modelled using partial models, then the boundary edges can be joined using a bridging technique similar to that used for lofting, as described in chapter 13.

9.4 Feature recognition

With feature recognition, the shape of a model is analysed to recover feature information. The kind of information recovered depends on the way the model is processed instead of being specified by the user. In simple terms, the model can be thought of as a kind of convex envelope or convex container (n.b. not strictly a minimal convex hull) with a convex kernel inside. The features lie between these two, the convex container and convex kernel. According to which of the two is regarded as the base object, the features are protrusive or depressive.

9.4.1 Classic feature recognition

Classic feature recognition techniques use edge concavity as a shorthand way of testing whether there is an interesting region. Kyprianou performed feature recognition by classifying edges and vertices as convex or concave and building facesets bounded by these. Classifying edges as convex or concave is done using the utility described in Appendix E. With Kyprianou's method, faces were first classified according to the number of hole loops and concave edges they contained. Faces with hole-loops or with a mixture of convex and concave edges were classified as 'primary' faces; all others were classified as 'secondary'. Primary faces were first sorted, with the faces with the largest number of inner loops considered first and used as 'seeds' for growing facesets. Facesets are grown by adding faces adjacent to the exterior loop of the primary faces to the faceset. Then faces of these are added, and so on. Once the facesets were built, they were classified to determine the nature of the feature.

This sort of classification is perhaps best explained with an example. The algorithm here differs a little from Kyprianou's original algorithm, but it performs basically the same task.

First, all edges in the body are classified as convex, concave or smooth. If no concave edges exist, then the body is a convex solid. Cubes and cylinders are examples of such convex solids, but most engineering objects contain at least one concave edge, usually several.

Subsequently, faces are classified according to their complexity. The effect

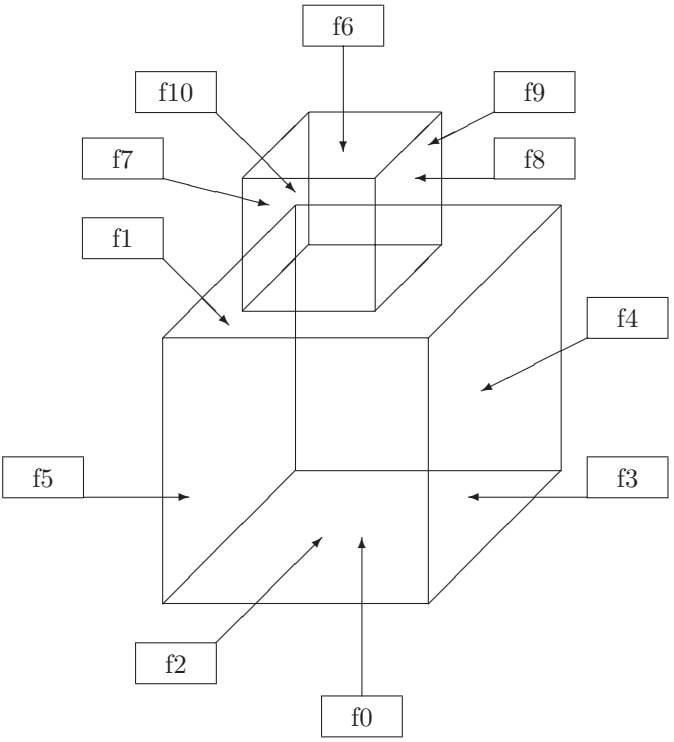


Figure 9.9: Object with boss

is to partition the boundary graph into subsets representing features and the basic object. The features can then be recognised by interpreting the subset graphs. Consider the cube with a boss, as shown in figure 9.9.

There is one face with a hole-loop (f1) and five faces bounded by a mixture of convex and concave edges, including the face with the hole-loop. These are sorted by complexity. The most complex face is the face with the hole-loop. After this come the other four faces, each with one concave edge (f7, f8, f9, f10). A boss could be then described as a face with convex edges surrounded by primary faces, each with one concave edge. The hole-loop is taken as a separation boundary, grouped together with the faces belonging to the boss (faces f7, f8, f9, and f10) and ultimately face f6. The external loop of the face is grouped with the other secondary faces adjacent to it (faces f2, f3, f4, and f5) and eventually with face f0, to form what Kyprianou termed the 'root'. Note, in passing, the block with the pocket, shown in figure 9.10. Here, the definition of the pocket might be a face with concave edges surrounded by

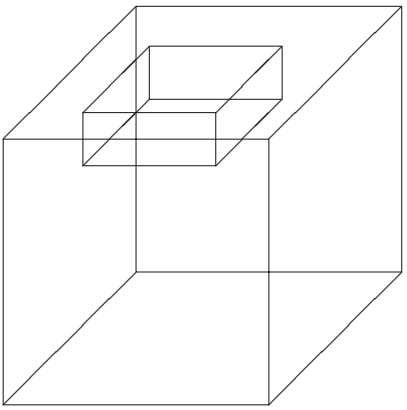


Figure 9.10: Object with pocket

primary faces, each with one convex edge. In fact, there are several of this sort of feature pair, where the role of convex and concave edges around the feature is swapped.

Some more features are shown in figure 9.11, and an object with several of these features is shown in figure 9.12.

A simple set of rules concerning these features is given below:

- **pocket:** base face with concave edges surrounded by primary faces each with one convex edge.
- **boss:** base face with convex edges surrounded by primary faces each with one concave edge.
- **slot:** base face with two sequences of concave edges separated by one or more convex edges.
- **partial slot:** base face with one connected set of concave edges separated by one or more convex edges.
- **rail:** base face with two sets of neighbouring primary faces.
- **through hole:** Two rings of convex edges bounding a connected set of through-faces. A through-face is either a closed concave curved face or a face with at least two neighbours in the set separated by convex edges.
- **bridge:** Two rings of concave edges bounding a connected set of bridge-faces. A bridge-face is either a closed convex curved face or a face with at least two neighbours in the set separated by concave edges.

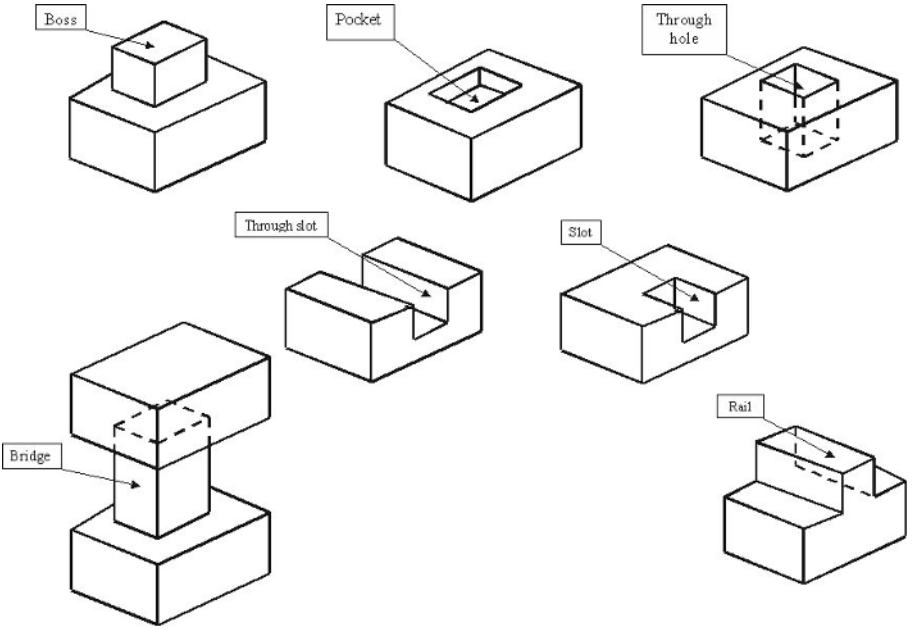


Figure 9.11: Possible feature set

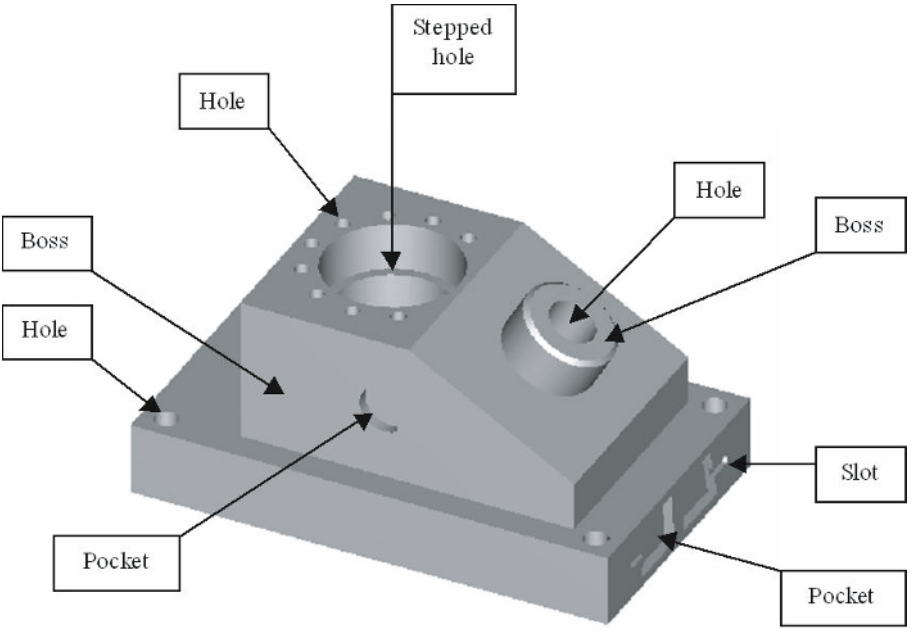


Figure 9.12: ANC101 object with some features

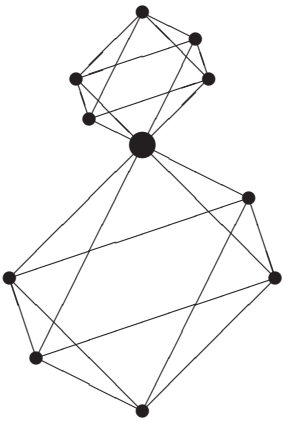


Figure 9.13: Dual graph of object with boss

(There is a formal definition of some manufacturing-related features in STEP in ISO 10303 AP214 and AP224. However, this is related to the manufacturing application area so the simpler terms are used here.)

Ansaldi et al. [1] performed feature recognition based on a sort of dual of the model, the Face-Adjacency Hypergraph or FAH. Features were determined as places where the dual graph of the model could be cut into separate pieces.

Looking at the dual graph of the object with a boss, as shown stylised in figure 9.13, the appeal of this idea is clear. The large node or vertex at the centre of the graph corresponds to the face with the hole-loop, and it is clear that it is a non-manifold vertex. Splitting the dual graph at this node breaks the graph into two basic elements, one corresponding to the base and one to the boss. The same basic arrangement is also evident for the block with pocket.

Regrettably, though, feature recognition for real objects is a problem rarely tractable with simple methods. Early on in the history of feature recognition Jared and his team compiled several examples of simple problem objects [96]. One example is shown in figure 9.14. In this object there is no convenient hole-loop to mark the boundary between object base and feature, and one of the faces surrounding the top of the boss is shared with the object base. Kyprianou's original algorithms were extended by Anderson to cope with several of these problem cases. Circuits of concave edges were identified, even though these might not form an explicit hole-loop. Other special cases were identified to handle shared elements. For the object in figure 9.14, for example, a notional missing edge was identified as connecting the extreme vertices of the open chain of concave edges surrounding the boss.

As described, explicit hole-loops in the model can be identified easily. However, there are some special cases, such as cylinders, that need to be handled, because some cylinder representations have no side edge, and hence,

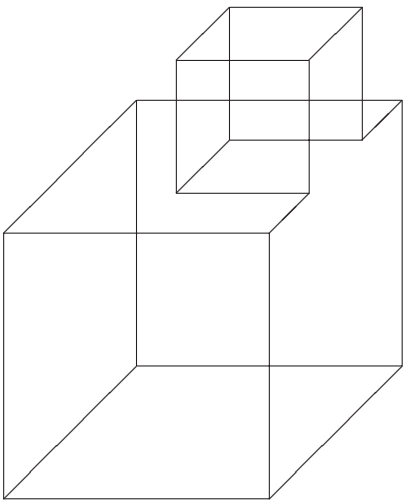


Figure 9.14: Object with boss with shared face

the cylindrical face has two boundaries.

Some feature recognition methods start by eliminating all hole-loops and classifying the sub-volumes that result. For some classes of objects, this works well enough, but feature interaction creates difficulties that are hard to handle with simplified schemes such as this. Consider the object in figure 9.15, which is loosely based on one of the STEP-NC test objects.

Removing the obvious holes gives the object in figure 9.16, but the large hole has a more complex boundary.

The boundary itself is not so much of a problem. The algorithm, roughly, involves looking for faces lying in a negative cylindrical surface. These faces should have two boundaries that surround the cylindrical axis, at least one of which should be composed of convex edges. The large hole fits into this category. There are two obvious results from removing the hole. Unfortunately they are both wrong, possibly. These are shown in figures 9.17 and 9.18.

Figure 9.19 shows what is probably required for manufacturing, with the large hole removed and the slots extended.

This very simple example is intended to show the problems involved in having too simple a strategy for hole removal. Using the maximum extent of the boundaries as the limit of the cylinder to be added back gives the object in figure 9.17. Using the minimum extent gives the object in figure 9.18. Figure 9.19 involves extending some boundary faces until they intersect.

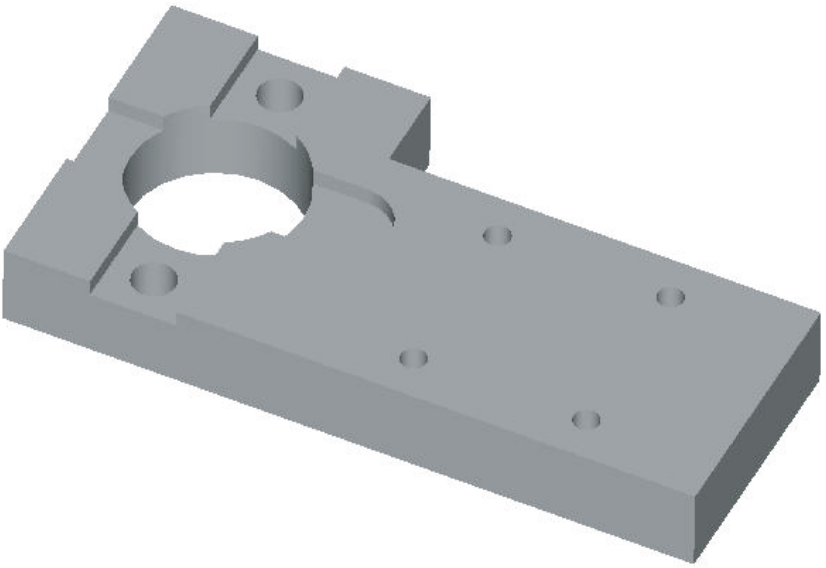


Figure 9.15: Simple object with holes

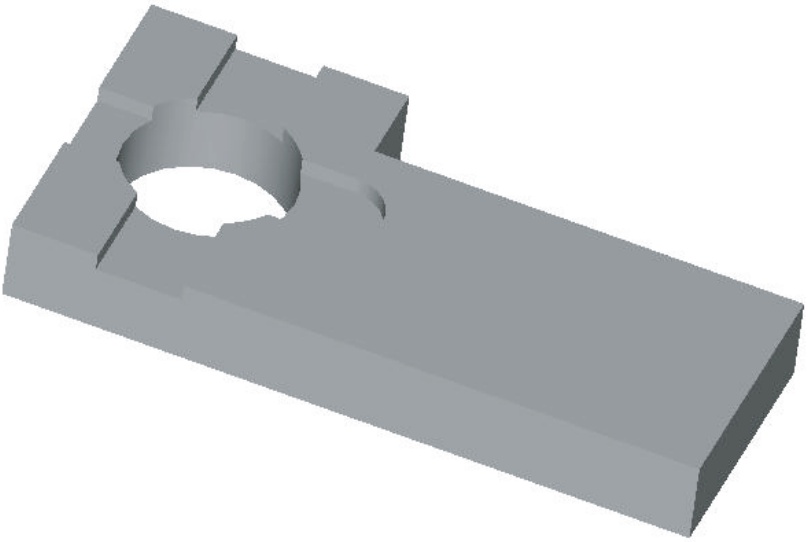


Figure 9.16: Simple object without the obvious holes

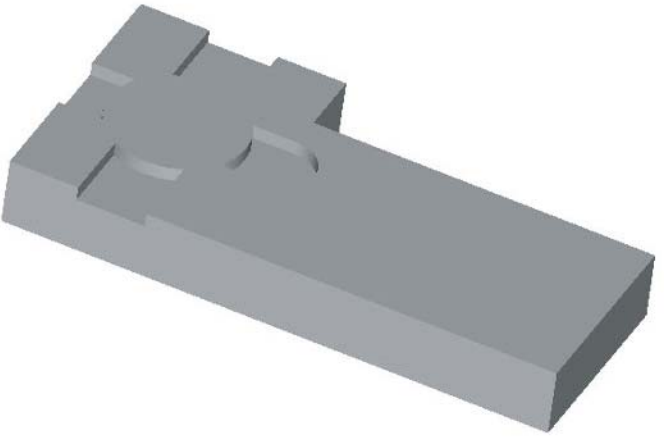


Figure 9.17: Simple object with large hole removed (version 1)

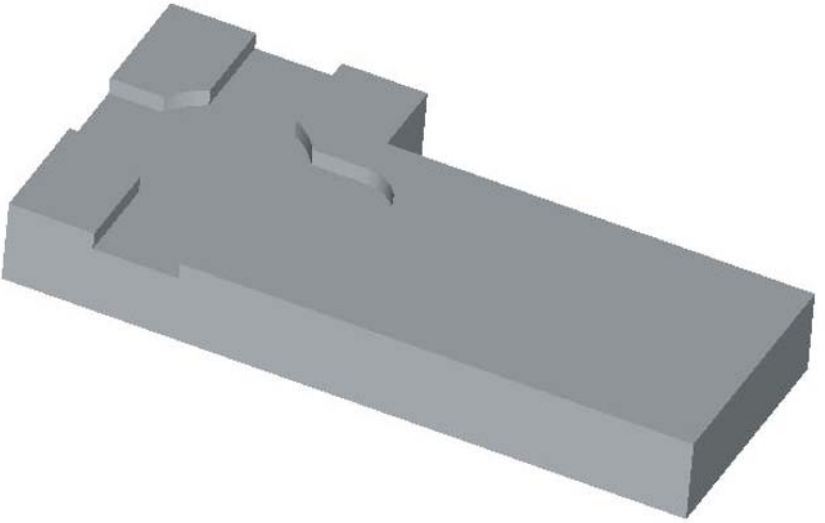


Figure 9.18: Simple object with large hole removed (version 2)

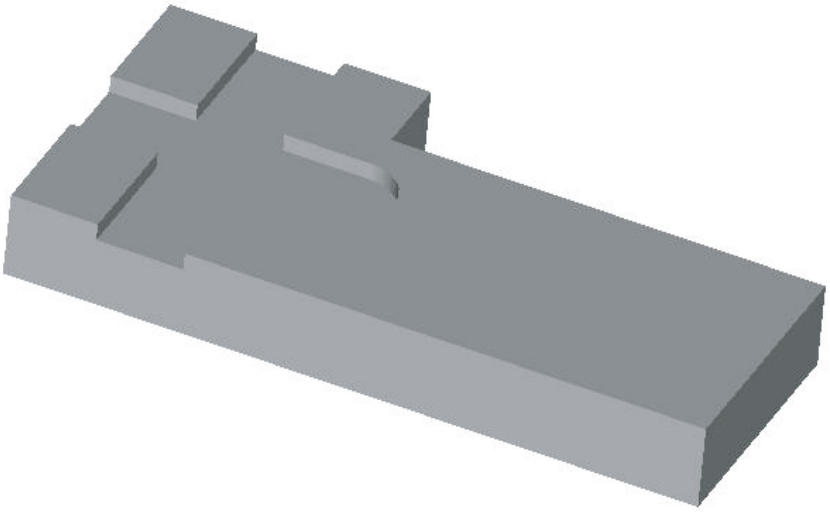


Figure 9.19: Simple object with large hole removed and extended slots

9.4.2 Features and applications

For some applications, such as machining, decomposition of a model into a base and a set of machinable elements is an interesting way of handling a model. Should you feel like shouting 'whoopee!' and think that this is all that needs to be done, then restrain yourself. Unfortunately this seems to have been the attitude which has appeared with some system developers. Merely identifying this kind of element is not the same as process planning, which is a complex task outside the scope of this book. If an object is to be made from a simple rectangular block of material, then this simple decomposition may be enough, but this is not the only way of machining a part. Another method, which immediately adds complications to the feature recognition process, is to machine the part from a moulded stock. It is then not clear, from the final object, which are the real machining features.

Figure 9.15 and figure 9.19 can also serve to illustrate some of the complexity. Taking the object in figure 9.15 as the final object, then the obvious holes removed in figure 9.16 may be made by drilling. However, the larger hole is more of a problem because it can be made in several ways and so one feature can be interpreted in many ways.

The notion of one feature having more than one corresponding machining operation can be illustrated using the really complex object shown in figure 9.20. There are several alternatives when considering machining operations for such a hole, as follows:

1. Drilling. This might be considered the 'obvious' method.

2. Milling. This is an option if the hole is larger than the size of the drills available or, alternatively, to avoid a tool change.

3. Pre-drilling and drilling. In this case, two machining operations correspond to one feature.

4. Pre-drilling and milling. Again an option if the hole is larger than a certain size.

5. Nothing. The hole is made by casting and is just to reduce weight; hence, there is nothing more that needs to be done. The hole may correspond to a manufacturing operation, but not necessarily to a subsequent machining operation.

6. And so on. The idea of this list is just to show that the simplistic idea that one feature corresponds to one machining operation is not justified. Also, the idea of a 'feature' does not always take into account the geometry, such as the size of the hole, which is an important parameter for manufacturing. Please also note that I am not a manufacturing expert; a real expert can probably think of several more alternatives.

For the hole in the object in figure 9.15, it might be manufactured by pre-drilling and then drilling for the sake of accuracy. If, to avoid a tool change, the hole is milled using the same tool as used to cut the slots, the hole is not really a hole but more a sort of through pocket; in which case, it changes identity. Also, in that case, the simple object with large hole removed in figure 9.18 makes some sort of sense as an intermediate shape. In any case, every through hole that is drilled can be considered as two manufacturing features, according to which side the tool enters. This is perhaps a fine point, but it can be important to make the distinction because of tolerance considerations.

In general, when it comes to any application, if you start with the question "What is a feature?" then the answer comes in interrogative form: "Why are you looking at this object?" You have to start from the point of view of the application in which you are interested before you can decide what is a feature in the context of that application. A very simple example of this is the much-used example of the block with a boss. It may well be natural for a designer to create this by extruding a square shape from the top face or by adding a small block, but it is not the boss that is interesting for a manufacturer, it is the material to be removed surrounding that block.

Jared and his team did much basic research into features and machining, as reported by Parkinson [96] for example. The lesson to be learned from their work is that, as they found, manufacturing is more than just feature recognition followed by simple machining of those features. They found that many decisions have to be made around the part, and correct, efficient manufacturing depends on these decisions; manufacturing is more than simply the part geometry. However, the over-simplified idea keeps reappearing that the

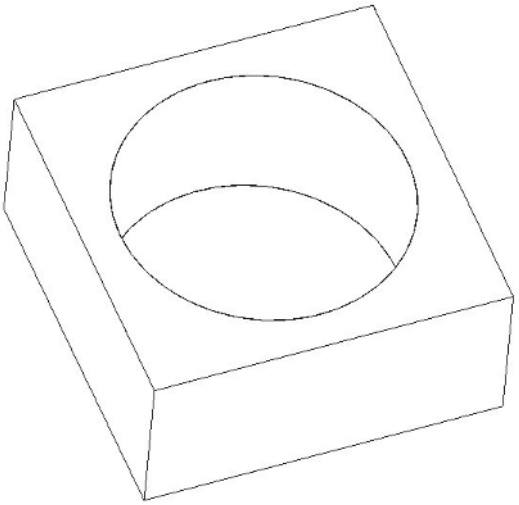


Figure 9.20: Really complex object with multiple machining possibilities

designer designs using manufacturing feature operations. The process planner is then supposed to simply pick these features up and define machining instructions for these.

The problem is that although each idea for automated manufacturing has some benefits, no idea is so superior that it excludes all others. For process planning, I have a vague assumption that the process planner would work as follows:

1. Removal of obvious features, either from design features or from simple recognition.

2. Identification and removal of other finishing features.

3. Manufacturing process decision.

4. Feature recognition adapted to manufacturing process, and removal of these features.

5. Shape adaptation for manufacturing (for example, adding draft angles and blends for moulding or casting)

6. Raw piece treatment (for casting, say, this would involve a separate mould-making step with mould-making feature recognition).

which is based on comments made by Sabin as a summary to a CAM-I conference at Robinson College, Cambridge, in 1983.

As can be seen from this cursory scenario, there is a need for several methods; hence, it is necessary to treat features in an intelligent way and not just to come up with over-simplified partial solutions that are not useful in practice because they do not cover enough cases.

9.4.3 Volumetric decomposition

The problem is knowing what a feature is. In psychology, features seem to be points where change occurs ([92]). Following this definition means that a totally convex object can have features, although these would not be identified using Kyprianou's methods. In fact, with this definition, a sphere can be defined as 'featureless', whereas any other object will have some features. The edges and vertices of a cube, for example, are places where a surface changes or where three surfaces meet, and are thus features. This is too loose a definition to be convenient, because visual recognition is not needed, but the definition of irregularity based on a global analysis of the shape to determine the features is a possible way to determine convex as well as concave features. The problem is that a global analysis, such as a three-dimensional Voronoi (or Medial Axis) analysis (see e.g. Hoffman [58] or Held et al. [54]) is expensive, if possible in general. Two methods for medial axis calculation are described in chapter 15.

The notion of a full-scale volumetric analysis such as that performed for Voronoi or Medial Axis is, however, appealing. This is implicit in Kyprianou's method, the concave edges being indicative of a change in volume. Corney and Clark [23], also, have this notion in their method. Another method was described by Renner and Stroud [107].

Yet another possibility for a subdivision was described by Renner and Stroud [108] based on the divide-and-conquer method described in chapter 15.

The whole topic is still a subject for research with no definite answer about how to recognise features.

9.5 Feature verification

A final short note concerns feature verification. It illustrates the need for a multitude of techniques. Feature verification is an application of feature recognition techniques for feature structures introduced explicitly into the model using, for example, design-by-features techniques. Where feature verification differs from feature recognition is that the elements of the model composing the feature are already known; it is simply their topological and geometrical relationships that are being examined.

With a frame-style representation for the features, the roles of different elements are assigned and it is 'simply' necessary to apply the feature recognition rules to these elements.

9.6 Structuring features

A general conclusion is that work on features is continuing and that no definite methods seem so superior yet to warrant exclusive treatment. As a parting shot in this chapter I would like to note another idea that needs to be followed up, that is, the structuring of feature information. In the whole of this chapter, I have ignored the fact that there are different reasons for features to exist in a model, and therefore that they arrive at different stages in the design process. For example, there are connection features ientified by Csabai [25] that need a higher tolerance than, say, pockets introduced to reduce weight. Structuring the shape and feature information so that other users can trace the reasoning of the designer is important. Also, passing isolated parts across to a process planner is not a technological necessity; it seems to be anchored more in the old habit of transferring shape information using drawings. There are a lot more possibilities than are currently used. It would be possible to create and exchange functional models to let the manufacturer know how the part fits with other elements in the product. A client, too, might benefit from such a functional model.

It is too early to describe these topics here. Suffice it to say that this area needs more work before it settles down into a subject sufficiently stable to be part of a book.

Chapter 10

Graphics

Computer graphics is a complicated subject, much too complicated to go into detail in a single chapter. Computer graphics is concerned with the production of images and, as such, is more concerned with the physics of light than with modelling directly although models may be used. For details, the reader should refer to specific textbooks, such as those by Newman and Sproull [91], Foley et al. [40], or Szirmay-Kalos et al. [133]. From the solid modelling point of view, it is necessary to determine the interfaces to the graphics system. Advanced graphics is not always necessary for visualisation of models; for example, it may be desirable to create a realistic image of the outside of a car, but line drawings of internal components are probably adequate. Another problem is that the interface between modelling and graphics is not altogether stable. It used to be that graphics devices were of the "Move to..." "Draw to ..." type. Sophisticated graphics, such as hidden line or surface elimination, had to be done in the modelling system and translated into simple graphics commands. Modern graphics devices can produce sophisticated images from simplified model representations with the help of hardware but the problem remains that modelling demands change as the model changes. This can be illustrated by the example of information attributes in the model. Some of these the modeller may know about and be able to translate, for example, shape modifiers or colour information, and others may need to be passed over as tags for display as labels. It is unlikely that a graphics device will be able to cope with this information directly, because this problem is more appropriate to the modelling domain.

From another point of view, the graphics methods described in this chapter are part of the history of solid modelling. Hidden line elimination methods were needed for producing 'realistic' solid images, and they are still needed for producing engineering drawings. Although high-level graphics has become cheap enough to be commonplace so that the contents of this chapter may be superfluous, I feel that it is worth recording anyway because trying to describe graphics output from a modeller when computer graphics is developing rapidly

is like trying to hit a moving target. Some of the contents of this chapter may be relevant anyway.

In general, the graphics output can be partitioned into two rough areas: 1) internal and 2) external graphics. With both of these types, information is output but the level differs. With internal graphics, the model is interpreted within the modelling system and information is output in the form of low-level graphics primitives. With external graphics, the model is output in a neutral form (e.g., triangles or convex planar polygons) that can be interpreted by a separate graphics processor. Along with the variation in the level of output so, too, varies the amount of control that the modeller exerts. Although it might seem beneficial to use high-level graphics devices and concentrate on modelling functions, it may be difficult to produce exactly the image required. A selection of techniques is presented here, which ones are needed, or whether more than one is relevant, depends on the environment in which the modelling system is used.

Splitting the techniques into these categories gives:

Model-space graphics

1. Line drawing
2. Hidden line removal
3. Special rendering techniques, (scan-line and ray-tracing)
4. Drawing free-standing geometry

Device-space graphics

1. "Communications models"
2. Hidden line removal

Special rendering techniques, e.g., scan-line or "bread-slicing" and ray tracing, are awkward to fit into the model-space graphics section because they can also be performed by dedicated graphics devices using a neutral model format. They are included in the section on model-space graphics to show how they use modelling tools, but the reader may also refer to the book by Szirmay-Kalos et al. [133] for details of graphics space-rendering techniques. In addition to drawing solid models, it may also be necessary to draw free-standing geometry, which is useful for showing free-standing or supplementary geometry that can be provide a designer, say, with a geometric framework within which to work, or for relating mechanisms. This chapter assumes that graphics output will be in general in three-dimensional form; it seems unnecessary, here, to devote space to mapping three-dimensions (3D) to two-dimensions (2D), clipping, and so on, which are dealt with in the graphics textbooks cited above in more than adequate form.

The simpler graphics forms are illustrated in figure 10.1 to figure 10.6, showing:

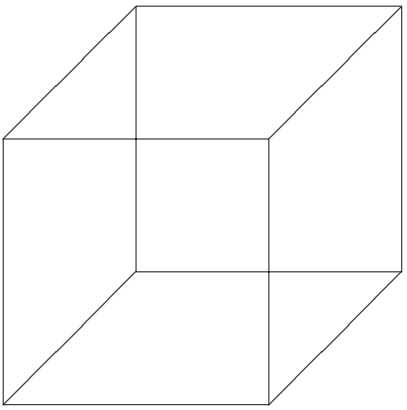


Figure 10.1: Wireframe drawing

- Figure 10.1–Wireframe drawing: All edges drawn regardless of whether or not they are really visible
- Figure 10.2–Hidden line: Invisible edges removed.
- Figure 10.3–Local hidden line: Simple local test for invisibility.
- Figure 10.4–Facetted model: Approximate model with convex facets with no hole-loops, generated from the exact model
- Figure 10.5–Facetted hidden line: Facetted model with hidden lines removed.
- Figure 10.6–Simple shaded picture: Facetted model with facets shaded.

10.1 Model-space graphics

As stated, what are called model space graphics techniques here are under the control of the modeller. Several variables and options affect or control the display of models. These control options and variations in the graphics output mode from the modeller.

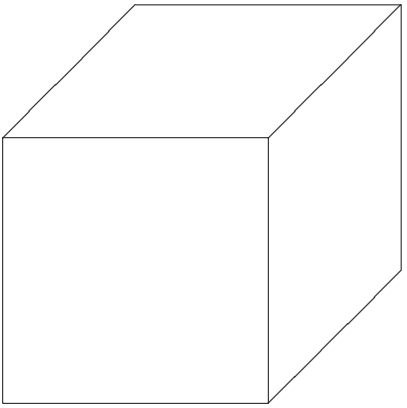


Figure 10.2: Hidden line

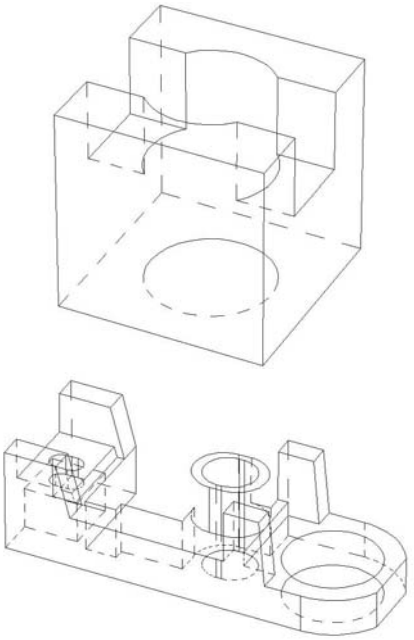


Figure 10.3: Local hidden line (lhl) images

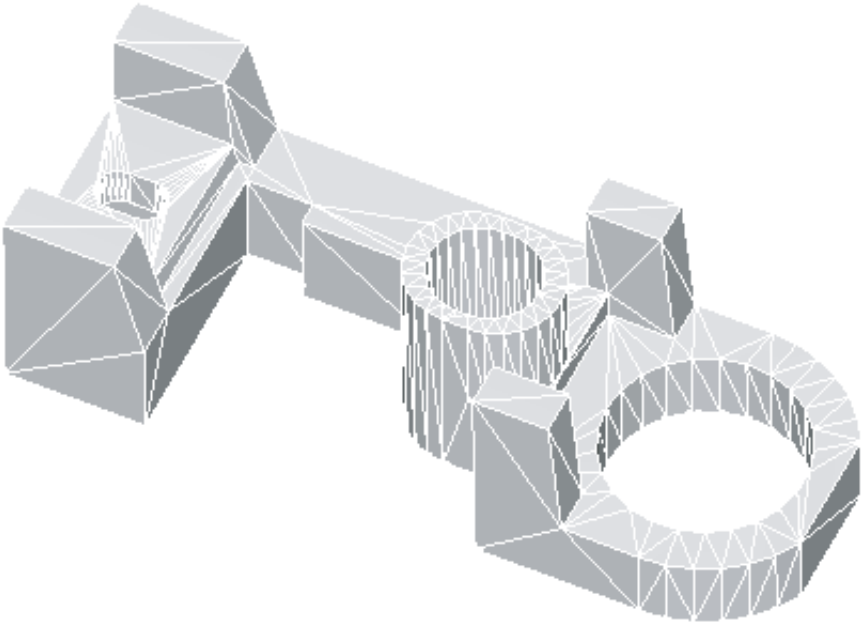


Figure 10.4: Facetted model (from the ACIS viewer)

10.1.1 Drawing styles

The drawing styles described here are based on those in the BUILD system developed by Dr. Ian Braid et al. in Cambridge. The user could set an eye position, with the direction of view defined as being towards the origin. Other options controlling the display are as follows.

Automatic scaling

It is necessary to set the scale of the drawing so that modeller world coordinates can be converted to graphics device coordinates. The scale can be set manually or automatically. Automatic scaling is relatively easy if there is a global spatial measure associated with the object to be drawn; otherwise the object has to be scanned, directly or indirectly, to determine the general size. If the object has a box or bubble associated with it to give the general spatial extent, then this can be used. Manual scaling is quicker, but it requires the user to know roughly what scale is being used. Automatic scaling has the advantage that it is more user friendly, but it means that drawings can shrink as the object gets larger, making comparison harder. It is useful as the default option so that a user can see the object at the largest possible scale and then decide what to do.

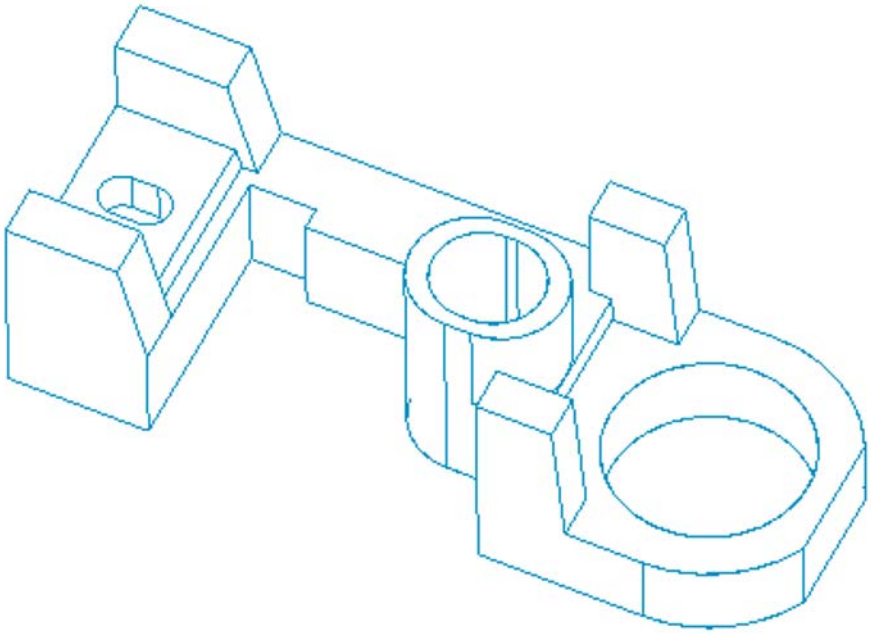


Figure 10.5: Facetted hidden line (from the ACIS viewer)

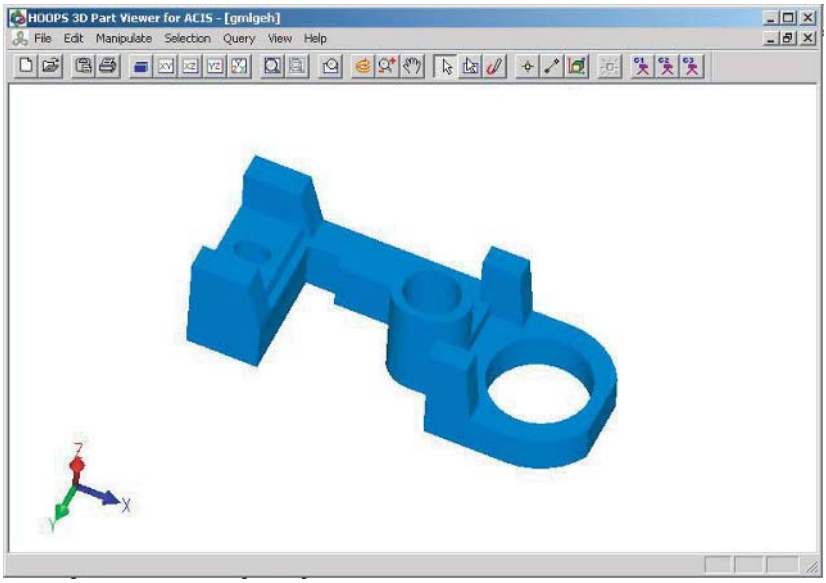


Figure 10.6: Simple shaded picture (from the ACIS viewer)

Axes

Axes are useful for showing the orientation of the object in 'normal' space. The axes can be drawn simply as three orthogonal lines at the origin labelled X, Y, and Z. Axes can be important for orienting some operations such as transformation specification.

Frame

A frame can be used to provide a system-specific or company-specific look to a drawing. In BUILD and GPM, this was a two-dimensional frame with the name of the system included and a space for the title, date, and user.

Hatching

Face hatching is useful for displaying object sections, for example. An alternative is to set the colour of the face. Face hatching is an example of a general class of graphics methods for identifying parts of a model. Colouring or highlighting parts of a model can be used to identify results of operations or targets for subsequent operations.

Labelling

Labelling a model provides a way of identifying model elements for modelling operations. Edges and vertices are easily identifiable, faces less so. Faces are gaps between edges and have no natural form. Drawing the face as a set of triangular facets leads to confusion. One possibility is to draw a small symbol on the surface of the face near one corner, perhaps a cone, and attach the label to it.

Local hidden line

Full hidden line removal is a complex process involving checking the whole object (see section 10.1.5). However, it is possible to perform quick local checks for visibility/invisibility. Edges that are convex with both neighbouring faces pointing away from the eye position, or concave edges with one neighbouring face pointing away from the eye position, are definitely 'invisible', hidden, and can be drawn dotted or simply not drawn according to taste.

Multiple views

Multiple views are normal in engineering drawings, hence, the inclusion of an automatic multiple view option. Normally such multiple views consist of three views of an object viewed along the X-, Y-, and Z-axes together with a general view. The same effect could be achieved manually by allowing the

user to set viewports and eye positions, but it is more convenient to provide an option to do this automatically. Multiple views can be used to compare relative positions of objects to be combined with Boolean operations. This is often done nowadays in CAD systems by a separate drawing module.

Perspective

Perspective is a depth cue that can give an added effect of realism to simple picture. For some purposes, though, a non-perspective view is more useful. For this reason, it is necessary to have this as an option.

Stereo

The stereo option in BUILD, developed by Robin Hillyard, was used to display a pair of superimposed images, one in red and the other in green, which could be viewed with suitable glasses. Stereo images can be produced by drawing the model from slightly different viewpoints. Instead of the simple red and green images mentioned above, there are now three-dimensional displays for producing images automatically.

Title

Used in BUILD to provide a title for drawings. The title consisted of the user, date, and a text description.

10.1.2 Viewing coordinate system and model transformations

The viewing system has its own coordinate system, which is defined from the viewing direction. This is defined from an eye position, under user control, and an arbitrary point, usually the approximate centre of the object being drawn, which can be determined from the centre of the box surrounding the object. This vector, from the arbitrary point to the eye position, say, is the normal vector of the viewing plane. Another vector, the "display up" vector, is needed to define the coordinate system (see figure 10.7). The X- and Y-axes are, as usual, orthogonal vectors lying in this plane. It is common that three-dimensional points can be specified directly to the graphics system, but the projection transformation can be calculated from this coordinate system. The other use for the coordinate system is for hidden-line elimination, as described in section 10.1.5. The origin of the coordinate system can be set anywhere on the line from the eye to the chosen point. If the origin is well behind the object (which can be determined from the object box or 'bubble'), then all z values will be positive. If the point is at the eye position, then all points will have negative z-components, if the eye position lies outside the object. If the eye position is inside the object, then points with a positive z-coordinate are implicitly behind the eye.

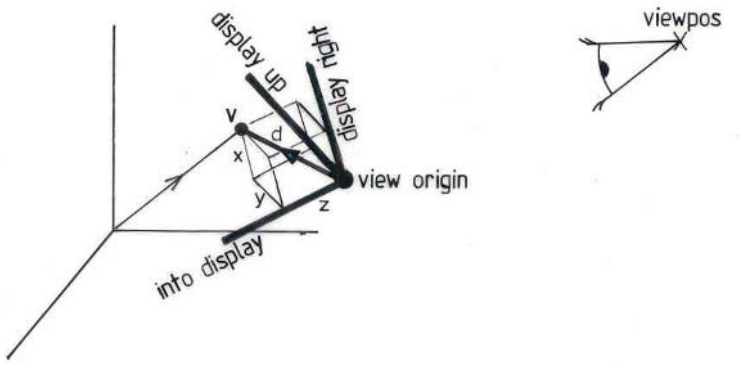


Figure 10.7: Graphics coordinate system

Another aspect to be noted is that objects often have a transformation matrix associated with them, as in the instances described in chapter 3. It is necessary to take this transformation into account when drawing the object. It may be possible to download this transformation to the graphics device and cope with it that way. Otherwise, all points from the body have to be transformed with this transformation before being used.

10.1.3 Drawing all edges in a body

Drawing all edges in the body is a basic graphics technique, suitable for wireframe- and solid-representation It is important because it uses the "all edges in body" list, not basic model connectivity; hence it can be used for some types of incorrect model, such as the results of incorrect modelling operations. It is the simplest and quickest method of drawing and can be implemented quickly to provide a basic graphics level, during the period that other, more complicated methods are being developed. Such images are not really sufficient and can be ambiguous, but producing more realistic 'solid' images takes time which can absorb effort while other parts of the modelling code are being produced. An example of the kind of image produced is shown in figure 10.8

Drawing an edge

The first step when drawing an edge is to check its curve type. If the edge has no curve reference then the edge can either be drawn as though it were straight, from the edge start position to the edge end position, possibly in a different colour, or not drawn at all. There are three basic cases for drawing edges with curve references: 1) straight edges, 2) circles and circular arcs, and 3) general curves. The reason for discriminating between these types is

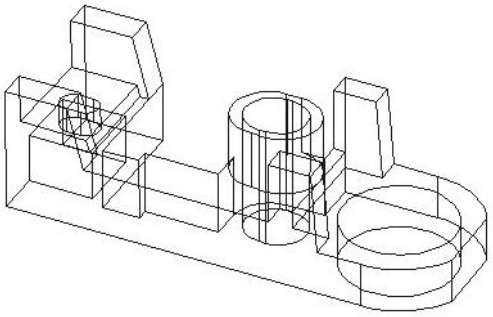


Figure 10.8: Wireframe image of MBB Gehause Rohteil

that standard drawing packages allow some curves to be defined as primitive drawing operations. However, the drawing operations based on the points described below have historical significance and form the background to current, more advanced methods. The point-based methods can be termed model-space methods as opposed to the graphics-device methods that use advanced graphics primitives. An advantage of point-based methods is that the points can be transformed using the model transformation, if one exists, to draw a transformed model.

Straight edges: If it is a straight edge, then the edge can be drawn from the edge start position to the end position. Note that this actually ignores any information in the curve, so there is the option to draw the edge based on the actual geometric information. It is probably sufficient to draw the line from the edge start point to the edge end point. If the edge is to be drawn dotted, one of the options, then the edge can be drawn as a series of line segments using parameter values. The edge parameter range is divided into an odd number of segments, say 11, and the odd-numbered segments drawn, i.e., segments 1, 3, 5, 7, 9, and 11.

Circular edges: Circles can be transmitted directly to graphics devices using the GKS Generalised Drawing Primitive, for example, and handled locally by a graphics device. If the body is untransformed, or if the graphics device can handle general transformations, then it may be better, if possible, to use such graphics mechanisms because this provides more flexibility for the graphics device. Alternatively, or if the graphics device cannot handle transformations, circles can also be drawn as a sequence of straight line segments, as for general curves. Unlike general curves, circles are uniformly curved, which means that uniform parametric division can be used to determine the straight lines segments representing the circle.

Generally curved edges: Edges with general curves can be drawn by determining several points along the edge and drawing straight line segments between them. There are two methods for determining the number of points: evenly spaced and unevenly spaced segmentation, which are illustrated in fig-

ure 10.9. Evenly spaced segmentation calculates a step-length and calculates points at regular intervals within the edge. This is easy to implement and gives not too bad an approximation for many practical curves, but it can be wasteful and give gross figures. Unevenly spaced segmentation is adapted to curvature so that more points are produced where the radius of curvature is small and fewer where the radius of curvature is large. Ellipses, parabolas, and hyperbolas are examples of simple curves with varying curvature. Free-form curves can be even more demanding, but usually smoothly varying free-form curves are to be expected.

A simple adaptive drawing method is to generate a set of evenly spaced points along the curve, as described above, and then examine the discrepancy between the midpoints of each line and the midpoint of the approximated curve section. If this discrepancy is greater than a certain amount, then the midpoint of the curve section is inserted. This splitting is applied recursively until the sequence of line segments approximates the curve sufficiently closely.

Another method is to determine the step-length along the curve from the curvature at the selected points along the curve. This is more satisfactory, from a mathematical point of view, than the two methods described above, which rely on the geometry being reasonable to work. Certainly under normal circumstances those methods will work, but they may not reveal problems in extreme cases with free-form geometry. However, analysing the curve to determine the step-length can be slower than the other methods and tie the graphics system closely to particular geometric forms.

The number of points used to draw curves is an optional user parameter. If an image is scaled up, then generally it is advisable to use more segments so that curves appear smooth. This may be difficult to arrange with device space graphics because it implies dynamic adaptation.

Visual identifiers: Finally, it may be necessary to provide face, edge, and vertex identifiers as part of the graphics output, which were referred to as 'labelling' when drawing options were described above. Labels allow a user to identify edges for subsequent modelling operations; they can also be useful for debugging. Note that, for end-user systems, graphical picking is more user friendly, and much more suitable than requiring a user to give model element identification numbers to control commands. Graphical labelling is useful for system development purposes.

In BUILD, labels were drawn approximately at the middle of the edge, and slightly offset to avoid being obscured by the edge. An arrowhead was also drawn on the edge to indicate its direction. Vertex numbers were drawn close to the vertex in question. Face labels were drawn close to one end of the first edge in the outer loop of the face. Label positioning is, to some extent, view dependent, reflecting their original use for static images. Another general observation is that it is helpful to be able to distinguish between labels. BUILD had a 'label style' for controlling which labels were drawn. Another possibility is to draw the labels in different colours. Although either method is possible, it can be useful to label model elements selectively, especially when

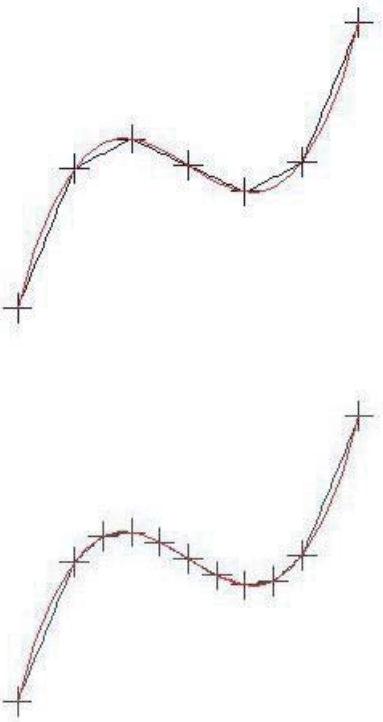


Figure 10.9: Even- and uneven-point spacing for drawing curves

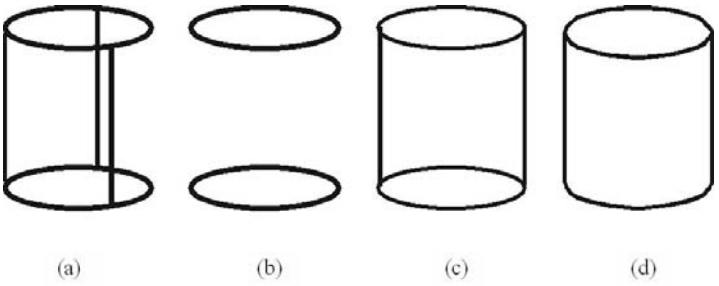


Figure 10.10: Fake edges and silhouette edges in the model

the model is very complicated. Another point to note is the scaling of the labels. These should be consistent with the model so that they maintain their size if part of the model is scaled up.

10.1.4 Drawing edges and silhouettes

The next level of sophistication is to draw silhouette lines as well as the edges of a model. Silhouette edges give an improved indication of curved faces, but they should be distinguished in some way from ordinary edges. I know of one apparent error when a user tried to pick a silhouette edge instead of an edge and complained because the operation did not work. Drawing silhouette lines changes the character of graphics from edge-based to face-based graphics. It is also a view-dependent graphics technique, not suitable for dynamic rotation because the silhouettes change as an object is rotated. However, model-space graphics is, essentially, for producing static pictures. Dynamic rotation falls into the domain of device-space graphics.

Figure 10.10 shows a cylinder with and without silhouette lines and 'fake' edges to illustrate their visual effects. In figure 10.10a three fake edges are shown. In figure 10.10b, the cylinder without fakes edges and without silhouette lines is shown. In figure 10.10c, no fake edges and silhouette lines are shown. The final image, figure 10.10d, shows the cylinder with silhouette lines and hidden lines removed.

To draw the silhouette lines, all faces in the body are examined. Planar faces can be ignored because they only have 'silhouettes' when their normals are perpendicular to the viewing direction; in which case, their edges provide the silhouettes. For a non-planar face, a silhouette curve is determined from the surface of the face. Silhouette curve calculation is part of the general interface to geometry, as mentioned in chapter 3. In a similar way to Boolean operations, the curve is then intersected back with the face; the intersection points, if any, are sorted; and the silhouette curve is drawn between these points. If the silhouette curve does not intersect any edge in the face, then the whole face is directed towards or away from the viewpoint. Silhouettes

divide faces and edges into visible and non-visible portions, which means that the points where the silhouette curves intersect edges in a face mark changes in the character of an edge from visible (at least potentially) to invisible. For local-hidden line images, this means that the edge changes from 'solid' to 'dotted' or invisible along its length. Edges intersected by the silhouette curve should be drawn in two (or more) pieces, say from visible point to silhouette intersection point, and from silhouette intersection point to invisible point. Edges extending through more than 180 degrees may be intersected twice by silhouette curves; in which case, they should be drawn as three pieces.

10.1.5 Exact hidden line algorithms

The face-based graphics mode is more evident for exact hidden line removal algorithms. There are many different algorithms for producing hidden line drawings, and it would require a lot of space to describe all of them in detail.

One reasonably simple algorithm involves converting the visible faces or visible face portions of the object into polygons in the viewing plane and then performing two-dimensional polygon clipping to remove overlapping parts. See figure 10.11.

With this method, it is convenient to define the viewing plane origin as the eye position, Z-axis towards the body being viewed, and discard any points with negative z-coordinates. This allows the eye to be put inside an object, not really useful for single objects, but it can aid visualisation of collections of objects.

The first step is to generate a set of polygons corresponding to visible faces and face portions. Each face is examined separately; planar faces with normals pointing away from the eye can be discarded immediately. For other faces, it is convenient to copy the face as a two-dimensional lamina. For curved faces, the silhouette curve is calculated and intersected with the lamina. If the curve does not intersect the lamina, then the whole face from which it was produced points towards or away from the eye, as determined by the surface normal at some point on the face. If the face points away from the eye, then the lamina can be deleted. If the silhouette curve intersects the lamina, then the face has visible and obscured parts. Edges are then inserted into the face copy corresponding to the silhouette curves and the lamina is divided into visible and invisible pieces. The invisible portions can be discarded. Note that there can be more than one invisible or visible portion from a single face.

After all faces have been examined, there will be a list of visible laminae that then have to be analysed to determine which lie in front of which. The first step is to approximate the lamina boundaries with a set of straight line segments and convert the vertex positions to the eye-based coordinate system, as mentioned above. The analysis involves intersecting each pair of polygon projections of the laminae and discarding obscured parts.

The intersection process involves examining pairs of edge projections in the viewing plane. If they intersect, then the z-coordinate at the intersection

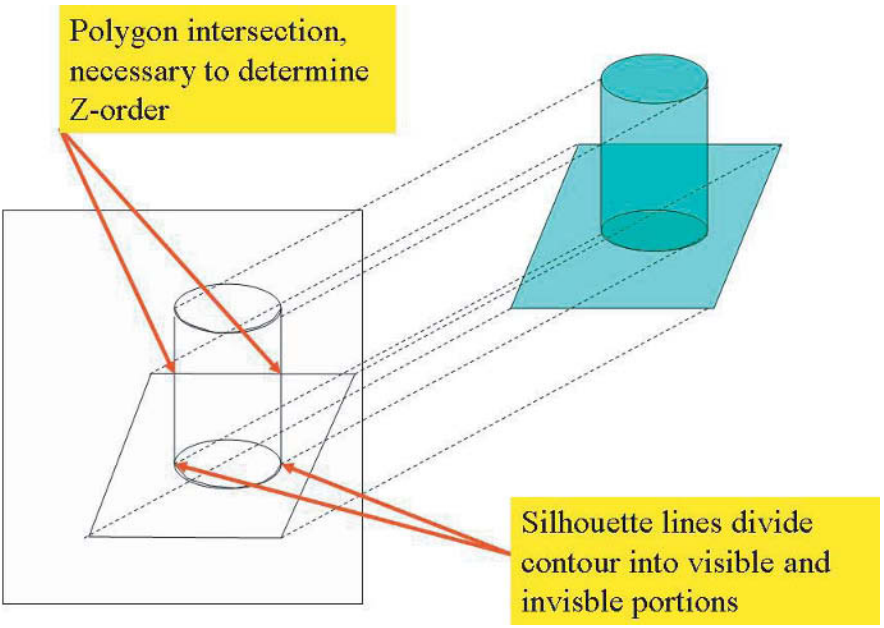


Figure 10.11: Simple hidden-line elimination

point for each edge determines which obscures which. As the origin of the coordinate system is at the eye position with the z-axis towards the object, then the edge with lower z-value obscures the other. The obscured part of the polygon boundary should be replaced by a copy of the obscuring polygon boundary between the points where the polygon projections intersect. If there is no intersection between the edge pairs, then it is also necessary to check whether one polygon projection completely obscures another; in which case, the obscured lamina is deleted. When all pairs of laminae have been compared, the polygonal projections of the remaining laminae are drawn.

10.1.6 Bread slicing

'Bread slicing' is a technique appropriate for raster devices, which was developed by Jared for BUILD. The image is built from a series of slices through the body. Each slice produces a closed contour in the object. This can be further processed to find the curve segments that are not obscured, or a z-buffer algorithm used to create the final image. Points along these curve segments are taken as the centres of rectangles on the screen, which are then coloured according to the surface normal at the point.

Obviously the method is expensive computationally, although it can produce high-quality images. The best quality pictures are produced when the rectangles used to create the picture are approximately one pixel large, but

this requires many sections through the model and surface normal calculations at many points along the curve sections.

The method of building the cross-sections through the body is very similar to the procedure for Boolean operations and planar sectioning described in chapter 6. First it is necessary to determine the section plane. Note, first of all, that there are two basic types of slicing: 1) with parallel planes and 2) with planes containing the eye-point. The second type will produce perspective pictures, but it also requires slightly different processing of the section contours. The planes should all lie on an axis through the eye position and orthogonal to the viewing direction. For the first type of slicing, the planes are all parallel to each other.

Depending on whether perspective or parallel bread-slicing is being performed, the contour segments are projected onto a line, which is effectively a strip of pixels on the screen. Occlusion calculations are performed to remove overlapping segments, and the rectangles are built. Finally, the strip of rectangles is displayed.

10.1.7 Ray tracing

Another method for producing good quality images is the ray tracing method. The method is well documented in the computer graphics literature so it is unnecessary to describe it in detail here. From a modelling point of view, though, the procedure involves defining a set of 'rays' as straight line geometric entities in the modeller. These can then be intersected with the model or models to be displayed by intersecting them with each face in the model (checking to see whether the surface–ray intersection point lies inside the face) and using the surface normal at that point to define new rays with the appropriate direction.

The tools needed, therefore, are traversal tools ("all faces in body" for each separate body), geometric intersections (straight–curve/surface), the point-in-face test, and surface normal calculations. Again, this graphics method will be extremely slow for complicated objects and scenes and is not (currently) viable as an interactive method. It can be useful, however, for building high-quality static images for, say, advertising or visualisation.

10.2 Device-space graphics

With device-space graphics, the model is transferred in some "neutral-form", a so-called "communications model". Sophisticated communications models have been examined by Carleberg ([16]). The problem with device space graphics lies in determining a satisfactory neutral form that incorporates the complexity of a general model but, at the same time, is simple enough to be general. There is no good answer because both computer graphics and solid modelling have independent developments so that trying to match the

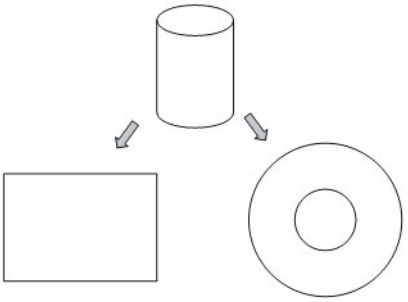


Figure 10.12: Cylinder parametrisation

two exactly is difficult. Much modern graphics is done by hardware, but hardware implementation requires simplified forms. Modelling development is tending toward model sophistication with more complicated models. It seems necessary, therefore, to come to some sort of compromise, with both sophisticated, device-space graphics and model-space graphics. Device-space graphics can be used to produce good quality images that can be rotated, for example, to aid visualisation, whereas model-space techniques, such as those described earlier in this chapter, are used to produce images more adapted to the model.

The simplest device-space graphics method uses facetted models. Facetting approximates the surfaces of the model with convex planar, singly connected elements. The easiest facets to handle are three-sided facets because they are bound to be planar, but many graphics devices can handle general facets as well, provided that they are planar.

An object can be facetted by facetting each face separately. The first step is to subdivide curved edges into short segments. Next each face is examined and converted to parameter space, the inner loops are joined to the outer loops, concave vertices are joined to close neighbours, and finally all subfaces are converted back to real space.

The edges are subdivided recursively until the difference between the midpoint of the start and end vertex positions and the midpoint of the edge is less than some tolerance. The subdivided edges can then be approximated by straight line segments. Some care must be taken, though, with free-form edges, where the midpoint may coincide within tolerance but the two halves of the edge may not be curved so that a straight line approximation is inaccurate. In this case, it is better to subdivide the edge into short segments regardless of the midpoint check, and then apply the midpoint check to the subedges.

Each face is facetted in parameter space so that the facetting is done in a plane. The first step, therefore, is to determine an appropriate parameter

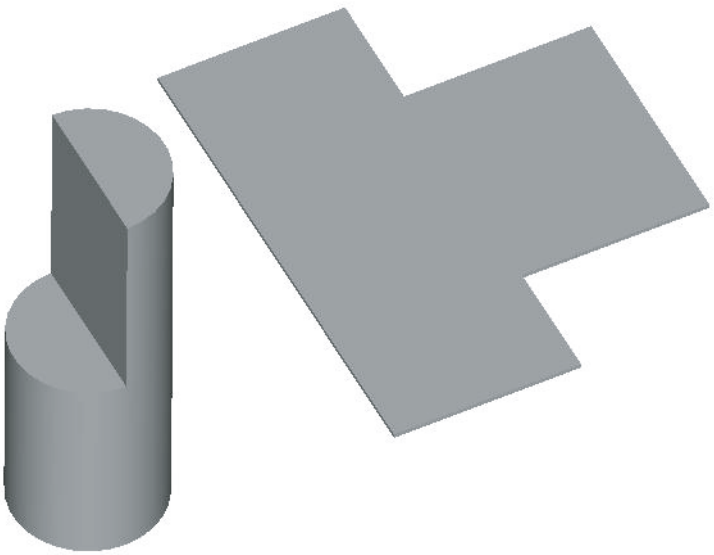


Figure 10.13: Cylinder parametrisation with split edge

space for the face. This is not a straightforward problem. Planes and free-form surfaces have a natural parametrisation that can be used. Other surfaces have to be converted, but the conversion introduces conflicts between maintaining connectivity and maintaining linearity. Consider a cylinder, for example, as shown in figure 10.12. The cylinder can be 'opened out' into a parameter space by selecting some arbitrary boundary and 'unwrapping' the cylinder. Alternatively the cylinder can be converted into a circular disc. The former parametric conversion preserves linearity, but an edge that crosses the arbitrary boundary (figure 10.13) will be cut into two pieces. If the face only uses a part of the cylinder, then the arbitrary boundary can be chosen to lie outside the face, but this is not always possible. An alternative is to convert the cylinder to a disc by choosing a notional 'apex' point lying on the cylinder axis (but away from the face). This point then becomes the origin of the parameter space, and the face is mapped onto a circular disc centred at the origin. Although this preserves continuity, circular edges around the cylinder at different heights have different lengths in parameter space, as illustrated in figure 10.14. Also, the midpoint of a straight line connecting two points at the same distance from the origin will lie at a different height on the cylinder from the endpoints.

The problem with facetting curved faces is because they cannot be naturally mapped onto planes in parameter space, except with free-form surfaces. This can be overcome by splitting the face up into regions that definitely

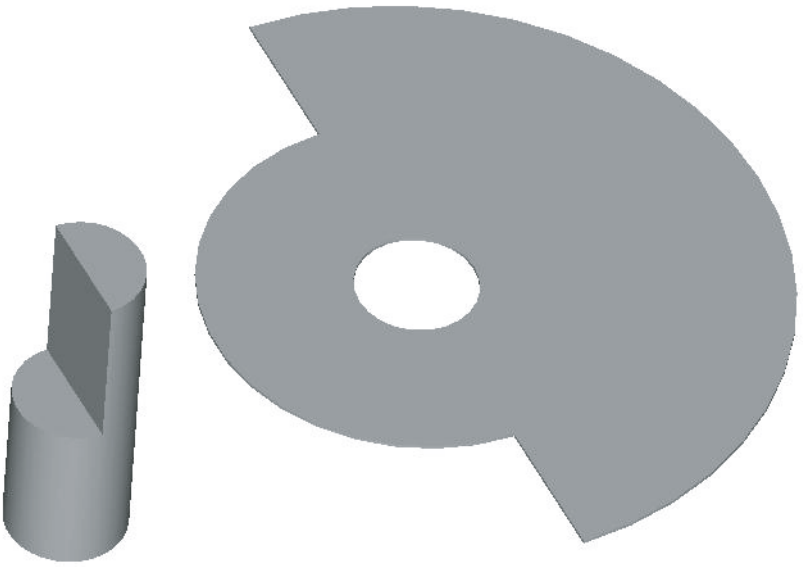


Figure 10.14: Non-uniform cylinder parametrisation

extend through less than 180 degrees and then to map these onto planes. For example, the cylindrical face in figure 10.13 is split into four faces using two perpendicular planes. Each of the four sub-faces can then be mapped into parameter space using a convenient parametrisation based on the artificially imposed boundaries. Conical surfaces can also be split using two planes; spheres and toroids need three orthogonal planes.

10.3 Facetting an object

The following method is destructive because it uses the body structures for traversal. If this is not desired, then it may be necessary to copy the object before facetting. Otherwise substructures could be used, or the object retraversed and unfacetted afterwards. If a copy is used, then it may be necessary to attach identity tags to the elements in the copy, referring to the entity in the original from which they derive.

Three common parameters are used to control facetting, as follows:

1. Chord height
2. Edge length
3. Internal triangle angle

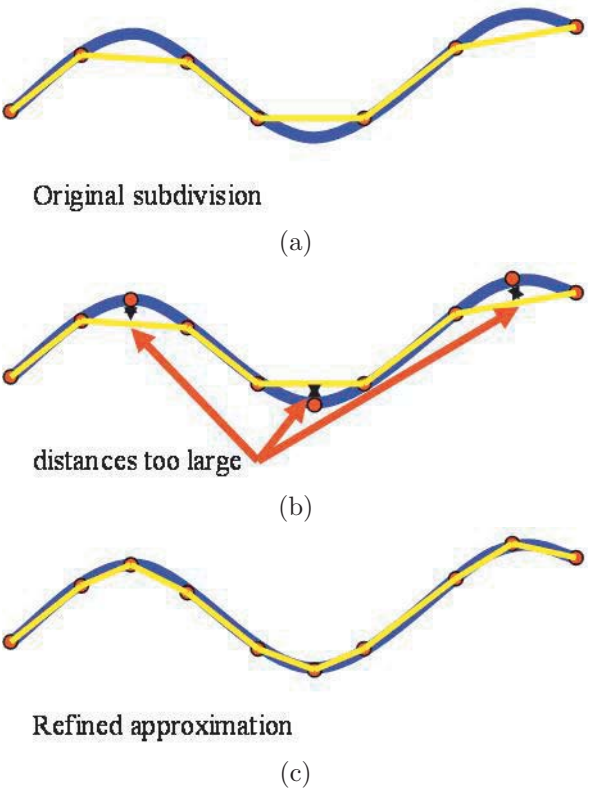


Figure 10.15: Subdividing a curved edge

The chord height parameter is the most commonly used and controls the discrepancy between the facetted approximation and the original object.

The edge length parameter defines the maximum length of edges. Using this can give more even facetting than using just the chord height.

The final parameter is the internal triangle angle. This controls the minimum internal angle of triangles and, hence, the evenness of the triangular facet shapes.

The first step in facetting is to traverse the body subdividing the edges into appropriate lengths. This is illustrated in figure 10.15. The edge is subdivided into intervals. If it is a straight edge, then this is only necessary if the edge length is longer than the maximum length, if set. If the edge refers to a quadratic curve (circle, ellipse, parabola, etc.), then it can be divided at intervals, as described earlier. If the edge is a complex curve, then it should be divided at inflection points, if any, and at intermediate intervals.

Once this has been done, you might have something like the curve shown in figure 10.15a. The middle points of each section are then compared with the

closest points on the curve. If the distance is greater than the chord height, as in figure 10.15b, the section is divided into two, as in figure 10.15c, and the process is repeated until every segment is within chord height distance from the curve.

The next step is to facet the faces. Faces are facetted by connecting any concave vertices (facets with an internal angle within the face of greater than or equal to 180 degrees) to a neighbouring 'visible' vertex. If none exist, then the face is convex and can be subdivided. Otherwise, once all such vertices have been connected, the original face consists of several convex sub-faces. Each convex face with more than three sides is then subdivided until only triangular faces exist. The new edges introduced during the subdivision process can have straight curves assigned. The midpoints of these edges need to be checked to see whether the distance between these and the nearest point on the surface is greater than the chord-height. If so, the edge is split into two, the new vertex is moved onto the surface, and it is considered to be a concave vertex.

A simple example is shown in figure 10.16.

The original face is shown at the top and has twelve vertices, which are labelled arbitrarily in the figure. Vertices v4, v5, v9, v10, v11, and v12 are concave vertices. First, in figure 10.16a, v12 is connected to v2. The candidate vertices for the join are v1, v2, and v3, because these are the only ones that are visible. Similarly, in figure 10.16b, v11 is connected to v3. In figure 10.16c, v9 is connected to v2, but it remains a concave vertex so it is connected to v1, in figure 10.16d. Next, v10 is connected to v3, in figure 10.16e, but, as with vertex v9, it remains concave so it is also connected to vertex v4, in figure 10.16f. Vertex v4 is still concave, so it is connected to vertex v1, in figure 10.16g. Vertex v5 is connected to vertex v7, in figure 10.16h, and again to vertex v8, in figure 10.16i, as it, too, remains concave after the first join. Now, the original face has been broken up into convex sub-faces, and those that have more than three sides are broken down, arbitrarily here, with edges connecting v12 to v3, v1 to v10 and v4 to v8. This gives the final facetting shown in figure 10.16j.

The new faces can be assigned the same geometry as the original face to help edge removal in a tidying-up phase.

10.4 Drawing free-standing geometry

As part of a design system, it may be necessary to draw free-standing geometry that has been created as a design aid. This is not particularly onerous, because the basic techniques are as outlined earlier in the chapter, but there are special points to note. However, it is important to note the difference between drawing free-standing geometry as geometry and free-standing geometry as, for example, hole centre-lines. Drawing general geometry is described here. When free-standing geometry is used for special purposes, then drawing it

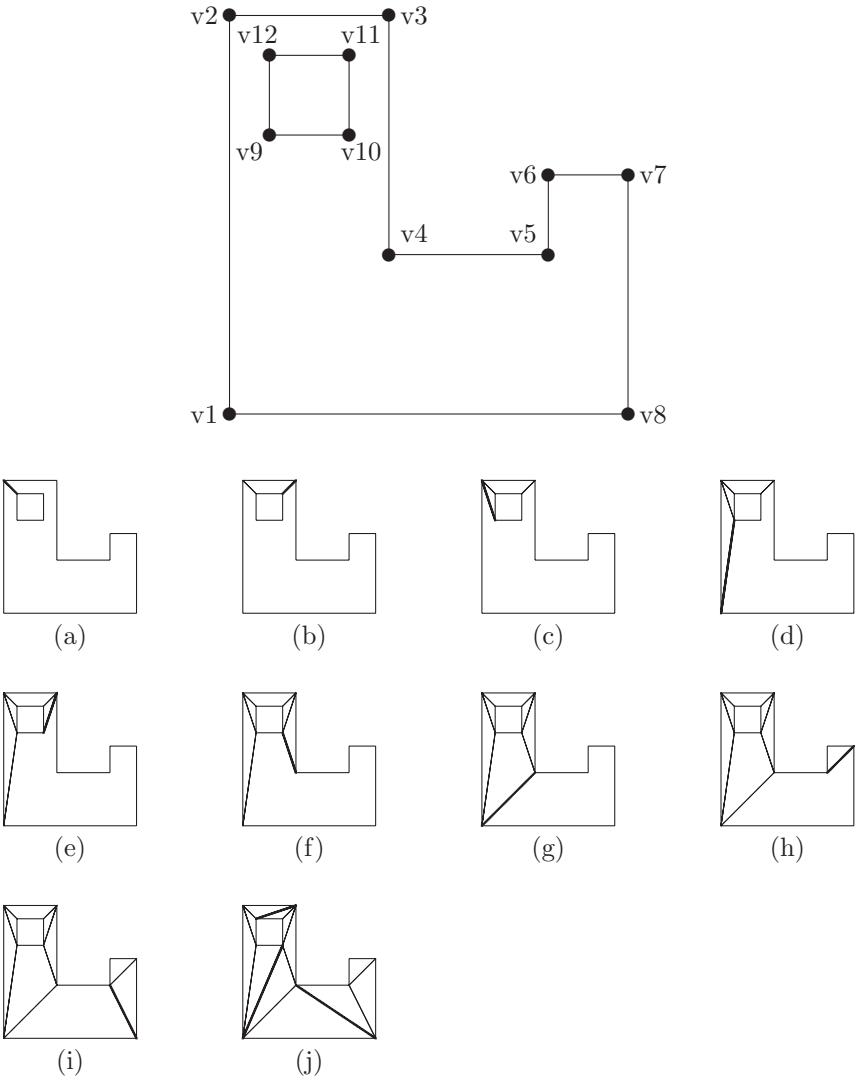


Figure 10.16: Facetting a face

falls into the domain of the application. It is unrealistic to try and foresee and cater for all possible uses; it is better to provide general tools that a user can adapt or use as guidelines, if necessary.

The first thing that must be decided is how much of the free-standing geometry is to be drawn. Whereas in a volumetric model the geometry is limited by the edge or face embedded in it, these limitations are not imposed in free-standing geometry that can extend infinitely. It is necessary, therefore, to define the portion of model space that is interesting and draw the part or parts of the geometry that lie within this portion of space.

The next thing to decide is how to display the geometry. Curves and points are not a problem; however, surfaces pose a problem. A plane, for example, extends infinitely; there are no natural edges to give an outline to it. It is necessary to adopt some artificial representation. The most general is to draw a set of hatching lines on surfaces, although an attractive representation is to draw free-standing surfaces as transparent. It is important that, as far as possible, the geometric entities can be distinguished from models. This can be done, for example, by using different colours or different line types.

The nature of the hatching lines will obviously vary according to the nature of the geometry as well as the taste of the system designer. A plane might be drawn with straight line cross-hatching (or equally well by a series of concentric circles and radial lines). A cylinder or a cone might be drawn as a set of circles about the axis and straight lines around the axis. A sphere might be drawn as a set of longitudinal and latitudinal circles, and similarly for a torus. A free-form surface might be drawn as a grid on the surface. Equally possible is to draw a set of points of the surface, although this may be more costly in graphics terms.

It is only possible to give outlines here, because there are various possibilities and no obviously superior one. If free-standing geometry is allowed in a modeller, then it is reasonable to provide facilities for drawing it, and so some method should be provided.

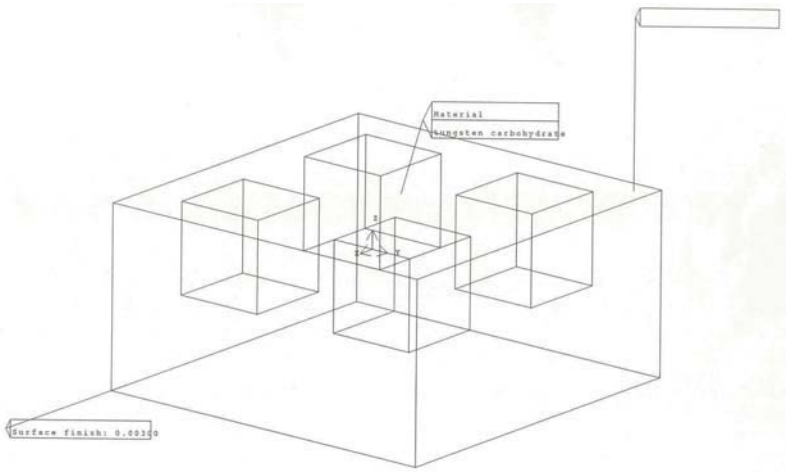
10.5 Drawing non-geometric information

Non-geometric information is another awkward area to be specific about. As mentioned, in chapter 8, there are no clear guidelines about how non-geometric information should be handled in a modeller; hence, only general principles can be outlined here. It is, though, important to have a clear philosophy about what to do with it so that it can be presented to the user in one form or another. However, which colours to use, which forms to use, and so on have to be decided.

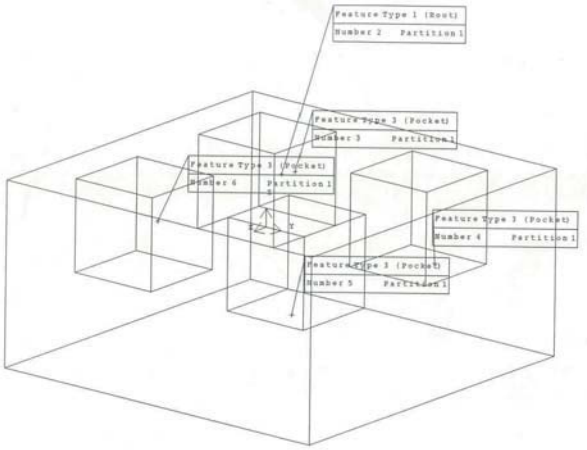
The simplest way of drawing non-geometric information is as tags attached to parts of the model. Each tag is drawn as a box containing appropriate information in text form. However, positioning the tags needs to be done with care to avoid obscuring the model. Even so, a model of even moderate complexity

may have a large quantity of data associated with it, for example, feature interpretations, manufacturing information such as surface finish, shape modifiers such as blends and threads, and associativity information. Not all may be needed at any one time, though; hence, it will probably be necessary to display the information selectively. Two examples of labelled images from a naive implementation I once did in BUILD are shown in figure 10.17.

Another possibility for displaying shape modifiers is to draw the appropriate part of the body with its modified shape. Braid developed this technique in 1979 [11] for implicit blending. The basic idea was that many blends were introduced into a model by a mould maker running his thumb along sharp edges. As such the blends were not necessary as explicit parts of the model and were represented by tagging edges. When the model was drawn, the tagged edges could be drawn either in their explicit sharp form or rounded. Other conventions for displaying model elements, for example, screw threads, can be used instead of modelling and displaying the explicit geometry.



(a)



(b)

Figure 10.17: Drawings with labels

Chapter 11

Inputting and outputting

Inputting and outputting models in character form is dealt with in this chapter. The basic discfile method described here was developed for the BUILD system. Unfortunately I do not know to whom to attribute the method, which I think is a very elegant method for reading and writing to disc, as well as general copying and other purposes.

The method uses logical pointers instead of the real memory addresses. Logical pointers are small integers, so that an edge might have left loop 1, right loop 2, and be adjacent to edges 2, 3, 4, and 5, say. The integers are used for accessing the appropriate entities from arrays, as described later.

11.1 Writing to disc

The basic steps for writing an object to disc are as follows:

1. Preserve the model entity numbers.
2. Renumber the model entities from 1 to n (where n is the number of that type of entity in the model).
3. Write the header.
4. Write the number of each type of entity to the discfile.
5. Write the entities to disc.
6. Replace the original entity numbers.

1 **Preserving the model entity numbers**

Here it is assumed that each entity has a system identity number. Not all systems do, but it is useful to have such an identity number for debugging, printing, and other purposes. The reason for

preserving the original numbers is that they are changed in the writing operation, and it is important that after the operation, the entities have their original numbers replaced.

First it is necessary to count the number of each type of entity in the model. For each entity type, an integer array of the appropriate size is created and the entities are scanned, inserting the identity numbers into the array.

2 **Renumbering the model entities**

The new entity numbers form the logical pointers in the structure in the discfile. Entities should be renumbered from 1 to n, where n is the number of that type of entity in the model. As mentioned, the new entity numbers will be used as array pointers when retrieving the object; writing these logical pointers saves searching and comparing when reading the object.

3 **Writing the header**

The file header can contain any desired information, relevant or even irrelevant for modelling purposes. Possible information in the header might include:

1. Modelling system name and version number
2. Creator identity
3. Date created
4. Access code and password
5. Comments
6. Tolerance information

The form and type of data recorded in the header depend on the general system requirements and vary between users; hence, it will be ignored here. The only important requirement is that it must be identifiable as header data.

4 **Writing the numbers of each type of entity**

The numbers of particular types of entity in the model is available from the first step. The reason for writing it out is so that the entities can be created at the beginning. Obviously this step will vary from modeller to modeller because the datastructure will vary. Because of this it is impossible to be exact about the disc format so the following is based on the suggested datastructure in Appendix A.

The first thing to do is to determine what entities there are in the datastructure. In chapter 3 the following are defined:

VERTEX
EDGE
FACE
LOOP
FACEGROUP
SHELL
POINT
CURVE
SURFACE
FEATURE
SHAPE-MODIFIER
CONSTRAINT-DATA
SUPPLEMENTARY-GEOMETRY
PASSIVE-DATA

WIREFRAME-OBJECT
SHEET-OBJECT
VOLUME-OBJECT
INSTANCE
MECHANISM-CONSTRAINT
SUPPLEMENTARY-GEOMETRY

GROUP-OF-OBJECTS

These can be arranged into three main groups: assemblies, single objects, and model elements. The number of each of the basic elements is first written to the file. For an assembly, the basic elements are the instances, volume-, sheet-, and wireframe objects; mechanism constraints; and suplementary-geometries. For a volume, sheet, or wireframe, the basic elements are the faces, edges, vertices, and so on in the first group.

It is not obligatory to have all these elements in a datastructure, and it is, in fact, more likely that a modelling system will start out with a basic datastructure that will be extended as new useful elements are identified. For this reason, it is important to try and make the disc format extensible for new software versions. One way of doing this is to precede the numbers of entities with keywords identifying the datastructure element, as shown in the example section. Similarly, keywords can be used in writing out the datastructure elements, as described in the next section. It is not obligatory to use keywords, but it can give more flexibility. Another alternative is to add numbers of new elements at the end of the list, but this can give an illogical order to the element numbers.

5 **Writing the entities to disc**

The exact disc format for an entity depends, of course, on the nature of the datastructure. Pointer fields are written as the number of the entity pointed to, integer, real, character data, and so on are written directly. Note, also, that NIL pointers have to be handled in some way. These can be written as 0 or -1, for example, provided that these are not used as normal entity numbers.

There is a question of whether to use keywords as part of the disc format. As pointed out, this can be useful if the disc format may change. It can also make the discfile easier to read, although this is probably not very important.

For example, using the edge datastructure definition given in Appendix A, section A.1.2, the record for an edge (assuming a box is used for the space definition) might appear as:

EDGE 1 RCW 3 RCC 2 LCW 8 LCC 10 RLOOP 1 LLOOP 6 START 1 END 2 NEXT 2 PREV 12 CURVE 1 FRIEND -1 COGEOM -1 INFO -1 BOX [-1.0,1.0,-1.0,-1.0,-1.0,-1.0] MARKER 0000

It could equally well be written as:

EDGE 1 3 2 8 10 1 6 1 2 2 12 1 -1 -1 -1 [-1.0,1.0,-1.0,-1.0,-1.0,-1.0] 0000

giving a more compact but less readable file format. The choice between the two is somewhat arbitrary, and neither is obviously better than the other. Having a compact disc format may be important enough for most implementors to make the second format preferable. In the examples in this chapter, keywords will generally not be included. It can also be pointed out that it is not necessary for the discfile to be readable because the discfile should not be read or altered directly. It should always be possible to read old discfiles and, if necessary, modify the model inside the system to maintain consistency. Making files readable can be useful to help people unfamiliar with the format to understand what has been written. This can be useful during system development, but for users it is possible that binary output is more appropriate because the users will most probably not read the files and need to reduce space.

One other point about keywords is that keywords are useful for writing entities with variable syntax. In the datastructure defined in Appendix A, some entities have fields that can be pointers to one of a set of different types of entity. For example, a direct-pointer variant LOOP can refer to an EDGE or to a VERTEX as its start. With these, it is necessary to distinguish in some way which type of pointer is being read in.

Note that the OWNER and SPACE fields are not written out, because the owner object is, by definition, the object being written out and so should be set to the new object being read. The space field is not written out as a pointer, but the information is written directly. There is an option not to write it out at all and to recalculate it when needed.

6 **Replacing the original entity numbers**

Finally, once all entities have been written to disc, it is necessary to replace the original numbers so that the object remains unchanged under the operation. This is done simply by scanning through the entities, retrieving the old entity number from the appropriate array using the current entity number as index.

11.2 Reading from disc

Reading from disc has to match writing to disc. There can be a problem with reading discfiles written with older versions of a modeller, which can mean that previous versions of model reading functions should be preserved to read old model files and perform an automatic conversion to the new model format (see also section 11.3). Preserving the system name and version number is important to be able to identify the file origin to know whether it can be read properly.

The basic steps for reading a model are as follows:

1. Read the model file header.
2. Read the number of each type of entity, and create the new model entities.
3. Read the discfile.

1. **Reading the model file header.** As described in section 11.1, the file header contains various information including important information about the readability of the file such as access code, if any, and the version of the modelling system used to write the file. The version number allows some automatic conversion from a former datastructure to the current one to take place, if this is possible, as described in section 11.3.

2. **Reading the number of each type of entity and creating the new model entities.** Having the number of each type of entity at the beginning of the file means that all necessary entities can be created initially and then defined as the entity records are read from the file. For example, suppose that an object being read has, among other things, 16 vertices, 24 edges, 12 loops, and

10 faces. Sixteen new vertex records are created, and pointers to these preserved in a 16element vertex pointer array. Similarly, 24 edge records, 12 loop records, and 10 face records are created, and pointers to these are preserved in a 24-element edge pointer array, a 12-element loop pointer array, and a 10-element face pointer array. When a reference to, say, vertex 3 is found, the vertex referred to in element 3 of the vertex pointer array is substituted.

3. **Reading the discfile.** Each entity record in the file should contain a complete description of the pointer fields of the original entity, which are recorded as logical pointers. The discfile record is a character 'image' of the internal memory record. To read an entity, the list of integers is read, interpreting each as a pointer reference and setting the appropriate field in the new entity.

For example, suppose an edge record is as follows:

EDGE 7 6 5 12 9 3 2 5 6 8 6 1 1 -1 -1 -1 [-1.0,1.0,1.0,1.0,1.0,1.0] 0000

The keyword EDGE indicates that an edge is being read, and the first integer indicates which edge is being defined. The fields are interpreted in the order:

RCW RCC LCW LCC RLOOP LLOOP START END NEXT PREV CURVE FRIEND COGEOM INFO SPACE MARKER

The integers 6, 5, 12, and 9 are interpreted as edge pointers to be retrieved from the corresponding elements of the edge array. The next two numbers, 3 and 2, are interpreted as loop pointers to the right and left loops, respectively, and retrieved from elements 3 and 2 of the loop pointer array. The two values 5 and 6 are interpreted as pointers to the start and end vertex from the vertex pointer array. The next two integers, 8 and 6, are again taken as edge pointer references for the next and previous edges in the 'all edges in body' list. The integer 1 is taken as a curve reference. The next three integers are all -1, which here is taken to indicate a NIL reference, so the FRIEND, COGEOM, and INFO fields are all set to NIL. The last information read in is the marker information indicating special status of the edge recorded as a bit pattern.

The owner field of the edge is set as a pointer to the volume object being created, and the space field of the edge is set to NIL.

The general principle is that each entity in the model is written as a separate, complete record. Entities are not cross-referenced so that, for example, if the Right ClockWise edge pointer of edge 7 is set to edge 6, no change is made in edge 6. The writing and reading procedures have to be matched so that the integer sequence read back from disc is correctly interpreted as references to elements in the new model. The problem of reading old discfiles, with different formats is discussed next.

11.3 Reading old discfiles

It is normally the case that a modelling system is developed over a period of time and that the datastructure is enhanced periodically, which means that models produced with early versions of the modelling system are not entirely compatible with later versions of the system. It is therefore necessary to be able to adapt the old model discfiles somehow so that the saved models can still be used.

There are several ways of doing this. One method is to edit the discfile (manually or automatically) and add the extra fields. This is not particularly desirable because of the complexity of the changes, but it is possible.

Another method is to preserve old versions of the discfile reading code and perform the changes within the reader, adding new elements as required to make the earlier model description complete. This method was developed in the BUILD modelling and was termed "Oldfromdisc". For example, suppose that version 1.3 of the software contains only faces, edges, and vertices but no facegroups that are introduced in version 1.4. A version 1.3 model can be read using the old reading code to read the model, creating a facegroup to contain all faces and setting the appropriate facegroup field of all faces to the new facegroup. The deficiencies in the discfile are made good according to some default strategy so that the model read back is consistent with the new version of the modelling code.

A third method is to make the discfile reading code capable of reading earlier versions by using the system version number, recorded in the file, to indicate what deficiencies are present in the discfile. The procedure would be to create any extra entities where they would normally be read for discfiles created with earlier versions of the modeller. In the above example, where facegroups are added to the datastructure, instead of reading the number of facegroups, the number would be set to 1 as default. The facegroup would be defined when the volume is created. Instead of reading the facegroup number as part of the face record, the facegroup field would be set to point to the artificially created facegroup instead.

Although it is not desirable to have very old model files preserved and expect the modelling system to always be able to read the files, this may not be possible. Old archived models may be needed during the lifetime of a product, and a commercial system based on a solid modeller needs to be able to read them. Possibly it is better to perform upgrades continuously using background processes so that a file is only one, possibly two, versions out of date. If not, special conversion code may be needed.

11.4 Examples

Consider the cube shown in figure 11.1.

The discfile may appear as follows:

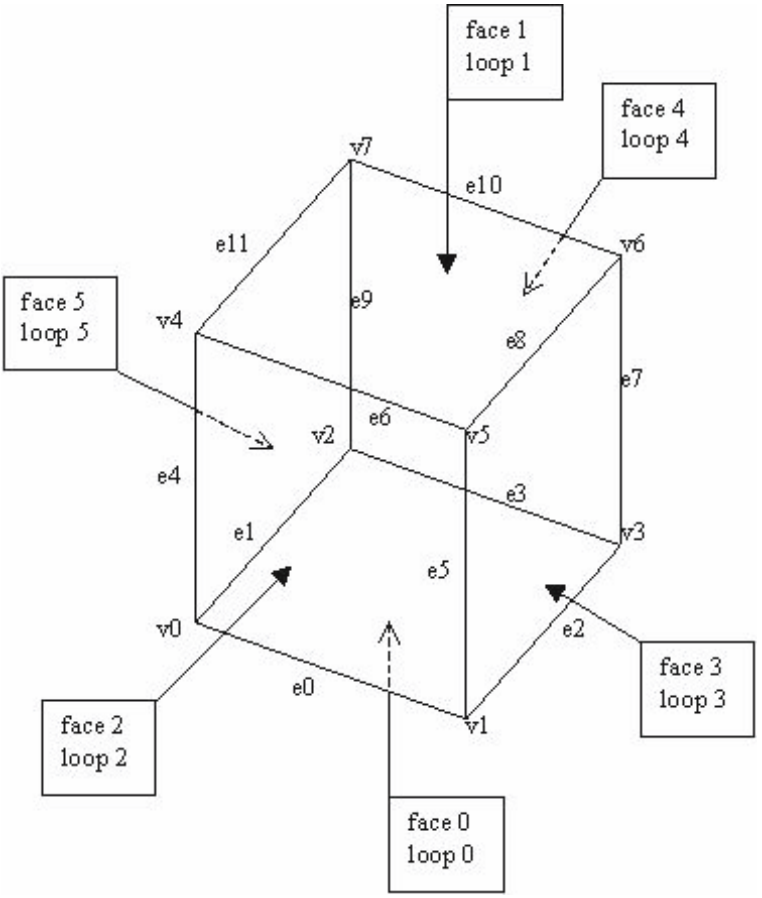


Figure 11.1: Basic cube

```
# System: WOMBAT I; Version: 5.3
 # Created by: STROUD
# Date: 17 JUN 1993
# Access: Unrestricted; Password: @@@@@@@@
# Comments: Simple cube example
# Tolerance 0.0000001 VOLUME:
v 8 e 12 f 6 l 6 p 8 c 1 s 6 fg 1 sh 1 fea 0 mc 0 sm 0 cd 0 sg 0 pd 0
volume -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000
shell 1 1 1 -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000
facegroup 1 1 1 1 s -1 -1 -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000
vertex 1 1 e 1 2 -1 -1 -1 0000
vertex 2 1 e 2 3 -1 -1 -1 0000
vertex 3 2 e 3 4 -1 -1 -1 0000
vertex 4 3 e 4 5 -1 -1 -1 0000
vertex 5 5 e 5 6 -1 -1 -1 0000
vertex 6 6 e 6 7 -1 -1 -1 0000
vertex 7 8 e 7 8 -1 -1 -1 0000
vertex 8 10 e 8 1 -1 -1 -1 0000
edge 1 3 2 8 10 1 5 1 2 2 12 1 -1 2 e -1 [-1.0,1.0,-1.0,-1.0,-1.0,-1.0] 0000
edge 2 6 8 1 4 4 1 1 3 3 1 1 -1 3 e -1 [-1.0,-1.0,-1.0,1.0,-1.0,-1.0] 0000
edge 3 4 1 10 5 1 6 2 4 4 2 1 -1 4 e -1 [1.0,1.0,-1.0,1.0,-1.0,-1.0] 0000
edge 4 5 6 2 3 3 1 3 4 5 3 1 -1 5 e -1 [-1.0,1.0,1.0,1.0,-1.0,-1.0] 0000
edge 5 7 4 3 12 3 6 4 5 6 4 1 -1 6 e -1 [1.0,1.0,1.0,1.0,-1.0,1.0] 0000
edge 6 9 2 4 7 4 3 3 6 7 5 1 -1 7 e -1 [-1.0,-1.0,1.0,1.0,-1.0,1.0] 0000
edge 7 6 5 12 9 3 2 5 6 8 6 1 -1 8 e -1 [-1.0,1.0,1.0,1.0,1.0,1.0] 0000
edge 8 11 1 2 9 5 4 1 7 9 7 1 -1 9 e -1 [-1.0,-1.0,-1.0,-1.0,-1.0,1.0] 0000
edge 9 8 6 7 11 4 2 6 7 10 8 1 -1 10 e -1 [-1.0,-1.0,-1.0,1.0,1.0,1.0] 0000
edge 10 12 3 1 11 6 5 2 8 11 9 1 -1 11 e -1 [1.0,1.0,-1.0,-1.0,-1.0,1.0] 0000
edge 11 10 8 9 12 5 2 7 8 12 10 1 -1 12 e -1 [-1.0,1.0,-1.0,-1.0,1.0,1.0] 0000
edge 12 5 10 11 7 6 2 8 5 1 11 1 -1 -1 e -1 [1.0,1.0,-1.0,1.0,1.0,1.0] 0000
loop 1 1 1 e -1 -1 0 0000
loop 2 2 7 e -1 -1 0 0000
loop 3 3 7 e -1 -1 0 0000
loop 4 4 9 e -1 -1 0 0000
loop 5 5 11 e -1 -1 0 0000
loop 6 6 12 e -1 -1 0 0000
face 1 1 1 1 2 -1 -1 -1 -1 [-1.0,1.0,-1.0,1.0,-1.0,-1.0] 0000
face 2 2 2 1 3 -1 -1 -1 -1 [-1.0,1.0,-1.0,1.0,1.0,1.0] 0000
face 3 3 3 1 4 -1 -1 -1 -1 [-1.0,1.0,1.0,1.0,-1.0,1.0] 0000
face 4 4 4 1 5 -1 -1 -1 -1 [-1.0,-1.0,-1.0,1.0,-1.0,1.0] 0000
face 5 5 5 1 6 -1 -1 -1 -1 [-1.0,1.0,-1.0,-1.0,-1.0,1.0] 0000
face 6 6 6 1 1 -1 -1 -1 -1 [1.0,1.0,-1.0,1.0,-1.0,1.0] 0000
point 1 1 v -1 1 0000 (-1.0,-1.0,-1.0)
point 2 2 v -1 1 0000 (1.0,-1.0,-1.0)
point 3 3 v -1 1 0000 (-1.0,1.0,-1.0)
```

```
point 4 4 v -1 1 0000 (1.0,1.0,-1.0)
point 5 5 v -1 1 0000 (1.0,1.0,1.0)
point 6 6 v -1 1 0000 (-1.0,1.0,1.0)
point 7 7 v -1 1 0000 (-1.0,-1.0,1.0)
point 8 8 v -1 1 0000 (1.0,-1.0,1.0)
curve 1 1 e -1 12 0000 short straight
surface 1 1 f -1 1 0000 plane (0.0,0.0,-1.0) -1.0
surface 2 2 f -1 1 0000 plane (0.0,0.0,1.0) -1.0
surface 3 3 f -1 1 0000 plane (0.0,1.0,0.0) -1.0
surface 4 4 f -1 1 0000 plane (-1.0,0.0,0.0) -1.0
surface 5 5 f -1 1 0000 plane (0.0,-1.0,0.0) -1.0
surface 6 6 f -1 1 0000 plane (1.0,0.0,0.0) -1.0
```

The first line after the header indicates that a volume object is being read, so the first step is to create a volume object entity. This will be the owner entity for shells and edges, because that field is not defined in the file records.

The next line defines how many entities of other types are defined in the file. The line:

```
v 8 e 12 f 6 l 6 p 8 c 1 s 6 fg 1 sh 1 fea 0 mc 0 sm 0 cd 0 sg 0 pd 0
```

defines that there will be 8 vertices, 12 edges, 6 faces, 6 loops, 8 points, 1 curve, 6 surfaces, 1 facegroup, 1 shell, no features, no mechanism constraints, no shape modifiers, no constraint data, no supplementary geometry, and no passive data entities. Normally this line would probably consist simply of numbers, but the letters make it easier to understand what the numbers refer to, hopefully.

Eight vertex entities are created, and pointers to these are preserved in an eight-element vertex pointer array. Twelve edges are created, and pointers to these preserved in an edge pointer array. Similarly, six faces, six loops, eight points, one curve, six surfaces, one facegroup, and one shell are created and preserved in appropriately sized pointer arrays. Note that the definitions of curve and surface entities are split into two parts, a constant sized part and a variable sized data part. The constant sized part is created at the beginning; edges and faces point to these entities that, in turn, point to the variable sized data entities that are created when the curve or surface is defined.

The first line following this:

```
volume -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000
```

defines the volume entity fields. The vertex, edge, and shell pointer fields are set to pointer to vertex 1, edge 1, and shell 1, found from the appropriate pointer arrays by default; hence, need not be defined explicitly. The -1 indicates the info field is NIL, the six reals enclosed in [] parentheses are the box minima and maxima, and the final 0000 defines the marker bits of the volume

object.

The next line:

shell 1 1 1 -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000

defines the shell entity. The first number is the shell identity number and indicates that the shell entity pointed to in shell array element 1 is being defined. As with all entities, the real identity number of the shell is immaterial; the identity number refers to an array element rather than a real shell identity number. The second number is the number of the next shell (in this case the shell's own number indicating that it is the only shell in the volume), and the next field of the shell is set to point to the shell in the shell pointer array element 1. The third number, also a 1, is the number of the first facegroup, which is retrieved from array element 1 of the facegroup pointer array. The -1 indicates that the information pointer field is NIL. The next six reals define a box, and the final four 0's are the marker bits of the shell. The owner pointer field is set to point to the volume.

The third line:

facegroup 1 1 1 1 s -1 -1 -1 [-1.0,1.0,-1.0,1.0,-1.0,1.0] 0000

defines the facegroup. As with all entities in the file, the first number is the identity number and indicates that the facegroup referred to in facegroup array element one is to be defined. The second integer indicates the first face in the facegroup, a pointer to which is contained in the face pointer array element 1. The third 1 is a reference to the next facegroup and indicates that the facegroup is the only one in the shell because it is a reference to the facegroup being defined. The fourth 1 indicates the owner of the facegroup. This field can be either a pointer to a facegroup or a pointer to a shell so an extra letter is needed to show that this is a shell pointer. The first -1 corresponds to the surface field of the facegroup, which is NIL because all faces in the facegroup have their own surfaces. The second -1 is for the "cogeom" field of the facegroup, which is set to NIL. The second -1 is corresponds to the "info" field, which is also set to NIL. The six reals define the box and finally, as before, the 0000 is to set the marker bits.

The fourth line:

vertex 1 1 e 1 2 -1 -1 -1 0000

is the first of the eight vertex definition lines. The fields for each are:

identity edge point next friend cogeom info marker-bits

As usual, the first 1 is the vertex identity number. The second 1 is the edge reference for the edge of the vertex, and the "e" following it indicates

that this is an edge reference and not a loop reference (for loops consisting of a single vertex). The third 1 is a reference to the point of the vertex. The 2 is a reference to vertex 2, the next vertex referred to in the all-vertices-in-body chain. The three -1's indicate that the "friend", "cogeom" and "info" are all NIL pointers. The four zeros are the vertex marker bits. The other seven vertices are read similarly.

After the vertex definition lines, come the edge definition lines, starting with the line:

edge 1 3 2 8 10 1 5 1 2 2 12 1 -1 2 -1 [-1.0,1.0,-1.0,-1.0,-1.0,-1.0] 0000

The fields are

identity rcw rcc lcw lcc rloop lloop start end next previous curve friend cogeom space marker-bits

Therefore the identity and the next four integers are edge references used to access the edge pointer array. The next two are loop references found from the loop pointer array. The two after these are vertex references; then come two more edge references and a curve reference. As before, the friend and info fields are set to NIL; the reals are the minima and maxima of the box, and the four zeroes indicate no marker bits. Note that all edges have the same curve reference, so the cogeom field is used to chain the edges sharing the curve together. This is because, in this example, the edge curves use a short form curve that contains no explicit geometric data and relies on the start and end position of the edge.

There are six loop definitions, the first being:

loop 1 1 1 e -1 -1 0 0000

The fields corresponding to these integers for the loop are

identity face edge next info holeloop marker-bits

So the first integer (the identity) is a loop reference and the second is a face reference. The third integer, also a 1, is an edge reference, but it needs the "e" key to allow loops consisting of a single vertex to be represented. The first "-1" is the next loop pointer of this loop and indicates that there is only one loop in the face. The second -1 is the information pointer, also NIL. The zero is not a pointer but a flag to say that this is a perimeter loop. The representation of perimeter- and hole-loops is a matter of modeller convention, and it can vary. The perimeter-loop could be the first loop in the face, but this presupposes only one perimeter-loop which can be insufficient for cylinders and cones with no fake edges. Another option is to make no distinction between perimeter- and hole-loops and use a geometric test should they need

to be distinguished. This representation is based on the datastructure in Appendix A, which distinguishes between perimeter- and hole-loops with a field in the structure. The last four zeroes are, as before, marker bits.

Matching the loop definitions are the face definitions, starting with:

face 1 1 1 1 2 -1 -1 -1 -1 [-1.0,1.0,-1.0,1.0,-1.0,-1.0] 0000

The corresponding field definitions are

identity loop surface owner next feature friend cogeom info space marker-bits

It should be fairly clear, by now, what these mean. The "owner" field refers to a facegroup and the "next" to the next face within that facegroup.

There are eight point definitions:

point 1 1 v -1 1 0000 (-1.0,-1.0,-1.0)

being the first. The fields here correspond to

identity user info use-count marker-bits geometry

The v after the second "1" is to indicate that the "user" reference is to a vertex and not to a supplementary geometry entity. Note that the geometry definition comes at the end of the record. Even though not necessary for a point, this is for compatibility with the other geometric entities, where the variable size geometric element is pushed to the end of the record.

The single curve definition:

curve 1 1 e -1 12 0000 short straight

has the fields

identity user info use-count marker-bits geometry

After the marker bits comes the geometry definition that can be of variable size because of the various types of curve. The curve entity itself is of fixed size, and it is defined by the first five fields up to and including the marker bits. The curve entity contains a pointer to a variable-sized geometric record that is created when the geometric definition is read. In this case, the curve form is essentially empty, because the straight curve data are obtained from the positions of the start and end vertices of the edge, as explained in Appendix A, section A.2.10. This means that the same straight curve can be used by all edges in the cube, hence, the use count of 12.

The surface definitions start with:

surface 1 1 f -1 1 0000 plane (0.0,0.0,-1.0) -1.0

and have the fields

identity user info use-count marker-bits geometry

Each surface is used by one face only, as for the points, so there are six surface definitions. As with the curve, there is a fixed size pointer section followed by a variable-sized geometry section. The type of geometry has to be given, as a keyword or code value, which can be used to select the correct geometry reading function to read the necessary data.

11.5 Supplementary information

Another topic that might need to be written is the derivation of the model. There is a tendency in commercial systems to keep a record of the creation of the model so that operations can be reapplied. This could be included with the object as part of the discfile or separately as part of a database system. The main concern with this is to preserve the references to parts of the objects used as input to modelling operations. This is discussed further in the next chapter.

One method of recording the information about how the model was created is as a list of commands in macro form. For the model parts, though, identification needs to be independent of pointers. This is discussed in section 12.3.

One method is to assign model elements names and use these names in the macro-command text. If this option is chosen, then this implies that it is necessary to have a name-table associated with the file. The problem is name maintenance between the writing and the reading systems. If there is a difference in the operations then there may be name mismatches. Trying to maintain names is the subject of research and is termed “persistent naming”.

Another method is geometry based. If the model elements were found using hit-testing (or ray casting) then, either the graphics orientation when the hit-test was done, or the transformed ray needs to be stored. It is also possible to store the whole model before and after to allow geometric matching of models before and after. This, though, will create very large files.

Recording this information may make the model easier to adapt later, but the information is less certain than the finished shape.

11.6 Communication with other modellers

Communication between modelling systems is important because the different systems have different properties and different applications. It is common that large firms have several different systems that are used for different parts of the production process, and thus, there is a need for inter-system communication. The degree of success of the communication depends on how compatible are the modelling systems. For example, reading boundary representation (B-rep) model files into a B-rep modeller, or reading CSG models into a CSG modeller is much more straightforward than reading a B-rep model into a CSG system. This is because reading a B-rep model into a CSG system implies performing some sort of B-rep to CSG conversion, for which no certain general method exists.

However, apart from these compatibility problems, there is a problem with converting between geometry representations. Different free-form surface representations may have different information and different interpolation methods. Geometric conversion problems are complex and will not be dealt with here.

Another problem that should be emphasised is the problem of information loss when transferring models. For example, suppose that there are two modellers, A and B, between which to transfer data. Modeller A has facegroups and no fake edges, whereas modeller B has no facegroups and likes faces that extend through at most 120 degrees. The change in data is illustrated in figure 11.2. The original model, a cube with a cylindrical extrusion, is shown in 11.2a, with a possible face structure described symbolically underneath. It is assumed, here, that the faces of the cuboid base are contained in one facegroup, whereas the cylindrical and planar faces of the extrusion are contained in a separate facegroup. When this model is transferred to system B the facegroup structure is removed and the faces merged into a single set, producing a structure such as that shown in figure 11.2b. Also the single cylindrical face of the extrusion has to be broken up into three faces, a not necessarily straightforward operation in general. If the model is then passed back to system A again, the resulting structure would be as shown in figure 11.2c, with a single facegroup containing ten faces instead of the original structure of two facegroups containing six and two faces, respectively.

There are two basic ways of communicating between modelling systems:

1. Special formats: Direct translation between modelling system structures.
2. Standard formats: Translation via a 'neutral form' common to many systems.

Special communication formats

Special communications formats are more efficient than standard communications formats but have the disadvantage that they must be written for each pair of modelling systems that need to communicate. There are three

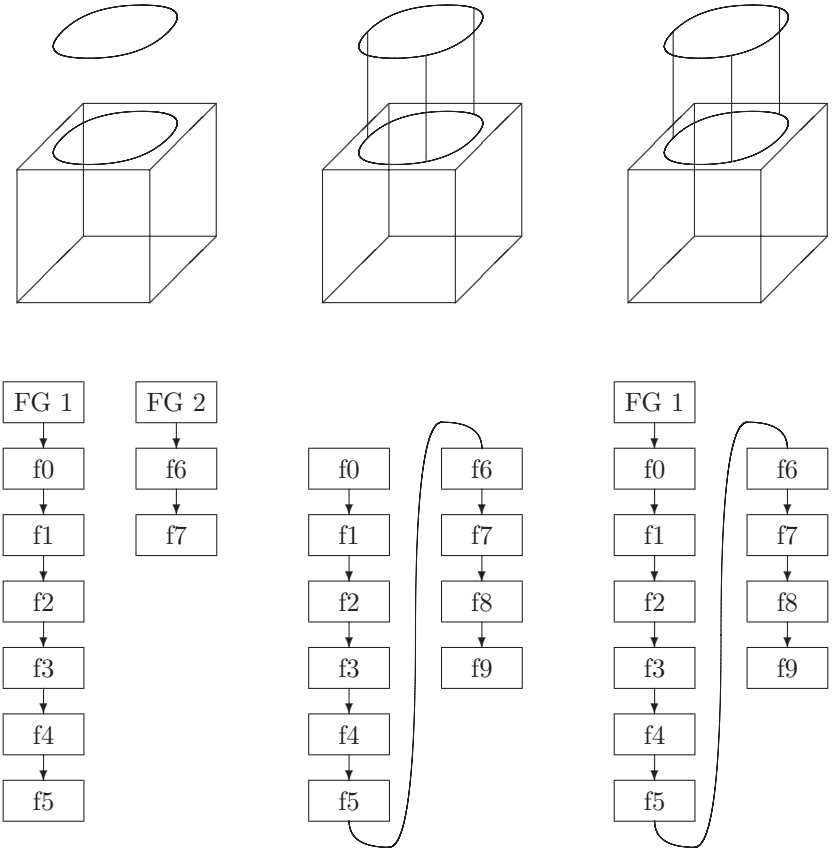


Figure 11.2: Structural changes during communication

basic strategies: 1) for the sending system to write a file in its own disc format and let the receiving system sort things out; 2) for the sending system to write a file in the receiving system's disc format; or 3) to have a separate conversion program to convert the output from the sending system into the input format of the receiving system. It is easier to choose the first alternative and let the receiving system interpret the sending system's structures in the best way possible. The second alternative can be used if, for example, the receiving system is inaccessible, a black box, whereas the sending system code can be changed or extended. If both systems are inaccessible, then the third alternative may be the only choice, but it is the least reliable of the three.

Standard communication formats

Standard communications formats, or neutral formats, started to come at the end of the 1970s and the beginning of the 1980s, and work is still going on. There are several well-known communication formats, e.g., IGES, XBF, CAD*I, and STEP. The basic idea is to create a neutral form. The term 'neutral' means that the form is not native to any particular system. Systems write their models in terms of elements in the neutral form and have to be able to interpret elements in the neutral form in terms of elements in the modelling system.

IGES (Interational Graphics Exchange Standard) appeared at the end of the 1970s. It was originally a standard for communicating drawings rather than solid models. It has steadily been enhanced over the years to include surfaces and curves and even CSG models. One drawback of IGES is that it is very extensive, containing a wide variety of possible information forms, which has meant that, at least in the past, commercial companies have only implemented subsets of IGES, so that communication has had only limited success.

After IGES appeared, the Computer Aided Manufacturing International (CAM-I) organisation (roughly speaking, an industrial/academic federation for promoting research in various areas of production) set about developing a B-rep communication format, the XBF (eXperimental Boundary Format). Although reasonably successful, XBF was used more to demonstrate feasibility. IGES, at that time, was insufficient for solid model communication, and XBF was intended to show how IGES could be extended. This, though, did not happen because the IGES developers did not accept it and work on solid communication went a different way.

CAD*I was an ESPRIT project to establish a common exchange format. There were four parts to the project resulting in recommendations for transfer of 1) wireframes, surfaces, and solids; 2) product analysis data; and 3) drafting data. The results of the project provided input to the STEP proposal for product model transfer.

It may be instructive to look at the CAD*I neutral format, although I do not intend to 'do it justice', I want to give a brief overview. As far as

this chapter is concerned, the most interesting of these is the first of these. The neutral file is described in a research report [115], and interested readers should refer directly to this. One important point to note about the neutral file is that it is designed to be traversable in a single pass. The intention was that the main work would be done when writing the file as it was felt that the file would be written once but may be read many times. This assumption may not, in fact, be altogether valid because although the model may be used several times, it is probably better to read the file once, convert it to the modeller format, and then write it to disc in the modeller's own database format. This avoids repeating the work done in converting the neutral file format into the system's own topological and geometrical entities. Nevertheless, the assumption seems basically valid.

The neutral file consists of a header followed by the "world", the object data in suitable form. The header consists of 14 pieces of information:

1. The file author
2. The company from which the file comes
3. The computer on which the file was produced
4. The operating system
5. File pre-processor
6. Pre-processing date and time
7. Optional disclaimer
8. Geometry level
9. Assembly level
10. Parametric level
11. Referencing level
12. Maximum number of digits per integer
13. Maximum number of significant digits in real mantissa
14. Maximum number of digits in real exponent

This information is aimed at providing basic system level information so that the reading system or user can judge the degree of success of the conversion. Pre-processing refers to the creation of the file by 'pro-processing' a model into neutral file format. The various level codes (8–11) describe the types of model that are contained in the file, which is described in chapter 6 of the report.

The "world" contains the model description that is of more interest here. The world part of the file starts with a world header that contains details about the general geometric state of the model. It contains:

1. A length scale factor so that a unit distance in the model corresponds to the scale factor times one metre.
2. An angle scale factor for converting recorded angles into radians
3. The world size
4. Distance tolerance
5. Angle tolerance
6. Curvature tolerance

Note that, unlike the BUILD system, the numbers of elements in the model are not recorded. The CAD*I neutral format does not allow forward references, which is possible in the BUILD format because all necessary entities are created before they are defined; hence, correct references can be inserted at any time. The lack of forward references means that there are problems in defining, say, loops and edges, An edge refers, directly or indirectly (via edge-loop links), to a loop. A loop also refers to an edge, the first edge in the loop. Similar problems exist with the definitions of edges and vertices that refer to each other. To get around this it is necessary to make some inferences while building the model. In the case of loops and edges, there is no loop reference in the edge, the loop refers to the edges, and the loop-references from the edges are set when the loop is set up. The same is true with vertices and edges.

The geometry definitions: Point, curve, and surface are more or less independent of each other, normally. The order of definition of the B-Rep topological entities in the file is

vertex
edge
loop
face
shell

avoiding forward references and filling in the gaps as outlined above.

Geometrically the standard allows the following curves:

lines
line segments
polygons
circles
ellipses
parabolas
hyperbolas
polycurves

B-spline curves

also trimmed curves, composite curves offset curves and surface curves. The following surfaces are allowed:

spheres
cylinders
cones
toruses
planes
surfaces of revolution
surfaces of translation
B-spline surfaces

Any other geometry forms must be defined in terms of these. For example, general quadric surface forms, defined in matrix form, are not included. Retrieving the data is usually easier than writing it because there is enough information to reconstruct the surface in the receiving system, although with the reservation that free-form surface representations can be difficult to convert. Writing surfaces can be more of a problem than reading them, even for simple surfaces. Take, for example, a general quadric in matrix form, as mentioned. If there is no specific information about what the surface is, then it has to be analysed to rediscover what it is. An elliptic cylinder, for example, then has to be converted into a surface of translation; hence, an elliptic cross section and the translation direction have to be determined from the matrix. Similarly rotational surfaces, except spheres, cylinders, and cones, need a rotation axis and a basic curve. Another alternative is to convert the surfaces into B-spline form and write them to a file as such. However, this means that simple forms degenerate into complex forms and that there is a resulting information loss, at least potentially. The CAD*I standard does include a form number in the B-spline surface representation to allow some information about the surface to be retained in higher level form. This is not a criticism of the standard; it is just a restatement of the problem with geometry.

The standard also allows complex assemblies to be transferred. Up to eight levels of assemblies can also be defined, with instances of solids and assemblies.

Another problem that has to be noted with this kind of transfer is that other kinds of entity exist than simple B-rep models. The standard allows for transfer of wireframe, surface, and solid models, which means that, for completeness, it is necessary to be able to represent wireframe and surface models somehow. Wireframe models are included in the recommended datastructure described in chapter 3. Surfaces can be represented as partial models, say, as described in chapter 5, although extra functions may be needed to integrate them properly. The solid model definition in the definition encompasses B-rep models, CSG models, polyhedral models, primitives, compound B-reps, con-

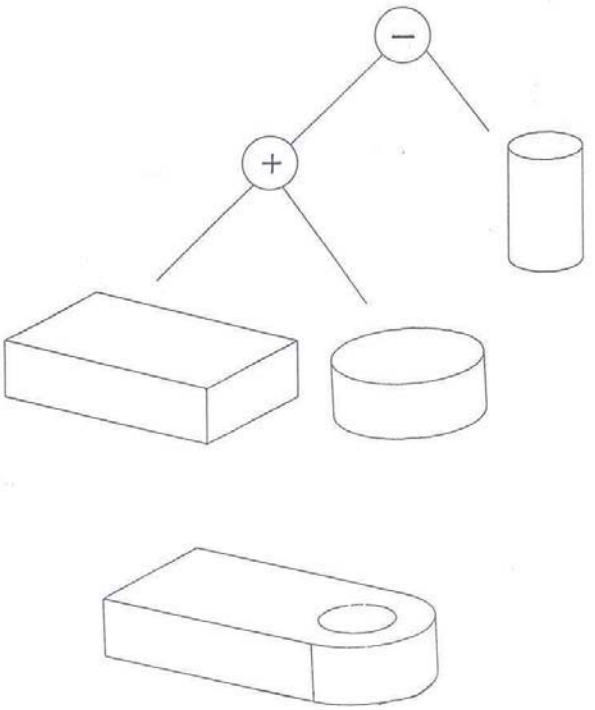


Figure 11.3: CSG model

structs, and hybrid solids. One problem about transporting models between systems with different basic modelling characteristics is again information loss, this time about the original construction of the object. For example, suppose that the object in figure 11.3 was made in a CSG system. The object can be constructed simply in a B-rep system by taking the B-rep primitives and combining them with Boolean operations. If the object is then re-exported to a CSG system, then a lot of work needs to be done to rediscover the CSG representation of the object.

This simple example illustrates the problem of transferring between different types of system. Of course it is possible to record information about the derivation of a model, but if, again for example, the CSG model is modified in the B-rep system, then the derivation information can be invalidated. This kind of transfer problem is not trivial, but it is one that should be handled at a high level; excessive transfer of models should be avoided.

11.7 STEP

STEP is too large and complicated an effort to describe here in full. There are other books about STEP, such as the one by Owen [94]. The purpose here is just to show how the STEP solid model format is related to normal modelling entities.

The following is a STEP model file describing a $10 \times 20 \times 30$ block:

```
ISO-10303-21;
HEADER;
/* Exchange file generated using ST-DEVELOPER v1.5 */
FILE_DESCRIPTION(
/* description */ ('STEPNET test file'),
/* implementation_level */ '2;1');
FILE_NAME(
/* name */ 'block.stp',
/* time_stamp */ '1998-01-28T11:02:16+01:00',
/* author */ ('Name'),
/* organization */ ('STEP Tools'),
/* preprocessor_version */ 'ST-DEVELOPER v1.5',
/* originating_system */ 'ST-ACIS',
/* authorisation */ 'Name');
FILE_SCHEMA (('CONFIG_CONTROL_DESIGN'));
ENDSEC;
DATA;
#10 = PLANE('',#40);
#20 = CARTESIAN_POINT('',(0.,0.,15.));
#30 = DIRECTION('',(0.,0.,1.));
#40 = AXIS2_PLACEMENT_3D('',#20,#30,$);
#50 = PLANE('',#80);
```

```
#60 = CARTESIAN_POINT('',(5.,0.,0.));
#70 = DIRECTION('',(-1.,0.,0.));
#80 = AXIS2_PLACEMENT_3D('',#60,#70,$);
#90 = LINE('',#110,#120);
#100 = DIRECTION('',(0.,1.,0.));
#110 = CARTESIAN_POINT('',(5.,0.,15.));
#120 = VECTOR('',#100,1.);
#130 = PLANE('',#160);
#140 = CARTESIAN_POINT('',(0.,10.,0.));
#150 = DIRECTION('',(0.,-1.,0.));
#160 = AXIS2_PLACEMENT_3D('',#140,#150,$);
#170 = LINE('',#190,#200);
#180 = DIRECTION('',(-1.,0.,0.));
#190 = CARTESIAN_POINT('',(0.,10.,15.));
#200 = VECTOR('',#180,1.);
#210 = PLANE('',#240);
#220 = CARTESIAN_POINT('',(-5.,0.,0.));
#230 = DIRECTION('',(1.,0.,0.));
#240 = AXIS2_PLACEMENT_3D('',#220,#230,$);
#250 = LINE('',#270,#280);
#260 = DIRECTION('',(0.,-1.,0.));
#270 = CARTESIAN_POINT('',(-5.,0.,15.));
#280 = VECTOR('',#260,1.);
#290 = PLANE('',#320);
#300 = CARTESIAN_POINT('',(0.,-10.,0.));
#310 = DIRECTION('',(0.,1.,0.));
#320 = AXIS2_PLACEMENT_3D('',#300,#310,$);
#330 = LINE('',#350,#360);
#340 = DIRECTION('',(1.,0.,0.));
#350 = CARTESIAN_POINT('',(0.,-10.,15.));
#360 = VECTOR('',#340,1.);
#370 = PLANE('',#400);
#380 = CARTESIAN_POINT('',(0.,0.,-15.));
#390 = DIRECTION('',(0.,0.,1.));
#400 = AXIS2_PLACEMENT_3D('',#380,#390,$);
#410 = LINE('',#430,#440);
#420 = DIRECTION('',(0.,-1.,0.));
#430 = CARTESIAN_POINT('',(5.,0.,-15.));
#440 = VECTOR('',#420,1.);
#450 = LINE('',#470,#480);
#460 = DIRECTION('',(-1.,0.,0.));
#470 = CARTESIAN_POINT('',(0.,-10.,-15.));
#480 = VECTOR('',#460,1.);
#490 = LINE('',#510,#520);
#500 = DIRECTION('',(0.,1.,0.));
```

#510 = CARTESIAN_POINT(”,(-5.,0.,-15.));
#520 = VECTOR(”,#500,1.);
#530 = LINE(”,#550,#560);
#540 = DIRECTION(”,(1.,0.,0.));
#550 = CARTESIAN_POINT(”,(0.,10.,-15.));
#560 = VECTOR(”,#540,1.);
#570 = LINE(”,#590,#600);
#580 = DIRECTION(”,(0.,0.,-1.));
#590 = CARTESIAN_POINT(”,(-5.,-10.,0.));
#600 = VECTOR(”,#580,1.);
#610 = LINE(”,#630,#640);
#620 = DIRECTION(”,(0.,0.,-1.));
#630 = CARTESIAN_POINT(”,(5.,-10.,0.));
#640 = VECTOR(”,#620,1.);
#650 = LINE(”,#670,#680);
#660 = DIRECTION(”,(0.,0.,-1.));
#670 = CARTESIAN_POINT(”,(-5.,10.,0.));
#680 = VECTOR(”,#660,1.);
#690 = LINE(”,#710,#720);
#700 = DIRECTION(”,(0.,0.,-1.));
#710 = CARTESIAN_POINT(”,(5.,10.,0.));
#720 = VECTOR(”,#700,1.);
#730 = MANIFOLD_SOLID_BREP(”,#740);
#740 = CLOSED_SHELL(”,(#750,#980,#1210,#1320,#1410,#1500));
#750 = ADVANCED_FACE(”,(#760),#10,.T.);
#760 = FACE_BOUND(”,#770,.T.);
#770 = EDGE_LOOP(”,(#780,#850,#900,#950));
#780 = ORIENTED_EDGE(”,*,*,#790,.T.);
#790 = EDGE_CURVE(”,#810,#830,#800,.T.);
#800 = INTERSECTION_CURVE(”,#90,(#10,#50),.CURVE_3D.);
#810 = VERTEX_POINT(”,#820);
#820 = CARTESIAN_POINT(”,(5.,-10.,15.));
#830 = VERTEX_POINT(”,#840);
#840 = CARTESIAN_POINT(”,(5.,10.,15.));
#850 = ORIENTED_EDGE(”,*,*,#860,.T.);
#860 = EDGE_CURVE(”,#830,#880,#870,.T.);
#870 = INTERSECTION_CURVE(”,#170,(#10,#130),.CURVE_3D.);
#880 = VERTEX_POINT(”,#890);
#890 = CARTESIAN_POINT(”,(-5.,10.,15.));
#900 = ORIENTED_EDGE(”,*,*,#910,.T.);
#910 = EDGE_CURVE(”,#880,#930,#920,.T.);
#920 = INTERSECTION_CURVE(”,#250,(#10,#210),.CURVE_3D.);
#930 = VERTEX_POINT(”,#940);
#940 = CARTESIAN_POINT(”,(-5.,-10.,15.));
#950 = ORIENTED_EDGE(”,*,*,#960,.T.);

```
#960 = EDGE_CURVE('',#930,#810,#970,.T.);
#970 = INTERSECTION_CURVE('',#330,(#10,#290),.CURVE_3D.);
#980 = ADVANCED_FACE('',(#990),#370,.F.);
#990 = FACE_BOUND('',#1000,.T.);
#1000 = EDGE_LOOP('',(#1010,#1080,#1130,#1180));
#1010 = ORIENTED_EDGE('',*,*,#1020,.T.);
#1020 = EDGE_CURVE('',#1040,#1060,#1030,.T.);
#1030 = INTERSECTION_CURVE('',#410,(#370,#50),.CURVE_3D.);
#1040 = VERTEX_POINT('',#1050);
#1050 = CARTESIAN_POINT('',(5.,10.,-15.));
#1060 = VERTEX_POINT('',#1070);
#1070 = CARTESIAN_POINT('',(5.,-10.,-15.));
#1080 = ORIENTED_EDGE('',*,*,#1090,.T.);
#1090 = EDGE_CURVE('',#1060,#1110,#1100,.T.);
#1100 = INTERSECTION_CURVE('',#450,(#370,#290),.CURVE_3D.);
#1110 = VERTEX_POINT('',#1120);
#1120 = CARTESIAN_POINT('',(-5.,-10.,-15.));
#1130 = ORIENTED_EDGE('',*,*,#1140,.T.);
#1140 = EDGE_CURVE('',#1110,#1160,#1150,.T.);
#1150 = INTERSECTION_CURVE('',#490,(#370,#210),.CURVE_3D.);
#1160 = VERTEX_POINT('',#1170);
#1170 = CARTESIAN_POINT('',(-5.,10.,-15.));
#1180 = ORIENTED_EDGE('',*,*,#1190,.T.);
#1190 = EDGE_CURVE('',#1160,#1040,#1200,.T.);
#1200 = INTERSECTION_CURVE('',#530,(#370,#130),.CURVE_3D.);
#1210 = ADVANCED_FACE('',(#1220),#290,.F.);
#1220 = FACE_BOUND('',#1230,.T.);
#1230 = EDGE_LOOP('',(#1240,#1270,#1280,#1310));
#1240 = ORIENTED_EDGE('',*,*,#1250,.T.);
#1250 = EDGE_CURVE('',#930,#1110,#1260,.T.);
#1260 = INTERSECTION_CURVE('',#570,(#290,#210),.CURVE_3D.);
#1270 = ORIENTED_EDGE('',*,*,#1090,.F.);
#1280 = ORIENTED_EDGE('',*,*,#1290,.F.);
#1290 = EDGE_CURVE('',#810,#1060,#1300,.T.);
#1300 = INTERSECTION_CURVE('',#610,(#290,#50),.CURVE_3D.);
#1310 = ORIENTED_EDGE('',*,*,#960,.F.);
#1320 = ADVANCED_FACE('',(#1330),#210,.F.);
#1330 = FACE_BOUND('',#1340,.T.);
#1340 = EDGE_LOOP('',(#1350,#1380,#1390,#1400));
#1350 = ORIENTED_EDGE('',*,*,#1360,.T.);
#1360 = EDGE_CURVE('',#880,#1160,#1370,.T.);
#1370 = INTERSECTION_CURVE('',#650,(#210,#130),.CURVE_3D.);
#1380 = ORIENTED_EDGE('',*,*,#1140,.F.);
#1390 = ORIENTED_EDGE('',*,*,#1250,.F.);
#1400 = ORIENTED_EDGE('',*,*,#910,.F.);
```

#1410 = ADVANCED_FACE('',(#1420),#130,.F.);
#1420 = FACE_BOUND('',#1430,.T.);
#1430 = EDGE_LOOP('',(#1440,#1470,#1480,#1490));
#1440 = ORIENTED_EDGE('',*,*,#1450,.T.);
#1450 = EDGE_CURVE('',#830,#1040,#1460,.T.);
#1460 = INTERSECTION_CURVE('',#690,(#130,#50),.CURVE_3D.);
#1470 = ORIENTED_EDGE('',*,*,#1190,.F.);
#1480 = ORIENTED_EDGE('',*,*,#1360,.F.);
#1490 = ORIENTED_EDGE('',*,*,#860,.F.);
#1500 = ADVANCED_FACE('',(#1510),#50,.F.);
#1510 = FACE_BOUND('',#1520,.T.);
#1520 = EDGE_LOOP('',(#1530,#1540,#1550,#1560));
#1530 = ORIENTED_EDGE('',*,*,#1290,.T.);
#1540 = ORIENTED_EDGE('',*,*,#1020,.F.);
#1550 = ORIENTED_EDGE('',*,*,#1450,.F.);
#1560 = ORIENTED_EDGE('',*,*,#790,.F.);
#1570 = (GEOMETRIC_REPRESENTATION_CONTEXT(3) GLOBAL_ UNCERTAINTY_ASSIGNED_CONTEXT((#1580)) GLOBAL_UNIT_ASSIGNED_CONTEXT((#1590, #1600, #1650)) REPRESENTATION_CONTEXT('ID1', '3-D'));
#1580 = UNCERTAINTY_MEASURE_WITH_UNIT(LENGTH_MEASURE(1.E-006), #1650, 'closure', 'Maximum model space distance between geometric entities at asserted connectivities');
#1590 = (NAMED_UNIT(*) SI_UNIT($,.STERADIAN.) SOLID_ANGLE_UNIT());
#1600 = (CONVERSION_BASED_UNIT('DEGREES',#1610) NAMED_UNIT (#1620) PLANE_ANGLE_UNIT());
#1610 = PLANE_ANGLE_MEASURE_WITH_UNIT (PLANE_ANGLE_MEASURE (0.01745329252), #1630);
#1620 = DIMENSIONAL_EXPONENTS(0.,0.,0.,0.,0.,0.,0.);
#1630 = (NAMED_UNIT(*) PLANE_ANGLE_UNIT() SI_UNIT($,.RADIAN.));
#1640 = ADVANCED_BREP_SHAPE_REPRESENTATION ('',(#730),#1570);
#1650 = (LENGTH_UNIT() NAMED_UNIT(*) SI_UNIT(.MILLI.,.METRE.));
#1660 = SHAPE_DEFINITION_REPRESENTATION(#1670,#1640);
#1670 = PRODUCT_DEFINITION_SHAPE('Version','Test Part',#2220);
#1680 = PRODUCT('1','Product','Test Part',(#1690));
#1690 = MECHANICAL_CONTEXT('3D Mechanical Parts',#1700,'mechanical');
#1700 = APPLICATION_CONTEXT('configuration controlled 3d designs of mechanical parts and assemblies ');
#1710 = APPLICATION_PROTOCOL_DEFINITION ('International Standard', 'config_control_design',1994,#1700);
#1720 = PRODUCT_RELATED_PRODUCT_CATEGORY('detail', $, (#1680));
#1730 = CC_DESIGN_PERSON_AND_ORGANIZATION_ASSIGNMENT(#1740, #1770, (#1680));
#1740 = PERSON_AND_ORGANIZATION(#1750,#1760);

#1750 = PERSON('1','Last Name','First Name', $, $, $);
#1760 = ORGANIZATION($, 'R&D','R&D');
#1770 = PERSON_AND_ORGANIZATION_ROLE('design_owner');
#1780 = PRODUCT_DEFINITION_FORMATION_WITH_SPECIFIED_SOURCE ('Version', 'Test Part', #1680, .MADE.);
#1790 = CC_DESIGN_PERSON_AND_ORGANIZATION_ASSIGNMENT(#1800, #1830, (#1780));
#1800 = PERSON_AND_ORGANIZATION(#1810, #1820);
#1810 = PERSON('2','Last Name','First Name', $, $, $);
#1820 = ORGANIZATION($,'R&D','R&D');
#1830 = PERSON_AND_ORGANIZATION_ROLE('creator');
#1840 = CC_DESIGN_PERSON_AND_ORGANIZATION_ASSIGNMENT(#1800, #1850, (#1780));
#1850 = PERSON_AND_ORGANIZATION_ROLE('design_supplier');
#1860 = CC_DESIGN_APPROVAL(#1870,(#1780));
#1870 = APPROVAL(#1880,'Version approval');
#1880 = APPROVAL_STATUS('not_yet_approved');
#1890 = APPROVAL_DATE_TIME(#1900,#1870);
#1900 = DATE_AND_TIME(#1910,#1920);
#1910 = CALENDAR_DATE(1996,1,1);
#1920 = LOCAL_TIME(0,0,0.,#1930);
#1930 = COORDINATED_UNIVERSAL_TIME_OFFSET(5,0,.BEHIND.);
#1940 = APPROVAL_PERSON_ORGANIZATION(#1960,#1870,#1950);
#1950 = APPROVAL_ROLE('Version approval');
#1960 = PERSON_AND_ORGANIZATION(#1970,#1980);
#1970 = PERSON('3','Last Name','First Name',$,$,$);
#1980 = ORGANIZATION($,'R&D','R&D');
#1990 = CC_DESIGN_SECURITY_CLASSIFICATION(#2000,(#1780));
#2000 = SECURITY_CLASSIFICATION('Version','Security for version',#2010);
#2010 = SECURITY_CLASSIFICATION_LEVEL('unclassified');
#2020 = CC_DESIGN_APPROVAL(#2030,(#2000));
#2030 = APPROVAL(#2040,'Version Security approval');
#2040 = APPROVAL_STATUS('not_yet_approved');
#2050 = APPROVAL_DATE_TIME(#2060,#2030);
#2060 = DATE_AND_TIME(#2070,#2080);
#2070 = CALENDAR_DATE(1996,1,1);
#2080 = LOCAL_TIME(0,0,0.,#1930);
#2090 = APPROVAL_PERSON_ORGANIZATION(#2110, #2030, #2100);
#2100 = APPROVAL_ROLE('Version Security approval');
#2110 = PERSON_AND_ORGANIZATION(#1970,#2120);
#2120 = ORGANIZATION($,'R&D','R&D');
#2130 = CC_DESIGN_PERSON_AND_ORGANIZATION_ASSIGN-MENT(#2140 , #2160, (#2000));
#2140 = PERSON_AND_ORGANIZATION(#1970,#2150);
#2150 = ORGANIZATION($,'R&D','R&D');

```
#2160 = PERSON_AND_ORGANIZATION_ROLE('classification_officer');
#2170 = CC_DESIGN_DATE_AND_TIME_ASSIGNMENT( # 2180, # 2190, ( #
2000 ));
#2180 = DATE_AND_TIME(#2200,#2210);
#2190 = DATE_TIME_ROLE('classification_date');
#2200 = CALENDAR_DATE(1996,1,1);
#2210 = LOCAL_TIME(0,0,0.,#1930);
#2220 = PRODUCT_DEFINITION('Version','Test Part',#1780,#2230);
#2230 = DESIGN_CONTEXT('3D Mechanical Parts',#1700,'design');
#2240 = CC_DESIGN_PERSON_AND_ORGANIZATION_ASSIGNMENT ( #22-
50, #2270, (#2220));
#2250 = PERSON_AND_ORGANIZATION(#1810,#2260);
#2260 = ORGANIZATION($,'R&D','R&D');
#2270 = PERSON_AND_ORGANIZATION_ROLE('creator');
#2280 = CC_DESIGN_APPROVAL(#2290,(#2220));
#2290 = APPROVAL(#2300,'Definition approval');
#2300 = APPROVAL_STATUS('not_yet_approved');
#2310 = APPROVAL_DATE_TIME(#2320,#2290);
#2320 = DATE_AND_TIME(#2330,#2340);
#2330 = CALENDAR_DATE(1996,1,1);
#2340 = LOCAL_TIME(0,0,0.,#1930);
#2350 = APPROVAL_PERSON_ORGANIZATION(#2370,#2290,#2360);
#2360 = APPROVAL_ROLE('Definition approval');
#2370 = PERSON_AND_ORGANIZATION(#1970,#2380);
#2380 = ORGANIZATION($,'R&D','R&D');
#2390 = CC_DESIGN_DATE_AND_TIME_ASSIGNMENT ( # 2400 , #2410, (
#2220 ));
#2400 = DATE_AND_TIME(#2420,#2430);
#2410 = DATE_TIME_ROLE('creation_date');
#2420 = CALENDAR_DATE(1996,1,1);
#2430 = LOCAL_TIME(0,0,0.,#1930);
ENDSEC;
END-ISO-10303-21;
```

The definition of the model elements is contained in an ISO 10303 document, part 42. The elements of this are sufficiently close to the model elements described in this book to make it relatively easy to make the correspondence.

For example, the planar surface definition:

```
#10 = PLANE('',#40);
#20 = CARTESIAN_POINT('',(0.,0.,15.));
#30 = DIRECTION('',(0.,0.,1.));
#40 = AXIS2_PLACEMENT_3D('',#20,#30,$);
```

is a point/normal definition. Instead of being in one line, the geometric in-

formation is spread over four lines.

A line definition is:

```
#90 = LINE(",#110,#120);
#100 = DIRECTION(",(0.,1.,0.));
#110 = CARTESIAN_POINT(",(5.,0.,15.));
#120 = VECTOR(",#100,1.);
```

This, too, is geometrically understandable as a point and direction.

Similarly, the topological information description is relatively straightforward:

```
#730 = MANIFOLD_SOLID_BREP(",#740);
#740 = CLOSED_SHELL(",(#750,#980,#1210,#1320,#1410,#1500));
#750 = ADVANCED_FACE(",(#760),#10,.T.);
#760 = FACE_BOUND(",#770,.T.);
#770 = EDGE_LOOP(",(#780,#850,#900,#950));
#780 = ORIENTED_EDGE(",*,*,#790,.T.);
#790 = EDGE_CURVE(",#810,#830,#800,.T.);
#800 = INTERSECTION_CURVE(",#90,(#10,#50),.CURVE_3D.);
#810 = VERTEX_POINT(",#820);
```

Line #730 corresponds with an object definition in this case, with a pointer to a shell. The CLOSED_SHELL definition corresponds to a shell definition, with a list of faces. An ADVANCED_FACE has a list of loops, called FACE_BOUNDs, a reference to a surface and an orientation flag. The FACE_BOUND refers to a ring of edges (it could be a single vertex) and STEP allows an extra type, a "POLY_LOOP", and an orientation flag to determine whether the edge order is as defined or reversed. The EDGE_LOOP refers to a set of ORIENTED_EDGEs, each of which, in this case, is an EDGE_CURVE, with references to bounding vertices and geometry. For the simple model above, there is a fairly straightforward correspondence between these entities and the datastructure defined in this book.

To translate to and from STEP, it is necessary to take the integrated resources (e.g., Part 42 of ISO 10303) and make a comparison between STEP entities and model entities. Once this has been done, then it is possible to decide what the meaning of the STEP entity is in terms of the target modelling system.

One way of doing this is described in figure 11.4.

As the file is read, the STEP structures are read and created. STEP uses line numbers, or, more correctly, record numbers, as logical pointers in the same way as described for reading objects. As each record is read, the structures referenced are created, if necessary, together with the real model entities.

Consider, again, the following lines:

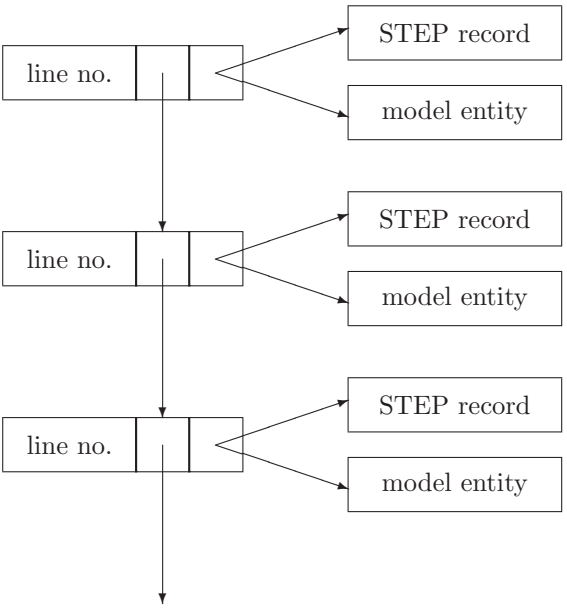


Figure 11.4: Building a STEP structure

#730 = MANIFOLD_SOLID_BREP(",#740);
#740 = CLOSED_SHELL(",(#750,#980,#1210,#1320,#1410,#1500));
#750 = ADVANCED_FACE(",(#760),#10,.T.);
#760 = FACE_BOUND(",#770,.T.);
#770 = EDGE_LOOP(",(#780,#850,#900,#950));
#780 = ORIENTED_EDGE(",*,*,#790,.T.);
#790 = EDGE_CURVE(",#810,#830,#800,.T.);
#800 = INTERSECTION_CURVE(",#90,(#10,#50),.CURVE_3D.);
#810 = VERTEX_POINT(",#820);

The "730" is read, and record 730 is sought in the list of entities. If this has not been created, then the record is created with the data (pointers represented by record numbers and other data) together with a VOLUME_OBJECT entity. The line contains a reference to a CLOSED_SHELL object at line 740. It is necessary to use this to set up the VOLUME_OBJECT pointers. Assume that the record has not been created; a numbered record is created with a NULL pointer to the STEP record and a pointer to a new SHELL entity. The back pointer of the SHELL object to the VOLUME_OBJECT is also set. When the next line is read, the numbered record is found, but the STEP record is NULL, of course, so this is created. The CLOSED_SHELL

record has a list of faces, which differs from the structure of the SHELL because a SHELL has a pointer to a single FACE and the FACEs are chained together. This means that, for each reference to a ADVANCED_FACE in the STEP record, a FACE has to be created within the context of the SHELL so as to create and link the model structure. Line 750 is a FACE definition with a reference to a loop and to a surface. The loop definition is spread over lines 760 and 770. Line 780 corresponds to a loop-edge link, whereas line 790 is an edge definition.

There are similar considerations when reading an edge to those when reading a shell because it may be necessary to create the new edge and link it into a chain of edges in a body. In this case, it is necessary to trace the back pointers up through the loop-edge link to the loop, the face, the shell, and finally the body. Problems occur, of course, if the body, or part of the chained structure, has not been defined. In this case, it is necessary to leave the back pointers unset and to add them as the higher level elements appear.

The curve definition is interesting from the point of view of stability. Because modellers may work with different tolerance values, it may be that a intersection curve calculated at one tolerance level in the sending modeller is not correct in the receiving modeller, which has a smaller tolerance. Association of surface definitions with a curve definition allows the receiving modeller to recalculate the intersection, if necessary.

After reading such a model, it is possible to check it for consistency. See section 14.1.

Finally, here are some general comments.

There are two major reasons for using standard data exchange formats: 1) exchanging models between different modelling systems and 2) archiving models. At present, data exchange is perhaps the major reason, but archiving is also an important topic when the life of products extends over relatively long periods of time. For example, the lifetime of an aircraft design may be 50 years or longer. Maintaining a product database for this length of time needs stability. A company may well change its CAD system several times during the product lifetime.

It may be tempting to want to add to or change standards to incorporate more advanced elements. This is not desirable. Creating dialects leads to an erosion of these standards. If changes or additions are really necessary, then it is better to use the official channels to ask for an update. These are not always quick, but they are more stable.

The correspondence between STEP elements and model elements may not always be evident. Translating standard elements into local model elements may involve a loss or change of information, as already described. It may be desirable to preserve the input information as a note or in some other manner.

11.8 Printing models

A related task is to produce printed forms of models. This is not really directly for system users, but it is an aid for system developers in debugging. The task is the same, really, that is to produce a representation of the entities in the datastructure, although in a more human friendly form than that needed for database input/output. The printed form is especially useful for cases where the object may be in a strange state as a result of an erroneous or an incomplete operation, or where there are special conditions such as non-manifold or coincident topology that cannot be distinguished with a simple graphical tool. When I need to understand what is happening in an operation, I often draw a sketch of an object, identifying relevant elements and connectivity from the printed form of the object, as an aid to visualisation. It is also possible to sketch in elements such as loop-edge links that would normally not be drawn.

Another small difference between the printed version and the database version of a model is the way in which the data are output. In the database version, the data can be output in a continuous stream. With the printed version, it is better to use an output buffer, keeping lines to a maximum length of 80, say.

The suggested order of printing, based on the order used in BUILD is

- OBJECT
- SHELL-FACEGROUP-FACE structure
- EDGES
- VERTICES
- SURFACES
- CURVES
- POINTS

It should be stressed, again here, the usefulness of having a unique identity number for each entity. It is more difficult to use pointers because they are longer than identifiers and non-consecutive. Having the numbers as part of the basic structure also means that these can be used in debugging, printing out the identity number as part of a trace. If the numbers are 'faked' by using, for example, arrays of entities and using the array element as the logical identity, then it is not usually possible to duplicate this numbering system in isolated routines; hence, only the pointer values can be used, which can be tedious, as mentioned.

The amount of information printed is controlled, in BUILD, by using a set of so-called "print styles" analogous to the drawing styles described in the previous chapter. These allow dynamic control of the amount of information printed. Possible styles include

- Faces: To enable/inhibit printing of the shell/facegroup/face structure of the object.
- Edges: To enable/inhibit printing of edge records.
- Vertices: To enable/inhibit printing of vertex records.
- Topology: To enable/inhibit printing of the topological records, i.e., equivalent to faces, edges, and vertices.
- Features: To enable/inhibit printing of feature records.
- Surfaces: To enable/inhibit printing of surface records. Note the additional "freeform" style for controlling printing of free-form records that can be large.
- Curves: To enable/inhibit printing of curve records. As with surfaces, the freeform style can be used to control printing of free-form curves.
- Points: To enable/inhibit printing of point records.
- Geometry: To enable/inhibit printing of geometry, i.e., surfaces, curves, and points.
- Freeform: To control printing of free-form records.

The print-out of a cube may appear as:
VOLUME OBJECT 1, unnamed, used 0 times, shell: 1, edge 1, vertex 1,
feature: NIL, info: NIL, marker bits (),
box [-1.0:1.0,-1.0:1.0,-1.0:1.0]

SHELL 1, next: NIL, facegroup: 1, object 1, info: NIL, marker bits (),
box [-1.0:1.0,-1.0:1.0,-1.0:1.0]

FACEGROUP 1, Facegroup: NIL, next: NIL, Face: 1, owner shell 1, surface: NIL,
cogeom: NIL, info: NIL, marker bits (),
box [-1.0:1.0,-1.0:1.0,-1.0:1.0]

FACE 1, surface 1, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 1 perimeter, info: NIL, edges: 1 3 4r 2r

FACE 2, surface 2, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 2 perimeter, info: NIL, edges: 7r 12r 11r 9r

FACE 3, surface 3, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 3 perimeter, info: NIL, edges: 7 6r 4 5

FACE 4, surface 4, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 4 perimeter, info: NIL, edges: 9 8r 2 6

FACE 5, surface 5, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 5 perimeter, info: NIL, edges: 11 10r 1r 8

FACE 6, surface 6, facegroup 1, cogeom: NIL, info: NIL, marker bits () box [-1.0:1.0,-1.0:1.0,-1.0:1.0]
LOOP 6 perimeter, info: NIL, edges: 12 5r 3r 10

EDGE 1 Start vertex 1, end vertex 2, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [-1.0,1.0,-1.0,-1.0,-1.0,-1.0]
LOOP-EDGE LINK (right) loop 1, prev: e2, next: e3
LOOP-EDGE LINK (left) loop 5, prev: e10, next: e8
EDGE 2 Start vertex 1, end vertex 3, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [-1.0,-1.0,-1.0,1.0,-1.0,-1.0]
LOOP-EDGE LINK (right) loop 4, prev: e8, next: e6
LOOP-EDGE LINK (left) loop 1, prev: e4, next: e1
EDGE 3 Start vertex 1, end vertex 4, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [1.0,1.0,-1.0,1.0,-1.0,-1.0]
LOOP-EDGE LINK (right) loop 1, prev: e1, next: e4
LOOP-EDGE LINK (left) loop 6, prev: e5, next: e10
EDGE 4 Start vertex 3, end vertex 4, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [-1.0,1.0,1.0,1.0,-1.0,-1.0]
LOOP-EDGE LINK (right) loop 3, prev: e5, next: e6
LOOP-EDGE LINK (left) loop 1, prev: e3, next: e2
EDGE 5 Start vertex 4, end vertex 5, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [1.0,1.0,1.0,1.0,-1.0,1.0]
LOOP-EDGE LINK (right) loop 3, prev: e4, next: e7
LOOP-EDGE LINK (left) loop 6, prev: e12, next: e3
EDGE 6 Start vertex 3, end vertex 6, curve 1, friend: NIL, cogeom: NIL, info: NIL, marker bits (), box [-1.0,-1.0,1.0,1.0,-1.0,1.0]
LOOP-EDGE LINK (right) loop 4, prev: e2, next: e3
LOOP-EDGE LINK (left) loop 3, prev: e10, next: e8
EDGE 7 Start vertex 5, end vertex 6, curve 1, friend: NIL, cogeom: NIL,

info: NIL, marker bits (), box [-1.0,1.0,1.0,1.0,1.0,1.0]
LOOP-EDGE LINK (right) loop 3, prev: e5, next: e6
LOOP-EDGE LINK (left) loop 2, prev: e9, next: e12
EDGE 8 Start vertex 1, end vertex 7, curve 1, friend: NIL, cogeom: NIL,
info: NIL, marker bits (), box [-1.0,-1.0,-1.0,-1.0,-1.0,1.0]
LOOP-EDGE LINK (right) loop 5, prev: e1, next: e11
LOOP-EDGE LINK (left) loop 4, prev: e2, next: e9
EDGE 9 Start vertex 6, end vertex 7, curve 1, friend: NIL, cogeom: NIL,
info: NIL, marker bits (), box [-1.0,-1.0,-1.0,1.0,1.0,1.0]
LOOP-EDGE LINK (right) loop 4, prev: e6, next: e8
LOOP-EDGE LINK (left) loop 2, prev: e11, next: e7
EDGE 10 Start vertex 2, end vertex 8, curve 1, friend: NIL, cogeom: NIL,
info: NIL, marker bits (), box [1.0,1.0,-1.0,-1.0,-1.0,1.0]
LOOP-EDGE LINK (right) loop 6, prev: e3, next: e12
LOOP-EDGE LINK (left) loop 5, prev: e11, next: e1
EDGE 11 Start vertex 7, end vertex 8, curve 1, friend: NIL, cogeom: NIL,
info: NIL, marker bits (), box [-1.0,1.0,-1.0,-1.0,1.0,1.0]
LOOP-EDGE LINK (right) loop 5, prev: e8, next: e10
LOOP-EDGE LINK (left) loop 2, prev: e12, next: e9
EDGE 12 Start vertex 8, end vertex 5, curve 1, friend: NIL, cogeom: NIL,
info: NIL, marker bits (), box [1.0,1.0,-1.0,1.0,1.0,1.0]
LOOP-EDGE LINK (right) loop 6, prev: e10, next: e5
LOOP-EDGE LINK (left) loop 2, prev: e7, next: e11

VERTEX 1 friend: NIL, cogeom: NIL, info: NIL, point 1 (-1.0,-1.0,-1.0)
e1 l1 e2 l4 e8 l5
VERTEX 2 friend: NIL, cogeom: NIL, info: NIL, point 2 (1.0,-1.0,-1.0)
e1 l5 e10 l6 e3 l1
VERTEX 3 friend: NIL, cogeom: NIL, info: NIL, point 3 (-1.0,1.0,-1.0)
e2 l1 e4 l3 e6 l4
VERTEX 4 friend: NIL, cogeom: NIL, info: NIL, point 4 (1.0,1.0,-1.0)
e3 l6 e5 l3 e4 l1
VERTEX 5 friend: NIL, cogeom: NIL, info: NIL, point 5 (1.0,1.0,1.0)
e5 l6 e12 l2 e7 l3
VERTEX 6 friend: NIL, cogeom: NIL, info: NIL, point 6 (-1.0,1.0,1.0)
e6 l3 e7 l2 e9 l4
VERTEX 7 friend: NIL, cogeom: NIL, info: NIL, point 7 (-1.0,-1.0,1.0)
e8 l4 e9 l2 e11 l5
VERTEX 8 friend: NIL, cogeom: NIL, info: NIL, point 8 (1.0,-1.0,1.0)
e10 l5 e11 l2 e12 l6

NO IDENTIFIED FEATURES

SURFACE 1 info: NIL, plane (0.0,0.0,-1.0) -1.0
SURFACE 2 info: NIL, plane (0.0,0.0,1.0) -1.0
SURFACE 3 info: NIL, plane (0.0,1.0,0.0) -1.0
SURFACE 4 info: NIL, plane (-1.0,0.0,0.0) -1.0
SURFACE 5 info: NIL, plane (0.0,-1.0,0.0) -1.0
SURFACE 6 info: NIL, plane (1.0,0.0,0.0) -1.0

CURVE 1 info: NIL, straight, short form

POINT 1 info: NIL, (-1.0,-1.0,-1.0)
POINT 2 info: NIL, (1.0,-1.0,-1.0)
POINT 3 info: NIL, (-1.0,1.0,-1.0)
POINT 4 info: NIL, (1.0,1.0,-1.0)
POINT 5 info: NIL, (1.0,1.0,1.0)
POINT 6 info: NIL, (-1.0,1.0,1.0)
POINT 7 info: NIL, (-1.0,-1.0,1.0)
POINT 8 info: NIL, (1.0,-1.0,1.0)

In the above-printed version of a cube, it is assumed that the model is represented using loop-edge links. With winged-edge pointers, an edge record might appear as:

EDGE 1 Start vertex 1, end vertex 2, curve 1, rloop: 1, lloop: 5,
rcw:3 rcc:2 lcw:8 lcc:10, friend: NIL, cogeom: NIL, info: NIL,
marker bits (), box [-1.0,1.0,-1.0,-1.0,-1.0,-1.0]

It is convenient to have two types of printing routine for each entity type: a brief form and a long form. The brief form simply prints the number of the entity or NIL for a NIL entity, to avoid extensive checking in the long form printing routines. The long form routine prints a representation of the fields of the entity record. For example, for an edge, the brief printing routine for an edge would be

```
print_brief_edge(EDGE e, STRING s) VOID:
IF ( e IS NIL ) print_string(s+"NIL")
ELSE print_int(s, number OF e)
FI;
```

and the long form printing routine might be:

```
print_long_edge(EDGE e) VOID:
IF ( e IS NIL )
THEN print_string("Null edge record"); flush_buffer();
```

```
ELSE
print_int("EDGE ", number of e);
print_brief_vertex(start OF e, " Start vertex ");
print_brief_vertex(end OF e, ", end vertex ");
print_brief_curve(curve OF e, ", curve ");
print_brief_edge(friend OF e, ", friend: ");
print_string(", cogeom: ");
CASE cogeom OF e
IN (EDGE xe): print_brief_edge(xe, ""),
(SUPPLEMENTARY_GEOMETRY sg): print_brief_sg(sg, "")
OUT print_string(" unknown type")
ESAC;
print_info(info OF e);
print_marker(marker OF e);
print_space(space OF e);
flush_buffer();
print_long_el_link(rlink OF e); flush_buffer();
print_long_el_link(llink OF e); flush_buffer()
FI;
```

The exact form of these is, of course, arbitrary. The important thing is to print out the relevant information from each entity. Other entity printing routines are similar.

The facegroup structure can be printed using recursion, i.e.,

```
print_long_facegroup(FACEGROUP fg) VOID:
IF ( fg IS NIL )
THEN print_string("Null facegroup record"); flush_buffer()
ELSE
print_int("FACEGROUP ", number OF fg);
print_brief_facegroup(facegroup OF fg, ", Facegroup: ");
print_brief_facegroup(next OF fg, ", next: ");
print_brief_facegroup(face OF fg, ", Face: ");
print_string(", owner ");
CASE owner OF fg
IN (SHELL s): print_brief_shell(s, "shell: "),
(FACEGROUP xfg): print_brief_facegroup(xfg,"facegroup: ")
OUT print_string("unknown type")
ESAC;
print_string(", cogeom: ");
CASE cogeom OF fg
IN (FACE xf): print_brief_face(xf, ""),
(FACEGROUP xfg): print_brief_facegroup(xfg, ""),
(SUPPLEMENTARY_GEOMETRY sg): print_brief_sg(sg, "")
OUT print_string(", unknown type")
```

```
ESAC;
print_info(info OF fg);
print_marker(marker OF fg);
print_space(space OF fg);
for_all_facegroups_in_facegroup(fg, print_long_facegroup);
for_all_faces_in_facegroup(fg, print_long_face);
FI;
```

For an object the entity data and then all entities are printed; thus:

```
print_long_volume(VOLUME_OBJECT vo) VOID:
IF ( vo IS NIL )
THEN print_string("Null volume record"); flush_buffer()
ELSE print_int("VOLUME OBJECT ", number OF vo );
IF ( name OF vo IS NIL )
print_string(", unnamed");
print_int(", used ", used OF vo); print_string(" times,");
print_brief_shell(shell OF vo, " shell: ");
print_brief_edge(edge OF vo, ", edge: ");
print_brief_vertex(vertex OF vo, ", vertex: ");
print_brief_feature(feature OF vo, ", feature: ");
print_info(info OF vo);
print_marker(marker OF vo);
print_space(space OF vo);

    IF style(faces)
THEN for all shells in object(vo, print_long_shell)
FI;
IF style(edges)
THEN for all edges in object(vo, print_long_edge)
FI;
IF style(vertices)
THEN for all vertices in object(vo, print_long_vertex)
FI;
IF style(features)
THEN for all features in object(vo, print_long_feature)
FI;
IF style(surfaces)
THEN for all surfaces in object(vo, print_long_surface)
FI;
IF style(curves)
THEN for all curves in object(vo, print_long_curve)
FI;
IF style(points)
THEN for all points in object(vo, print_long_point)
```

FI;
FI;

This is a rough outline of how the printing routines work. It would be tedious to write down pseudo-code for all of these here, but the ones above should provide a sufficient idea of what the rest should look like.

Chapter 12

Modeller access

The main purpose of this chapter is to describe how to get at the modeller. There are three ways described here: command interpreters, macro-languages, and programming interfaces. Command interpreters are the most complex, and most of this chapter is devoted to them. Macro-languages are based, to some extent, on these in that they provide a symbolic programming interface. Programming interfaces, known usually as Application Programming Interfaces, or APIs, turn the modeller into a software package around which it is possible to create applications.

The function of a command interpreter is fairly obvious; it provides a 'symbolic interface' to the modeller. As such, it is not necessary to limit the description to the classic notion of a command interpreter. Work on a LISP interface to a modeller was done during the 1980s in Sweden by Kjellberg and others; there are commercial LISP-like interfaces to other modellers as well. In the same way as the classic command interpreter, LISP provides a symbolic interface through a standard language. Some of the description below is also relevant to such language interfaces, the use of a stack to preserve results, for example. Syntax analysis, perhaps, less so, but similar parameter checking can be advantageous even for a language-based system. Menu systems have a similar function to command interpreters. The complexities of "Computer-Human Interaction" will not be discussed here, though.

Interaction with the modeller occurs on at least two levels: the user level and the system developer level. User level interaction is also outside the scope of this book, as it depends on many other factors; this chapter is aimed at code developers who want to test modelling code rather than designers who will use it in practice. It is possible to build menu systems on top of command interpreters to provide a friendlier interface to the system. Building a user interface this way means that the command language is readily available as a 'history' of what the user has done in a session as well as providing a macro-language capability. With the increase in power and functionality of computers, more possibilities are available than in the early days when

interaction was limited to textual commands. It is possible to say that in general, some kind of interaction is necessary with the system, though, to provide a flexible basis for testing modeller code. This chapter describes some possibilities to be considered when developing modeller command interpreters; how they are then tailored is, of course, up to particular code developers.

In general the command interpreter should provide access to as wide a range of the modelling functions as possible. Complete user functions provide a way for creating objects for testing, but it is also useful to be able to test partial functions under controlled conditions for debugging purposes. The command interpreter is also a means for imposing a 'philosophy' for using the system. A general modelling kernel may contain facilities that are not required, or there may be alternative ways to perform similar tasks. For any particular application, the modeller needs to be examined and the necessary philosophy decided upon.

Another consideration is that the commands should be structured in some way. Providing all commands on a single level makes it harder to see what is available. Suitably grouped commands can provide a more comprehensible overview. In general the modelling functions can be roughly grouped into the categories: system commands; basic modelling functions; and applications.

Finally, the chapter describes multi-user command interpreters as an alternative to the 'usual' single-user architecture.

This description in this chapter is based largely on Graham Jared's work on the BUILD command interpreter, reported briefly by Braid [11], and on experience with similar test systems.

12.1 Command interpreter architecture

Graham Jared's original command interpreter for BUILD used a stack as a repository for objects and command parameters. The resulting command interpreter has been described as being like a 'solid modelling calculator'. Jared also had a separate syntax analyser that checked the command line and picked off the information and put it on the stack. These two ideas proved very useful, and the description here is based on these.

12.1.1 Working stacks

The importance of the stack is that it is a unified mechanism, similar in principle to that used for parameter passing in some languages. This means that the code for the various commands can have a general form, picking the command data from the stack so that there is no need to have to cope with a variety of parameters to different command functions.

As a repository the stack has to contain all possible model entities as well as the standard standard types such as reals, integers, and strings. When a command was encountered, the syntax analyser (see section 12.2) was called to

check the command line. The syntax analyser picked the elements, e.g. numbers, vectors, strings, and model elements, from the command line and placed them on the stack from where they were retrieved by the command code. The results of the operations were also placed on the stack from where they could be accessed by other commands, such as drawing or printing. Having a stack also means that the user does not have to define names, for example, for everything that needs to be referred to, although a naming mechanism is a useful addition for being able to access special results, so it should be provided as well.

Jared used a single stack in his interpreter, but it can be useful to have auxilliary stacks to contain partial results and for moving elements around. Another command interpreter used three stacks, the main one for modelling results and the other two under user control.

12.1.2 Command structures (sub-interpreters)

As stated, modelling commands can be roughly divided into the categories:

1. System commands
2. Kernel modelling commands
3. Applications and logical command units

The System commands category contains general utility commands such as PRINT, DRAW, debugging, and status commands. These commands are useful at all levels of the command interpreter.

Kernel modelling commands are, for example, Boolean operations, sweeping, local operations that are used to build up basic models. These commmands are useful for setting up a model for testing code and for testing applications.

Applications and logical command units are easily identifiable groups of commands that can be broken out from the rest of the commands. In general, the applications commands provide the final stages in testing.

A simple example of a command set is:

System commands
DEBUG DRAW EXECUTE LOG PRINT Stack-transfer

Kernel modelling commands
CONE CUBE CYLINDER PRISM PYRAMID TORUS
SWEEP SWING ADD INTERSECT SECTION SUBTRACT BEND BLEND CHAMFER
IMPRINT JOIN_FACES REFLECT FIX_TRANSFORM TRANSFORM UNTRANSFORM

COPY (DUPLICATE) DELETE

Application commands

2.5D:	CIRCLE END EXTEND LINE START
MODIF:	LIFTF MAKE_EDGE MOVEV MOVEE SMOOTH SPINF SPLIT_EDGE
NC:	GENERATE_PATHS SELECT_TOOL
SHEET:	DETACH_FACE JOIN_EDGES MAKE_SKIN THICKEN FLATTEN

System commands

The DEBUG system command is responsible for turning on or off debugging. This is almost certainly necessary for development work, but methods vary. BUILD used a bit system with which there were several debug words, the bits of which could be set or unset. Different parts of the code could then test particular bits to tell whether to output debugging information. The commercial system ACIS has module names and debug levels to control the debug output.

The PRINT and DRAW commands are self-explanatory. The stack transfer commands are for moving entities between the command interpreter stacks.

The EXECUTE command is for executing a command file and the log file is for turning on or off command sequence logging.

Kernel modelling commands

The function of the kernelling modelling commands is more or less obvious from the names. There should be functions to create a set of primitives, to extrude shapes in straight lines and circles, and to combine models with Boolean operations. Included in the list are some local operations, as examples. Note, though, the transformation commands and the copying commands. One philosophy is to create instances of objects, instead of producing new topological structures, and to associate transformations with models to describe general changes (SCALE, ROTATE, TRANSLATE) instead of actually modifying the geometry. This is quicker than performing real copy and duplication operations, but it is useful to have commands to really copy an object and to apply the transformation matrix to the geometry of the object.

The use of 'instances' of objects rather than the objects themselves is worth clarifying a little. As described, in chapter 3, an instance is, essentially, a reference to a transformation matrix and a reference to a model. This is a useful mechanism for creating assembly models with multiple examples of the same part. For example, an assembly that contains ten screws need have only one screw model and ten instances of that model with relevant

transformations to rotate and translate the different instances to the correct positions (see figure 12.1). This is fairly straightforward, but it is important to consider how these instances are managed, as described in section 3.4.2. Suppose, for example, that one of the ten screws in the above-mentioned assembly has to be shortened. If the original model is shortened then all ten instances will be shortened; hence, it is necessary to make a real copy of the screw and to shorten the copy. This has to be handled automatically in a command interpreter and is part of the general philosophy of using the model. In general, modelling operations on a body, such as the Boolean operations, can be given instances instead of actual object models as parameters. If the model is multipli-instanced (there should be a count in the datastructure to determine such multiplicity), then the referenced object model should be copied and the copy instanced instead of the original model. With local operations, the picture is not so clear. Usually the operation parameters consist of model parts, typically vertices, edges, and faces, and it is unclear how to manage such operations. It is possible to find the model containing these elements and, if that model is multipli-instanced, to make a copy. It is then necessary to find the elements corresponding to the original target elements in the copy and give these as parameters. It is not clear, though, whether it is worth doing.

Another point to note, concerning transformations, is that the order in which the transformations are applied produces different results. Transformation matrices responsible for different effects (rotation, scaling, translation) can be concatenated to produce a single transformation matrix. New transformations can be concatenated by pre-multiplying or post-multiplying. Pre-multiplation is effectively the same as post-multiplation of the same transformations but in the reverse order. Pre-multiplation applies the transformation in the model coordinate system, whereas post-multiplication applies the transformation in the world coordinate system. Pre-multiplication is useful, for example, in scaling or rotating a model, which has been already positioned with other transformations.

Missing from the list of kernel modelling functions are interrogation functions and partial functions. Point-in-body and point-in-face tests are useful, and certainly important; hence, it can be useful to include these so that they can be tested. Partial functions, such as producing the intersection seams in Booleans, may also be useful. The inclusion of such functions is useful for the system developer for testing, perhaps less so for the end-user of the system.

Application commands

The particular applications available in the modeller depend, of course, on the interest of the system developer. A few examples, given above, are 2.5D, MODIF, NC, and SHEET.

The 2.5D command set is for creating two-dimensional shapes that can be extruded into solids, or for inscribing shapes on faces. The BUILD shape

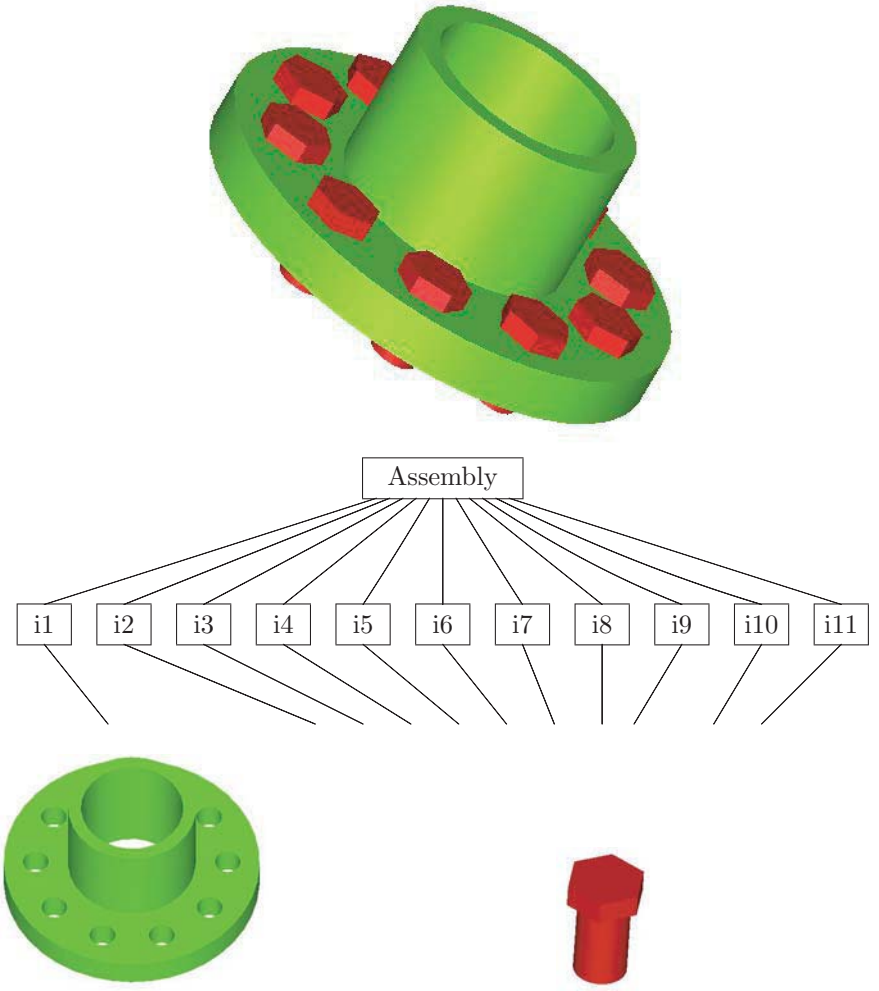


Figure 12.1: Simple instance structure

creation set included facilities for creating a geometric framework (help geometry) consisting of a set of points, LINEs and CIRCLEs that could then be used define a two-dimensional lamina body. The body definition sequence consisted of a START command, a sequence of EXTEND commands, and an END command. Each point specified in the sequence could be given either in coordinate form, as a previously created point, or as an implicit point defined from other help geometry. The geometry of the new edge could be obtained from the help geometry or, by default, was a straight line.

MODIF was a Japanese system developed by Kimura and Chiyokura [18] used for creating polygonal models that were then smoothed to produce the final shape. Taking an input, polygonal shape, commands to define and manipulate the polyhedron might include LIFTF (lift, or sweep a face), MAKE_EDGE (make and edge between two vertices), MOVEV (move a vertex), MOVEE (move an edge), SPINF (spin a face), and SPLIT_EDGE (split an edge by inserting a vertex). The SMOOTH command produces the final shape.

Numerical Control, NC, packages were developed early on for automating the geometric part of machining. The purpose of the example commands above, GENERATE_PATHS SELECT_TOOL is self-explanatory.

The SHEET command set is an example of special-purpose modelling commands for building sheet models. Sheet models can be made by the 2.5D command set and manipulated in some ways by general commands such as REFLECT. Having a separate command set means that special conditions can, to some extent, be controlled. Sheet objects and sheet modelling require special representations, as described in chapter 5, hence, the reason for separating them out. A user might define a volumetric model, convert it to a special sheet model with the MAKE_SKIN command, DETACH_FACEs, JOIN_EDGES, and finally THICKEN the sheet model to produce a volumetric model. The FLATTEN command goes the other way and takes a volumetric model that is then converted to a flat sheet model.

Some examples of using the command set are given below. The command syntax and philosophy are based on the BUILD command interpreter. Commands are given in capital letters with comments undeneath.

1. CUBE
   Creates an instance of a cube from (-1,-1,-1) to (1,1,1) and places it on the stack.

2. COPY
   Creates second instance of the top object on the stack, the cube, and places it on the stack.

3. MODIFY BY MX 10 SZ 4
   Changes the transformation matrix of the second instance by a translation of ten units in the X direction and a scaling by 4 in

the Z direction.

4. CYLINDER
   Creates a cylinder, radius 1, height 2, centred at the origin, and axis parallel to the Z axis.

5. MODIFY BY SZ 4 SX 0.5 SY 0.5 RY 90 MX 5
   Modifies the transformation of the cylinder (the top of the stack) with a scaling by 4 in the Z direction and scaling by half in the X and Y directions, a rotation about the Y axis by 90 degrees, and a translation of 5 units in the X direction. The cylinder is now, effectively, a cylinder, radius 0.5, axis parallel to the X axis, running from (1,0,0) to (9,0,0).

6. ADD
   Calls the Boolean add (or unite) function to create a solid from the cylinder and block. However, the block is instanced twice, so before it can be added to the cylinder, it must be copied and the second instance is made to point to the copy. The transformation matrices of both the cylinder and the block can then be applied to both bodies before performing the Boolean operation. The two objects are removed from the stack before the operation and the result placed back onto the stack.

7. ADD
   Creates the final object combining the compound block/cylinder just created with the original block.

The following sequence creates the object shown in figure 12.2.

```
2.5D
CIRCLE 1 CENTRE (28.9,6.4) RADIUS 6.4
CIRCLE 2 CENTRE (18.3,6.4) RADIUS 2.9
LINE 1 START (34.6,0) DIR (0,1)
START (0,0)
EXTEND TO (28.9,0)
EXTEND ROUND C1 CCW TO LXC L1 C1 NEAR (34.6,3.5)
EXTEND TO LXC L1 C1 NEAR (34.6,9.3)
EXTEND ROUND C1 CCW TO (28.9,12.8)
EXTEND TO (21.4,12.8)
EXTEND TO (21.4,9.3)
EXTEND TO (18.3,9.3)
EXTEND ROUND C2 CCW TO (15.4,6.4)
EXTEND TO (15.4,4)
EXTEND TO (9.4,4)
EXTEND TO (9.4,3)
EXTEND TO (6.4,3)
```

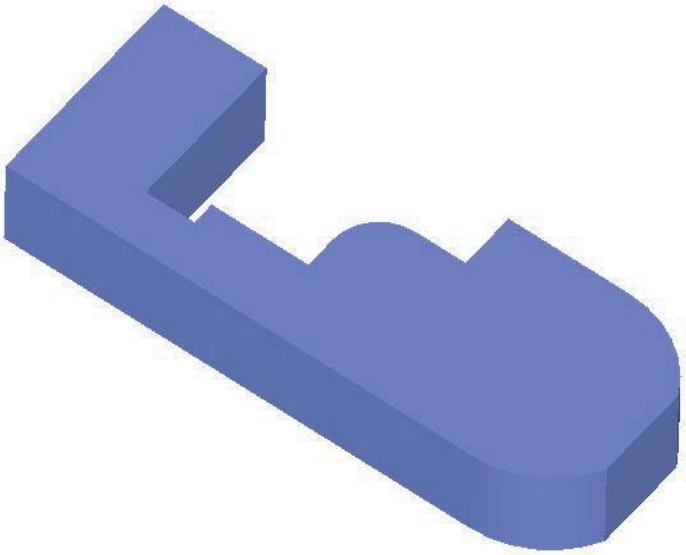


Figure 12.2: Simple shape

```
EXTEND TO (6.4,12.8)
EXTEND TO (0,12.8)
CLOSE
SWEEP BODY (0,0,5)
```

The use of the construction geometry created in the 2.5D command, which is CIRCLE 1, CIRCLE 2, and LINE 1, helps in defining both curves and intersection points. In the above case, this saves the user from having to work out square roots and copy the numbers into the command line by hand. In command lines 6 and 7, the "LXC" key stands for Line intersection with Circle.

12.2 Syntax handling

It is possible to make each command responsible for handling its own syntax, each command explicitly checking keywords and reading appropriate values. This means that the command syntax is together with the command, which can be useful when adding or modifying commands. However, this hides the syntax away in several places and necessitates code duplication. It can be better to record the syntax separately and have a general syntax handler. However this has the disadvantage that discrepancies between what the com-

mand code expects and the declared syntax can creep in.

Graham Jared's command interpreter in the BUILD system contained a coded syntax declaration in the form of a call to a central syntax handler. The first line of each command contained the call to the syntax handler, which put the values read from the command line onto the stack from where they could be retrieved by the command code. The same central syntax handler also provided a uniform HELP mechanism by printing the required command syntax.

A related approach to Jared's is to declare the syntax seperately from the command and preserve it in the form of a tree structure. Having the syntax explicitly recorded in this way means that it can be interrogated by other applications, such as menu handlers. Another point about declaring the syntax explicitly is that it makes it easier to understand the commands.

Syntax definition keys might be:

1. key: A mandatory keyword
2. opt_key: An optional keyword
3. value: A mandatory value
4. opt_value: An optional value
5. key_value: A mandatory key+value pair
6. opt_key_value: An optional key+value pair
7. def: A default value

An optional value or an optional key_value pair must be followed by a default specification so that the syntax controller can supply the missing value.

The important syntactic value types for a solid modeller command interpreter are:
REAL
INTEGER
VECTOR
Topological element
Geometric element

Ignoring specialised testing commands, for the majority of commands, the relevant topological elements are BODY, FACE, EDGE, or VERTEX, although others might be useful for particular operations. The use of FACESETs is generally not well defined, in modelling terms, because they were introduced for efficiency purposes and hence can be interpreted in different ways. SHELLs and LOOPs have a clearer meaning so there may be some cases where a command may usefully refer to them. The geometric elements are SURFACE, CURVE, or POINT.

The CUBE command might have the syntax:
CUBE opt_key_value FROM vector def (-1,-1,-1) optkey_value TO vector def (1,1,1)

The "opt_key_value" term specifies that the next two items are an optional key_value pair. The key and the value type follow, then "def" is to specify the default value if the pair is missing.

With this command syntax the command: "CUBE" would then generate a cube, side length 2 centred at the origin.

The command: "CUBE FROM (-4,0,-1)" would create a cube with side lengths 5, 1, and 2 in the X-, Y-, and Z-directions, centred on the point $(-2.5, 0.5, 0)$.

The command: "CUBE TO (20,10,5)" would generate a cube with side lengths 21, 11, and 6 centred on the point (10.5,5.5,3).

The command: "CUBE FROM (1,1,1) TO (3,3,3)" would generate a cube with all side lengths 2 as the default, but centred on the point (2,2,2).

The commands "CUBE FROM", "CUBE TO" and "CUBE TO (1,1,1) FROM (-1,-1,-1)", for example, would generate syntax errors; the first two because the vector value is missing, and the third because the key-value pairs are in the wrong order.

The command: "CUBE FROM (1,1,1) TO (-1,-1,-1)" might generate an error because the FROM point is greater than the TO point, although the software could really cope with this. The command: "CUBE FROM (1,-1,-1) TO (1,1,1)" should certainly generate an error because the X side length would be zero, but this is a semantic error rather than a syntactic one, and so it has to be discovered by the modelling software.

12.3 Model element identification

A major problem with command interpreters is identification of the targets of operations. Operations such as the Boolean operations depend solely on the position of the target models. However, many local operations need an edge, face, or vertex upon which to operate. Identification is discussed more fully by Várady et al. [140]; only a brief summary of various methods is given here.

Note, also, that there is much interest in what is termed "persistent naming". The idea behind this is that as the model develops the elements that appear are named and these names are propagated 'somehow' as they are split or changed. This means that if, say, an edge is chamfered and then the model is changed at an earlier stage so that the chamfered edge is split into two, the chamfer will be applied to both parts of the now divided edge. See also the next section.

Briefly there are five simple ways to identify elements:

1. Identity number

2. Topological/geometric navigation
3. Hit testing
4. Names
5. Geometry

1. Identity numbers
In the datastructure described in Appendix A, each entity in the datastructure has an associated number by which it can be identified. In the original BUILD system, this was used to identify the targets of operations, e.g.,

CHAMFER e9 - to chamfer edge number 9
SWEEP f10 BY (0,0,4) - to sweep face number 10 by 4 in the Z direction

However, although this was not too bad when using the system interactively, it was clumsy when writing command scripts to be used offline. This was basically because elements were modified or disappeared during modelling operations, so changing the sequence meant that the wrong elements were modified. It was frequently necessary to print out the object, identify the new entity numbers, and modify the script files. Note that this also happens if the entity numbers are used to record the commands in interactive sessions.

2. Topological/geometric navigation

In 1979, Chris Cary developed a method of topological navigation that was much more flexible than using entity numbers. The original method used the six qualifiers: TOP, BOTTOM, LEFT, RIGHT, FRONT, and BACK, together with the topological elements: FACE, EDGE, or VERTEX. Later on "INNER LOOP" was added as an additional qualified topological element. A sequence might be, for example:

BOTTOM VERTEX OF RIGHT EDGE OF LEFT FACE OF BODY

The starting point is the last entity, the body. From this the left-most face is found (if there were two or more, then a warning was given). The face is then used as a starting point to find the right-most edge bounding the face. The bottom vertex of this edge is returned as the result. The qualifiers are view dependent and were determined from the current view position.

To find an extreme element involves calculating the extents of an entity in the defined direction and in the opposite direction. The extreme element is defined as having the maximum extent in the given direction and, if there are two or more candidates, the minimum extent in the opposite direction.

In the body in figure 12.3, face 1 might be identified as the FRONT FACE. Edge 3 would be the TOP EDGE OF FRONT FACE. Vertex 4 would be the

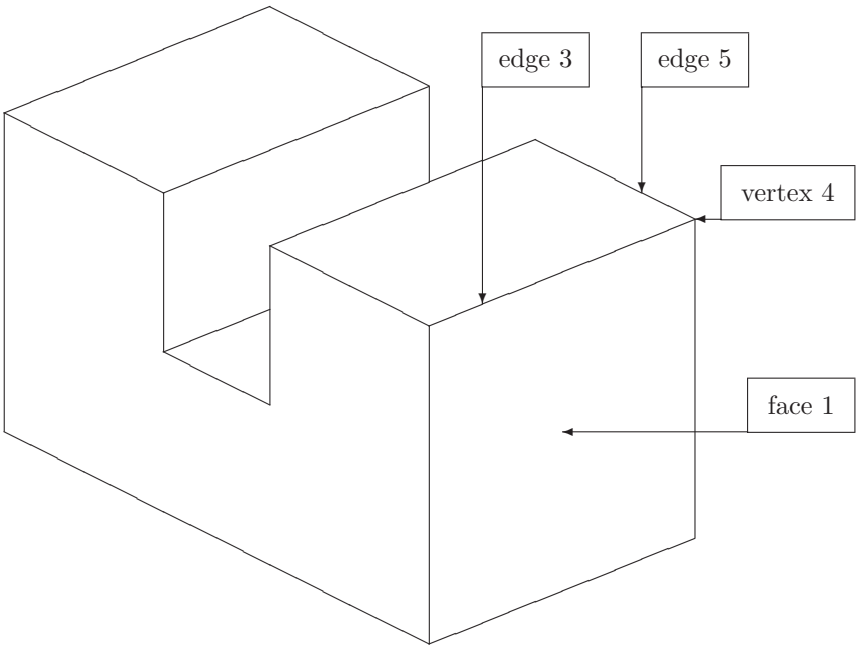


Figure 12.3: Topological traversal example

RIGHT VERTEX OF TOP EDGE OF FRONT FACE and edge 5 the BACK EDGE OF RIGHT VERTEX OF TOP EDGE OF FRONT FACE. Note that edge 5 could not be found as the RIGHT EDGE OF BODY or TOP EDGE OF BODY, because these are both ambiguous. Finding elements in complex bodies could be quite complicated, but the big advantage of Cary's method over using entity numbers was its independence from the modelling operations used to create the model.

Finally, note that a variant on Cary's original technique is to reverse the order of the terms so that the sequence can be decoded as it is read. The sequence given at the start of this section would be represented as:

BODY LEFT FACE RIGHT EDGE BOTTOM VERTEX

The sequence above to find edge 5 would become:

BODY FRONT FACE TOP EDGE RIGHT VERTEX BACK EDGE

3. Hit testing or ray casting
Hit testing involves 'firing' a ray at a body to find a topological entity given a view point and a direction. This method derives from graphics techniques. A ray is defined as a point, a direction, and a thickness. The thickness is needed because it is very difficult to point precisely at an entity, so the thickness is a sort of tolerance value. In modelling terms, ray casting involves intersecting the ray with all faces in the body and checking whether the resulting point lies inside or on the boundary of the face, checking the closest point to edges or checking the distance of vertices from the ray.

Hit testing is the most user-friendly of these methods and is convenient for a user, although it is the most complex in the amount of modelling work that needs to be done. Like Cary's topological navigation method, it is view dependent, so that if a ray is defined interactively and recorded in a log file, then it will not usually find the same element if a new viewing point is defined. This, however, is a minor consideration.

4. Names
Names are dangerous. It is fairly simple to define names and name tables so that any entity can be named and found. However, the problem, as pointed out in chapter 8, concerns what happens to names during modelling. If, say, a face is named and then split, to which geometric portion of the face should the name be associated? This topic is currently receiving attention under the term: "permanent naming".

5. Geometry
The final identification method is to identify parts of models using geometry, especially the vertex coordinates. A vertex is identified directly from its co-ordinates. An edge can be identified from the coordinates of its start and end

vertices. A face is identified by giving the start and end vertex positions of two of its edges.

This method is easy to implement and can be useful for simple objects, at least. However, it can be difficult to identify coordinate positions that derive from complicated intersections. To overcome this, some tolerance value may need to be used to find approximated positions.

Another problem is that the definition can be ambiguous. There can be more than one edge starting and ending at the given positions. Two faces in a sheet object may share two or more edges. In these cases, an extra geometric test may need to be performed.

12.4 Multi-user modelling

This section concerns "multi-user modelling", whereby more than one person can use the modeller at the same time.

Most, if not all, current solid modelling systems have been developed as single-user systems for use on isolated workstations, communicating with other applications via discfiles or some such static interface. It is also possible to allow multiple access to a modeller. This is related to the complex topic of concurrent engineering, which aims at allowing cooperative work practices. Multi-user modelling also allows cooperative work practices, letting users access the same part or parts to design or modify a part in collaboration. This is illustrated symbolically in figure 12.4. For the first case, a supervisor, or teacher, can monitor the work of several trainees or students, intervening if necessary and providing online guidance. In the second case, a chief designer or product controller can supervise the work of a design team while the designers can see each other's work. In the third case, a designer and process planner, or other expert, can collaborate to produce a design adapted for downstream applications.

The same sort of technique can be used to create dual-functionality systems that use two different systems to increase the functionality range of a system, or to help when switching between modellers.

There are several ways to implement this, as a network of workstations or as a large machine and several slave computers. It remains a possibility, but it needs more work.

12.4.1 Splitting up the command interpreter

Note, first, that this is not a great departure from a single-user system; the most difficult parts to handle are possibly the system-level facilities to allow input from multiple sources. The initial part is to separate out the interfaces from the central 'kernel' modelling system and adapt the interfaces to allow input from multiple sources and output to multiple destinations.

The interfaces are as follows:

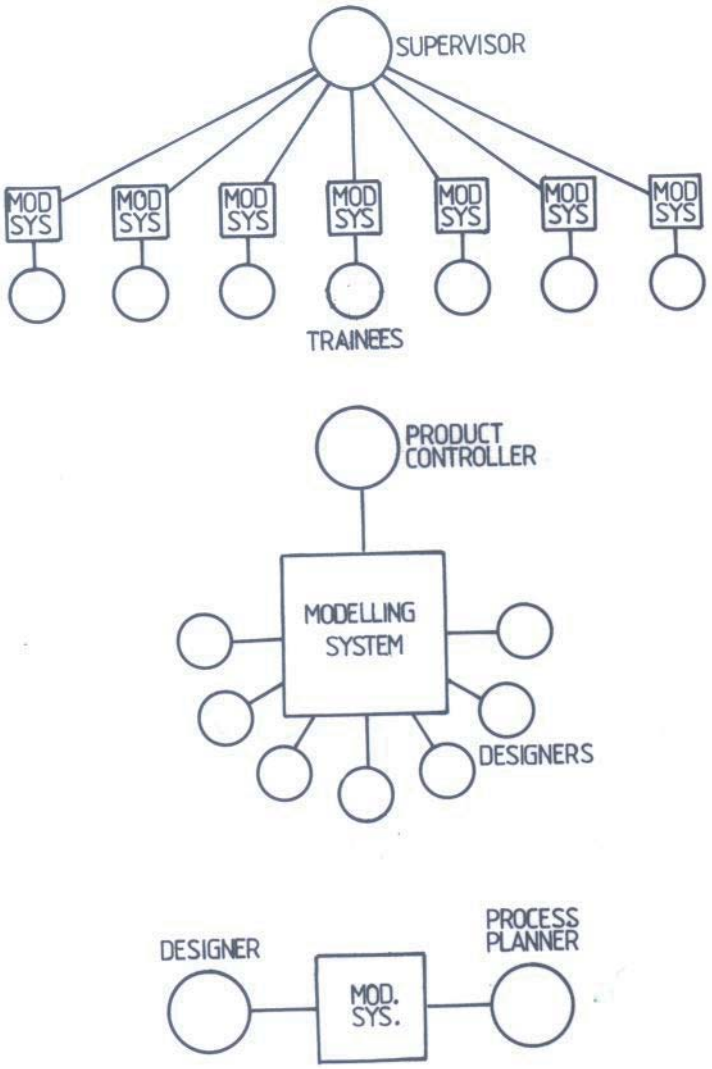


Figure 12.4: Conflicting and non-conflicting modelling sequences

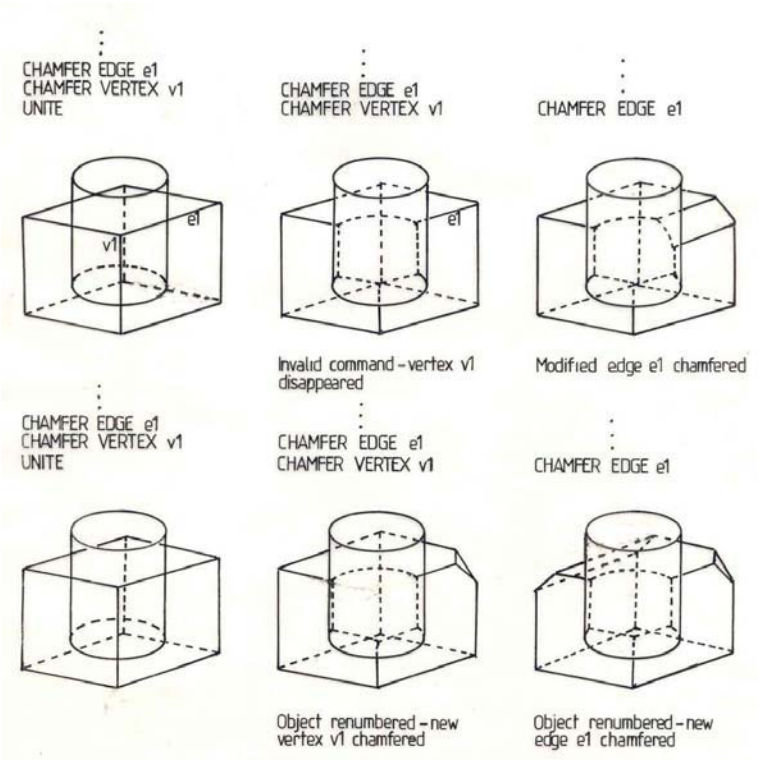


Figure 12.5: Conflicting and non-conflicting modelling sequences

1. command stream
2. graphics output
3. error stream
4. modeller–modeller transfer

Command stream

A multi-user command stream can be implemented as a list mechanism where each user is allowed to add commands for processing. This simple mechanism, though, requires some sophisticated handling mechanisms to make it work properly. The system may need to control access so that some parts of a model can be protected, in the same way that a database controls access. It is also necessary to check the targets of commands in the command list to see if they have been modified or deleted by an earlier command. An example of the sort of problems that might arise is shown in figure 12.5.

In the first sequence, the edge that is the target of the command is not checked and is reused with the result that the wrong edge is eventually cham-

fered. In the second sequence, the command list is checked and the edge reference is updated as the model is modified so that the correct edge is eventually modified. This illustrates why it is necessary to check the command list. How to do this is more complicated. It is necessary to be able to match the records of changed elements in a model to the targets of commands. This means that it is necessary to convert the command as it is received and decode any references into actual pointers. This is what the topic of "persistent naming" is intended to solve. Persistent naming has become more important as commercial systems have adopted the strategy of recording the user operations in a sort of "history list". If the user wishes, it is possible to access commands in this history list and change them. This almost inevitably results in changed topology, i.e., structure, and hence the need to try and keep track of changed elements.

(I will leave this topic here, though, as it is changing. There is a limit to the usefulness of this, and it is getting close to the dream function, which is described nowhere in this book, the function "Do What I Mean". Trying to infer what the user really wants is more-or-less impossible, and hence, any approximation is already very complex.)

A command structure might contain the following:

user identifier	The user who gave the command
user access code	Access code so as to be able to restrict use
user priority	For controlling queue order
user location	For error reporting, graphics etc.
status flag	Online/offline etc.
command string	The actual command

Graphics output

As related in chapter 10, there are several different styles of graphics output from a modeller, from simple wireframe pictures to realistic rendered images. It is necessary to maintain a list of graphics destinations so that when a 'draw' or similar command, is invoked all destinations can be found. Each of these destinations must also contain some 'method' reference to determine what type of graphics transfer is to take place.

A graphics destination record might look like:

destination address method user status

Error stream

The error stream is for sending error messages. These might be simply displayed or cause a more user-friendly help subsystem to be invoked to try and give more help about what happened.

Modeller-modeller transfer

This is needed for transferring between collaborating modules in the system. When using two different modellers, say, to have increased functionality, it is necessary to be able to switch back and forth frequently and fluently. This is not a trivial problem, but one that has bearing on the ability to link modellers

and applications into complex systems.

12.4.2 Blocking access to sensitive model parts

Analogously to the way in which database systems work, it is necessary to be able to control access to models when more than one person is creating or modifying a model. Things like interfaces between parts in a mechanism, or basic design elements, may need to be protected. This can be done using non-geometric 'NOLI-ME-TANGERE' elements attached to parts of the model to cause operations that modify them to fail. In practice this is another part of the complexity surrounding attributes in modelling.

12.5 Command files and macro-languages

When testing a modeller, it is useful to be able to create command interpreter instructions in files, called here "command files", which can be read and executed. This makes it possible to repeat testing sequences and to create test suites to check that functionality is still available. A similar facility can also be used internally to create macro-sequences to build up commands.

An important aspect of this is language. For command files, it is normal to use the command line sequences as instructions. It can be useful, though, to spend a little time, then, on language design when defining the command interpreter syntax. As should be obvious from chapter 3, modelling systems are heavily typed. These types may need to be transferred to the command interpreter as well. Other options, such as repetitions and numerical calculations, may also need to be included. It is common, currently, to create patterns in CAD models by copying a sub-model as a separate object and creating patterns of these that are then added to or subtracted from the original model. However, it may be more correct to apply a command repetitively instead, especially if it uses model parts to adapt the command.

Another possibility is to use an existing language as input. Kjellberg [72] proposed the use of LISP as a language shell around a modeller. One reason was that mentioned above, that it is often necessary for users to be able to do extra things, such as performing calculations. LISP has a simple but powerful structure that has been exploited for many purposes. Important also, was that LISP is interactive, and so, earlier on, complex programs could be monitored, even altered, interactively. Other interactive languages, such as visual basic, have appeared since.

12.6 Application programming interfaces

Application programming interfaces (APIs) make the modeller available as part of an application. This is usually done by making them libraries. It could be done by having the modeller as a separate process as part of an

object-oriented environment. However, this section will concentrate on the library type of implementation.

There are two parts to this section. The first describes the classic method. The second part concerns a communal effort by a top group of experts in the United Kingdom to define a general interface, called DJINN.

12.6.1 Classic programming interfaces

The usual scientific way of doing this is to sit down, scratch your head, and try and work out what to put in the interface. It is necessary to determine what is useful for applications. The main operations, such as those described in chapters 5 and 6, are usual candidates. However, support operations are also useful. The functions to traverse sets, like all edges in a body, are also useful. These might return lists instead of applying a function, though. The set of functions described in chapter 3 are probably useful as well.

An API might consist of roughly the following elements:

- High-level modelling operations
- Disc input and output
- Data exchange standard interfaces
- Interrogation functions
- Graphics functions

This will probably not be enough for code developers, though. Unless the modelling code is given in source code format, then there will probably be useful hidden functions that are not included. It is difficult, if not impossible, to foresee all of the uses to which a modelling system might be put. The set above, therefore, represents a minimal high-level set, but it can be useful also to provide a way for users to get at the basic datastructure so that they can build up their own extra functions.

12.6.2 DJINN

The DJINN effort (see [28]) is well worth looking at not only because it provides a guide but also because it takes a broad view of geometric modelling. This means that it is more generally applicable for applications than the API of a single implementation.

DJINN specifies an interface in which geometry, wireframe, sheet objects, and solids can coexist. It is also intended that different modelling methodologies can coexist, so that the implementation details are hidden. Solids could be represented by boundary representation (B-rep) but also as CSG or cellular models, for example.

The DJINN philosophy is to provide a minimum number of elements. It assumes that programmers know what they are doing and can supplement the basic API. It assumes also that the modelling system behind the interface will exist as part of a larger application that has a significant functionality. Instead of trying to duplicate functionality, such as logging for example, this is left for the application. Instead of just looking at what a particular modeller can offer, DJINN is based on both sides of the interface. This means that general modelling functionality is combined with general knowledge about the way that modelling is used in applications.

Chapter 13

Free-form geometry

Incorporating free-form geometry into a solid model is a requirement for many purposes, but it is not always easy to achieve the desired effect. This chapter describes some aspects of working with free-form surfaces from a solid modelling point of view. The mathematical aspects are dealt with in several texts (Faux and Pratt [34], Mortenson [88], Boor [9], Farin [35], Bartels et al. [6], Hoschek and Lasser [61], Rockwood and Chambers [113], for example), geometric design systems are common. There remain aspects of how to put together solids and free-form surfaces.

Originally this was done by setting single surfaces into faces (section 6.11, section 13.1). Another alternative is to build a model from surface patches, as described in section 13.5. Finally, polyhedral smoothing (Chiyokura and Kimura [18], Peters [98], and Varady and Stroud [141]) offers a very interesting three-dimensional sculpting method which offers a method for three-dimensional design. This technique is also dealt with in section 13.6.

13.1 Basics

To summarise: Topology defines the structure of a model, and the geometry defines the shape. There are several types of geometry, but it is possible to draw a distinction between analytic geometry and numeric geometry. With analytic geometry, the shape information is explicit, for example, that a curve is circular or that a surface is planar. With numeric geometry, there is no inherent notion of the shape that is represented. The shape is controlled by the positions of a set of points called the control points. In addition there can be weights and knot points in more advanced forms.

This numeric geometry is more difficult to use but is very useful, not only for representing complex shapes such as car bodies, boats, aircraft or aesthetic products, but also to 'close the geometric set'. The 'geometric set' is the set of analytic geometry in a modelling system. Normally this covers many useful

forms, but usually these are not enough to cover everything. For example, if you intersect two cylinders that are perpendicular, with intersecting axes, but have different radii, then the resulting curve is complex. Numeric geometry can be used to represent this type of curve rather than having to have an explicit analytic curve.

The following is a very basic description of some geometrical aspects. This is not intended to replace the use of the literature cited above but to illustrate some basics. Numerical geometry has been and is still the subject of a lot of research.

Put very simply, numerical geometry combines a set of points, called the control points, with different weighting functions. The weighting functions are called "basis functions" and different sets of functions give different geometry. Some are described briefly below.

The geometry is parametric. One well-known method is to have an explicit formula, such as $y = x^2 - 2x + 1$, for example, where y is given explicitly as a function of x. Alternatively x and y may be related, as in the formula $x^2 + y^2 = r^2$. For parametric geometry, though, x, y, and z are given as functions of another variable, say t.

One example of this is $x = rsin(t)$; $y = rcos(t)$, $z = 0$; $t = 0$ to 2Π, which gives a circle in the plane $z = 0$. If z is also a function, $z = kt$, then the curve is a spiral. However, the forms used here are all polynomials in t instead of being functions like cosine or sine.

The shape of a curve is determined by a set of points, called the "points of control". Exactly where these points are for a given curve depends on the so-called "Basis functions". The first of these is called the "Bézier" form and is described next. A more general form, the "B-spline", for "Basis" spline is described briefly afterwards, and finally, the currently popular form, the NURBS form is described. The descriptions are for curves, but there is an extension to surfaces, solids, and even higher degrees.

13.1.1 Bézier geometry

Bézier geometry is very convenient for illustrating the basic notions of numerical geometry. The basis functions for a curve of degree n are simply the expansions of the function:

$$((1 - t) + t)^n$$

The general formula for a Bézier curve is:

$$\sum_{i=0}^{n} \frac{n!}{i!(n-i)!}(1 - t)^{n-i}t^i B_i$$

where t is the curve parameter and the B_i are the control points. For example, for a cubic Bézier curve, which has four control points, you have:

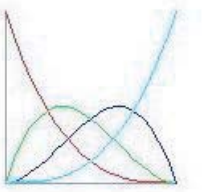


Figure 13.1: Bézier basis functions

$$P(t) = (1-t)^3 B_0 + 3t(1-t)^2 B_1 + 3t^2(1-t)B_2 + t^3 B_3 \tag{13.1}$$

As the value of t varies from 0 to 1, the sequence of points traces out a curve. Bézier curves have some nice properties as well, which made them attractive as a starting point for design. For example, the curve passes through the first and last points, and the tangent directions at the start and end of the curve are given by the control points. The tangent at the start of the curve is $n(B_1 - B_0)$. The tangent at the end of the curve is $n(B_n - B_{n-1})$.

To illustrate the notion of basis functions and points of control, take three polynomials in t defining x, y, and z:

$$\begin{gathered} x = t^3 - 3t^2 + 9t - 1 \\ y = 11.5t^3 - 18t^2 + 9t - 1 \\ z = -5t^3 + 9t^2 + 3t \end{gathered}$$

The Bézier basis functions for a cubic are:

$$\begin{gathered} (1-t)^3 = 1 - 3t + 3t^2 - t^3 \\ 3t(1-t)^2 = 3t - 6t^2 + 3t^3 \\ 3t^2(1-t) = 3t^2 - 3t^3 \\ \text{and } t^3 \end{gathered}$$

These are shown plotted in figure 13.1

To write the functions for x, y and z in terms of combinations of the Bézier basis functions, one can write:

$$t^3 - 3t^2 + 9t - 1 = a.(1-t)^3 + b.3t(1-t)^2 + c.3t^2(1-t) + d.t^3$$

where a, b, c and d are the unknown terms. Comparing the terms gives:

$$t^3 - 3t^2 + 9t - 1 = (d - 3c + 3b - a)t^3 + (3c - 6b + 3a)t^2 + (3b - 3a) + a$$

which gives:

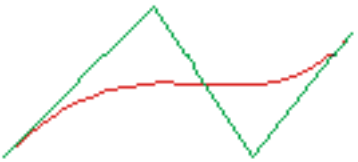


Figure 13.2: Simple Bézier curve and control polygon

$$\begin{aligned} -1 &= a \\ 9 &= 3b - 3a \\ -3 &= 3c - 6b + 3a \\ 1 &= d - 3c + 3b - a \end{aligned}$$

and from that one obtains: $a_x = -1, b_x = 2, c_x = 4$, and $d_x = 6$.

Doing the same thing for y and z gives:
$a_y = -1, b_y = 2, c_y = -1, d_y = 1.5$
$a_z = 0, b_z = 1, c_z = 5, d_z = 7$

Writing these as points gives (-1,-1,0), (2,2,1), (4,-1,5), and (6,1.5,7). These are the control points of the Bézier curve shown in figure 13.2.

Note how the control polygon indicates the tangent directions at the start and end of the curve.

Another interesting tool is that developed by P. de Casteljau. This can be used both to find a point on the curve by a set of linear interpolations and to determine the control points of a subdivision of the curve. This method uses a 'corner cutting' scheme based on linear interpolation.

To find a point with a particular parameter value, say $t = 0.5$, on the curve, a set of points on the control polygon is formed. These points are at a distance of 0.5 along each edge of the polygon. These points are then joined with two lines. The midway points along these two new lines are found and joined. Finally the halfway point along this is the point on the curve.

For $t = 0.5$, the result is shown in figures 13.3 and 13.4

The de Casteljau method can be written in pyramid form as:

$$\begin{array}{ccccccc} & & & B_0^3 & & & \\ & & B_2^0 & & B_2^1 & & \\ & B_1^0 & & B_1^1 & & B_1^2 & \\ B_0 & & B_1 & & B_2 & & B_3 \end{array}$$

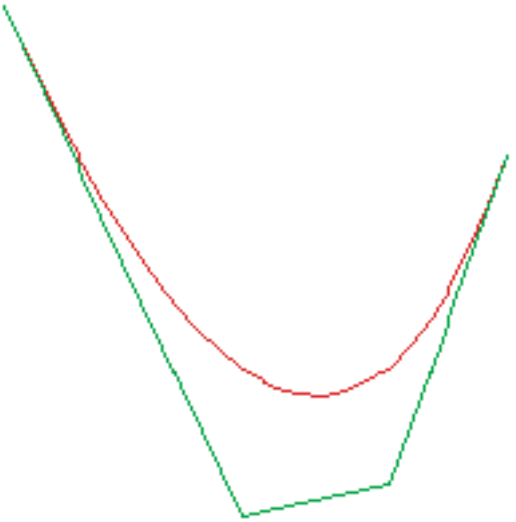


Figure 13.3: Bézier curve and control polygon

$B_0^3 = (1-t)B_2^0 + tB_2^1$

$B_2^0 = (1-t)B_1^0 + tB_1^1$

$B_2^1 = (1-t)B_1^0 + tB_1^1$

$B_1^0 = (1-t)B_0 + tB_1$

$B_1^1 = (1-t)B_0 + tB_1$

$B_1^2 = (1-t)B_0 + tB_1$

For the previous example, to find the point at $t = 0.4$, you would have

$$\begin{array}{cccc} & & (2.184, 0.456, 2.32) & \\ & (1.24, 0.44, 1.28) & & (3.6, 0.48, 3.88) \\ (0.2, 0.2, 0.4) & & (2.8, 0.8, 2.6) & & (4.8, 0, 5.8) \\ (-1, -1, 0) & (2, 2, 1) & (4, -1, 5) & (6, 1.5, 7) \end{array}$$

The point P(0.4) is (2.184,0.456,2.32). Moreover, the two diagonals are the sets of control points of two curves with the same shape as the original, meeting at that point.

That is $(-1, -1, 0)$, $(0.2, 0.2, 0.4)$, $(1.24, 0.44, 1.28)$, and $(2.184, 0.456, 2.32)$ are the control points for one part of the curve, whereas $(2.184, 0.456, 2.32)$,

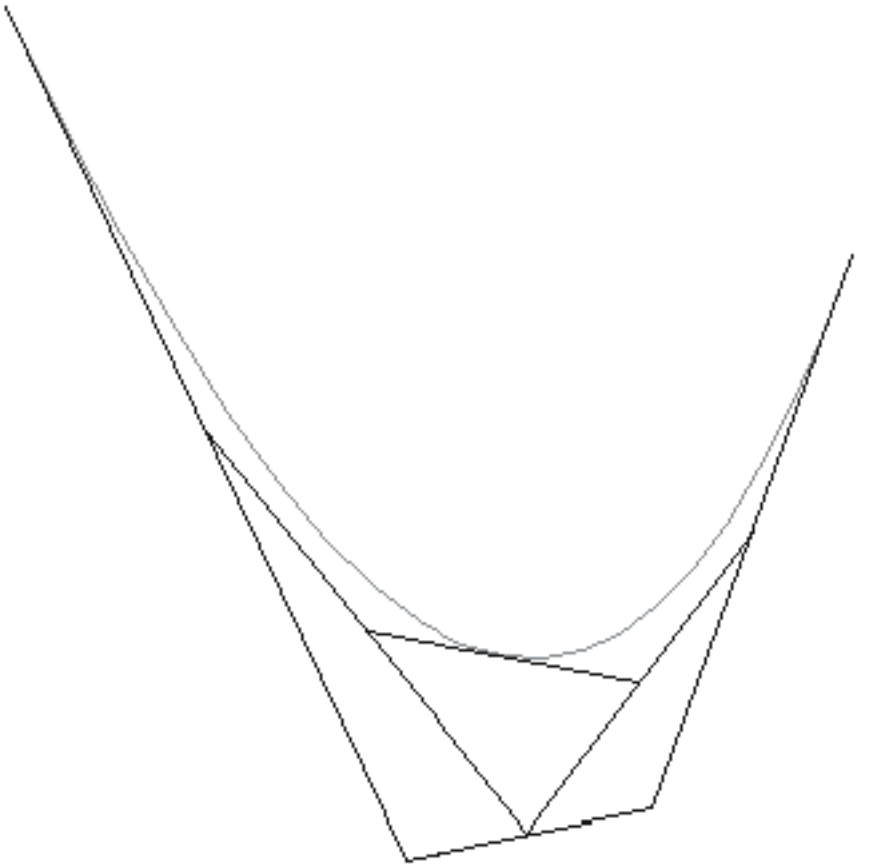


Figure 13.4: de Casteljau scheme

$(3.6, 0.48, 3.88)$, $(4.8, 0, 5.8)$, and $(6, 1.5, 7)$ are the control points of the other part of the curve.

The de Casteljau method also works with values of t outside the range 0 to 1. For example, if you use the parameter value 2, you get:

$$\begin{array}{cccccccc} & & & (13,37,2) & & & & \\ & & (7,-13,16) & & (10,12,9) & & & \\ & (5,5,2) & & (6,-4,9) & & (8,4,9) & & \\ (-1,-1,0) & & (2,2,1) & & (4,-1,5) & & (6,1.5,7) & \end{array}$$

The control points are a little different now, though. $(-1, -1, 0)$, $(5, 5, 2)$, $(7, -13, 16)$, and $(13, 27, 2)$ are the control points of a single extended curve that can be divided into two pieces, one with the original control points: $(-1, -1, 0)$, $(2, 2, 1)$, $(4, -1, 5)$, and $(6, 1.5, 7)$ and a second extension curve with the control points: $(6, 1.5, 7)$, $(8, 4, 9)$, $(10, 12, 9)$, and $(13, 37, 2)$.

13.1.2 B-spline geometry

The same idea of basis functions lies behind B-spline curves as well. However, the basis functions are different. For a cubic B-spline, these are:

$t^3/6$
$(1 + 3t + 3t^2 - 3t^3)/6$
$(4 - 6t^2 + 3t^3)/6$
$(1 - 3t + 3t^2 - t^3)/6$

Repeating the procedure of rewriting polynomials in terms of these new basis functions one gets:

$$t^3 - 3t^2 + 9t - 1 =$$
$$a_x.(1-3t+3t^2-t^3)/6+b_x.(4-6t^2+3t^3)/6+c_x.(1+3t+3t^2-3t^3)/6+d_x.t^3/6$$
$$= (d_x-3c_x+3b_x-a_x)t^3/6+(3c_x-6b_x+3a_x)t^2/6+(3c_x-3a_x)t/6+(c_x+4b_x+a_x)$$

or: $d_x - 3c_x + 3b_x - a_x = 6$
$3c_x - 6b_x + 3a_x = -18$
$3c_x - 3a_x = 54$
$c_x + 4b_x + a_x = -6$

Solving this set of equations gives:

$$a_x = -12, b_x = 0, c_x = 6, d_x = 12$$

In the same way it is possible to find the values for y and z which gives the points of control as: $(-12, -22, 3)$, $(0, 5, -3)$, $(6, -4, 9)$ and $(12, 20, 9)$. The curve is shown in figure 13.5

Since the initial polynomials for x, y, and z are the same, then it is clear that the resulting curve is the same. This example is mainly intended to show the difference between the Bézier points of control and the B-spline points of control.

Note, though, that the curve does not pass through any control points. Note also that the tangent conditions are also different. Differentiating the basis functions and substituting $t = 0$ and $t = 1$ you find that the tangent at the start of the curve is $0.5 \times (B_2 - B_0)$ and at the end of the curve is $0.5 \times (B_3 - B_1)$, where B_0, B_1, B_2, B_3 are the B-spline control points.

The basis functions do not have the same easy form as the Bézier basis functions; it is necessary to derive them. To derive the B-spline basis functions, it is necessary to follow a complicated procedure. The basis functions placed end-to-end form a bell-shaped curve. This bell-shaped curve is a composite of four curve pieces, for the cubic case, noted below as f_0, f_1, f_2, and f_3.

The theory of B-splines imposes some conditions on the function formulae, as follows:

$f_0(0) = 0, f_0(1) = f_1(0), f_1(1) = f_2(0), f_2(1) = f_3(0), f_3(1) = 0$

Also, to preserve continuity conditions:

$f_0'(0) = 0, f_0'(1) = f_1'(0), f_1'(1) = f_2'(0), f_2'(1) = f_3'(0), f_3'(1) = 0$

and

$f_0''(0) = 0, f_0''(1) = f_1''(0), f_1''(1) = f_2''(0), f_2''(1) = f_3''(0), f_3''(1) = 0$

Rewriting these equations as polynomials we have:

$f_0(t) = a_0t^3 + b_0t^2 + c_0t + d_0$
$f_1(t) = a_1t^3 + b_1t^2 + c_1t + d_1$
$f_2(t) = a_2t^3 + b_2t^2 + c_2t + d_2$
$f_3(t) = a_3t^3 + b_3t^2 + c_3t + d_3$

and also:

$f_0'(t) = 3a_0t^2 + 2b_0t + c_0$
$f_1'(t) = 3a_1t^2 + 2b_1t + c_1$
$f_2'(t) = 3a_2t^2 + 2b_2t + c_2$
$f_3'(t) = 3a_3t^2 + 2b_3t + c_3$

and

$f_0''(t) = 6a_0t + 2b_0$
$f_1''(t) = 6a_1t + 2b_1$
$f_2''(t) = 6a_2t + 2b_2$
$f_3''(t) = 6a_3t + 2b_3$

Finally, we have the condition that the polynomials must sum to 1 over the range from 0 to 1, hence:

$a_0 + a_1 + a_2 + a_3 = 0$
$b_0 + b_1 + b_2 + b_3 = 0$
$c_0 + c_1 + c_2 + c_3 = 0$
$d_0 + d_1 + d_2 + d_3 = 1$

To satisfy the conditions described above, we have the following relations:

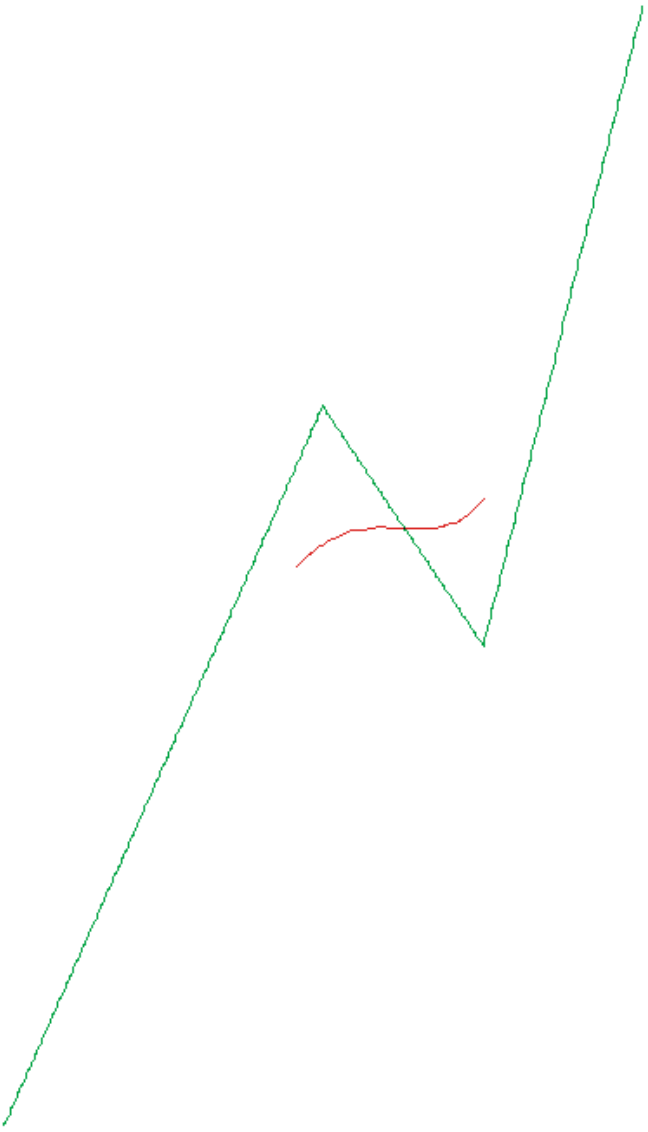


Figure 13.5: Simple B-spline curve and control polygon

$b_0 = c_0 = d_0 = 0, d_1 = a_0,$
$a_1 + b_1 + c_1 + d_1 = d_2,$
$a_2 + b_2 + c_2 + d_2 = d_3,$
$a_3 + b_3 + c_3 + d_3 = 0,$
also: $c_1 = 3a_0$ and $2b_1 = 6a_0,$
from the derivative conditions, so:
$a_1 + 7a_0 = d_2,$
$3a_1 + 2b_1 + c_1 = c_2,$ or
$3a_1 + 9a_0 = c_2$ and $6a_1 + 6a_0 = 2b_2$ or
$3a_1 + 3a_0 = b_2,$ $3a_2 + 2b_2 + c_2 = c_3,$ or
$3a_2 + 9a_1 + 15a_0 = c_3$ and
$6a_2 + 6a_1 + 6a_0 = 2b_3,$
$a_2 + 7a_1 + 19a_0 = d_3,$
$3a_3 + 2b_3 + c_3 = 0,$ or
$3a_3 + 9a_2 + 15a_1 + 21a_0 = 0$ or
$a_3 + 3a_2 + 5a_1 + 7a_0 = 0$ so
$a_3 = -3a_2 - 5a_1 - 7a_0$

There is also the condition that:
$a_0 + a_1 + a_2 + a_3 = 0$
So $-2a_2 - 4a_1 - 6a_0 = 0$
or $a_2 = -2a_1 - 3a_0$
hence:
$a_3 = a_1 + 2a_0,$
$b_3 = -3a_1 - 6a_0,$
$c_3 = 3a_1 + 6a_0$ and
$d_3 = 5a_1 + 16a_0$
because $a_3 + b_3 + c_3 + d_3 = 0$
we have $6a_1 + 18a_0 = 0,$ or $a_1 = -3a_0$
Finally, using $d_1 + d_2 + d_3 = 1$
we have $6a_0 = 1$ or $a_0 = 1/6$
hence:
$a_1 = -1/2, b_1 = 1/2, c_1 = 1/2$ and $d_1 = 1/6,$
$a_2 = 1/2, b_2 = -1, c_2 = 0$ and $d_2 = 2/3,$
$a_3 = -1/6, b_3 = 1/2, c_3 = -1/2$ and $d_3 = 1/6$

Rewriting we get the familiar form of the B-spline basis equations:
$t^3/6$
$(1 + 3t + 3t^2 - 3t^3)/6$
$(4 - 6t^2 + 3t^3)/6$
$(1 - 3t + 3t^2 - t^3)/6$

These equations are shown plotted in figure 13.6.

In fact the conditions determine that the first basis function is a multiple of t^3 and the last a multiple of $(1 - t)^3$.

A simple conversion between Bézier and B-spline control points can be

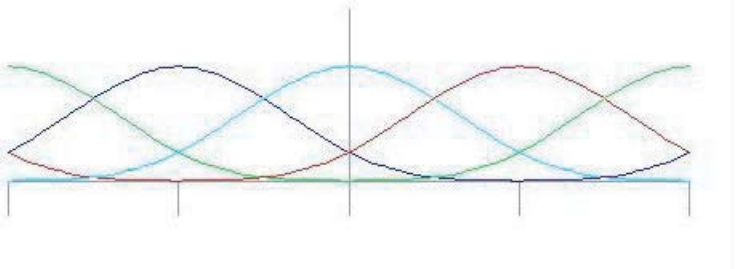


Figure 13.6: Bézier basis functions

found by comparing the basis functions. Suppose the control points of two equal curves are $B_0^e, B_1^e, B_2^e, B_3^e$, and $B_0^s, B_1^s, B_2^s, B_3^s$, respectively.

Evaluating both sets of basis functions at $t = 0$ and at $t = 1$ gives:
$B_0^e = (B_0^s + 4B_1^s + B_2^s)/6$
$B_3^e = (B_1^s + 4B_2^s + B_3^s)/6$

Differentiating the basis functions and evaluating these at t = 0 and t = 1 to find the tangents gives:
$3(B_1^e - B_0^e) = (B_2^s - B_0^s)/2$
$3(B_3^e - B_2^e) = (B_3^s - B_1^s)/2$

or:
$B_1^e = (2B_1^s + B_2^s)/3$ and
$B_2^e = (B_1^s + 2B_2^s)/3$

Solving the other way gives:
$B_0^s = 6B_0^e - 4B_1^s - B_2^s$
$B_1^s = 3B_2^e - 2B_2^s$
$B_2^s = 3B_1^e - 2B_1^s$
$B_3^s = 6B_3^e - 4B_2^s - B_1^s$

Substituting in B_1^s and B_2^s gives:
$B_1^s = 2B_1^e - B_2^e$
$B_2^s = 2B_2^e - B_1^e$

and substituting these in the expressions for B_0^s and B_3^s gives:
$B_0^s = 6B_0^e - 7B_1^e + 2B_2^e$
$B_3^s = 6B_3^e - 7B_2^e + 2B_1^e$

Examining the control points for the example in Bézier and B-spline, that is:
Bézier: (-1,-1,0), (2,2,1), (4,-1,5) and (6,1.5,7)
B-spline: (-12,-22,3), (0,5,-3), (6,-4,9) and (12,20,9)
shows that these relationships hold.

The basis functions for a quadratic B-spline can be derived in the same way to give the functions:

$t^2/2$,
$(-2t^2 + 2t + 1)/2$,
$(t^2 - 2t + 1)/2$

For a quartic B-spline, you get the following basis functions:
$t^4/24$,
$(-4t^4 + 4t^3 + 6t^2 + 4t + 1)/24$,
$(6t^4 - 12t^3 - 6t^2 + 12t + 11)/24$,
$(-4t^4 + 12t^3 - 6t^2 - 12t + 11)/24$,
$(t^4 - 4t^3 + 6t^2 - 4t + 1)/24$

Substituting $(1 - t)$ for t in the B-spline basis equations and calculating the new equations gives the same equations as a result but in reverse order. This demonstrates that the equations are symmetric.

Although the basis functions and other properties seem more complex, there are several advantages of the B-spline form. The most general form, and the currently most popular form, is the NURBS form of the B-spline. Before mentioning this, though, it is necessary to explain briefly rational forms.

13.1.3 Rational forms

Bézier and B-spline curves are limited in the type of geometry they can represent. It is, for example, impossible to represent a circle in a simple Bézier form, because the shape of a simple Bézier is a parabola. To get around this difficulty, it is necessary to introduce weights to the control points. The curve forms with these weights are called "rational forms".

Now try and define a 90 degree circular arc with a Bézier.

Assuming that the circular arc is to be on the $Z = 0$ plane, has unit radius, and a quadratic Bézier will be used, you would have the three control points: $(0, 1, 0)$, $(1, 1, 0)$, and $(1, 0, 0)$.

No problem.

Except that this is not a circular arc.

Evaluate the curve at the point $t = 0.5$, you have:

$P(0.5) = (1 - 0.5)^2 * (0, 1, 0) + 2 * 0.5 * (1 - 0.5) * (1, 1, 0) + (0.5)^2 * (1, 0, 0)$

or: $P(0.5) = 0.25 * (0, 1, 0) + 0.5 * (1, 1, 0) + 0.25 * (1, 0, 0)$, which gives $(0.75, 0.75, 0)$ and not $(\sqrt{2}/2, \sqrt{2}/2, 0)$

To have a true circular arc, it is necessary to use weights on the control points. The cubic rational Bézier form looks like:

$P(t) = \frac{(1-t)^n B_0 w_0 + nt(1-t)^{n-1} B_1 w_1 + ... + nt^{n-1}(1-t) B_{n-1} w_{n-1} + t^n B_n w_n}{(1-t)^n w_0 + nt(1-t)^{n-1} w_1 + ... + nt^{n-1}(1-t) w_{n-1} + t^n w_n}$

For the quadratic Bézier form you get:

$P(t) = \frac{(1-t)^2 B_0 w_0 + 2t(1-t) B_1 w_1 + t^2 B_2 w_2}{(1-t)^2 w_0 + 2t(1-t) w_1 + t^2 w_2}$

Keeping the weights of w_0 and w_2 as 1, and substituting the point above you get:

$P(0.5) = \frac{(1-0.5)^2 * (0,1,0) + 2 * 0.5 * (1-0.5) * (1,1,0) w_1 + (0.5)^2 * (1,0,0)}{(1-0.5)^2 + 2 * 0.5 * (1-0.5) w_1 + (0.5)^2} = (\sqrt{2}/2, \sqrt{2}/2, 0)$

which gives:

$P(0.5) = \frac{0.25*(0,1,0)+0.5*(1,1,0)w_1+0.25*(1,0,0)}{0.5+0.5*w_1} = (\sqrt{2}/2, \sqrt{2}/2, 0)$

Rearranging gives you:
$(0.25, 0.25, 0) + 0.5 * (1, 1, 0)w_1 = 0.5 + 0.5 * w_1 * (\sqrt{2}/2, \sqrt{2}/2, 0)$
so
$(0.5, 0.5, 0) + (w_1, w_1, 0) = (1 + w_1) * (\sqrt{2}/2, \sqrt{2}/2, 0)$
since both x and y are the same, this gives:
$w_1 = \sqrt{2}/2 - 0.5 + w_1 * \sqrt{2}/2$

or
$w_1(1 - \sqrt{2}/2) = \sqrt{2}/2 - 0.5$

which gives:
$w_1 = (\sqrt{2}/2 - 0.5)/(1 - \sqrt{2}/2)$

Multiplying the top and bottom by $(1 + \sqrt{2}/2)$ gives: $w_1 = \sqrt{2}/2$.

So, to represent an exact circle you reduce the weight of the middle control point. Reducing the weight even lower gives an ellipse. In effect, rational forms of these curves offer the possibility of using only numerical geometry instead of having a mixture of analytic and numerical geometry. This, though, is a strategic decision and is described in chapter 3.

One thing to note, though, is that some operations may need to use numerical geometry because they distort analytic geometry so that it can no longer be represented by an analytic form. This is another important support role for numerical geometry.

13.1.4 NURBS geometry

The term NURBS stands for Non-Uniform Rational B-Spline, and this form adds an extra degree of complexity, and, hence, flexibility, to the B-spline. This comes in the form of a so-called 'knot vector'. The values in the knot vector mark the starts and ends of the B-spline basis functions. Each knot in the knot vector is the same as, or greater than, the preceding value. A useful reference is Piegl and Tiller [100], which explains the manipulation methods needed for solid modelling.

Examining figure 13.6 you see at the bottom some small marks. These are the knot points. If they are regularly spaced, then the B-spline is a Uniform B-spline. Changing them gives the non-uniform character of the curve. One special case is where two or more knots coincide. If n knots coincide at the beginning of the curve (parameter $t = 0$), and another n knots coincide at the end of the curve ($t = 1$), then you have a Bézier form of the curve.

A NURBS curve can be defined as:
$P(u) = \frac{\sum_{i=0}^{n} w_i * B_i * N_{i,k}(u)}{\sum_{i=0}^{n} w_i * N_{i,k}(u)}$

The B-spline basis functions $N_ik(t)$ are defined as:

$N_{i,k}(u) = \frac{u-t_i}{t_{i+k}-t_i} * N_{i,k-1}(u) + \frac{t_{i+k+1}-t_{i+1}}{t_{i+k+1}-t_{i+1}} * N_{i+1,k-1}(u)$ with: $N_i, 0(u) = 1$ if $t_i <= u < t_{i+1}$, 0 otherwise.

13.1.5 Curves with multiple pieces

Often single cubic curves are not enough to represent a complex form. Raising the degree of the curve is possible, but this increases the complexity. Another solution, a solution used in many systems, is to create forms as several pieces of simple curves. Naturally this solution is not without difficulties either, because it is necessary to 'harmonise' the curve pieces.

The term 'harmonise' needs a more precise definition. The notion of harmony is formalised in mathematical terms for curves under the term "continuity". The notion of continuity contains different levels:

0 – Position continuity
1 – Tangency continuity
2 – Curvature continuity
3 – Torsion continuity
etc.

between the curve pieces. These imply constraints on the derivatives of the curve pieces and hence on the positions of various control points.

This is illustrated in the following example. Taking the functions used in the section on Bézier and B-spline curves and differentiating them gives:

$X(t) = t^3 - 3t^2 + 9t - 1$
$Y(t) = 11.5t^3 - 18t^2 + 9t - 1$
$Z(t) = -5t^3 + 9t^2 + 3t$
$X'(t) = 3t^2 - 6t + 9$
$Y'(t) = 34.5t^2 - 36t + 9$
$Z'(t) = -15t^2 + 18t + 3$
$X''(t) = 6t - 6$
$Y''(t) = 69t - 36$
$Z''(t) = -30t + 18$
$X'''(t) = 6$
$Y'''(t) = 69$
$Z'''(t) = -30$

The derivative of a Bézier curve is a new Bézier curve with degree one less and with the control points:

$$\begin{array}{cccc} & & (6,69,-30) & \\ & (-6,-36,18) & & (0,33,-12) \\ (9,9,3) & & (6,-9,12) & & (6,7.5,6) \\ (-1,-1,0) & & (2,2,1) & & (4,-1,5) & & (6,1.5,7) \end{array}$$

The bottom row consists of the control points. The row above that is constructed as the differences between the points in the bottom row multiplied by the degree, three in this case. The next row up is the differences between

the points in this row, multiplied by two, the degree of this row. To construct a continuation to the right with the same characteristics, the right-hand diagonal is copied as the left-hand side of a new pyramid and the missing points constructed:

$$\begin{array}{ccccccc} & & & (6,69,-30) & & & \\ & & (0,33,-12) & & A? & & \\ & (6,7.5,6) & & B? & & C? & \\ (6,1.5,7) & & D? & & E? & & F? \end{array}$$

$D = (6,1.5,7) + (6,7.5,6)/3$; $B = (6,7.5,6) + (0,33,-12)/2$; $E = D + B/3$;
$A = (0,33,-12) + (6,69,-30)$; $C = B + A/2$; $F = E + C/3$;
$A = (6,102,-42)$
$B = (6,24,0)$
$C = (9,75,-21)$
$D = (8,4,9)$
$E = (10,12,9)$
$F = (13,37,2)$

Usually, though, not all elements are needed because this is just a continuation of the same curve. If the curvature is to be preserved, then the third derivative need not be the same and one has:

$$\begin{array}{ccccccc} & & & A? & & & \\ & & (0,33,-12) & & B? & & \\ & (6,7.5,6) & & C? & & D? & \\ (6,1.5,7) & & E? & & F? & & G? \end{array}$$

Here only C, E and F can be determined. In effect this means that the first three control points of the extended curve are fixed; the fourth can be moved to get a curvature continuous but different curve. Similarly, if a tangent continuous extension is required, then the pyramid diagram is:

$$\begin{array}{ccccccc} & & & A? & & & \\ & & B? & & C? & & \\ & (6,7.5,6) & & D? & & E? & \\ (6,1.5,7) & & F? & & G? & & H? \end{array}$$

and only F is determined; the control points G and H are free to be placed elsewhere to get the extension.

Which type of continuity to use is not fixed. In some cases, tangent continuation is enough; in other cases, curvature continuity may be required. Imagine a line connected to a circular arc. Tangent continuity is possible, but curvature continuity would distort the line or the circle.

This, though, concerns the case where the tangent and curvature are the same size for the continuation. It is also possible to have continuity with reduced size elements. The tangent direction of the continuation at $t = 0$ must be the same as the tangent direction of the original curve segment at

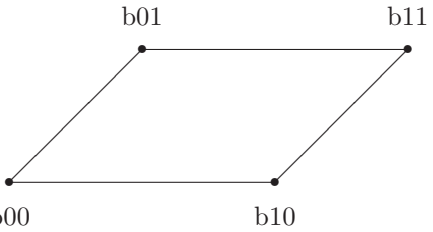


Figure 13.7: Simple linear-linear surface

$t = 1$, but the magnitude may be different. The formula for the curvature of a curve, from Hoschek and Lasser [61], for example, is:

$$\frac{\left|x' X x''\right|}{|x'|^3}$$

So, if the tangent length is halved, the curvature length must be one quarter to preserve the curvature.

13.1.6 Surfaces

Surfaces work in the same way as curves, just in two directions. These are called "tensor product" surfaces. Because of this, instead of one parameter, there are now two, u and v usually. To illustrate this, here is the simplest example, with four control points. To find a point on the surface, you interpolate first in one direction and then in the other. The following is to illustrate this and to show that it does not matter in which direction you interpolate first.

See figure 13.7.

Interpolation u followed by v

Calculate two points on the lines $b00$, $b10$ and $b01$, $b11$ using linear interpolation.

$Pu0 = b00 * (1 - u) + b10 * u$
$Pu1 = b01 * (1 - u) + b11 * u$
$Puv = Pu0 * (1 - v) + Pu1 * v$
$= (b00 * (1 - u) + b10 * u) * (1 - v) + (b01 * (1 - u) + b11 * u) * v$
$= b00 * (1 - u) * (1 - v) + b10 * u * (1 - v) + b01 * (1 - u) * v + b11 * u * v$

Interpolation v followed by u

Calculate two points on the lines $b00$, $b01$ and $b10$, $b11$ using linear interpolation.

$P0v = b00 * (1 - v) + b01 * v$
$P1v = b10 * (1 - v) + b11 * v$

$Puv = P0v * (1-u) + P1v * u$
$= (b00 * (1-v) + b01 * v) * (1-u) + (b10 * (1-v) + b11 * v) * u$
$= b00 * (1-u) * (1-v) + b10 * u * (1-v) + b01 * (1-u) * v + b11 * u * v$

The general formula for Bézier surfaces is:

$$\sum_{i=0}^{n} \sum_{j=0}^{m} \frac{n!}{i!(n-i)!} \frac{m!}{j!(m-j)!} (1-u)^{n-i} u^i (1-v)^{m-j} v^j B_{ij}$$

For the simple surface above, $m = n = 1$ and there are four points of control: B_{00}, B_{10}, B_{01} and B_{11}.

B-splines surfaces are similar but have different basis functions.

Note that the general formula above allows the use of different degrees, i.e., m and n do not need to be the same. So, for example, if m or n has degree 1, then you have a ruled surface. If they are both greater than 1 then you have a general surface. Figure 13.8 shows a simple translational surface ($m = 1$) with the same curve shape on the top and bottom, just offset. Figure 13.9 shows a ruled surface ($m = 1$) where the top and bottom curves are different. Figure 13.10 shows a cubic–quadratic surface ($n = 3, m = 2$). Figure 13.11 shows a cubic–cubic surface ($m = n = 3$).

As with curves, surfaces can be combined so that a face can reference a multi-patch surface. In general, though, these surfaces should join smoothly with at least continuity of curvature. If not, then there is a good case for inserting an edge and a new face.

Surface patches always have four boundaries! Actually this is not true, because there is work on triangular patches; see Farin [35], for example. At the time of writing, though, for practical purposes, surfaces should have four sides.

If a triangular patch is needed, then one way of doing this is to make the control points along one edge coincide. However, this disrupts the surface property calculations and, if possible, should be avoided. The topology of the face accessing the surface may be used to trim the patch into a triangular section.

If there is a region with more than four sides, then it may be necessary to subdivide it into a set of four-sided regions. The classic way of doing this is the method that uses a central point connected to the midpoints of each side of the region. So, instead of one n-sided region, you get n four-sided regions. The shape of these subregions is determined by the position and normal direction of the central point. This provides a degree of freedom to control the shape of these sub-regions.

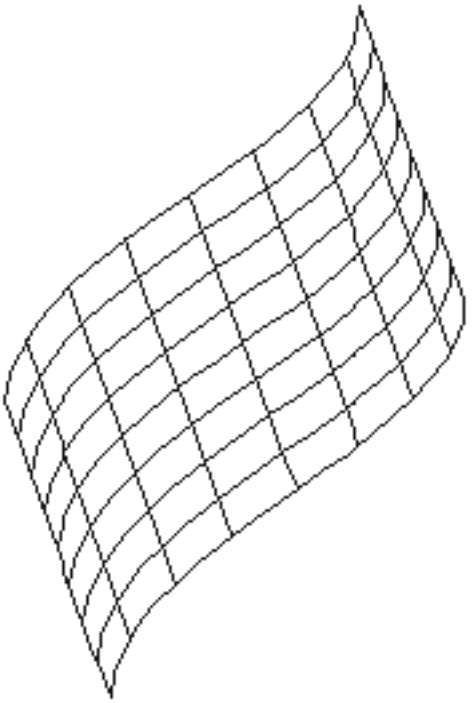


Figure 13.8: Translational surface

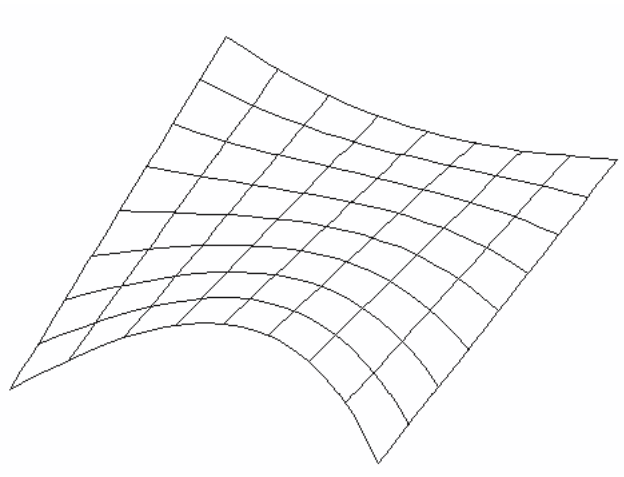


Figure 13.9: Ruled surface

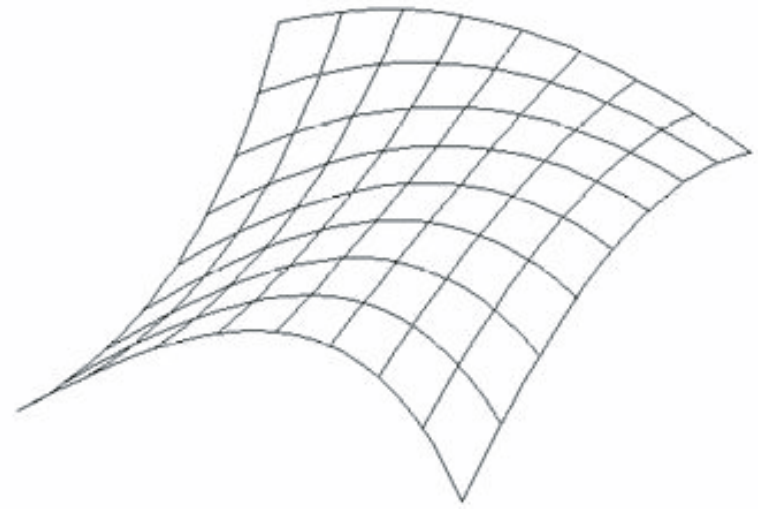


Figure 13.10: Cubic-quadratic surface

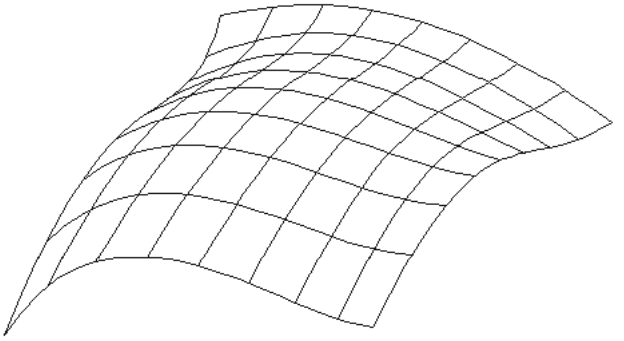


Figure 13.11: Cubic-cubic surface

13.2 Interpolation

13.2.1 Interpolation for a single curve

Interpolation concerns a different kind of geometric construction, where instead of having a known function or known control points, only the points on the curve are known. The examples here use interpolation of four points to give a cubic Bézier; other formats are similar.

For simplicity, the points lie on the plane $Z = 0$, and the four points for the examples are:

(-2,-2) (1,2) (4,-2) (16,3)

The parameter values for the first and last points are 0 and 1, respectively, but, to perform the parametrisation, it is necessary to choose parameter values for the middle points. The number of control points is equal to the number of points through which the curve has to pass. For the above example, a Bézier cubic is needed. This has the form:

$$f(t) = (1-t)^3 b_0 + 3t(1-t)^2 b_1 + 3t^2(1-t)b_2 + t^3 b_3$$

For a Bézier curve, the points b_0 and b_3 are known; they are the same as the first and last given points. Unfortunately there is an infinite number of solutions depending on the values of t_1 and t_2 corresponding to the points (1,2) and (4,-2). With these two values, there are two relationships:

$$(1,2) = (1-t_1)^3 b_0 + 3t_1(1-t_1)^2 b_1 + 3t_1^2(1-t_1)b_2 + t_1^3 b_3$$
$$(4,-2) = (1-t_2)^3 b_0 + 3t_2(1-t_2)^2 b_1 + 3t_2^2(1-t_2)b_2 + t_2^3 b_3$$

With these two expressions, it is possible to find the unknown control points b_1 and b_2.

Interpolation with uniform division in t

In other words, $t_1 = 1/3$ and $t_2 = 2/3$.

Substituting these values into the cubic Bézier equation, using the points (a, b) and (c, d) as the unknown control points, gives:

$$(1-1/3)^3 * (-2,-2) + 3 * 1/3 * (1-1/3)^2 * (a,b) + 3 * (1/3)^2 * (1-1/3) * (c,d) + (1/3)^3 * (16,3) = (1,2)$$
$$(1-2/3)^3 * (-2,-2) + 3 * 2/3 * (1-2/3)^2 * (a,b) + 3 * (2/3)^2 * (1-2/3) * (c,d) + (2/3)^3 * (16,3) = (4,-2)$$

or:

$$0.296296 * (-2,-2) + 0.444444 * (a,b) + 0.222222 * (c,d) + 0.037037 * (16,3) = (1,2)$$
$$0.037037 * (-2,-2) + 0.222222 * (a,b) + 0.444444 * (c,d) + 0.296296 * (16,3) = (4,-2)$$

Figure 13.12: Interpolation with uniform control points

Note that the sum of the weighting factors equals one; that is:
$0.296296 + 0.444444 + 0.222222 + 0.037037 = 1$

Rearranging the equations gives:
$-0.592593 + 0.444444a + 0.222222c + 0.592593 = 1,$
$0.444444a + 0.222222c = 1$
$-0.074074 + 0.222222a + 0.444444c + 4.740741 = 4,$
$0.222222a + 0.444444c = -0.666667$

$-0.592593 + 0.444444b + 0.222222d + 0.111111 = 2,$
$0.444444b + 0.222222d = 2.481481$
$-0.074074 + 0.222222b + 0.444444d + 0.888889 = -2,$
$0.222222b + 0.444444d = -2.814815$

Solving the equations gives:
$a = 4, b = 11.66667, c = -3.5, d = -12.166667.$

The resulting curve with the points of control

$(-2, -2)$ $(4, 11.6667)$ $(-3.5, -12.16667)$ $(16, 3)$

is shown in figure 13.12.

Interpolation with arc length

Here, the values t_1 and t_2 are calculated from the distances between the given points. In the example, the distances are 5, 5, and 13. The total length of the curve is estimated as $5 + 5 + 13 = 23$, and so the values of t_1 and t_2 are estimated as $t_1 = 5/23$ and $t_2 = 10/23$. Using the same method as before, the equations are:

$(1 - 5/23)^3 * (-2, -2) + 3 * (5/23) * (1 - 5/23)^2 * (a, b) + 3 * (5/23)^2 * (1 - 5/23) * (c, d) + (5/23)^3 * (16, 3) = (1, 2)$

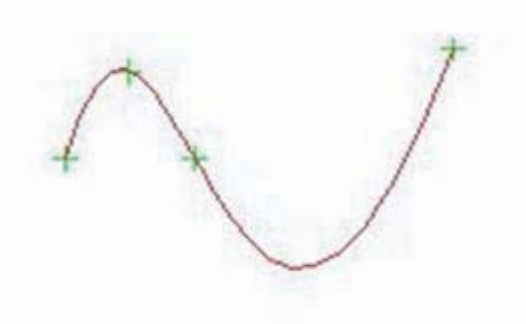


Figure 13.13: Interpolation with arc-length

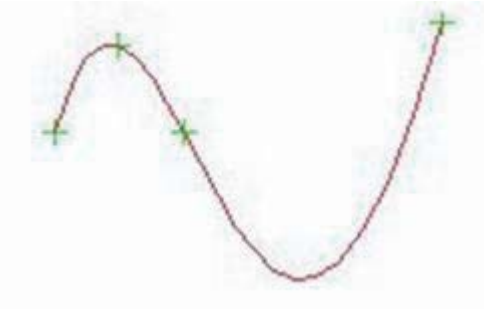


Figure 13.14: Interpolation with $t_1 = 0.2$ and $t_2 = 0.4$

$(1-10/23)^3 * (-2,-2) + 3 * (10/23) * (1-10/23)^2 * (a,b) + 3 * (10/23)^2 * (1-10/23) * (c,d) + (10/23)3 * (16,3) = (4,-2)$

or:

$0.479329*(-2,-2)+0.399441*(a,b)+0.110956*(c,d)+0.010274*(16,3) = (1,2)$
$0.180570 * (-2,-2) + 0.416701 * (a,b) + 0.320539 * (c,d) + 0.082190 * (16,3) = (4,-2)$

The resulting curve, with the control points:
$(-2,-2)$ $(2.8991, 14.0302)$ $(5.7342, -24.1213)$ $(16,3)$
is shown in figure 13.13.

Interpolation with $t_1 = 0.2$, $t_2 = 0.4$

The parameter values are, arbitrarily: $t_1 = 0.2$ and $t_2 = 0.4$.

The rest is left as an exercise. The resulting curve is similar to the previous example, but the figures for the calculation are easier. See figure 13.14.

Interpolation with $t_1 = 0.1$, $t_2 = 0.2$

Again, with arbitrary parameter values of $t_1 = 0.1$ and $t_2 = 0.2$:

$(1-0.1)^3 * (-2,-2) + 3 * 0.1 * (1-0.1)^2 * (a,b) + 3 * (0.1)^2 * (1-0.1) * (c,d) + (0.1)^3 * (16,3) = (1,2)$
$(1-0.2)^3 * (-2,-2) + 3 * 0.2 * (1-0.2)^2 * (a,b) + 3 * (0.2)^2 * (1-0.2) * (c,d) + (0.2)^3 * (16,3) = (1,2)$

or:

$0.729 * (-2,-2) + 0.243 * (a,b) + 0.027 * (c,d) + 0.001 * (16,3) = (1,2)$
$0.512 * (-2,-2) + 0.384 * (a,b) + 0.096 * (c,d) + 0.008 * (16,3) = (4,-2)$

In the result, it is 'obvious' that these values are too small because they give an extended curve, but it is still a valid interpolation.

The resulting curve, with the points of control $(-2,-2)$ $(7.8889, 27.6759)$ $(19.4444, -121.1204)$ $(16,3)$, is shown in figure 13.15.

Interpolation with $t_1 = 0.25$, $t_2 = 0.75$

More arbitrary parameter values: $t_1 = 0.25$ and $t_2 = 0.75$.

The resulting curve with the points of control: $(-2,-2)$ $(6.6667, 10.3333)$ $(-8.6667, -11.1111)$ $(16,3)$ is shown in figure 13.16.

Interpolation with $t_1 = 0.6$, $t_2 = 0.3$

Yet another example with arbitrary parameter values of $t_1 = 0.6$ and $t_2 = 0.3$, which, in effect, means that the order of the interpolated points is reversed.

The resulting curve with the points of control: $(-2,-2)$ $(16.7381, -6.4841)$ $(-16.5476, 7.7487)$ $(16,3)$ is shown in figure 13.17.

The whole set of curves in the examples above is shown superimposed in figure 13.18, but an infinity of solutions is possible.

Interpolation with a matrix

It is also possible to formulate the problem in terms of a matrix, which gives a method of solution more tractable for a computer. Formulating the problem for a Bézier curve in matrix terms gives:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ (1-t_1)^3 & 3t_1(1-t_1)^2 & 3t_1^2(1-t_1) & t_1^3 \\ (1-t_2)^3 & 3t_2(1-t_2)^2 & 3t_2^2(1-t_2) & t_2^3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

where B_0, B_1, B_2, and B_3 are the unknown control points and P_0, P_1, P_2, and P_3 are the points to interpolate. In order to find the points of control it is necessary to multiply by the inverse, giving:

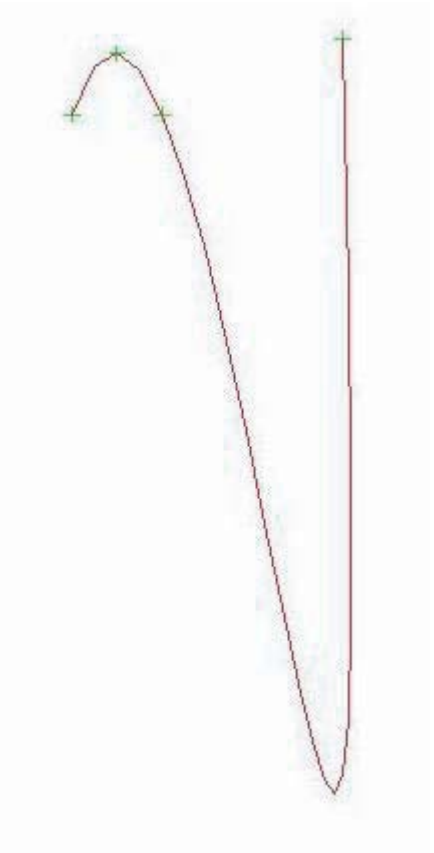


Figure 13.15: Interpolation with $t_1 = 0.1$ and $t_2 = 0.2$

Figure 13.16: Interpolation with $t_1 = 0.25$ and $t_2 = 0.75$

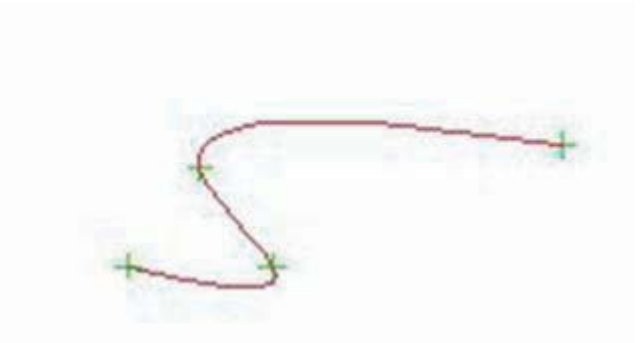


Figure 13.17: Interpolation with $t_1 = 0.6$ and $t_2 = 0.3$

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

where the 4×4 matrix is the inverse of the matrix formed from the Bézier basis functions. For the Bézier curve case above it is possible to simplify the inverse directly because $a = 1$, $b = c = d = 0$ and also $m = n = o = 0$ and $p = 1$. Similarly f, g, j and k are simple to write down because they are just the inverse of the central 2 portion of the matrix, giving the simplified matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ e & \frac{3}{q}t_2^2(1-t_2) & \frac{-3}{q}t_1^2(1-t_1) & h \\ i & \frac{-3}{q}t_2(1-t_2)^2 & \frac{3}{q}t_1(1-t_1)^2 & l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

where $q = 9(t_1t_2(1-t_1)(1-t_2))(t_2(1-t_1) - t_1(1-t_2))$.

The final values can be determined easily. Using the arc-length values, that is, $t_1 = 5/23$ and $t_2 = 10/23$ gives:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.479329 & 0.399441 & 0.110956 & 0.010274 \\ 0.180570 & 0.416701 & 0.320539 & 0.082190 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} (-2,-2) \\ (1,2) \\ (4,-2) \\ (16,3) \end{bmatrix}$$

or:

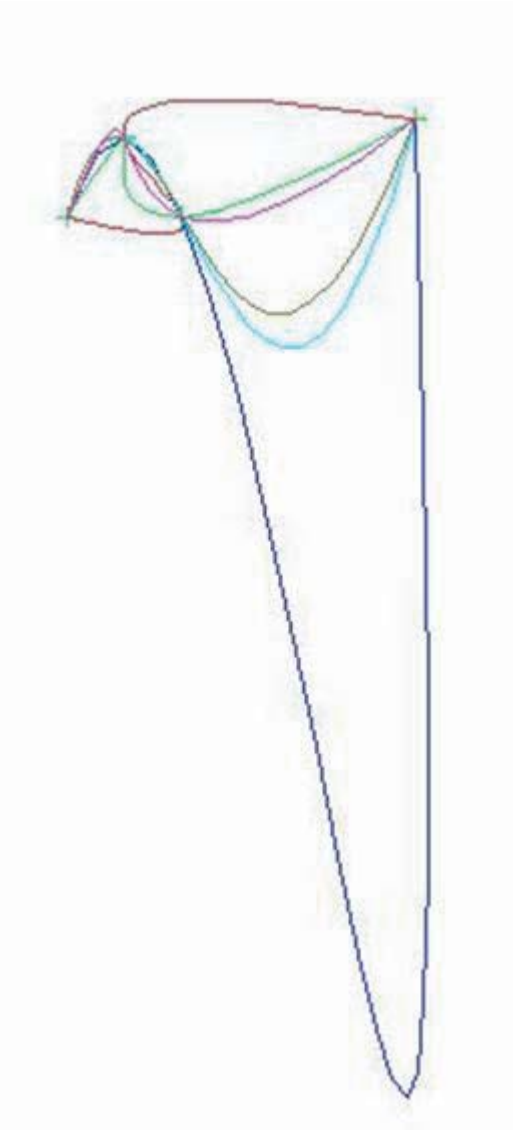


Figure 13.18: Combined image of interpolated curves

$$\begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ -1.633333 & 3.918519 & -1.356410 & 0.071225 \\ 1.56 & -5.094074 & 4.883077 & -0.349003 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} (-2,-2) \\ (1,2) \\ (4,-2) \\ (16,3) \end{bmatrix}$$

which gives the control points:
$(-2,-2), (2.899146, 14.030200), (5.734186, -24.121311), (16,3)$

13.2.2 Interpolation with a point and a tangent vector

An alternative to using just points is to determine the control points using a combination of points and tangents. Each imposed tangent adds a control point, so a cubic can be defined using three points and a tangent vector condition, for example. To do this, it is necessary to use the derivative of the curve, for a Bézier this means:

$b(t) = (1-t)^3 b_0 + 3t(1-t)^2 b_1 + 3t^2(1-t)b_2 + t^3 b_3$
$b'(t) = -3(1-t)^2 b_0 + 3((1-t)^2 - 2t(1-t))b_1 + 3(2t(1-t) - t^2)b_2 + 3t^2 b_3$

For a b-spline the equivalent is:

$b(t) = b_0(1 - 3t + 3t^2 - t^3)/6 + b_1(1 + 3t + 3t^2 - 3t^3)/6 + b_2(4 - 6t^2 + 3t^3)/6 + b_3 t^3/6$,
$b'(t) = b_0(-1 + 2t - t^2)/2 + b_1(1 + 2t - 3t^2)/2 + b_2(-4t + 3t^2)/2 + b_3 t^2/2$

Manual interpolation

If you have three known points: $(-3,0)$ $(0,-4)$ $(6,4)$ and the tangent vector at $(0,-4)$ is $(8,0)$, then it is possible to create two systems of equations to find the unknown control points. Here we will use the method of curve-length estimation to find the value of t corresponding to the given point; that is, $t = 1/3$. Substituting this in the Bézier equations gives:

$(1-1/3)^3 * (-3,0) + 3 * 1/3 * (1-1/3)^2 * (a,b) + 3 * (1/3)^2 * (1-1/3) * (c,d) + (1/3)^3 * (6,4) = (0,-4)$
$-3 * (1-1/3)^2 * (-3,0) + 3 * ((1-1/3)^2 - 2 * 1/3 * (1-1/3)) * (a,b) + 3 * (2 * 1/3 * (1-1/3) - (1/3)^2) * (c,d) + 3 * (1/3)^2 * (6,4) = (4,0)$
Or,
$0.296296 * (-3,0) + 0.444444 * (a,b) + 0.222222 * (c,d) + 0.037037 * (6,4) = (0,-4)$
$-1.333333 * (-3,0) + 0 * (a,b) + 1 * (c,d) + 0.333333 * (6,4) = (8,0)$

Which gives the values of c and d directly, which fixes the control point b_2 .

$c = 2$ and $d = -1.333333$

From the first equation you get:

$0.444444 * a = 1.1111111$, or $a = 0.5$ and $0.444444 * b = -3.851852$, or $b = -8.666667$

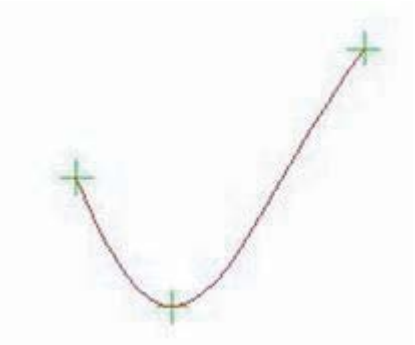


Figure 13.19: Interpolation with point and normal vector

The points of control are:
$(-3,0)(0.5,-8.6666667)(2,-1.3333333)(6,4)$
The resulting curve is shown in figure 13.19.

Matrix interpolation with tangent vectors

An alternative is, again, to construct the matrix solution, which gives:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ (1-t)^3 & 3t(1-t)^2 & 3t^2(1-t) & t^3 \\ -3(1-t)^2 & 3((1-t)^2-2t(1-t) & 3(2t(1-t)-t^2) & 3t^2 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} P_0 \\ P_1 \\ tv \\ P_2 \end{bmatrix}$$

where tv is the given tangent vector. Substituting the values gives:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 8/27 & 12/27 & 6/27 & 1/27 \\ -12/9 & 3(4/9-4/9) & 3(4/9-1/9) & 3/9 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} (-3,0) \\ (0,-4) \\ (8,0) \\ (6,4) \end{bmatrix}$$

The main matrix is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 8/27 & 12/27 & 6/27 & 1/27 \\ -4/3 & 0 & 1 & 1/3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and the inverse is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -4/3 & 9/4 & -1/2 & 1/12 \\ 4/3 & 0 & 1 & -1/3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Multiplying this with the given points and tangent vector gives the points of control:

$(-3, 0)(0.5, -8.6666667)(2, -1.333333)(6, 4)$

13.2.3 Interpolation with compound curves

For this part, the same four points are used as before:

$$(-2, -2)(1, 2)(4, -2)(16, 3)$$

With this type of interpolation, it is necessary to use the property that a curve passes through the first and last control points. Instead of using just one curve, however, three smaller curve pieces are used, the first between $(-2, 2)$ and $(1, 2)$, the second between $(1, 2)$ and $(4, -2)$ and the third between $(4, -2)$ and $(6, 3)$. To create this compound curve it is necessary to use continuity properties. There is already C0 continuity between the curves because they share common points, but C1 or C2 continuity is also needed.

The interpolation problem can be formulated in this way, i.e., as a sequence of three curves with the points of control:
$(-2, -2)(a, b)(c, d)(1, 2)$
$(1, 2)(e, f)(g, h)(4, -2)$
$(4, -2)(i, j)(k, l)(16, 3)$
where the vectors $(a, b), (c, d)(e, f)(g, h)(i, j)$ and (k, l) are the unknown points of control. To have three curve pieces with tangency continuity, you have the relationship:

$(e, f) - (1, 2) = m * ((1, 2) - (c, d))$ and
$(i, j) - (4, -2) = n * ((4, -2) - (g, h))$

The values of m and n are constants. The values of the vectors (a, b) and (k, l) are arbitrary and are a degree of freedom for shape control. For the internal points it is possible to choose the vector differences between the neighbouring points as direction. For the example given:
Tangent vector at $(1, 2) = (4, -2) - (-2, -2) = (6, 0)$
Normalised tangent vector: $(1, 0)$.
Tangent vector at $(4, -2) = (16, 3) - (1, 2) = (15, 1)$
Normalised tangent vector: $(0.997785, 0.066519)$

As an example, the first and last control points $(a, b) = (-2, 0)$ and $(k, l) = (16, 1)$ are chosen. Using the normalised vectors for the internal points gives the following three curves:

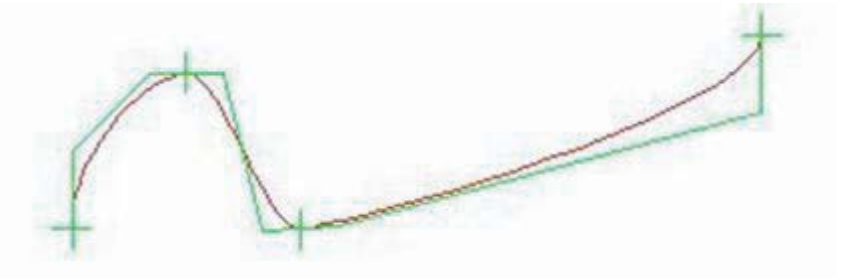


Figure 13.20: Interpolation with multiple curve pieces

Figure 13.21: Interpolation with multiple curve pieces, again

$(-2,-2)(-2,1)(0,2)(1,2)$
$(1,2)(2,2)(3.002215,-2.066519)(4,-2)$
$(4,-2)(4.997785,-1.933411)(16,1)(16,3)$

See figure 13.20.

Changing the scale of the tangent vectors, for example, by 1.3, gives the following points of control:

$(-2,-2)(-2,1)(-0.3,2)(1,2)$
$(1,2)(2.3,2)(2.702879,-2.086475)(4,-2)$
$(4,-2)(5.297121,-1.913525)(16,1)(16,3)$

See figure 13.21.

It is also possible to vary the values of the tangent vectors to produce curves in accordance with the length of each piece of curve. Looking at the lengths by using the differences between the points to be interpolated as an estimate gives the lengths, as before, as 5, 5, and 13. It is possible to assign values at $(1,2)$ and at $(4,-2)$ according to their positions on the curve to be constructed, that is, that $(1,2)$ has the value $5/13$ and $(1,2)$ the value $10/23$, as for the arc-length interpolation.

Using these values, it is possible to define control points in relation to the length of each curve piece, giving, for example:

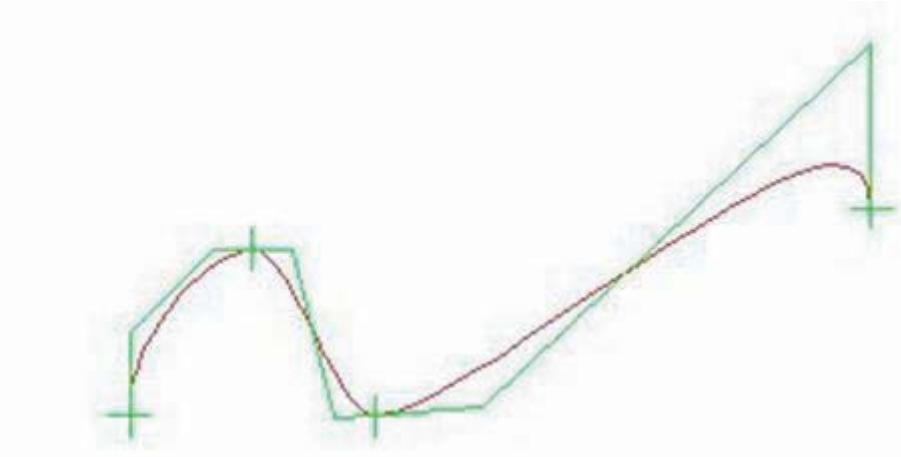


Figure 13.22: Interpolation with multiple curve pieces for the third and last time

$(-2,-2)(-2,1)(-0.3,2)(1,2)$
$(1,2)(2.3,2)(2.702879,-2.086475)(4,-2)$
$(4,-2)(6.594241,-1.827951)(16,7)(16,3)$

Figure 13.22.

The curve also shows a changed tangent vector at $(16,3)$.

While keeping the tangent vectors aligned, it is possible to vary their respective sizes. Hoschek and Lasser, for example, name three possible schemes for the weighting factors, due to Bessel, Akima, Renner/Pochop.

13.3 Lofting

Lofting is an operation that is topologically simple but geometrically complex.

For the topological changes, see figure 13.23. If one or both sections are new, that is, if lofting is not between existing solids, then the new section is converted into a sheet object. Then, one of the opposing faces (which face towards the interior of the lofted part), is made into an inner contour, or hole-loop of the other face, as in figure 13.23a. Matching vertices are then connected to give the topology of the lofted part (figure 13.23c). If the topology of the sections does not match, as in figure 13.23b, then it is necessary to adjust one of them so that they have the same number of edges and vertices and then connect them (figure 13.23d).

This process of matching is not always simple. Consider trying to match a circle to a square, for example. Matching also applies to every section in the sequence; each one should have the same topology. One way of doing this is to consider the boundary of each section as a continuous sequence parametrised

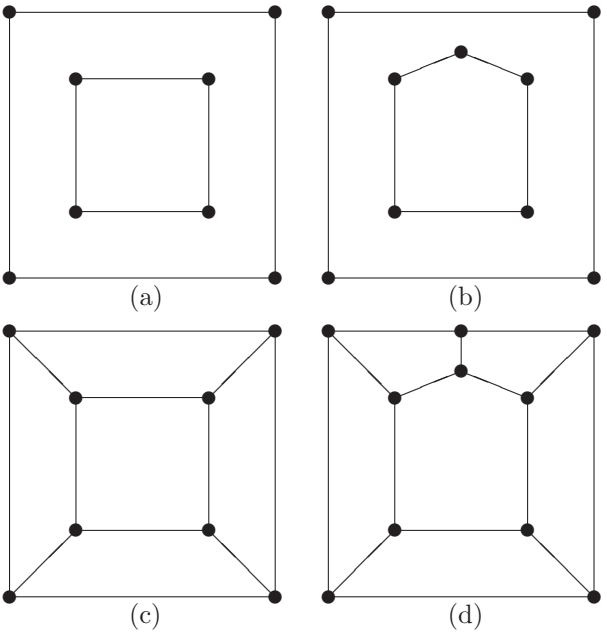


Figure 13.23: Topological changes for lofting

from 0 to 1, say. The parameter values of the vertices are then matched, and new vertices are inserted to minimise the discrepancies. The first thing to note is the importance of matching the start vertices. Another important element is that the direction of parametrisation is the same for each section. Sometimes this can only be done with user assistance, maybe to adjust the topology of the sections manually and to help pick matching starting points and directions. This cannot really be done automatically. Suppose that a twisted object such as that in figure 13.24 is what is wanted. The shape is made by matching the top edge of the first section, the left-hand edge of the second section, the bottom edge of the third section, and the right-hand edge of the last section. All sections are the same shape and size, and an automatic method would not be able to tell what the user really wants.

Assuming that the matching process has been done successfully, then for each edge in the sections, a series of matching edges defines the surfaces of the lofted part. Consider figure 13.25.

The first section, section 1 at the left of the figure, has five edges, and the other three each have four. Because of the placement of the extra vertices, the top edges of the three other sections should be split. The next section, section 2, has a cubic Bézier as the top edge. The third section is simply rectangular, and the fourth section has a symmetric quadratic Bézier as the top edge.

The corner points of each of these sections are:

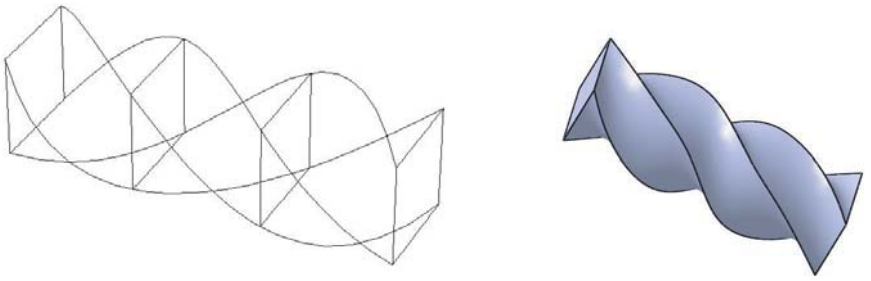


Figure 13.24: Twisted lofted object

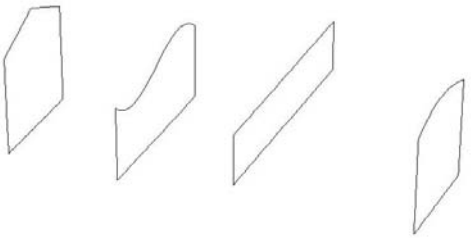


Figure 13.25: Multi-section lofting example

Section 1
$(-75, -20, 20)$ $(-75, -20, -20)$ $(-75, 20, -20)$ $(-75, 20, 20)$ $(-75, 0, 30)$

Section 2
$(-25, -30, 20)$ $(-25, -30, -10)$ $(-25, 30, -10)$ $(-25, 30, 20)$
The top cubic Bézier of section 2 has control points: $(-25, 30, 20)$ $(-25, 10, 40)$ $(-25, -10, 0)$ $(-25, -30, 20)$

Section 3
$(25, -40, 30)$ $(25, -40, 10)$ $(25, 40, 10)$ $(25, 40, 30)$

Section 4
$(75, 0, 20)$ $(75, 0, -20)$ $(-25, 40, -20)$ $(-25, 40, 20)$
The top quadratic Bézier of section 4 has the control points: $(75, 40, 20)$ $(75, 20, 30)$ $(75, 0, 20)$.

To match the topology of the sections, it is necessary to split the top edge of sections 2, 3, and 4. For the cubic Bézier, this is done using de Casteljau's method:

$$
\begin{array}{cccc}
 & & (-25,0,20) & \\
 & (-25,10,25) & & (-25,-10,15) \\
(-25,20,30) & & (-25,0,20) & & (-25,-20,10) \\
(-25,30,20) & (-25,10,40) & & (-25,-10,0) & & (-25,-30,20)
\end{array}
$$

This gives two half curves with the control points:
$(-25,30,20)$ $(-25,20,30)$ $(-25,10,25)$ $(-25,0,20)$

and

$(-25,0,20)$ $(-25,-10,15)$ $(-25,-20,10)$ $(-25,-30,20)$

As the top curve of the second section is a cubic curve, then all matching edges have to be cubics. For the first section, this gives the control points:

$(-75,20,20)$ $(-75,40/3,70/3)$ $(-75,20/3,80/3)$ $(-75,0,30)$

and

$(-75,0,30)$ $(-75,-20/3,80/3)$ $(-75,-40/3,70/3)$ $(-75,-20,20)$

For the third section, this is also easy, giving:

$(25,40,30)$ $(25,80/3,30)$ $(25,40/3,30)$ $(25,0,30)$

and

$(25,0,30)$ $(25,-40/3,30)$ $(25,-80/3,30)$ $(25,-40,30)$

For section 4, the top curve is a quadratic Bézier, which means that there are two options. The first option is to raise the degree of the curve and then divide it, which gives:

$(75,40,20)$ $(75,20,30)$ $(75,0,20)$

goes to

$(75,40,20)$ $(75,100/3,70/3)$ $(75,40/3,70/3)$ $(75,0,20)$,

and then the division gives:

$$
\begin{array}{cccc}
 & & (75,20,25) & \\
 & (75,80/3,25) & & (75,40/3,25) \\
(75,100/3,70/3) & & (75,20,80/3) & & (75,20/3,70/3) \\
(75,40,20) & (75,80/3,80/3) & & (75,40/3,80/3) & & (75,0,20)
\end{array}
$$

and hence two sub-curves:

$(75,40,20)$ $(75,100/3,70/3)$ $(75,80/3,25)$ $(75,20,25)$

and

$(75,20,25)$ $(75,40/3,25)$ $(75,20/3,70/3)$ $(75,0,20)$

The second way is to subdivide the curve and then to raise the degree of the two halves, which gives:

$$(75, 20, 25)$$
$$(75, 30, 25) \quad (75, 10, 25)$$
$$(75, 40, 20) \quad (75, 20, 30) \quad (75, 0, 20)$$

$(75, 40, 20)$ $(75, 30, 25)$ $(75, 20, 25)$ goes to
$(75, 40, 20)$ $(75, 100/3, 70/3)$ $(75, 80/3, 25)$ $(75, 20, 25)$
and
$(75, 20, 25)$ $(75, 10, 25)$ $(75, 0, 20)$

goes to

$(75, 20, 25)$ $(75, 40/3, 25)$ $(75, 20/3, 70/3)$ $(75, 0, 20)$

Surprise, surprise, the results are the same.

The corner points of the sections are:

$$(-75, -20, 20)(-75, -20, -20)(-75, 20, -20)(-75, 20, 20)(-75, 0, 30)$$
$$(-25, -30, 20)(-25, -30, -10)(-25, 30, -10)(-25, 30, 20)(-25, 0, 20)$$
$$(25, -40, 30)(25, -40, 10)(25, 40, 10)(25, 40, 30)(25, 0, 30)$$
$$(75, 0, 20)(75, 0, -20)(75, 40, -20)(75, 40, 20)(75, 20, 25)$$

Interpolating the corner points vertically gives five curves:

$(-75, -20, 20)$ $(-22.21, -19.0, 6.20)$ $(41.45, -77.04, 51.60)$ $(75, 0, 20)$

$(-75, -20, -20)$ $(-18.84, -20.82, -28.21)$ $(43.52, -75.76, 54.80)$ $(75, 0, -20)$

$(-75, 20, -20)$ $(-20.79, 28.04, -31.41)$ $(29.46, 48.52, 53.91)$ $(75, 40, -20)$

$(-75, 20, 20)$ $(-23.59, 26.95, 5.23)$ $(23.59, 48.10, 49.81)$ $(75, 40, 20)$

$(-75, 0, 30)$ $(-24.91, 6.10, -1.83)$ $(29.42, -15.81, 50.02)$ $(75, 20, 25)$

See figure 13.26.

Three of the bounding surfaces, for the left-hand side, the bottom, and the right-hand side, are cubic-linear surfaces, or ruled surfaces, so the control points are determined by the bounding curves. The remaining two surfaces, on the top of the object, are cubic–cubic patches. The bounding curves give 12 control points, leaving four to be determined; thus:

$$(-75, -20, 20)(-75, -40/3, 70/3)(-75, -20/3, 80/3)(-75, 0, 30)$$
$$(-22.21, -19.0, 6.2)(a, b, c)(d, e, f)(-24.9, 6.1, -1.8)$$
$$(41.5, -77.0, 51.6)(g, h, i)(j, k, l)(29.4, -15.8, 50.0)$$
$$(75, 0, 20)(75, 20/3, 70/3)(75, 40/3, 25)(75, 20, 25)$$

The remaining four points are found by interpolating the sets of second and third control points for each section. That is, interpolating the points:

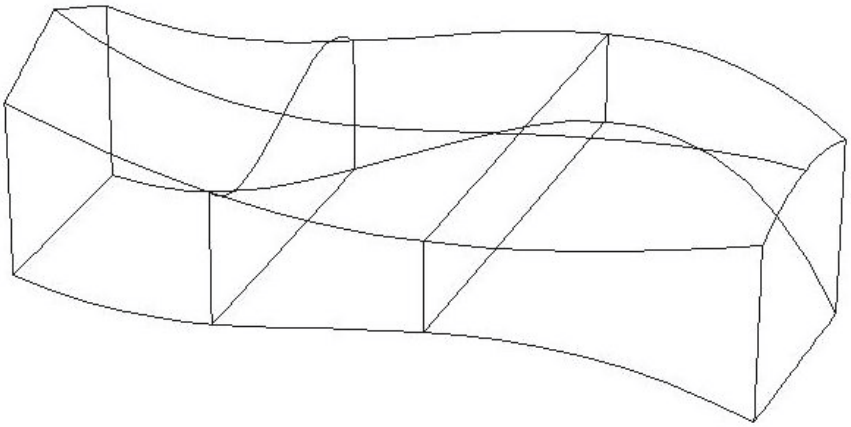


Figure 13.26: Lofting example — sections and interpolated curves

$(-75, -40/3, 70/3)$ $(-25, -20, 10)$ $(25, -80/3, 30)$ $(75, 20/3, 70/3)$
and
$(-75, -20/3, 80/3)$ $(-25, -10, 15)$ $(25, 80/3, 30)$ $(75, 40/3, 25)$

Interpolating these two gives curves with control points:
$(-75, -13.33, 23.33)$ $(-21.56, -9.78, -26.49)$ $(32.87, -57.25, 66.72)$
$(75, 6.67, 23.33)$
and
$(-75, -6.67, 26.67)$ $(-9.89, -51.35, -11.05)$ $(10.58, 73.98, 52.44)$ $(75, 13.33, 25)$

This means that $(a, b, c) = (-21.6, -9.8, -26.5)$
$(d, e, f) = (-19.2, 14.5, -14.0)$
$(g, h, i) = (32.9, -57.2, 66.7)$
and $(j, k, l) = (36.3, -76.1, 59.7)$ and the control points for the surface are:

$$(-75, -20, 20)(-75, -40/3, 70/3)(-75, -20/3, 80/3)(-75, 0, 30)$$
$$(-22.2, -19.0, 6.2)(-21.6, -9.8, -26.5)(-19.2, 14.5, -14.0)(-24.9, 6.1, -1.8)$$
$$(41.4, -77.0, 51.6)(32.9, -57.2, 66.7)(36.3, -76.1, 59.7)(29.4, -15.8, 50.0)$$
$$(75, 0, 20)(75, 20/3, 70/3)(75, 40/3, 25)(75, 20, 25)$$

See figure 13.27.

Doing the same for the other side gives the surface

$$(-75, 0, 30)(-75, 6.7, 26.7)(-75, 13.3, 23.3)(-75, 20, 20)$$
$$(-24.9, 6.1, -1.8)(-24.6, 13.2, 16.2)(-24.6, 13.2, 16.2)(-23.6, 26.9, 5.2)$$
$$(29.4, -15.8, 50.0)(27.4, 5.4, 40.7)(25.7, 26.8, 33.3)(23.6, 48.1, 49.8)$$
$$(75, 20, 25)(75, 26.7, 25)(75, 33.3, 23.3)(75, 40, 20)$$

See figure 13.28.

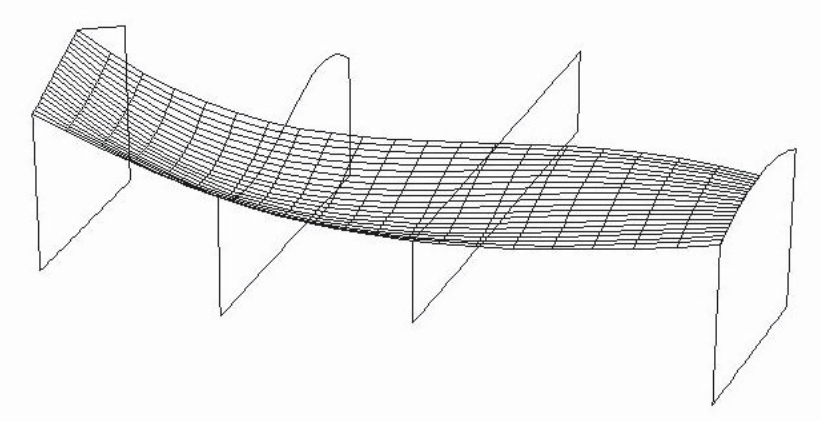


Figure 13.27: Lofting example — sections and first top surface

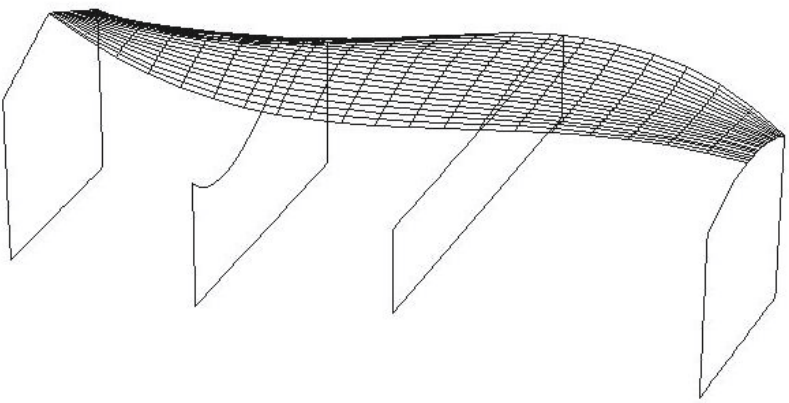


Figure 13.28: Lofting example — sections and second top surface

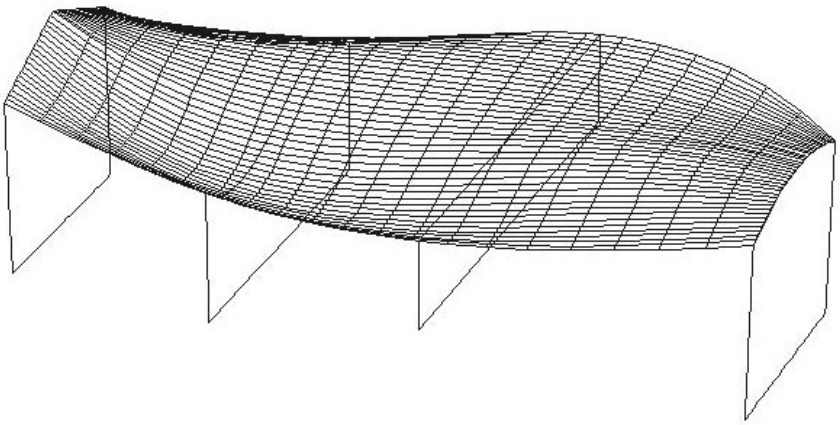


Figure 13.29: Lofting example — Top surfaces

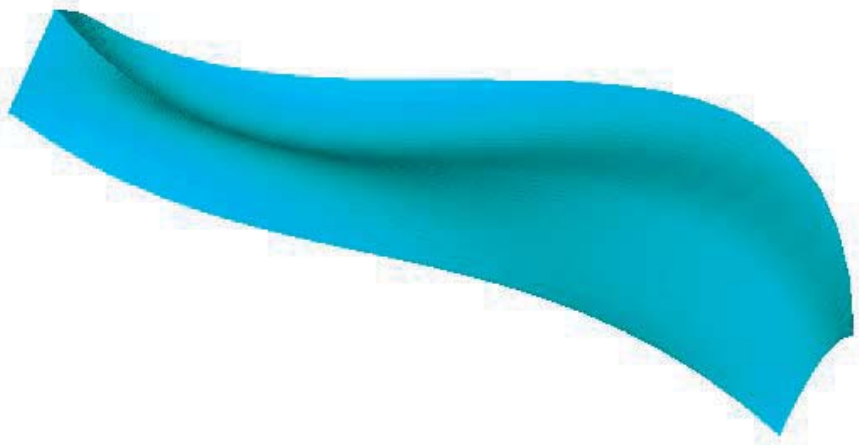


Figure 13.30: Lofting example — Top surfaces – shaded

The two surfaces together are shown in figures 13.29 and 13.30.

Naturally it is also possible to interpolate the sections with a set of curves and surfaces to keep the degree low.

13.4 Adding free-form surfaces to a face

In the simplest case a single surface patch has to be set into a face. This can be done using the SETSURF algorithm developed originally by Graham Jared.

13.4.1 SETSURF

The SETSURF algorithm retains the topology of the model and resets the geometry to be consistent. The algorithm is general and works with any surface, subject to the restrictions mentioned below.

The algorithm works by traversing all edges surrounding the face, intersecting the surface of the face on the other side of the given face. This is described in section 6.11 and illustrated in figure 13.31. The face in which the new surface is to be set is f0.

The new positions for vertices v1, v2, v3, and v4 are found by intersecting the edges e5, e6, e7, and e8, respectively, with the new surface. See figure 13.32.

Starting with edge e1, the new curve of e1 is obtained by intersecting the surface of f1 with the new surface. Next, the surface of f2 is intersected with the new surface to give the new curve for edge e2. The surface of f3 is intersected with the new surface to give the curve for edge e3, and finally, the curve for edge e4 is found by intersecting the surface of s4 with the new surface. See figure 13.33.

This algorithm is straightforward and correct, but there are several potential problems with the application. The algorithm works well if the surface being set into the face is not too curved and slightly larger than the face. The surface also has to be oriented consistently with the rest of the geometry. If any intersection fails to produce a result, then the whole process stops and the original geometry is reset.

The problem with this tool is not the correctness of the technique, but the appropriateness for the introduction of free-form surfaces into a model. If the free-form surface has been designed separately, then there is no guarantee that it will be exactly adapted to a face in the model in which it is to be set. The next two sections describe alternative strategies for comparison.

13.4.2 Multi-patch SETSURF

A generalisation of this technique is needed if the surface to be set into a face is a multipatch surface. Here the surface has a more complex topology

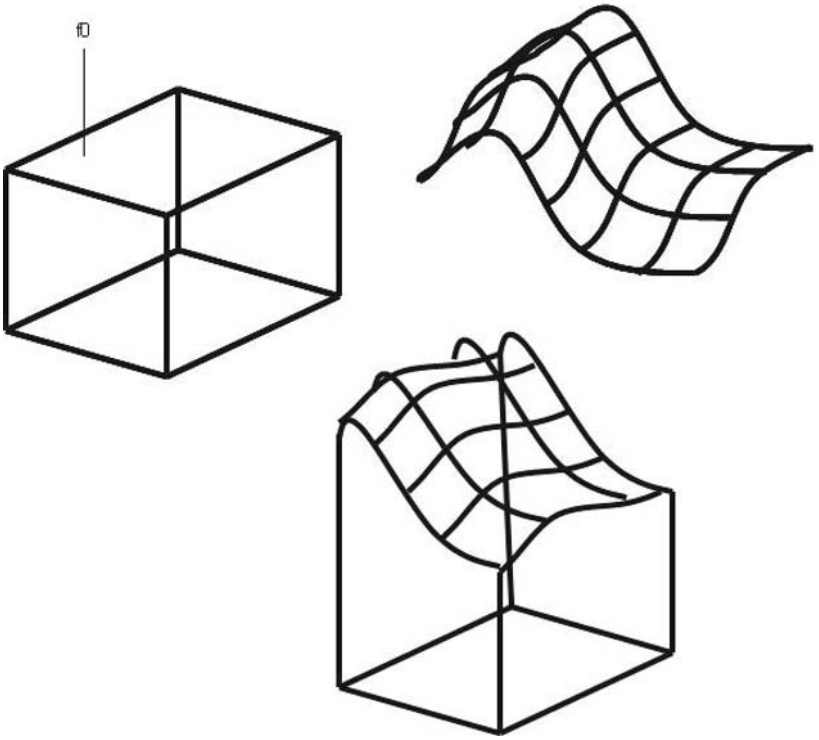


Figure 13.31: Illustration of SETSURF

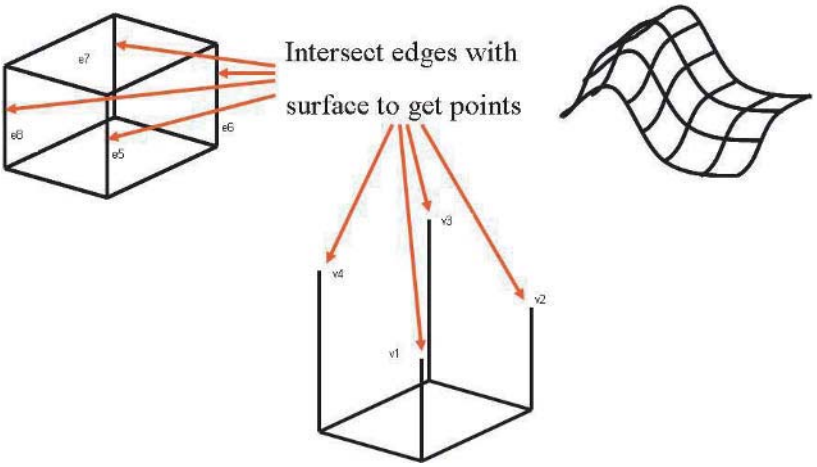


Figure 13.32: Illustration of SETSURF point calculation

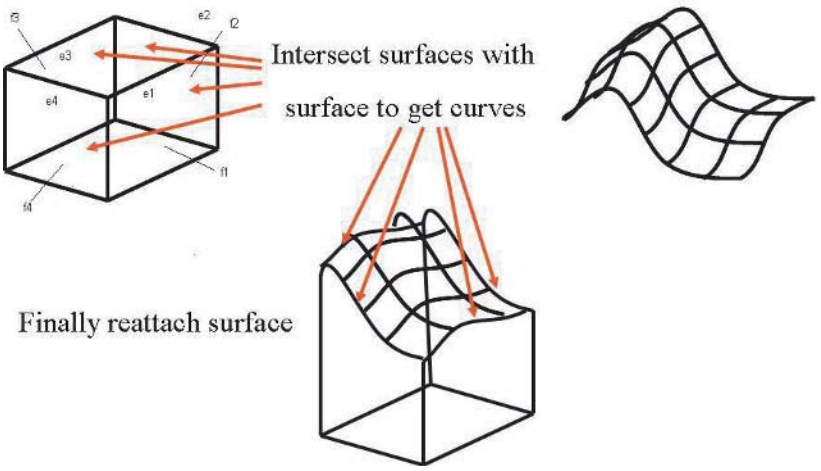


Figure 13.33: Illustration of SETSURF curve calculation

and intersection results may span more than one surface. A slight variation on the SETSURF algorithm, related to the notion of imprinting mentioned in section 6.9, can be used to create the appropriate result. (The following algorithm has not been implemented.)

The first step is to create a partial object (see chapter 5 and section 13.5) from the surface patches. The face in which the surface patches are to be set is then imprinted on the partial object, and where the imprinted image of an edge crosses one of the patch boundaries, that edge is split by inserting a new vertex. The result is that the topology of the face matches the imprinted image topology in the partial object. The edges and vertices of the imprinted boundary are then substituted for the boundary edges and vertices of the face to create the final object. This is illustrated in figure 13.34.

If the surface does not entirely cover the face into which it is to be set, then it is possible to trim the face or faces by imprinting the relevant free-form surface boundary into the face and then building one or more side faces in the same manner as described in section 3.1.3. However, there is always the question of how general should be the command. Should the command really try and cope with any case? Or should the operation be made to fail and the user asked to sort out problems?

13.4.3 Embossing a free-form surface

Another untried technique is related to the imprinting operation described in section 6.9 involving projecting the boundaries of the free-form surface back to the object to create a shaped boss.

A necessary piece of information is the projection direction, which may be the normal direction of the face if the surface is to be projected down onto

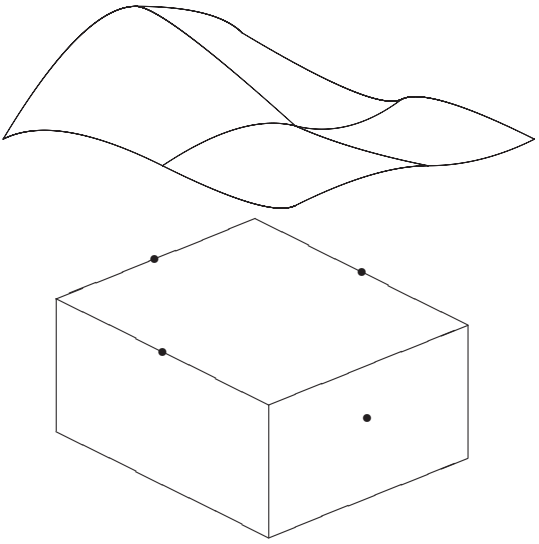


Figure 13.34: Inserting a multi-patch surface into a face

a single face, or arbitrary; in which case, it is necessary to search for intersections. The direction is used to generate swept surfaces from the boundary of the surface. These surfaces are then intersected with the face (or body) in the same way as for imprinting. The imprinted boundary is then connected to the topological structure created from the surface to create the side faces and the boss is complete. This is illustrated in figure 13.35.

This approach could work reasonably enough if applied to free-form surfaces that are not too curved and that can be projected onto the object. If the boundaries of the surface are too highly curved, then the swept surfaces may be self-intersecting. Figure 13.36 summarises the topological changes to create the object.

The original face is shown in figure 13.36a. For simplicity, it is shown as four-sided, but the topology and geometry could be more general. A four-sided inner contour is added, as shown in figure 13.36b. This contour is four-sided because a surface is, usually, four-sided. In the new face created, a second, four-sided, inner contour is created, which is the face in which the surface patch is set (figure 13.36c). Finally the matching corners of the two four-sided contours are connected to create the topology (figure 13.36d). The surfaces of the side faces created are calculated as swept surfaces of the matching surface patch sides. The curves of the edges marked e0, e1, e2, and e3 are the intersection curves of these swept surfaces with the surface of the original face.

This algorithm is simple. Obviously there are possibilities for problems. One possibility includes if the surface patch, as projected, overlaps or touches

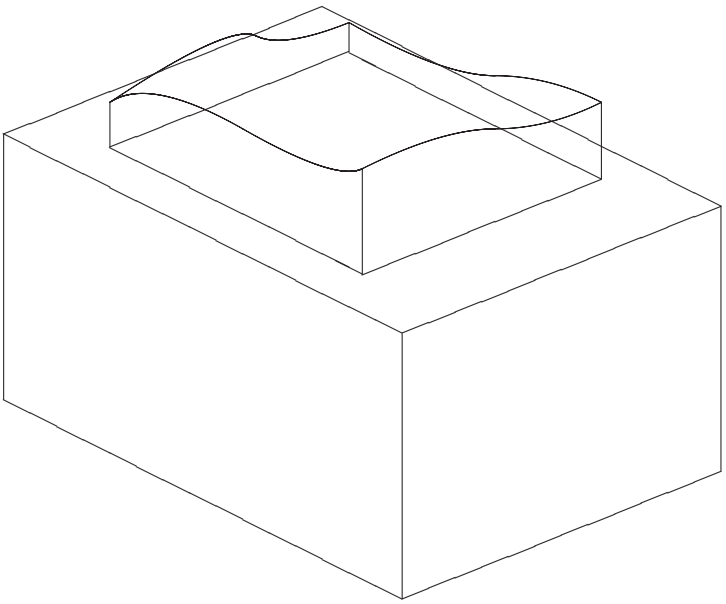


Figure 13.35: Creating a boss from a free-form surface

the boundary or boundaries of the face on which the boss is being created. This could be solved by making the boss a separate object (sweeping the surface patch downwards and doing a SETSURF on the bottom face to create a boss just touching the face) and then combining this object with the original object using an ADD Boolean operation. This is the current tendency. The other way is to tidy up the topology locally using the same techniques as the Boolean operations, but applying them more restrictively. This is similar to the local Boolean operations described in chapter 6.

13.4.4 Miscellaneous comments

Note, finally, that the algorithms described in this section deal mainly with with setting a surface into a single face. However, in the SETSURF algorithm and derivatives, the existing surface of this face is not, in fact, used for the operation. The single face is needed to relate the edges and vertices whose geometry is to be recalculated. As a result, it is possible to set a surface into multiple faces by preprocessing them to remove common edges, creating one super face with notional, or undefined, geometry. The new geometry is then set into the "super-face" as above. Given a list of faces and a surface that covers them, the edges to be removed are edges that lie between two faces in the list. Such edges should, in any case, be removed. The basic algorithm would be:

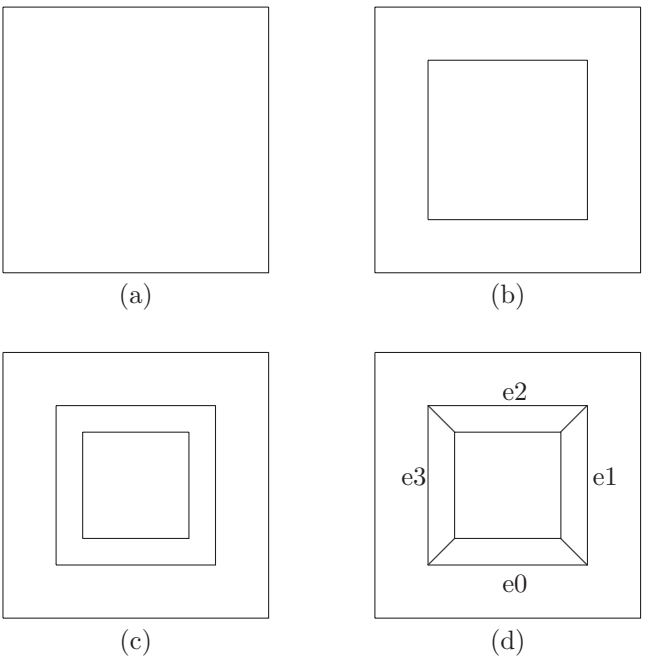


Figure 13.36: Creating the topology of the boss

Compile a list of all edges bounding the faces in the given face list. Process the edge list. Edges between two faces in the list are added to an "internal edge list", and the other edges are added to a "boundary edge list". Edges in the "internal edge list" are deleted, merging faces, creating hole-loops, or removing spur-vertices depending on the surrounding edge topology. Edges in the "boundary edge list" are processed as for the single face case.

Note that such a procedure also covers the case where there are several non-connected facesets.

13.5 Building a model from surface patches

This is another algorithm which has not been implemented but is related to the STL reconstruction algorithm in the STL reconstruction algorithm in Chapter 14, which has been implemented. The algorithm makes use of the basic fact that (crudely) a B-rep model is a set of bounded surface pieces associated via topology.

Assuming that Father Christmas, the Tooth Fairy, the Easter Bunny, and Superman all exist, which of course they do, you might have the following sequence:

1. Create the surface topology as explicit partial objects.
2. Sew together the partial objects.
3. Handle problems.

Reality is not that kind.

If this were so, then the process of creating a volume would be relatively simple. What has to be expected is that the surface patch corners will sometimes lie in the middle of the boundary of another patch, that the boundary curves of two adjacent surfaces will not be exactly coincident, and that the surface orientations will not be consistent. Creating the object involves, then, a process of juggling the topology and geometry to match and then reporting errors, such as unclosed boundaries.

A more realistic sequence might be

1. Create the surface topology as explicit sheet objects.
2. Sew together the sheet objects.
3. Handle problems.
4. Throw away the unwanted negative object created as a side effect.

The sheet objects are needed to get around two problems: the first is the orientation of the surfaces, and the second is that the surface set may not be closed.

13.5.1 Creating surface topology

For 'normal' numerical geometry, the surface topology is very simple; each face is four sided. However, the positions of the control points may create different topologies that can be slightly different. If, for example, the control points along one edge of the surface patch all coincide, then that edge has zero length and can be collapsed into a vertex, creating a three-sided patch. Mathematically triangular patches can be treated in an obviously analogous manner so they will not be considered further here.

Another possible complication is where the surfaces are represented by trimmed patches. Trimmed patches are similar to faces in a normal B-rep model. Associated with the surface is a set of curves in parameter space that delimit a portion of the surface. Each of these parameter-space curves can be considered to correspond to an edge in the sheet object created from the surface patch.

Another example of the difference between the surface topology and the B-rep topology occurs if the surface is periodic. Here, two boundary curves of the surface coincide and can be merged in the B-rep partial model producing a seam. If, in addition, the surface is C2 continuous at the seam, the edge can be removed entirely.

The boundary curves of the patch can also have discontinuities of curvature and/or tangency while still being geometrically single curves. It seems more logical, when converting the patches to divide the boundary edges at these points and possibly to recalculate the curves to be single C2 continuous.

13.5.2 Sewing together partial objects

The big problem in joining the partial objects is to match the topology of the separate partial models so that they can be merged. With completely matching surfaces, the joining method is simple.

First the surface patches are created, and normal surface patches generate four-sided faces, each surrounded by edges with two face pointers. The back-side faces are set into a negated copy of the original surface, or a reference to the original surface with a negation flag. Subsequently the vertices are joined; then, if any edges run between the same two vertices, these edges are merged. When all vertices have been merged, then the object should be complete and each edge should have two adjacent faces. If all faces form one connected set, then the process is finished; otherwise, the facesets have to be merged into one or more objects.

13.5.3 Handling problems

Two major problems can arise. First of all, there may be gaps where surfaces do not exactly match. The second problem is the object may not be closed because surface patches are missing.

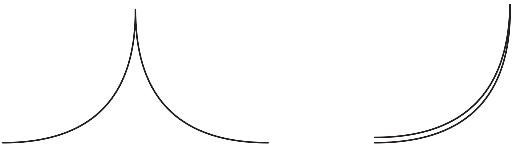


Figure 13.37: Possible sharp edge geometry

The first problem is difficult. Mismatching geometry might arise because geometry, such as intersection curves, may have been calculated at a lower tolerance than when the surface patches are joined. This can come about especially if the geometry is imported. One solution is to extend the two surfaces at the mismatching area, if they are not long enough, and then intersect them to give the required edge curve. Another possibility is to introduce a small blend surface, but this may cause other problems because it is likely to be so thin that geometric errors may result.

If all patches are joined, then it is necessary to check whether there are gaps. If it were possible to join surface objects as partial objects, then this would simply be a matter of looking for edges that have only one neighbouring face. Using sheet objects for joining means that it is necessary to check for 'sharp' edges. A sharp edge is one where the surface normals of the faces meeting at the edge are in the opposite direction; however, there are special cases. Figure 13.37 shows a possible section through a sharp edge to illustrate this case.

Two conditions have to be distinguished. On the left of the figure, the edge does not indicate that it is part of the boundary of an incomplete object, whereas on the right, the edge would be, but both are 'sharp' by the definition above. In this case, it is necessary to check the curvature to distinguish between the two conditions. However, this is much simpler if the sheet objects use a system where both faces refer to the same surface, with one of them having a flag to indicate that it is oriented in the opposite direction to the surface. This means that, to check for sharp edges, it is only necessary to check for edges between two faces referring to the same surface and that one of them has the orientation flag set to indicate that it is in the opposite direction.

This will leave the object either completely negated or completely positive, which can be determined with a point-in-body test for a point lying well outside the object. The point lies inside a negative object but outside a positive object.

13.6 Model 'sculpting' methods

The description in this section concerns tools implemented in the BUILD/-FFSolid system. This implementation is in turn based on an earlier imple-

mentation in the Swedish/Finnish GPM solid modeller, used by Fjällström for smoothing [36], [37]. BUILD/FFSolid is a Hungarian development based on the BUILD research system described by Braid [11].

The general philosophy is to make the model like a 'clay lump', to be pushed or pulled into the desired shape. The basic model can also be defined as a two-dimensional shape and then extruded several times, shaping the cross-sections, to produce the polyhedral model. The requirement that models are maintained as geometrically consistent means that general modelling operations such as the Boolean operations, chamfering, and imprinting, available in BUILD/FFSolid can be used as well as the basic sculpting tools. However, some special requirements and limitations on the sculpting tools meant that they were implemented separately from the rest of the modelling code. An example is that of the "lift face" operation. Superficially it performs a similar task to the "sweep" (straight extrusion) operation. However, the sweep operation moves some edges that can be considered to 'slide' with the face being swept, as described in Chapter 6. The lift face operation leaves edges in place, except where back-to-back coplanar faces would be created.

Only models with straight curves and planar surfaces are manipulated in the package. One possible extension is to allow general models to be manipulated by facetting curved faces and approximating curved edges by straight line segments. This has not been implemented, partly because there was, when the work was done, no automatic facetter in BUILD/FFSolid, so a fair amount of extra work would have had to have been done to implement one. However, the main reason was that there is the problem of reestablishing the original shape and topology from the facetted model, especially where a single face is selected for sculpting. Although neither problem is insurmountable, it was felt that it was unnecessary to expend effort on them in the test implementation phase.

13.6.1 Implemented operations

The following operations were implemented in the BUILD/FFSolid sculpting module:

1. Move a vertex
2. Move an edge
3. Move a face
4. Spinning a face
5. Polygonal 'swinging'

The 'move' operations (1–3) are specified with the identity of the vertex, edge, or face to be moved and a vector specifying the move direction and distance. The face spinning operation is specified by the face identity and the

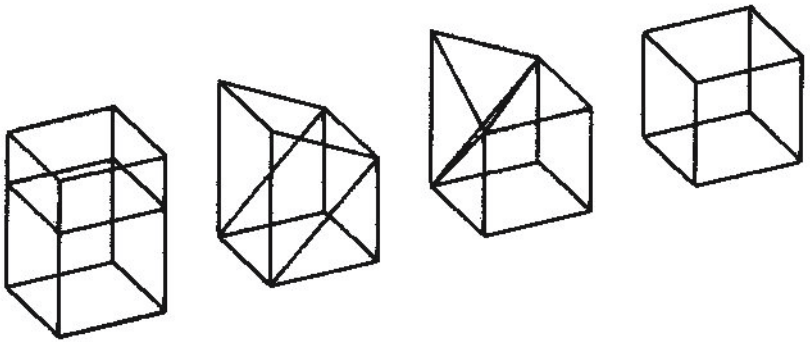


Figure 13.38: Sculpting operations

spin angle. The polygonal swing operation is specified with a face identifier, the swing axis, and the swing angle, optionally also with the number of steps.

The move operations are illustrated in figure 13.38.

Moving a vertex

To maintain geometric consistency, the operation to move a vertex first isolates the vertex so that it is surrounded only by three-sided faces; then it modifies the vertex position and finally recalculates the surfaces of the faces around the vertex.

Isolating the vertex is illustrated in figure 13.39. Figure 13.39, top left, shows the original arrangement of edges and faces around the vertex. One of the edges, say edge 1, here is selected, and the vertex at the opposite end (vertex 2) is found. The next edge counter-clockwise around the vertex from edge 1, here labelled edge 2, is found, and the vertex at the opposite end (vertex 3) is found. Vertex 2 and vertex 3 are then joined (figure 13.39 top right). The next edge around the vertex from edge 2, edge 3, and the opposite vertex (vertex 4) are found. Vertex 3 and vertex 4 are already joined, so there is no need to add a new edge (figure 13.39 bottom left). The next edge around the vertex is the first edge, edge 1, and the final step is to join vertex 3 to vertex 1 with a new edge. Vertex 1 is now 'isolated', surrounded by faces with three edges only (figure 13.39 bottom right).

This works when the angles between adjacent edges meeting at the vertex are less than 180 degrees. For a vertex such as that shown in figure 13.40a, there is a problem with this simple algorithm. Where the angle is greater than 180 degrees, then an edge is added bisecting the angle between the edges (figure 13.40b), and the new edge is connected to the opposite end vertices of the adjacent edges (figure 13.40c, figure 13.40d).

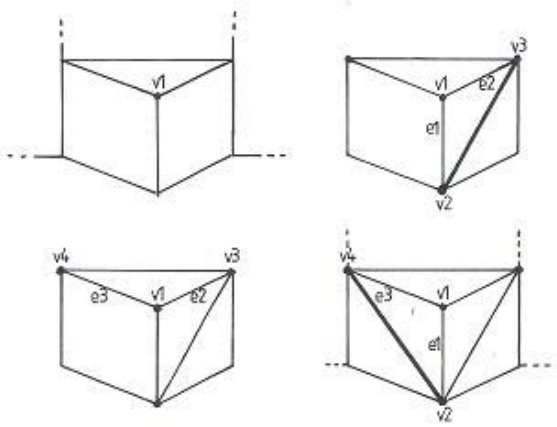


Figure 13.39: Isolating a vertex for moving

Once the vertex has been isolated, it can be repositioned and the geometry recalculated. In the implementation, the curves of the edges meeting at the vertex are determined by the positions of the endpoints. Recalculating the surface equations is straightforward because the modified faces are all three-sided, and hence, the corner points define the plane of the face.

A special case arises when the edges of a face are coincident; i.e., the face becomes degenerate. In this case, there is a choice between leaving the face as it is and collapsing it back to edges. Strictly speaking the face should be collapsed, because one aim of the sculpting tool implementation was to maintain the model as geometrically and topologically correct so that general modelling tools can be used in addition to the sculpting tools. In the current implementation, however, there are no checks for creating degenerate faces.

Moving an edge

Moving an edge is similar to moving a vertex in that the edges are first 'isolated', then the position is transformed and the new surrounding geometry is calculated.

Isolating the edge means creating four-sided faces on the left and right of the edges, and building triangular faces around the start and end vertices, when necessary, as illustrated in figure 13.41.

It is possible to have four-sided faces adjacent to the edge, rather than three-sided faces, if the left and right faces are isolated with edges parallel to the edge being moved, as described later. The three-sided faces are needed if the start or end vertex of the edge moves out of the surface of any face adjacent to the vertex.

It is easiest to describe the edge isolation process in terms of the "winged-

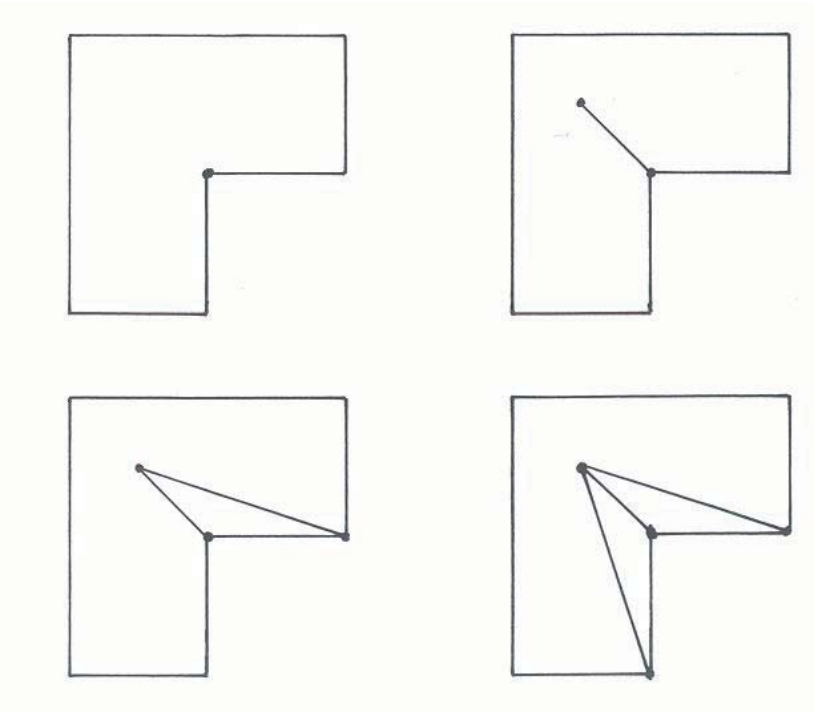
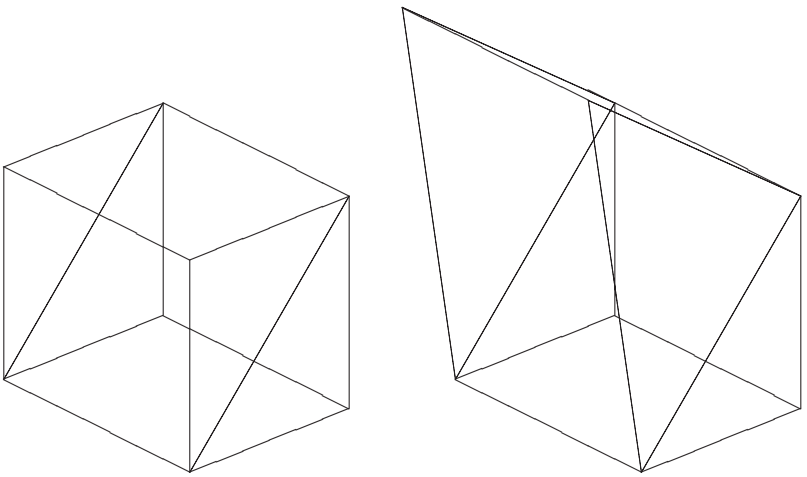


Figure 13.40: Isolating a vertex with edge-angle more than 180 degrees



Isolated edge Translated edge

Figure 13.41: Moving an edge

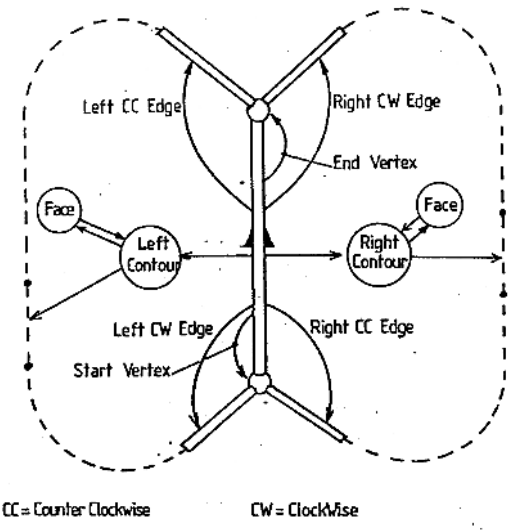


Figure 13.42: Winged-edge representation (after Braid et al. [13])

edge" edge representation shown in figure 13.42.

The edges adjacent to the edge around the right and left loops are called the RCC (Right Counter-Clockwise), RCW (Right ClockWise), LCC (Left Counter-Clockwise), and LCW (Left ClockWise) edges. Label the vertices at the opposite ends of the RCC and LCW edges from the start vertex as vertex 1 and vertex 4, respectively, and the edges at the opposite ends of the RCW and LCC edges from the end vertex as vertex 2 and vertex 3, respectively; then the right side of the edge is isolated as follows:

1. If vertex 1 and vertex 2 are not already connected, a new edge is inserted between them.

2. The distances of vertex 1 and vertex 2 from the curve of the edge are calculated.

3. If vertex 1 and vertex 2 are not the same distance from the curve, then a new edge is inserted between the base vertex of the wing-edge connected to the more distant vertex and to the closer vertex. So, for example, if vertex 1 is further from the curve than vertex 2, then a new edge is inserted from the start vertex of the edge to vertex 2.

The same process is repeated for the left face of the edge. For the end vertex, each pair of edges between the RCW and LCC edges is examined. If the movement vector is not perpendicular to the surface of the face between the edges, then it is necessary to isolate the vertex from the rest of the face by building a triangular face enclosed by the two adjacent edges and a new edge.

Once the edge has been isolated, the positions of the start and end vertex of the edge are modified and the surface equations of all new faces are recalculated.

The same problems can occur when moving an edge as when moving a vertex; i.e., faces and edges can disappear under some circumstances, and the same comments apply as before.

Moving a face

As mentioned, moving a face is functionally similar to the sweep (straight extrusion) operation. The difference is simply that the sweep operation tries to extend faces, where possible, leaving edges at discontinuities of slope and moving edges. The role of the face moving function is, in this implementation, to create objects extruded through a series of profiles. For this it defaults to the simplest sweep form of the well-known sequence of Make Edge Vertex — Make Face Edge, illustrated in figure 13.43 (from chapter 6). However, it has to be able to detect and cope with some extra cases as well to avoid having back-to-back coplanar faces.

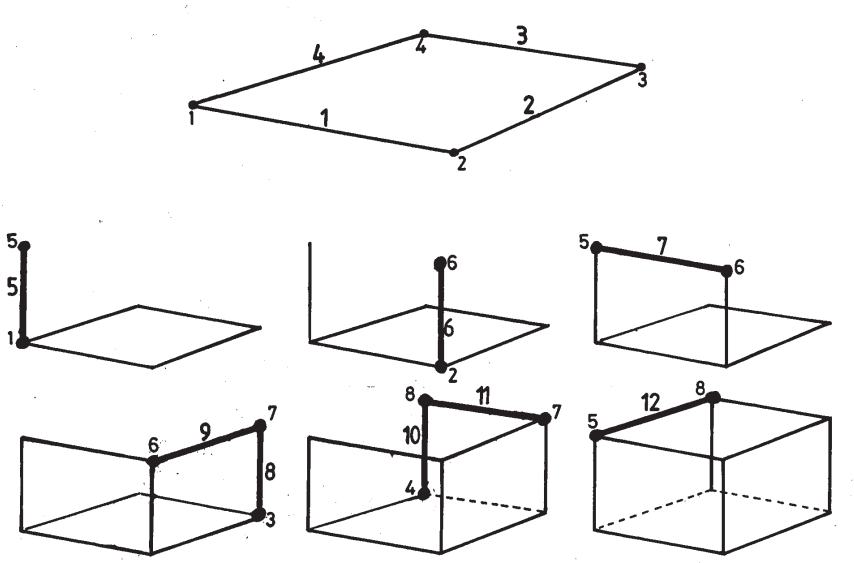


Figure 13.43: Sweep operation as series of MEV, MFE operations

The conditions where back-to-back faces can be created are illustrated in figure 13.44. Simply, these can occur when an edge adjacent to the face being moved, face f1 say, lies between f1 and a face with normal perpendicular to the move direction. If the edge is concave and the move direction is the same as f1's surface normal, or if the edge is convex and the move direction is opposite to f1's surface normal, then special action has to be taken. Under these conditions, the edge is moved. Edge moving for general sweeping is described in chapter 6 and in [124], so it will not be repeated here.

Spinning a face

Face spinning was implemented really as a 'special effect' rather than for any serious purpose. It is described here as an example of a special purpose function for creating a particular kind of shape. There are obviously many examples of such operations, depending on the kinds of tools natural for a particular application.

An example of the use of the spin-face function is shown in figure 13.45. The face spinning function uses the vertex moving function, described above, for each vertex around the face.

The face spinning tool in the implementation is a central face spinning function, although it is easy to see how this could be extended by allowing the user to specify an arbitrary rotation axis. For the central face spinner, the spin axis is calculated from the centre of the box surrounding the face and

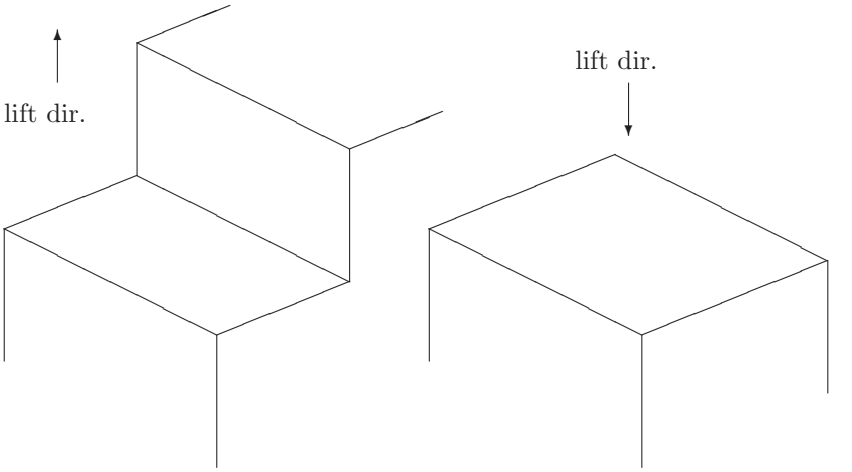


Figure 13.44: Edge conditions requiring special handling

the surface normal, the spin angle being the only data supplied by the user.

Two simple steps are involved. The first is to traverse all vertices adjacent to the face being spun to calculate the centre of the face and, hence, the spin axis. The second step is to traverse again the vertices, rotating each around the axis to its new position. The vertices are moved using the vertex moving procedure described previously.

Polygonal 'swinging'

Polygonal swinging is an old modelling function in BUILD/FFSolid. Instead of creating an object with rotational surfaces, the object is created with a series of facets. This is so as to be able to produce planar approximations of rotated objects or rotational parts of objects.

Polygonal swinging was implemented as a special case of the general swing operation described in [124], replacing the rotational geometry with planar geometry. A shape is swept around the axis in suitably sized steps. The operation is approximately the same as the move face operation described before, using a sequence of MEV, MFE Euler operators to produce the shape. However, there is the obvious special case where an edge or vertex lies on the rotation axis.

Other comments

Basic Euler operators are also useful for manipulating shapes, especially edge splitting (inserting a vertex in an existing edge), and making a face and an

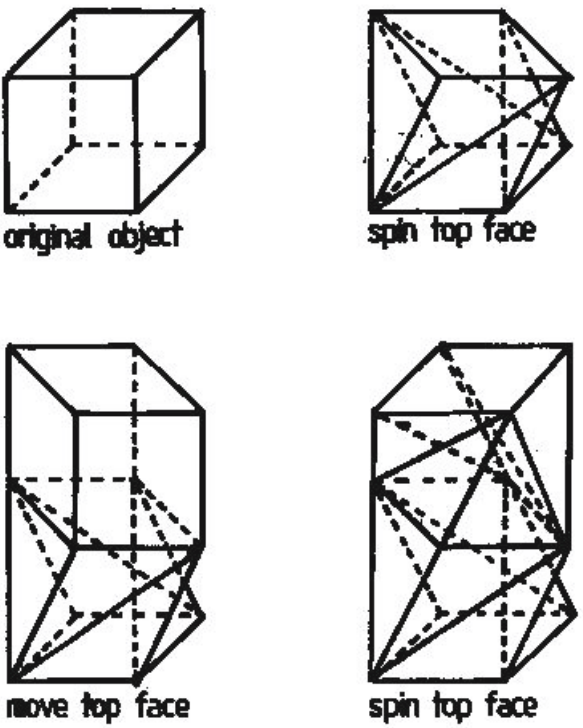


Figure 13.45: Face spinning examples

edge. However, the same care must be taken not to facilitate creating geometrically incorrect objects. The implementation was intended to preserve the geometrical integrity of the object, and the basic Euler operators do not necessarily preserve it. General Euler operators can create 'dangling' edges, for example, which have no meaning for the smoothing algorithm. Preserving the geometric integrity of the object means that other tools, such as the face inscription utility in BUILD/FFSolid, can be used to modify the model. However, the edge splitting Euler operator is very useful for inserting extra vertices for moving. Similarly the make edge face Euler operator is useful for trimming faces being moved or swept.

Finally, it should be noted that if there are spatial occupancy measures in the body, then these have to be removed or reset as the result of the operations.

13.6.2 Smoothing

The planar polyhedron smoothing algorithm was not fully completed when work on the modeller was suspended at the end of the development project. There were three parts to smoothing:

1. Marking edges and vertices to be rounded or static
2. Modifying the topology
3. Modifying the geometry

Marking edges and vertices to be rounded or static

By default all edges and vertices in the model can be rounded off during smoothing. However, it was decided that it should be possible to place vertices at specific positions and fit the surrounding geometry to these positions by marking the vertices as 'static'. The reason for doing this is so that objects that are defined in terms of cross-sections preserve set positions. Figure 13.46 shows the topological changes in a cube with all vertices rounded and a cube with one vertex marked as static.

Similarly, edges can be marked as to be rounded or as static. Figure 13.47 shows the topological changes in a cube with all rounded and one with a single static edge.

Modifying the topology

For each edge to be rounded, a pair of faces is added, and vertices are surrounded by four-sided faces. Because new topology is being added, care must be taken to avoid modifying the new edges and vertices. BUILD uses marker bits for this purpose. First, all vertices and edges in the original object are marked. Then, all marked edges and vertices are unmarked and modified;

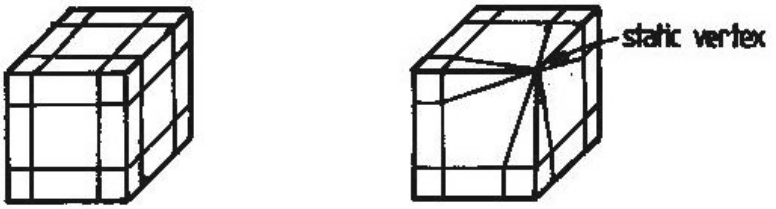


Figure 13.46: Topological modifications for rounded and static vertices

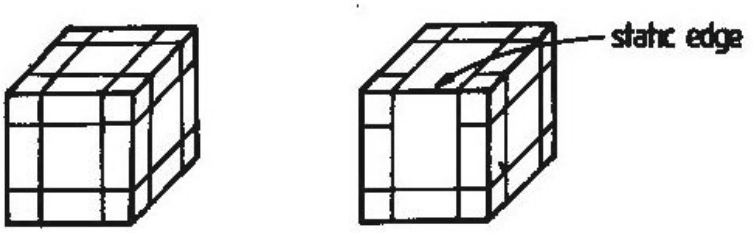


Figure 13.47: Cubes with all-rounded and one static edges

unmarked edges and vertices are ignored. There are other possible mechanisms for doing this; it is not interesting really how it is done, but it must be possible to make the distinction between original and new topology.

First, all original edges in the body are traversed and shortened proportionally. The proportion is given by a global ratio value, with a default value of 0.25. This means that the edge will be shortened in the ratios 0.25:0.5:0.25. A ratio value of zero means that no smoothing will take place. A ratio value of 0.5 means that the edge disappears entirely. Figure 13.48 shows a basic cube, and the cube with the edges shortened proportionally.

Once the edges have been shortened each original vertex is surrounded by short edges ending with a vertex with only two edges. For each vertex, v, an edge is chosen as the start edge, and the vertex opposite v along the edge is found, called v1, say. The next edge clockwise from the start edge is found, and the opposite vertex along this from v is found, v2 say. Vertices v1 and v2 are then joined, creating a triangular face. The new edge is then split by inserting a new vertex at a suitable position, creating a four-sided face. See figure 13.49.

The process is repeated until the vertex is surrounded by four-sided faces. As with vertex moving, care must be taken when the angle between adjacent edges around the vertex is greater than 180 degrees. In this case an extra edge is added between the two edges, and two four-sided faces are produced

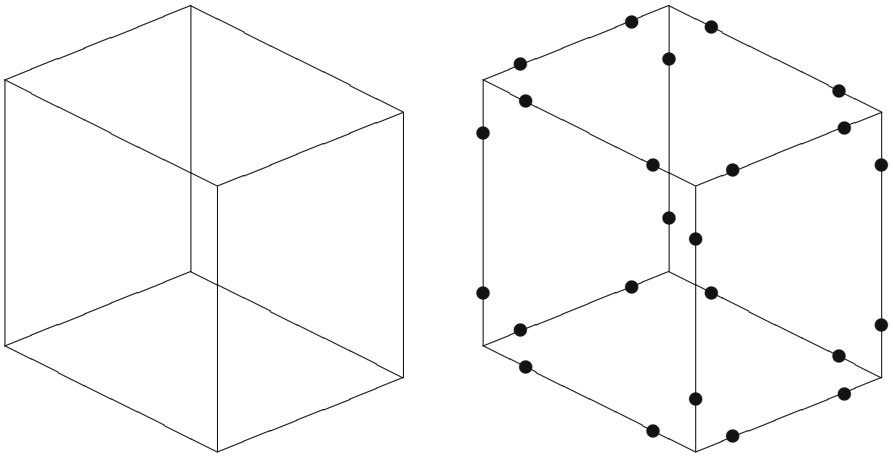


Figure 13.48: Original cube and cube with shortened edges

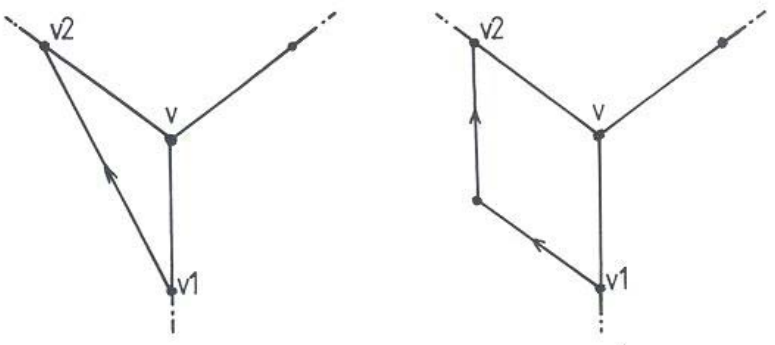


Figure 13.49: Creating a four-sided face at a vertex

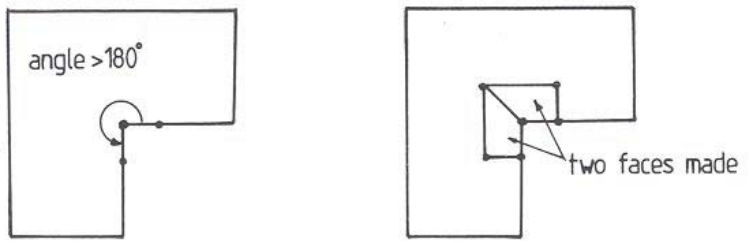


Figure 13.50: Creating four-sided faces at concavities

(figure 13.50).

Once the vertices have been modified, faces are added on the left and right of the original edges of the object. Figure 13.51 illustrates this.

If the global ratio variable has the value 0.5, the original edges will have zero length after the modification procedure and so must be removed and the coincident topology merged.

Modifying the geometry

Once the topological modifications are complete, the planar geometry of the new edges and faces created in the topological modification procedure is replaced with free-form curves and surfaces. In the implementation, this stage was not completed when work on the BUILD/FFSolid projected was halted; only vertex repositioning and curve recalculation were completed.

The free-form geometry used for the smoothed shapes is the double-quadratic geometry introduced by Várady into BUILD as part of his dissertation work [138]. The double-quadratic curve form is defined using a series of points together with tangent vectors at those points. For the smoothing implementation, only two points were used for each curve.

If a vertex is marked as static, it is not repositioned, the edge curves are fitted to it, and the vertices to be rounded are moved inwards. The average of the normals of the faces meeting at the vertex defines the surface normals of the modified surfaces and the plane in which the curve tangents lie. The geometric calculations are based on edge lengths, which are determined in the topological modification stage. Figure 13.52 shows the topological changes for three cubes with rounding ratios 0.1, 0.25, and 0.5.

Determining the rounding from the edge length means that rounding is non-uniform, so an edge can be rounded more at one end than at the other. Figure 13.53 shows an object (figure 13.53a) and the topological framework (figure 13.53b) for non-uniform rounding.

The vertex positions and edge directions provide a framework for the def-

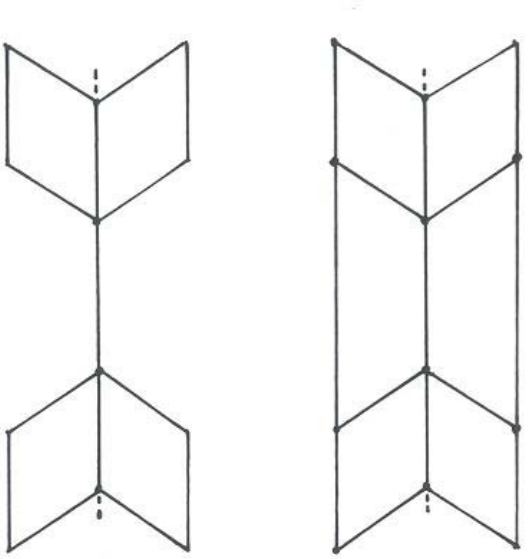


Figure 13.51: Creating side faces at an edge

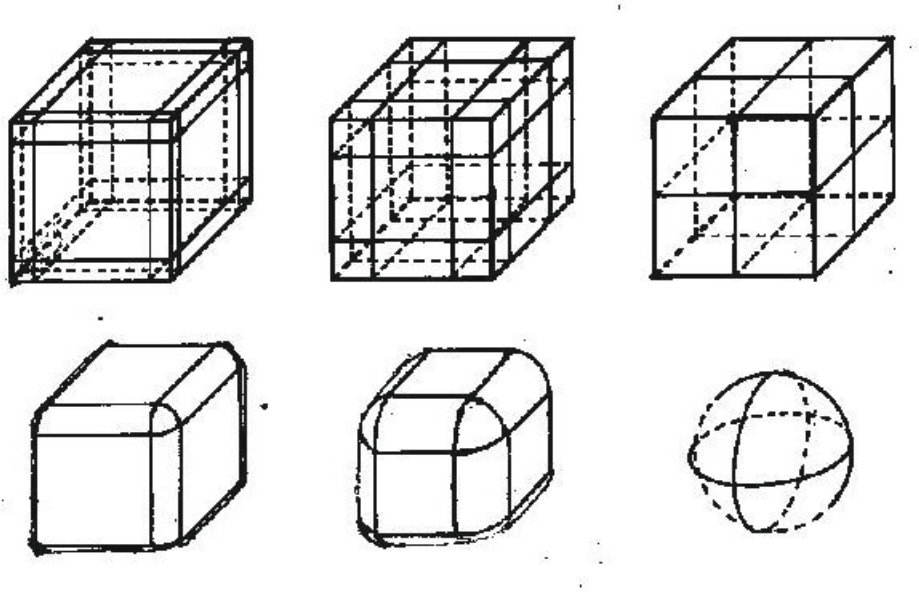
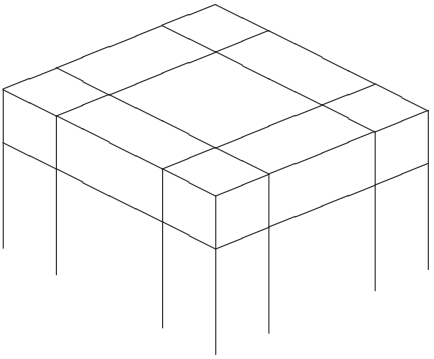
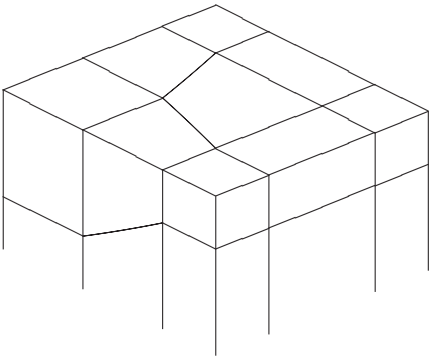


Figure 13.52: Topological changes in cubes rounded with ratios 0.1, 0.25, and 0.5



(a)



(b)

Figure 13.53: Uniform and non-uniform rounding

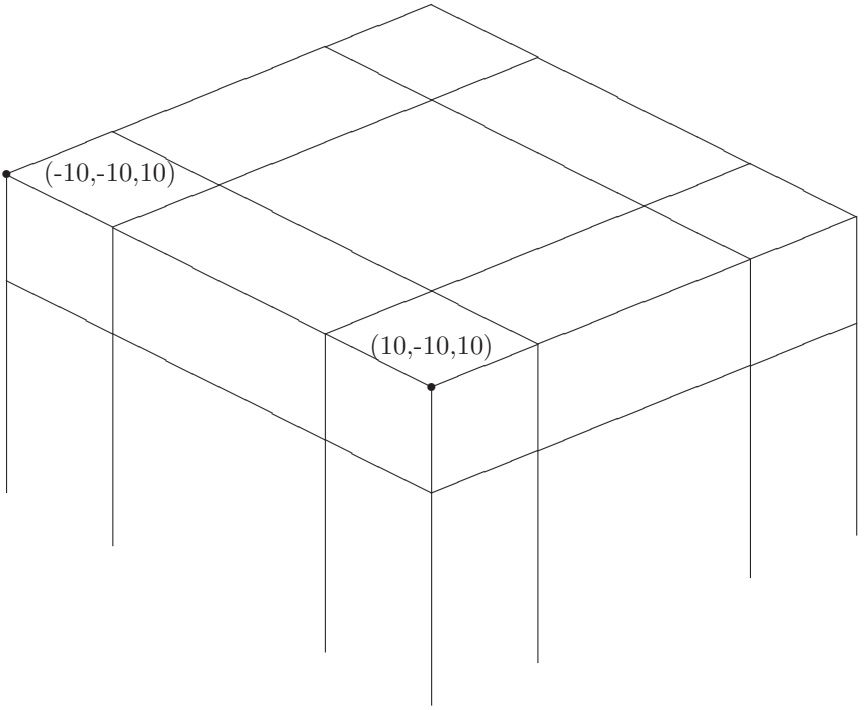


Figure 13.54: Uniform edge rounding

inition of the control points of the surfaces. The original edges of the object disappear, being replaced by surfaces. Note, though, that if the rounding ratios of both the start and the end vertex of an edge is 0.5, then there will be no corresponding surface. The original vertices of the body are repositioned, but the edges surrounding them are retained, defining the borders of a set of surfaces surrounding the vertex.

The uniform rounding case is shown in figure 13.54.

Assuming that the rounding ratios are 0.25 for both vertices, this means that the edge surface control polygon is as shown in figure 13.55.

The non-uniform case is similar, but instead of having a linear-quadratic surface, the surface becomes a cubic-quadratic surface patch to preserve the bounding curve directions at the start and end as parallel to the original edge. The original topological changes are shown in figure 13.56.

Assuming that the rounding ratios are 0.5 for one vertex and 0.25 for the other, this means that the edge surface control polygon is as shown in figure 13.57.

The control points are as follows:

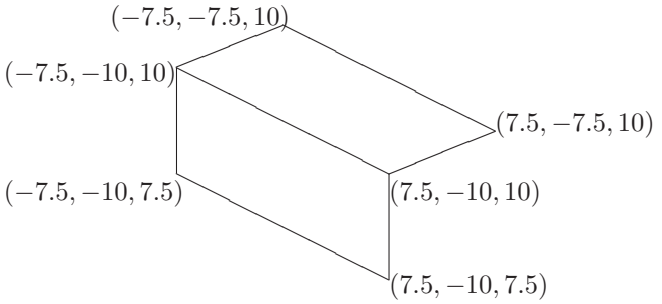


Figure 13.55: Uniform edge surface

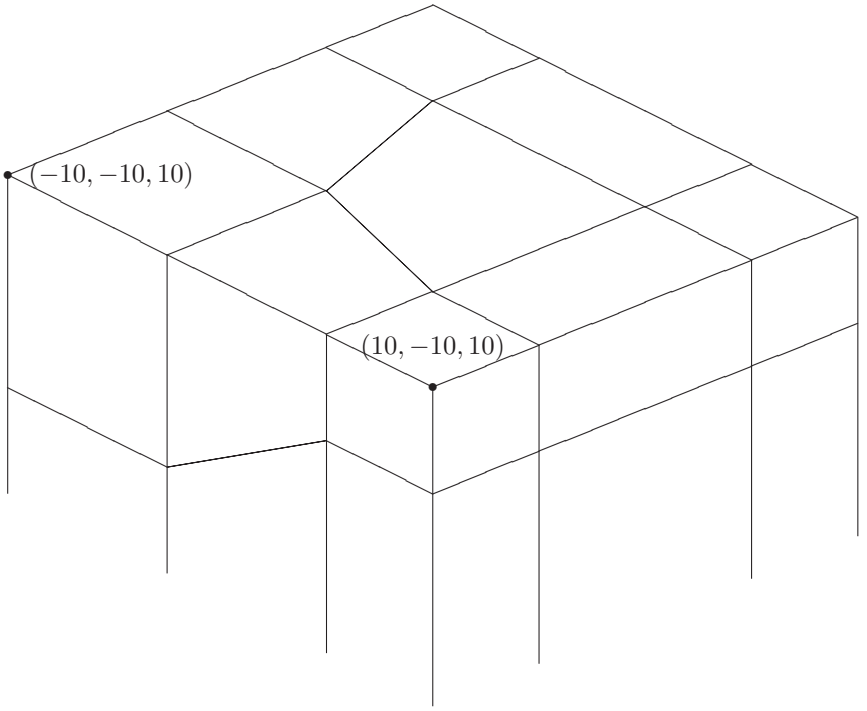


Figure 13.56: Non-uniform edge rounding

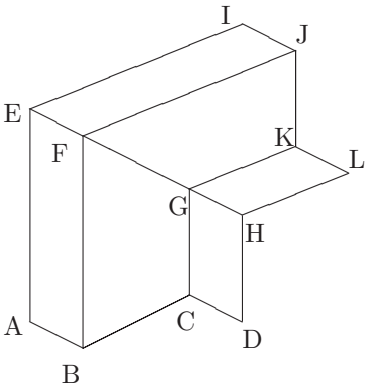


Figure 13.57: Non-uniform edge surface

$I(-5,-5,10)$	$J(-1.875,-5,10)$	$K(4.375,-7.5,10)$	$L(7.5,-7.5,10)$
$E(-5,-10,10)$	$F(-1.875,-10,10)$	$G(4.375,-10,10)$	$H(7.5,-10,10)$
$A(-5,-10,5)$	$B(-1.875,-10,5)$	$C(4.375,-10,7.5)$	$D(7.5,-10,7.5)$

13.6.3 Conclusions

The main conclusion drawn from the work described here is that object sculpting needs to be developed in conjunction with users to provide the right tools and results. Modelling techniques can be developed for many purposes, but it is difficult to know what is required.

The first part describes a few basic manipulation tools for research purposes. Keeping the object geometrically correct means that other tools, for example, the Boolean operations, reflect, chamfer, and so on, are also available. However, the special 'trick' tools, such as face spinning, need to be developed according to need. Wesley and Fournier [39] developed a planar polyhedral bending tool, for example, which could be useful.

The changes described subsequently are for smoothing, but it is relatively easy to see how 'spiky' shapes could be produced by moving vertices outwards. This illustrates a problem in the geometric transition stage of how to go from the gross polyhedral model to the final model. Producing spikes instead of smoothed objects is an example of a 'trick' or option that can be offered by the system to produce shapes. The problem is, though, to discover what tools a designer wants or needs. Object sculpting is an interesting example of a new, three-dimensional design tool, something more akin to clay modelling than traditional design. It could also conceivably be used for local free-form editing of parts of an object. The geometric transformation of the polyhedron

or polyhedral faces being modified is, for those purposes, most likely to be smoothing. However, practical experience is necessary to discover what is needed.

In the BUILD/FFSolid implementation, the topological modification stage is weak; it can potentially create self-intersections in the face. It would have been better to have built a Voronoi diagram, as done by Held [53] or Lukacs et al. [77] for machining, and to have used the Voronoi diagram as a basis for the topological modifications.

Another problem with the implementation is that a great many new faces are added during the smoothing stage, such as four extra (at least) for each smoothed edge. For a complex polyhedral model, this may cause problems because of the new model size and may slow down subsequent modelling operations.

Chapter 14

Applications

Since the beginning of the 1980s more and more attention has been paid to applications based on solid modelling instead of to solid modelling itself. This chapter describes an assortment of techniques relevant to applications built on top of modelling systems.

14.1 Model verification

Boundary representation (B-rep) modelling techniques have grown up for more than thirty years now to become complex systems. See [10], [11], and [82] that span the initial stages, early developments, and some later stages. It hardly needs to be pointed out that the B-rep is a 'piecewise' representation with a lot of interrelated parts between which consistency has to be maintained to guarantee correct functioning of the modeller. Such problems are dealt with by Hoffmann [58], for example. Other work has been done on improving modeller reliability, e.g., by Benouamer et al. [7] on improving numerical calculations, Sugihara [130] on evaluation procedures, or Zhu et al. [150] on improving robustness in Booleans. Other experiments have been made using rational arithmetic packages to ease the problem of numerical inaccuracy, by Solomon in 1979 (private communication) and by Jared et al. (private communication). This chapter describes a slightly different approach to the problem and presents methods for checking the validity of a model.

The idea of model verification is not new. For example, the GPM volume module specifications by Kjellberg et al. [70] contain specifications for a consistency checker, but little has been reported on the subject. Verification means, effectively, checking that the model obeys the implicit and explicit rules built into a solid modelling system. The implicit rules involve general assumptions about the model, such as its geometric consistency. The explicit rules are built into the datastructure, governing the connectivity between elements. It is relatively easy to determine what connectivity checks are required,

but the necessary set of geometric checks are less well-defined and selecting them depends more on the imagination of the implementor. It is possible to use set theory as a partial basis for determining correctness, but this is not sufficient for special representations, for example.

This section describes briefly the theoretical background to determining the checks and then goes on to describe an implementation based on the ACIS modelling system.

14.1.1 Justification

One important justification for developing model verification software is to improve knowledge about solid modelling systems. Model verification facilities can help users who find that systems do not work as expected and want to know why. Although they are not rule-based in the knowledge based system (KBS) sense, they encapsulate many rules for model validity and can indicate problems without having to resort to time-costly error reporting procedures and waiting for a developer to locate the problem.

Briefly, automatic model verification helps in four main areas, as follows:

1. Checking the correctness of any operation, especially new or user developed operations (i.e., topology correct).

2. Checking a model is correct during or at the end of the design process (i.e., no self-intersections or degenerate parts).

3. Checking an imported model is correct (i.e., that the imported geometry is consistent to within the importing system's tolerances).

4. Helping to explain operation failure by highlighting critical parts of a model.

The kinds of errors that may occur vary greatly. Simple topological problems may occur as the result of unexpected cases. There are also common geometric problems, such as when geometric inaccuracies creep into the model.

The problems arising from geometric accuracy are well known. Geometrically models are a set of surface portions bounded by curves that are in turn bounded by points. The geometry is related by topology, which indicates dependencies between geometrical elements without guaranteeing consistency. The geometric representations are often separate and sometimes approximative, which can cause problems in configurations where consistency is critical.

Apart from general positional inaccuracies, such as points or curves not lying on surfaces, there are also problems where the geometry compromises the topology, such as the case of edges that intersect each other, as illustrated in figure 14.1. Similar problems occur when faces or edges degenerate so that a face is approximately an edge, or an edge has almost zero length, approximating a vertex. Such problems can be caused either by numerical inaccuracy

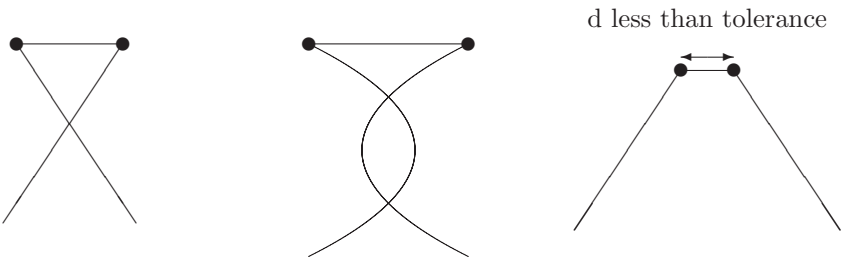


Figure 14.1: Positional inaccuracy problems

or by incorrect application of a modelling operation. This is because there are two 'truths' in the model, a geometric truth and a topological one, and errors occur when these two are in conflict.

14.1.2 Verifications

Two levels of checking can be identified: single model checking and assembly checking. Classically, an assembly is essentially a tree structure, with a root node containing instances of single models and other sub-assemblies, each with a transformation specification. Assembly checking involves checking that no subtree contains a reference to itself and checking the validity of the transformation information.

The exact nature of the single model checks is determined by the datastructure. The first step is to check the basic connectivity of a model because the connectivity is needed when traversing the model to find other elements. If there are circular lists where simple lists are expected or connectivity problems with the basic structure, then this should be discovered before the checker needs to rely on these structures, otherwise the checker may get into an infinite loop. When checking the datastructures, it is possible to make use of redundancy in the model structure for comparison. Assuming that a correct model pointer is given, whatever can be reached can then be checked, so the 'domain' of operations is established working outwards from the model to successive levels. If the whole structure cannot be traversed properly then it may be impossible to perform full checking and no automatic checks are possible. The user would then have to abandon the model or repair it enough manually to be traversable.

To illustrate the differences between modellers, figures 14.2, 14.3, and 14.4 show three datastructures, for the BUILD/FFSolid modeller, the GPM volume module, and ACIS, respectively. The ACIS datastructure is from 1993 when the work was done. Since then, it has been changed to some extent but the implementation illustrates many of the checking techniques.

As an example of the differences between these modellers, consider the

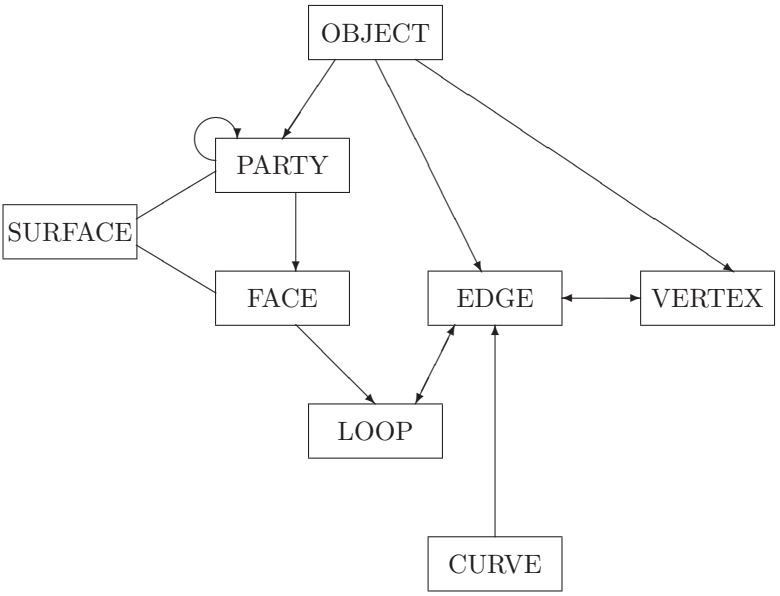


Figure 14.2: BUILD/FFSolid datastructure

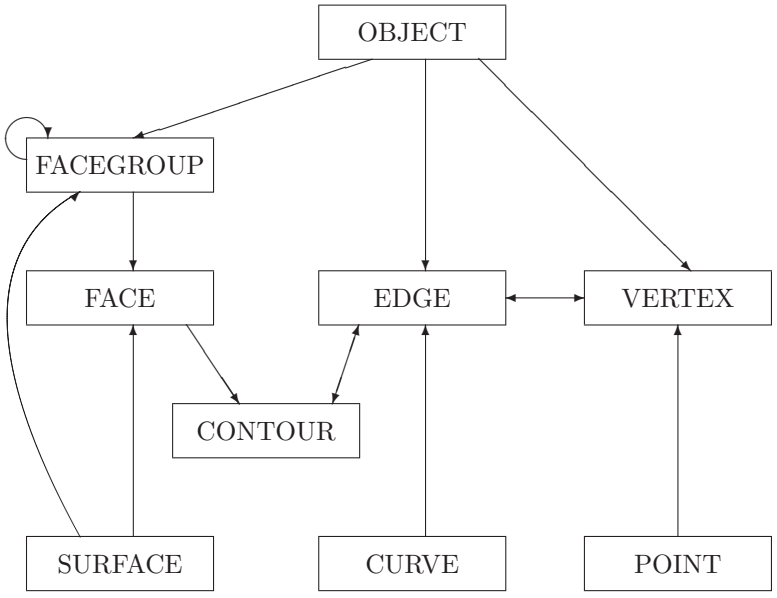


Figure 14.3: GPM Volume module datastructure

way that edges and vertices are connected. In BUILD/FFSolid and GPM, edges and vertices are linked in a chain from the model, whereas in ACIS, they can be accessed only via faces. This means that in BUILD/FFSolid or GPM it is possible to check that all edges and vertices can be reached through the face structure. If there are any edges in the model edge list which are not accessible through the face structure, then these are 'lost' parts of a model. In ACIS it is not possible to detect such discrepancies because the explicit model-edge/vertex links are not included. The ACIS datastructure is more efficient, but the redundancy in the other datastructures allows extra checks to be performed.

Body model datastructures contain both simple and circular lists. In a circular list, the last element in the list must point back to the first element. In a simple list, the last element must point to NIL. It is also necessary to check the back pointers, where these exist, pointing from list members to the list owner.

In addition to simple topological lists, some lists have a geometric component, such as the list of edges in a loop or around a vertex or faces meeting at an edge (for non-manifold edges). The connectivity of these lists can be checked in the same way, but they have an additional geometric component that needs to be checked. The geometric component determines the relative order in the list whereas with, say, the list of faces in a shell, the order of the faces is irrelevant (at least, usually).

The second type of check involves the geometric validity of a model. Some general rules about the geometry of the object make it easier to specify some of the checks required. Faces may have one or more outer loops of bounding edges and any number of inner loops of edges. The edges bounding a face may not intersect or touch each other except at the vertices. The implicit face orientation must agree with the surface orientation. A single model may have a single outer shell and any number of inner shells, or cavities. The object may not be self-intersecting, although parts of it may be coincident or 'non-manifold' parts. The curves and surfaces of the model must be curvature continuous in the portion used in an edge or face.

There are also checks based on the characteristics of particular modellers. Some modellers may also have requirements such as that faces or edges may not extend through more than 180 degrees. Also, the geometry needs to be checked for validity and stability. If, for example, a straight line is specified by two endpoints, then the distance between the points should be checked to see that they are not too close together. If the line is specified as a point and a normalised direction together with parameter values, then the direction has to be checked to see that it really is normalised, and the parameter values have to be checked.

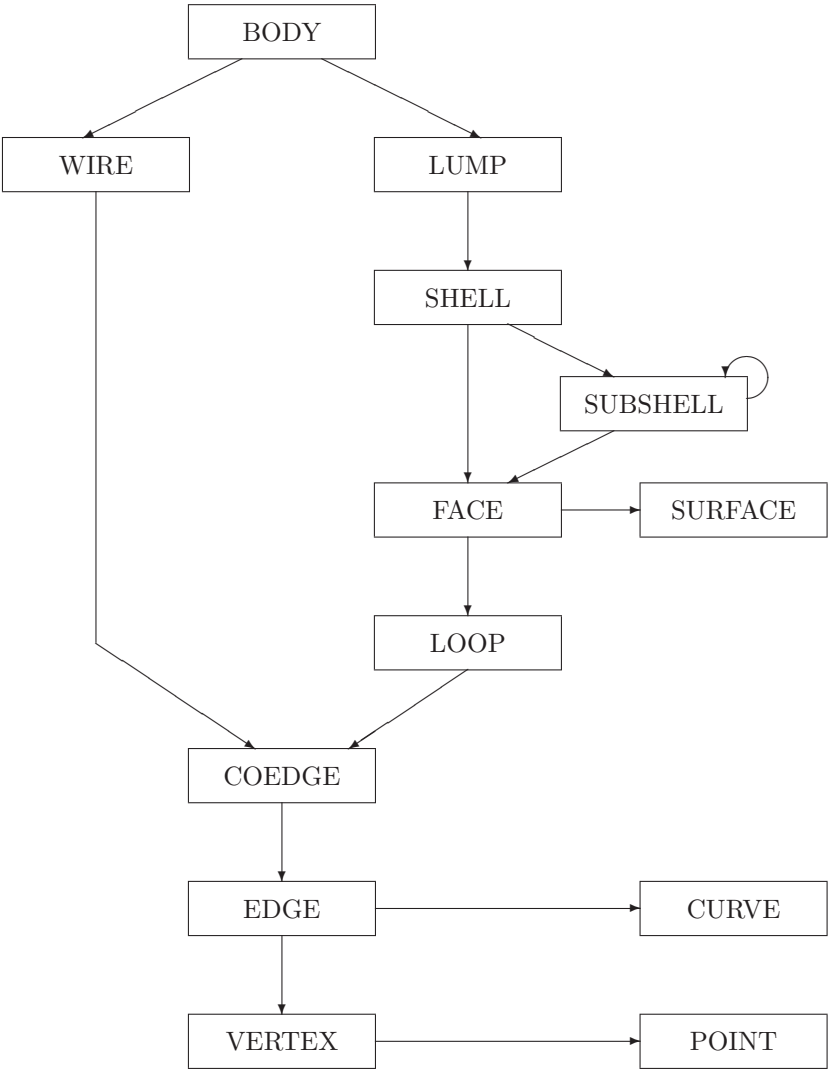


Figure 14.4: ACIS datastructure

14.1.3 Model verification in ACIS

It is easiest to describe model verification for a specific system because of the amount of variation in basic modeller design. This section describes a model verification package, or body checker, developed for the ACIS modeller. ACIS is interesting because it is a commercial kernel modeller containing general mechanisms to allow system developers to use it in different ways, which means that the checks have to be developed to take into consideration a wide variety of basic conditions. For example, ACIS allows incomplete objects, partially closed loops, or partially bounded edges. Although a particular application or commercial system based on ACIS may exclude partial models, for the ACIS kernel, they must be allowed.

When the work was done, there was an existing body checker in ACIS (which has since been improved) that was used to compare results with particular test objects. The body checker described here was an extension of that checker, performing the same checks and others.

The use of tolerances in solid modelling is described by Pfeifer [99] who identified six tolerance values in the GPM module for volume geometry. See also chapter 16. ACIS has four tolerance values: resabs, resnor, resfit, and resmch, which are a distance tolerance, relative epsilon value, curve fitting tolerance and machine precision value, respectively. The main tolerance value used in the body checker was the resabs value, which is a distance tolerance. This means that, for example, comparing angles means converting the angles into distances and comparing the distances.

This section describes the checks implemented and the next section the way the results were communicated.

The tests

The interface to the implemented checker consisted of four main sets of checks that test 1) a body (a single solid or wireframe object, or a collection of these), 2) a lump (the ACIS term for a connected solid piece), 3) a wire (a model containing edges and vertices), and 4) a faceset. These took an array of option flags to apply/ignore individual tests and a complete/incomplete flag to control whether incomplete objects were accepted.

Results are returned as a set of result records, which are described later. To provide a record of the checking process, at least one result record is produced for each relevant test. Even if a test is not applied, then this fact is recorded in the result record.

The verification methods and scans for checking ACIS models were the following:

1. The body transformation correct
2. The basic body lists valid (lumps, wires, etc.)
3. Subshells/faces correctly partitioned

4. Loops consist of single edge sets
5. Coedges correctly ordered around edges
6. Vertices have adjacent single edge sets
7. General model scan
8. No degenerate geometry present
9. Consistent geometry (curves, surfaces, points)
10. Curve continuous and no self-intersection on edge
11. Surface continuous and no self-intersection in face
12. Numerically stable geometric forms
13. Geometry counts correct
14. Shells consist of single connected facesets
15. Shells in lumps
16. Loops in faces and face/surface orientation correct
17. No unnecessary topological elements (empty subshells, wire, spur edges and co-curve vertices)
18. Spatial occupancy measures (boxes) correctness checked
19. Body or lump self-intersecting or non-manifold
20. Consistent information associated with the model and model elements

When testing a body, tests 1, 2, 7, 13, 18, and 19 are relevant directly for the body, and then the individual wires and lumps contained in the body are checked separately. For a lump, all tests except tests 1 and 13 are relevant if the lump is being tested in isolation. For a wire model only, tests 2, 5, 7, 8, 9, 10, 12, 18, and 20 are relevant, the other checks are ignored. When testing partial facesets, tests 3, 4, 5, 7, 8, 9, 10, 11, 12, 16, 17, 18, 19, and 20 are relevant.

Transformation check

Check 1 is a numerical check independent of the structure, so it can be performed without checking that the basic model structure is traversable.

ACIS separates the transformation information into separate pieces: a 3×3 rotation matrix, a single scaling value, and a translation vector. There are also three logical labels: rotate, reflect, and shear. Numerical checks examine the values in the matrix to exclude obviously erroneous values, i.e., to check that the determinant of the rotation matrix is 1 or -1 (for reflection), and that

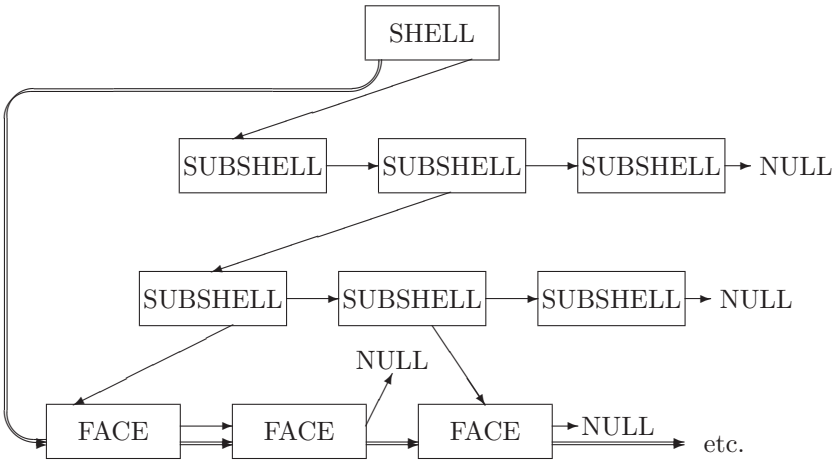


Figure 14.5: Shell-subshell-face structures

the scale value is non-zero. Consistency checks ensure that the information contained in the labels is consistent with the information in the rest of the transform.

Topological checks

Checks 2–4 verify the simple lists connecting the elements in the body. Checks 5 and 6 verify the geometrically ordered lists. As stated, these checks verify whether the connectivity of the model is correct. If the connectivity is correct, then it is safe to traverse the rest of the structure applying the other checks. If it is incorrect, then it is safer to stop and allow the user to repair the model if possible. However, a circular list where a simple list is expected could cause infinite looping and make examination and repair by the used difficult, so these should be broken if found.

The simple lists in the datastructure are as follows:

Wires in body

Lumps in body

Shells in lump

Coedges in wire

Checking these lists involves building lists of the dependent entities following 'next' pointers. If an entity is found that has already been inserted, then this indicates a circular list. With genuine circular lists, the procedure is similar, the entities found are added to a list, and the process terminates when a reference is found to an entity already in the list. For a circular list

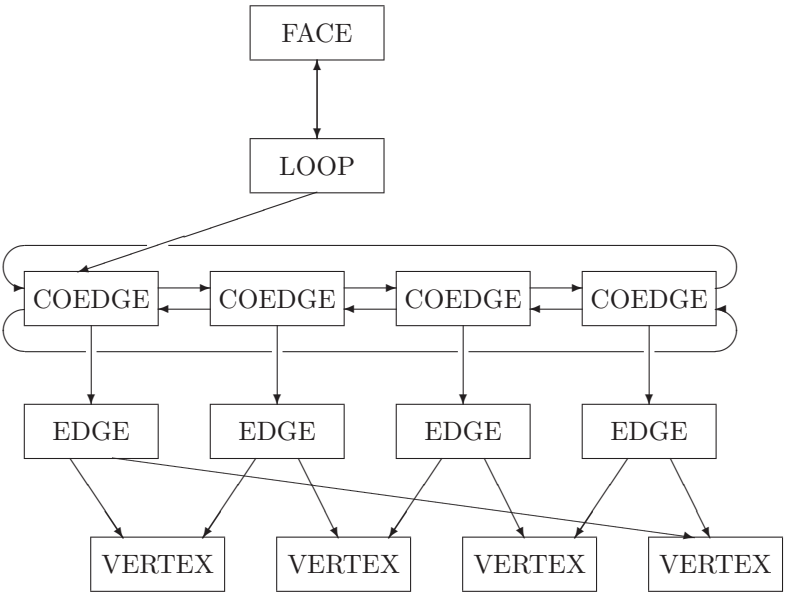


Figure 14.6: Loop-coedge-edge structure

to be valid, only the first element may be referenced, and the list must not terminate with a NULL pointer.

For the elements in each of the lists mentioned above, there must be a pointer back to the owner so that the datastructure can be traversed properly. This is easily checked while building the lists.

The face tree-structure associated with each shell can be handled similarly, although it is basically recursive. Each shell has an associated list of faces and a list of subshells; either may be empty. Each subshell also has a list of face children or subshell children, and so on (figure 14.5). All faces within a shell should be linked together in one chain and have consistent pointers back to the owner shell. Faces directly under a shell must be linked in a separate chain. Subshells in a shell must be linked together and have pointers back to the owner shell. Faces in a subshell must be linked into a chain and have pointers back to the owner subshell. The check works basically in the same way as the previous check.

A separate question concerns the validity of empty shells or subshells. Theoretically they are not really erroneous but merely superfluous. Hence it was decided not to deal with them here, but to report them in check 17, the check for no redundant topology.

The check that loops are single edge sets verifies the loop-coedge relationship. Each loop should normally consist of a single doubly connected closed ring of coedges (figure 14.6), although ACIS does allow incomplete loops as

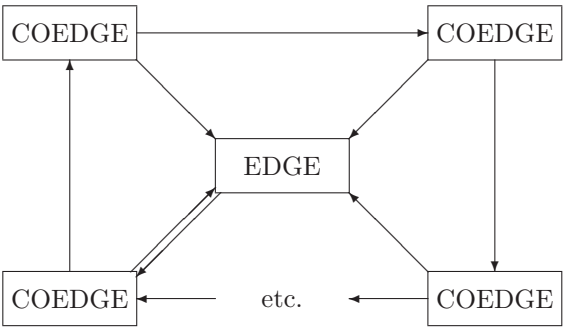


Figure 14.7: Structure of coedges round edge

well. Each loop is checked by checking the first coedge (to see whether it refers to the loop) and then using the coedge chains to find successive edges, checking that adjacent edges have both a common vertex (also with the correct orientation) and refer to the loop. The coedges are ordered counter-clockwise around the loop (using the next() relation), either in a closed ring or terminating with a NULL pointer. The test uses the coedge chain together with a vertex for checking consistency. To start checking, the first coedge in the loop is taken, the edge to which it refers is found, and the vertex counter-clockwise from this edge is obtained. This vertex should, at any step, lie between the current edge and the next edge referred to from the chain of coedges.

Check 5 verifies that coedges are correctly ordered around edges, i.e., that adjacent coedges around an edge should form pairs bounding solid regions. The general arrangement is illustrated in figure 14.7. If a small circular slice, centred on the edge, is made through the body then the elements of material around the edge form a set of notional wedges around the edge. There are two basic questions: whether the material segments are correctly ordered and if there is 'material-within-material'. See figure 14.8.

The check is performed for each edge. Edges with two or fewer coedges can be ignored. For the other edges, to check whether the coedges are correctly ordered involves using a geometric check. The angles of the tangent planes to the faces meeting at the edge can be calculated, and it can be checked that they form the correct sequence around the edge. However, there is a special case: if the model is a sheet model, then the tangent plane angles of the matching faces will be the same, but the next check is enough to decide whether the two coedges of these matching faces are correctly ordered. One special condition that has to be noted is where one of the faces at the edge is a double-sided face. With such faces, a single face is equivalent to a back-to-back face pair.

To check whether there is 'material-within-material' is relatively straightforward, because every coedge should be oriented in the opposite direction

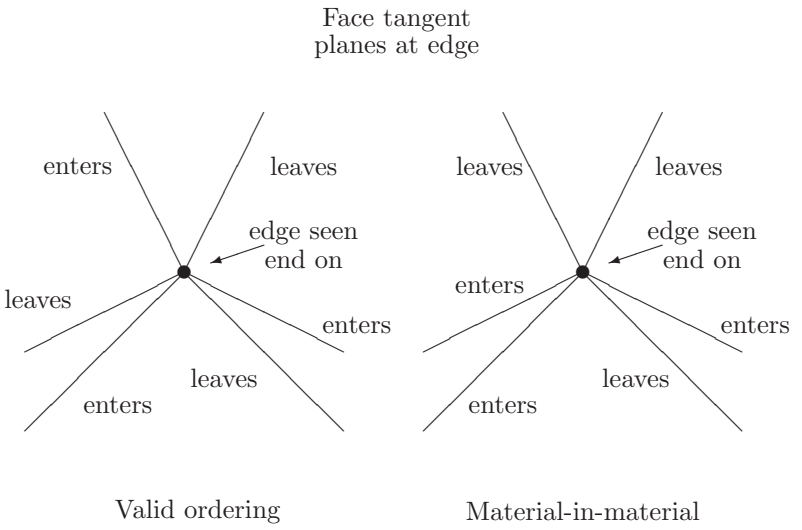


Figure 14.8: Coedge ordering round edge

to its neighbours. Checking this involves traversing all coedges around the edge checking their orientations, again making due allowance for double-sided faces.

Check 6 is the equivalent check for vertices, i.e., that the edges meeting at the vertex form single edge sets. The method of performing the check is analogous to that for checking loops. The first edge at each vertex is accessed, and then the coedges are used to find adjacent edges. For each edge, it is necessary to check that they refer to the vertex and that the coedges have a loop in common. For non-manifold vertices, where there are several edge sets at each vertex, each edge set has to be checked separately. For non-manifold vertices the vertex refers to a list of edges. Each edge in the list should be adjacent to a separate faceset, and edges should not be duplicated in the list. For each edge in the list, the edges are traversed clockwise or counter-clockwise until the original edge is found in the same way as for manifold vertices. Adjacent edges round the vertex are found using coedges. See figure 14.9.

Geometric checks

If checks 2–6 found no errors, then the datastructure can be traversed in relative safety. The next set of checks, checks 8–13, are general geometric checks before checking interactions.

Check 8 verifies that no degenerate geometry is present in the model. This is a check to see whether there are any zero-length edges or zero-area, narrow-, or self-intersecting faces. These conditions may affect performance

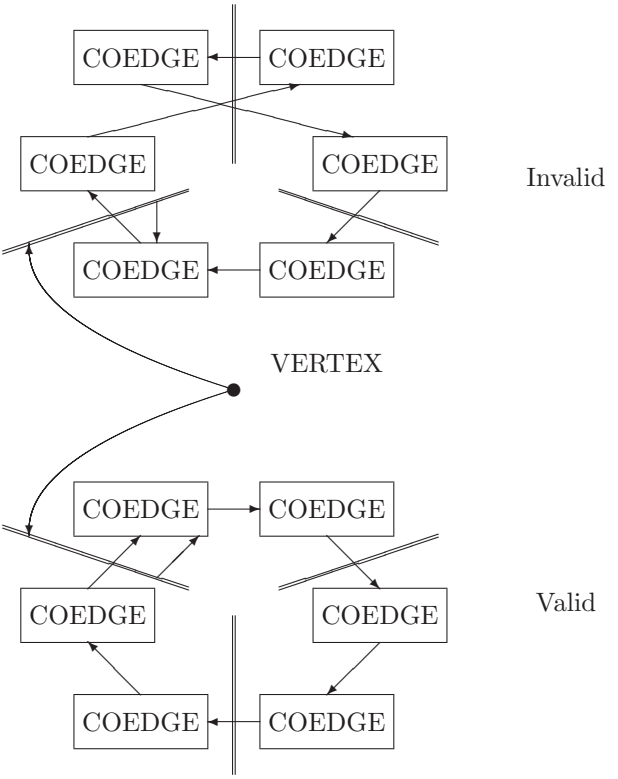


Figure 14.9: Edge sets around vertices

of modelling operations such as the Boolean operators. This check should also verify that all edges and faces have geometry attached, except for the special case of edges used to represent isolated points. The distance between the vertices along the curve is measured to check that the edge has a length greater than the tolerance.

Checking for faces that coincide internally is related to the Boolean operations. It involves checking all edges in pairs to see whether they intersect in points within the edges. Checking for narrow faces involves traversing vertices, examining the edges meeting at each to check that neighbouring edges are suitably spaced. If, at a distance greater than the tolerance (resabs for ACIS) from the vertex along the neighbouring edges, the separation between the edges is less than the tolerance, then there is a potential problem. If the separation is too small, then this means that a curve cutting through the face close to that vertex, but at a distance greater than the tolerance from it cuts the face boundary at points less than the tolerance apart. So, what can be considered as a single intersection point geometrically does not lie at a vertex, creating an awkward special case.

The next check, 9, verifies that the geometric information in the model is consistent. This involves traversing all edges in the lump or wire, or adjacent to faces in the faceset checking that the curve of the edge lies in the bounding surfaces and that the positions of the end vertices lie on the curve and on the surface. Obviously this check has to be performed within a given tolerance where the geometry is approximate and to allow for small, acceptable inaccuracies. Again, the ACIS resabs distance tolerance was used for this purpose. This is useful for checking that retrieved models, or models transmitted from another system, are acceptable within the current modelling tolerance of the system. It is also necessary to see whether the parameter values for the vertices, recorded in the datastructure, are correct and to test the correctness of coedge geometry.

In ACIS, the coedges, the links between edges, and loops, can have geometry associated with them as well: so-called "pcurves". Pcurves are parameter-space curves that provide a dual shape representation for intersection curves together with the three-dimensional edge curve. They are, therefore, closely associated with surfaces, and it is necessary to check that the curve shapes 'agree'. The curve parametrisations are not necessarily the same, but the pcurve should not be shorter than the curve of the edge. The pcurve representation is as a two-dimensional free-form curve, whereas that of the edge is a three-dimensional free-form curve. The check involved marching along the two-dimensional curve, converting the two-dimensional points to three dimensions using the associated surface representation, and checking that the three-dimensional point was on (or sufficiently close to) the three-dimensional curve.

There are two checks on the free-form geometry: a check on curves and a check on surfaces. The checks are related and verify that the portion of a curve or surface referred to is continuous and not self-intersecting. The check

on the curves is relatively simple, because there is an ACIS function to check free-form curves. The built-in check on surfaces, though, was incomplete. As implemented, the check examines the control points and reports an error if there are coincident control points. Because of the ACIS philosophy of separating the free-form geometry from the modeller, as far as possible, it was not possible to implement a geometric check based on any particular geometric form. This has to be done as part of the free-form geometric package.

Check 12 verifies the numerical form of the geometry, i.e., that the data defining the non free-form geometry are 'viable'. Possible geometric format problems are as follows:

1. Unit vectors with a magnitude different (more than 'resnor') from one
2. Non-orthogonal axis vectors in ellipses (i.e., dot product greater than resnor)
3. Inconsistent sine and cosine of semi-angle in cones (i.e., sum of squares greater than resnor from 1).
4. Invalid radii in ellipses (and hence also cones), spheres, or tori

The final check on the geometry, check 13, verifies that the geometry counts are correct. The geometry 'use counts' record the number of times that geometry is used so that shared geometry is not deleted while needed. These use counts can be checked by examining the number of times the geometry is actually used and comparing this with the recorded use count. The check is only reasonable if geometry is used in one body only and is only relevant when checking a complete structure.

General checks

Once the topology and geometry have been checked, it is possible to verify model structures. Checks 14 to 16 are checks to verify that the topological relationships are geometrically correct. Checks 17 and 18 perform 'marginal' checks, check 17 testing for unnecessary topology and check 18 testing the correctness of the boxes in the model.

Check 14 verifies that shells (groups of faces) consist of single connected facesets, i.e., that each shell contains only a single closed set of faces, edges, and vertices, and that connected topology is not spread over two or more shell entities. To perform the check, a list is compiled of faces that are adjacent across edges. This list is then checked against the shell pointer structure to see that 1) the faces all refer to the same shell and 2) that the shell consists of only those faces in the list.

Checks 15 and 16 are similar. Check 15 verifies that shells are not disjoint, and check 16 ensures that loops are not disjoint. Neither check is sufficient on its own, because they both use a simple test (point-in-body and point-in-face) test to classify containment, so self-intersections are not found. However, check 8 verifies whether edges in faces collide, and geometric self-intersection

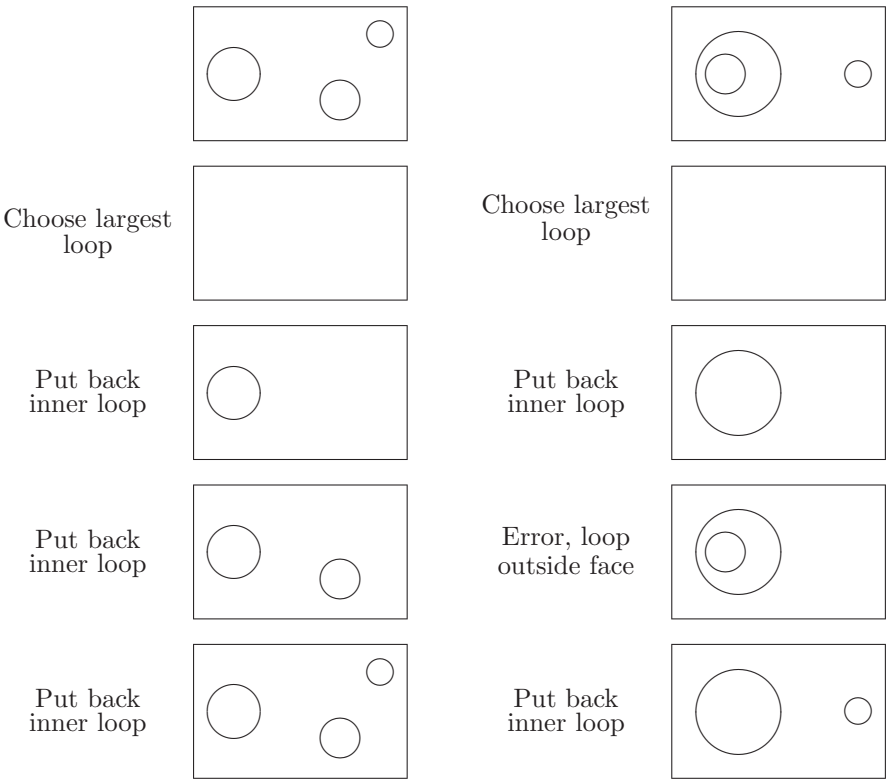


Figure 14.10: Loop containment testing

is tested for in check 19, which is why these checks do not perform a full test. The procedure for testing loops is illustrated in figure 14.10. The column on the left shows a valid loop set whereas the column on the left shows an invalid loop set for a face.

The check for shells in a lump involves finding the containing shell and then checking the other shells against it. When finding the containing shell, there are two cases to note: 1) when the object is positive and 2) when it is negative. Usually positive objects will be checked; in which case, the containing shell is the largest. For negative objects, the containing shell is the smallest. Checking whether an object is negative is done using the point-in-body test. A point is chosen that lies well outside the box containing the body. If this point is classified as 'inside' then the object is negative. The shells in the body are ordered according to size and checked. After each shell has been tested, it is reinserted into the original lump structure to pick up errors where one internal cavity shell surrounds another.

Check 16 verifies that loops are contained in faces. It involves basically the same steps as the previous test. First of all, the largest loop is found and then the other loops are tested to see that they are contained within it using the point-in-face test to classify one point of each loop. As with shells, the loops are initially separated from the face and then successively put back as they are tested. One small problem concerns complete cylinders where the curved face is bounded by two disjoint loops in ACIS. When the first loop is chosen it essentially divides the cylindrical surface into two infinite portions: one 'inside' the face and one 'outside'. The point-in-face test has to be able to cope with such partially bounded faces using the above strategy. Check 16 also tests that the face/surface orientations are consistent.

Check 17 tests that there are no unnecessary topological elements. If such unnecessary elements are present, it does not really make the model invalid; it just means that it is not minimal. Unnecessary elements are empty shells and subshells; adjacent faces lying in the same surface; adjacent edges lying in the same curve, where those edges are the only ones meeting at the common vertex; edges that appear twice in the same loop; or isolated points where there is no surface discontinuity.

The final check in this section, check 18, tests that the ACIS spatial occupancy measures, boxes, are correctly set. Boxes are used for quick interference tests, which is useful for improving efficiency in, for example, Boolean operations, so incorrect boxes may cause incorrect operation. They are tested rather than merely corrected to check that an operation has not modified the geometry or topology without deleting or resetting the box.

Final checks

Once the structures of the model have been verified, it is necessary to check whether the model is self-intersecting. This check is related to the Boolean operations, so it is easiest if it is built into these. However, in the body checker implemented around ACIS, it was not possible to alter the existing operations as these are part of the ACIS kernel. Instead it was necessary to perform the

same sort of tasks using subsidiary models.

The method implemented copies each face as a sheet object and then intersects these with each other using the basic method described by Braid [11] and in chapter 6. Each sheet pair is intersected with each other, and the results are compared back with the original faces so that intersections corresponding to existing edges can be removed. If any intersection results remain, then the faces corresponding to the sheet object pair are recorded as self-intersecting.

The last check on the structures is to verify that there is consistent information associated with the model and model elements. It is not possible to predetermine what kind of information will be associated with a model, and hence, it is not possible to check the validity. What can be done is to check for duplicate information, although this may not be an error. Unfortunately there was no way of automatically verifying the information content of attributes.

General model scan

Finally, the model is scanned to provide an overview of the entities in the model. The scan provides brief information about the object dimensions, the number of different types of entity in the model, non-manifold edges and vertices, partially bounded topology and the geometry types.

Communicating results

There were two main choices for communicating the body checker results: to output them directly or to return pointers to erroneous entities in datastructures. It was decided not to output results directly. The reason was that it was felt important to enable the body checker to be called by a user program and return problem entities for automatic correction. The body checker interface functions return a pointer to the head of a list of result records, one for each check (if no errors were found) or one for each error found. A utility for formatting the result records was also developed to provide a readable version.

14.1.4 Final remarks

It is possible to detect many potential problems in models. Automatic model verification encapsulates knowledge about the nature of modeller requirements about the model that can be used to highlight problems. It is not always easy to identify potential problems manually, especially in complicated models where problems may not be obvious from images, e.g., small tolerance problems or bad surface orientations.

The checks rely partly on basic considerations about the nature of the model and partly on 'redundant' information in the datastructure. This means that there can be positive benefits to including redundant information in the datastructure: analogous to the inclusion of extra bits in error checking codes.

Verification in kernel modellers, such as ACIS, is more difficult than checking in modellers linked to a specific application because kernel modellers contain general mechanisms. Particular systems can exclude certain conditions, such as incomplete models, which kernel modellers may allow. This section is intended to describe some general principles for determining checks in a modelling system as well as specific checks determined for the ACIS modelling system.

14.2 Model 'healing'

Model healing is more difficult than model verification. What this means, in effect, is to patch up the model and repair the errors found in the checking stage. There is a limit to what is possible using automatic methods, so an option is to provide a model editor to repair a model manually. Obviously this is dangerous and needs a lot of support and to be restricted in use. Such an editor would allow pointers to be set, numerical values to be changed, and so on. This would probably only be useful for system developers to solve client problems.

Based on the list from section 14.1.3, a list of checks for healing for a general datastructure with assemblies might be:

1. Instance transformations correct
2. Instance structure correct (no instance loops)
3. The basic body lists valid
4. Faces correctly partitioned
5. Loops consist of single edge sets
6. Loop-edge links correctly ordered around edges
7. Vertices have adjacent single edge sets
8. No degenerate geometry present
9. Consistent geometry (curves, surfaces, points)
10. Curves continuous and no self-intersection on edge
11. Surfaces continuous and no self-intersection in face
12. Numerically stable geometric forms
13. Geometry counts correct
14. Shells in bodies
15. Loops in faces and face/surface orientation correct

16. No unnecessary topological elements (empty shells, wire edges, spur edges, and co-curve vertices)

17. Spatial occupancy measures (boxes) correctness checked

18. Body or lump self-intersecting or non-manifold

19. Consistent information associated with the model and model elements

14.2.1 Assembly healing

Assemblies are relatively simple structures. An assembly is a list of instances. Each instance has a reference to a transformation and to an object or subassembly.

Instance transformations correct

For an assembly, all transformations should be translations and rotations. This may be controversial because some systems allow other transformations, but a personal view is that it is bad practice to allow shape-changing operations in an assembly. In real physical assembly, only rotations and translations are possible. If a part model should be another shape, then this should be done at an earlier stage. If the transformation is not a combination of rotations and translations, then the object of the instance should be copied, if necessary, and the transformation used to change the geometry of the object.

Transformations that have a zero determinant should be simply removed.

Instance structure correct (no instance loops)

This should be checked while building the structure, so it should never happen, but a bad structure could be imported. No assembly should contain a reference to itself, and no subassembly should contain a reference to itself. Any such reference should be removed along with the instance referring to it.

14.2.2 Topological healing

The basic body lists valid

All simple lists should end with a NULL pointer or a pointer to the first element in the list. If a pointer to an earlier is found, then that pointer is replaced with a NULL pointer or a pointer to the first in the list. If owner pointers are included, then these can be set, if necessary, during the traversal.

With tree structures, the same process is used. The structures are traversed depth first. Pointers to elements already in the list are replaced with NULL pointers, or pointers back to the first element at the same level.

The difference with doubly linked lists is that there are two lists that should have the same elements but in different directions. The lists are traversed

putting the elements into a separate list structure. Then the links are re-established from this list to make sure that all elements are linked.

With multiple lists there is also the possibility of cross checking the elements. For example, the list of all faces in the object can be traversed, building a list of all edges. This may also need to be done if one object structure is imported to another system with different list structures. The list of all vertices can be traversed, adding all edges at the vertices to the list of edges. Finally, this list can be compared with the list of all edges in the object if one exists. Any extra edges found can be added to the list of edges in the object. Note, though, that the edges in a face and the edges around a vertex may not be correct and traversing these should be done with care.

Similarly, the list of all edges in the object can be traversed, adding the faces of each edge to a list. This list can be compared with the the list of all faces in the object. Any extra faces can be added to the appropriate shell. This last sentence hides extra work. To find the correct shell, it is necessary to recheck the edges. If an edge has two faces, one face in the list of object faces and the other not in the list, then the face not in the list can be added to the same shell as the 'known' face. If the edge is a non-manifold edge, then the faces adjacent to the edge have to paired correctly. Face pairs with one face in a known shell and one unknown face can be repaired by adding the unknown face to the known shell. Any other faces need to be divided into linked sets and assigned to shells, if necessary. These new shells need to be added to the list of shells in the body.

The list of edges in the body can also be traversed to create a list of the start and end vertices of the edge. This is compared with the list of all vertices and any unknown vertices linked in to the body-vertex list.

Faces correctly partitioned

The face partitioning is done by comparing the faces with all neighbours to create linked sets of faces. Each linked set should correspond to one shell, and each shell should have one linked faceset. If this is not so, then shells are created to correspond to each faceset. A part of this is done in the previous repair, when the edge lists are used to check that all faces are linked into the object. Note, too, that this check relies on the face-edge connections using the loops. If these have not been checked, then care is needed when traversing them.

Loops consist of single edge sets

The loops can be checked first to see that the lists of loop-edge links is a correct doubly linked list in the manner described earlier.

A list of edges referring to a loop can be compiled from the list of all edges and this compared with the list of edges from the loop-edge links. Neighbouring edges in the loop should share at least one vertex. There may, however,

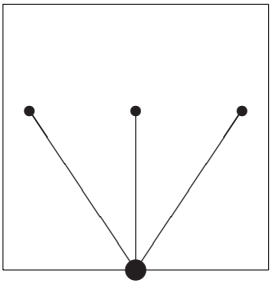


Figure 14.11: Multiple loop edges at a vertex

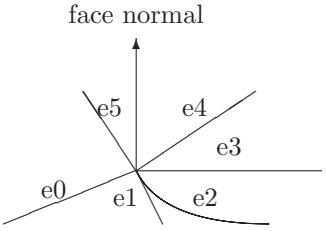


Figure 14.12: Sorting edges at a vertex

be multiple edges in the same loop that share a vertex, as in figure 14.11.

If this occurs, then it is necessary to check the order of the edges around the vertex, as described by Müller in [89], using the normal of the face surface at the vertex position. See figure 14.12.

The next edge counter-clockwise around a loop from a given edge is the next edge clockwise around the vertex from that edge.

Loop-edge links correctly ordered around edges

This is similar to Müller's edge ordering method for edges around a vertex. The edge directions into each face around the edge are used for the check in the same way as the curve tangents of the edges around the vertices are used. Also, as with the curves, curvature is also needed to sort out cases where neighbouring directions are coincident, as with edges e1 and e2 in figure 14.12.

Vertices have adjacent single edge sets

The relationships of edges around vertices is found using the winged-edge links or the loop-edge link entities. What is also needed for healing is to make sure that every edge set referring to the vertex is referred to by that vertex. If the vertex is a manifold vertex, then there will only be one edge set, but for non-manifold vertices, several edge sets occur. The same process is used, and

the list of all edges in the object is traversed to build a list of edges referring to the vertex. This edge list is then divided into groups of edges using the edge-loop links. This is done by picking one edge from the list, and then the 'ecwev' or 'eccev' operation is used to find neighbours that are put into a list, until the original edge is found. The list of all edges referring to the vertex is then scanned to see whether there are edges not in this secondary list. If there are, these are treated as seed edges to find other secondary sets until all edges are in secondary sets. One edge from each secondary set is then put into a list of edge references from the vertex.

14.2.3 Geometric healing

The geometric healing methods establish the correctness of the geometry.

No degenerate geometry present

Examples of degenerate geometry are zero-length edges or zero-area faces. A zero-length edge (figure 14.13a) needs to be deleted, merging the vertices at either end (figure 14.13b). If there are zero area faces (figure 14.13c), then some edges (the zero-length ones) are deleted in these faces leaving a set of coincident edges (figure 14.13d) that are then merged (figure 14.13e). The merging process involves identifying vertices where there are coincident adjacent edges in the loop. These are then merged. If the vertices at the opposite ends of the edges are in the different places, then the longer edge is split so that the two coincident edges are the same length before they are 'sewn' together.

Consistent geometry (curves, surfaces, points)

The geometric consistency repair involves making the curves of edges lie in the surfaces of the faces at the edge and the vertices lie in the surfaces of all faces meeting at the vertex. The difficulty comes when this is impossible with the topology of the object as given.

In its simplest form, the repair just means looking at all edges and intersecting them to produce the correct curve. All vertices are then traversed, intersecting the curves of the edges meeting at the vertex to produce a common point that is assigned to the vertex.

One problem comes with numeric geometry if the curve of an edge is the limit geometry of a patch that does not reach to the other surface. In this case there is no intersection result when intersecting the left and right surfaces. It is, however, possible to extend numerical surfaces if this happens and, hence, to obtain an intersection result. Extension, though, does not guarantee an intersection result. If the surfaces are almost tangent, then the surfaces might miss each other anyway. In this case, it is necessary to establish a bridge between them.

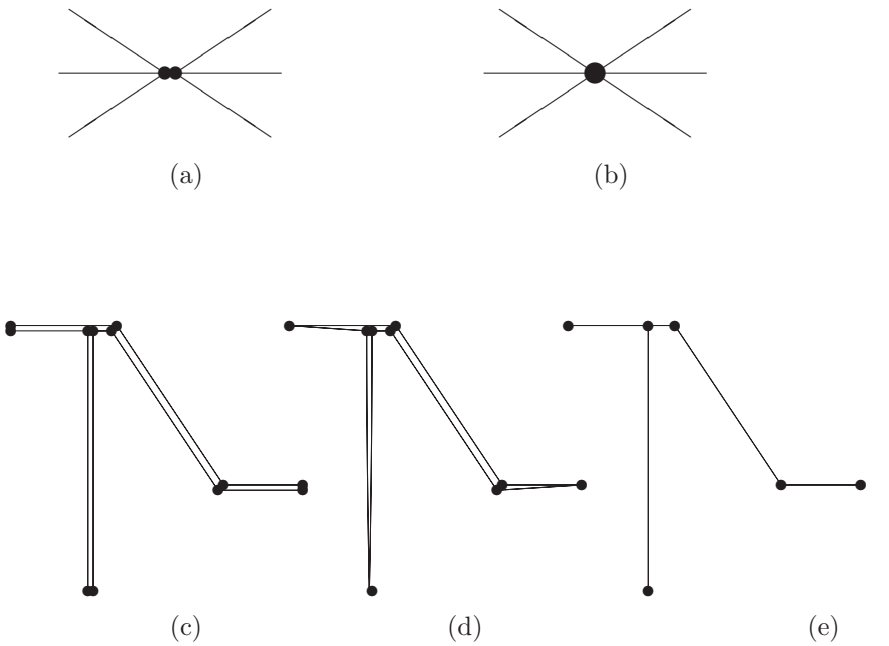


Figure 14.13: Degenerate edges and faces

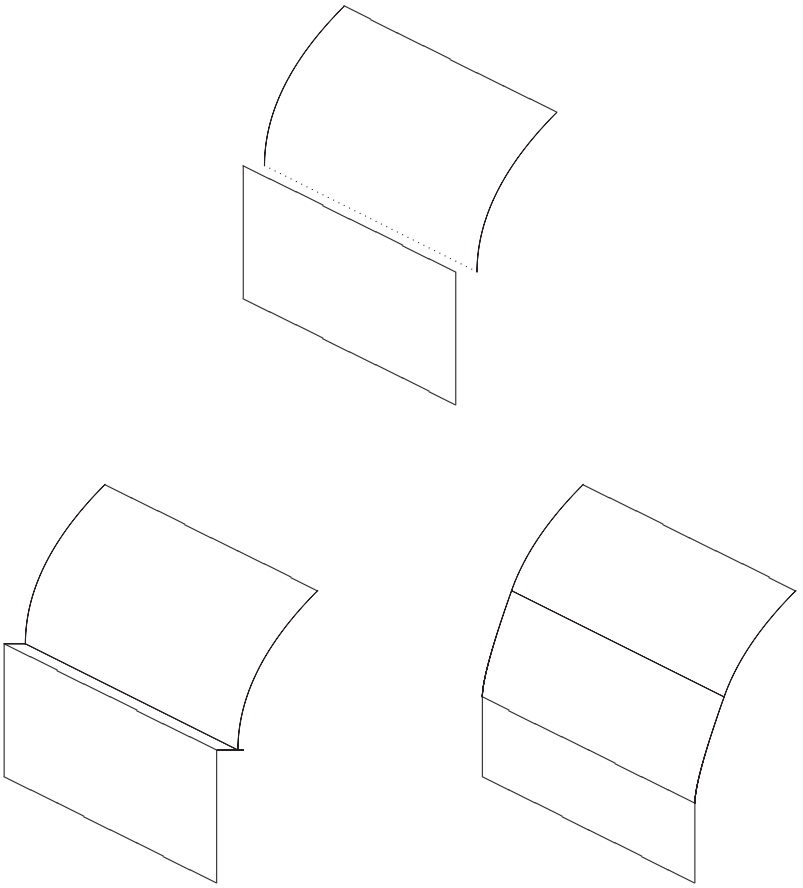


Figure 14.14: Edge healing at non-intersecting surfaces

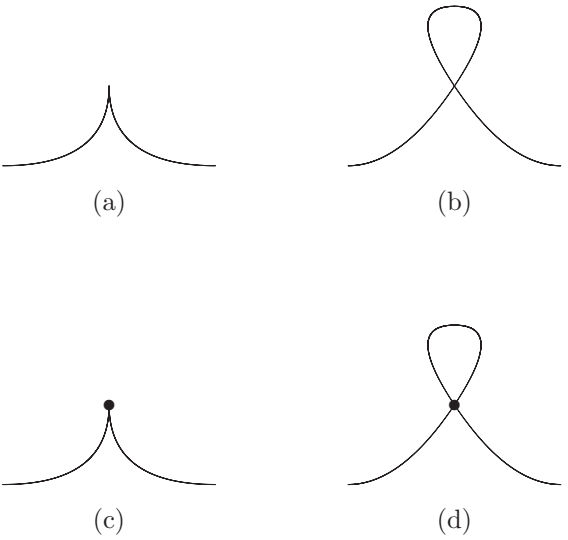


Figure 14.15: Adding vertices at discontinuities and self-intersections

This is a dangerous solution, though, because the new face is a long narrow one that may disturb other operations. Another related solution is to replace it with a larger blend face that then provides a larger face. Both of these solutions, though, disturb the vertices at either end by adding additional topological elements. See figure 14.14.

When treating vertices, it may be necessary to add 'eaves' as in Jared's chamfer algorithm. These eaves are three-sided faces linking the ends of edges that do not meet at the vertex to the vertex.

Curves continuous and no self-intersection on edge

If there is a discontinuity in the curve (figure 14.15a), then a new vertex should be added at the discontinuity (figure 14.15c). Similarly, if there is a self-intersection of the curve (figure 14.15b), then the edge should be split with a new vertex (figure 14.15d). Note, though, that this split involves dividing the edge into three pieces, so the edge has to be split twice and the two new vertices merged.

Surfaces continuous and no self-intersection in face

This process is analogous to the case of discontinuities or self-intersection on edges, but an edge or vertex needs to be added. For a face with a conical surface with the apex in the face, the apex needs to correspond to a vertex. Otherwise, new edges are added at discontinuities and at self-intersections.

At self-intersections, the edge needs to be a non-manifold edge.

Numerically stable geometric forms

An example of where numerical stability of geometric forms is important is if a straight line is defined by two points. The closer the points are, the less stable the form of the line. Similarly, normal vectors defined by three points that are arranged in a tight vee formation, such as in long thin triangles, are subject to instability. Repairing the geometry involves identifying such cases and replacing the unstable elements as far as possible.

Geometry counts correct

Geometric elements have counts to identify multiple usage. The repair method is to traverse all elements, i.e., vertices, edges, and faces, checking how many times the corresponding points, curves, and surfaces are used. The counter of the geometry is simply replaced with the calculated count.

14.2.4 Combined healing

Combined elements need both geometry and topology to work.

Shells in bodies

The shell-in-body repair means sorting the shells and handling overlap. The largest shell, measured using the bounding box, is taken as the outer shell. This needs to be checked to see whether it is positive or negative. Assuming that it is positive, the rest of the shells are sorted in descending order of size of bounding box. For each shell, it is necessary to check whether the shell is positive or negative and a point needs to be checked for containment in the body. A positive shell inside the body needs to be moved to a new body. A negative shell outside the body needs to be checked with new bodies, or made into a separate body.

Overlapping shells are handled with the self-intersection evaluator.

Loops in faces and face/surface orientation correct

This is analogous to the 'shells-in-bodies' repair. Loops are sorted in descending order. The largest is taken as the perimeter and the others checked for containment. If a loop lies outside the face, then it should become a new face.

14.2.5 Making things nice

Removing superfluous topology

The BUILD system had an operation, "makenice", which was used to 'repair' a model after some local operations. Making things nice means removing

extraneous topological and geometrical elements. This is sort of related to checking and healing because the extraneous elements are unnecessary.

In general an object is considered to be minimal; that is, it should not contain edges lying between faces lying in the same surface, and it should not contain vertices between edges lying in the same curve, providing there are no other edges at the vertex. Of course there are some exceptions sometimes with curved geometry. One basic decision that needs to be made is how to represent a cylinder, for example. In the BUILD system, there were three 'fake' edges around the cylinder to avoid having faces extending through more than 180 degrees. Some other systems like two 'fake' edges (to avoid having wire edges) or one 'fake' edge to have at least one edge around the cylinder.

The general method can be described as

for all edges in body(b, check edge);

where the **check edge** function is

```
if ( spur edge(e) ) remove spur edge(e);
else if ( wire edge(e) ) remove wire edge(e);
else if ( unnecessary edge(e) ) remove edge(e);
```

The definition of the **unnecessary edge** function is

```
lf = face OF left loop OF e; rf = face OF right loop OF e;
ls = surface OF lf; rs = surface OF rf;

if ( ls IS rs ) return TRUE;
else if ( identical surfaces(rs, ls) AND
NOT basic dividing edge(e)) return TRUE;
```

where the **basic dividing edge** function checks for edges that are needed for curved geometry, as described above.

The same is true for for vertices. It is possible to write this as

for all vertices in body(b, check vertex);

where the **check vertex** function is

```
if ( number of edges at vertex(v) == 2 )
BEGIN
e1 = edge OF v; e2 = ecwev(e1, v);
if ( (e1 ISNT e2) AND identical curves(curve OF e1, curve OF e2) )
kill vertex(v);
END
```

Spatial occupancy measures correctness checked

The spatial occupancy measures are used for improving the efficiency of Boolean operations, for example. They are simply recalculated for all model elements for which they are used.

14.2.6 Self-intersection repair

The self-intersection repair is a form of Boolean operation, described in section 6.1. Instead of using face lists from two bodies, the face list of one body is checked against itself.

The general procedure is

```
for all faces in body(b, check face(f));
separate body along boolean boundary;
recombine body parts;
```

The face checking procedure is:

```
check face(FACE f0)
for all faces in body(b,
if ( f0 ISNT current face) intersect faces(f0, current face);
```

The "intersect faces" operation is the same as for normal Boolean operations except that existing edges between the two faces being compared should be ignored. The "separate body along boolean boundary" and "recombine body parts" are also the same as for the Boolean operations.

14.2.7 Information repair

This is to ensure that consistent information is associated with the model and model elements. As pointed out in chapter 8, information is difficult to handle automatically. Apart from checking common information types, like material, surface finish, or thread notes, for example, there is little that can be done.

If the same information entity type occurs twice for an entity, then it is likely that one of these should be removed, but not definite. For example, it makes little sense to have two material specifications for the same body. It is also dubious to have two surface finish specifications for the same face.

A shape modifier such as a note to specify that a hole is to be threaded can be checked to see whether it is attached to a face in a cylindrical surface, for example.

Apart from these simple examples, it is difficult to see how to create a general automatic information repair procedure because there are so many possible information elements.

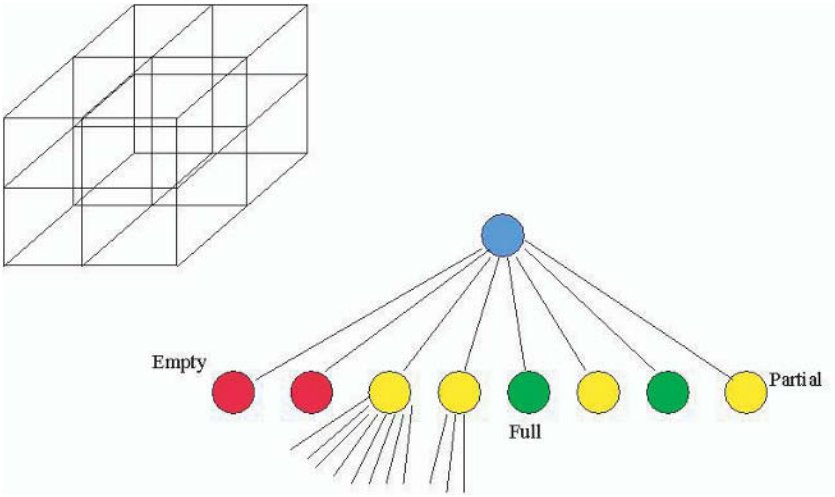


Figure 14.16: Octree subdivision

14.3 Generating an octree model from a B-rep

Octree models are described briefly in chapter 1. To approximate a model, the first step is to create a cube surrounding the model. If the model does not fill the cube, the cube is subdivided into eight subcells and the process is repeated. The result is a tree of cells and subcells. Subcells that are full or empty are not subdivided; subcells that are partially full are subdivided until some size limit is reached. This is shown schematically in figure 14.16.

One way of generating an octree model from a B-rep model is to cover the exterior with points and then subdivide these points into sets corresponding to octree cells. It is also necessary to have a normal vector out from the body associated with each vertex.

Again, using a sort of pseudo-code to describe the algorithm, you get

```
for all faces in body(b, generate points with normals);
b1 = box point set(point set);
create tree from point set(b1, point set);
```

The function **create tree from point set** is defined recursively so that it can be called on the subdivided point sets:

```
if ( box size(b1) > limit )
BEGIN
subdivide point set(point set, pset0, pset1, pset2, pset3, pset4, pset5, pset6,
pset7);
sb0 = box point set(pset0); link node(b1, sb0);
```

```
sb1 = box point set(pset1); link node(b1, sb1);
sb2 = box point set(pset2); link node(b1, sb2);
sb3 = box point set(pset3); link node(b1, sb3);
sb4 = box point set(pset4); link node(b1, sb4);
sb5 = box point set(pset5); link node(b1, sb5);
sb6 = box point set(pset6); link node(b1, sb6);
sb7 = box point set(pset7); link node(b1, sb7);
if ( NOT empty(pset0) AND NOT full(pset0) )
create tree from point set(pset0);
if ( NOT empty(pset1) AND NOT full(pset1) )
create tree from point set(pset1);
if ( NOT empty(pset2) AND NOT full(pset2) )
create tree from point set(pset2);
if ( NOT empty(pset3) AND NOT full(pset3) )
create tree from point set(pset3);
if ( NOT empty(pset4) AND NOT full(pset4) )
create tree from point set(pset4);
if ( NOT empty(pset5) AND NOT full(pset5) )
create tree from point set(pset5);
if ( NOT empty(pset6) AND NOT full(pset6) )
create tree from point set(pset6);
if ( NOT empty(pset7) AND NOT full(pset7) )
create tree from point set(pset7);
END
```

The definition of an empty cell is one in which there are no points, or one where all points are close to the boundary or boundaries of the cell with their normals pointing towards the interior of the cell. The definition of full is that all points are close to the boundaries of the cell with their normals pointing away from the interior of the cell. Note the use of the fuzzy variable 'close' which is a parameter that it is necessary to translate into a numerical value. This might be 10% of the box size, for example.

14.4 Recreating a B-rep from an STL file

The basic method for creating a facetted model from an STL file is similar to that described by Mäkelä and Dolenc [80], although it was derived without knowledge of their algorithm.

STL was the first exchange standard used for rapid prototyping. It was developed by 3D Systems, the manufacturers of the first commercial machine for Stereolithography and manufacturers of special graphics systems. STL soon became accepted by other manufacturers as more machines became available. However, STL was not developed as a solid model exchange format but as a graphics exchange standard, but it is, even so, outdated. As graph-

ics applications have different needs from the rapid prototyping process, or layered manufacturing process, modelling problems arise with the simplified description offered by STL. As is well known, STL consists of a simple list of triangular facets with corner points with limited accuracy. The facets lack adjacency information, and it is easy that holes appear in the communicated model.

An example of an STL file is given below:

```
solid model STL
facet normal -7,219088e-001 6,919882e-001 0,000000e+000
outer loop
vertex -3,684905e+000 3,379567e+000 5,752512e+000
vertex -3,532432e+000 3,538634e+000 5,500000e+000
vertex -3,684905e+000 3,379567e+000 5,247488e+000
endloop
endfacet
facet normal -4,492708e-002 2,328569e-001 9,714727e-001
outer loop
vertex -5,000000e+000 6,123032e-016 3,125388e+000
vertex -5,091192e+000 -1,744887e-001 3,162994e+000
vertex -4,998532e+000 -1,211429e-001 3,154493e+000
endloop
endfacet

and so on... until

facet normal 2,802502e-001 9,551557e-001 9,558970e-002
outer loop
vertex -1,220263e+001 -3,386880e+001 2,861102e-014
vertex -8,031230e+000 -3,509272e+001 2,861102e-014
vertex -1,170019e+001 -3,331335e+001 -7,023252e+000
endloop
endfacet
endsolid model STL
```

Strictly speaking, all values for the vertex coordinates in an STL file should be positive, but this is not a requirement for the rebuilding process.

Ideally communication should be done at as high a level as possible; it is always possible to simplify later, but rediscovering information once it is lost is time-consuming and should not be necessary. However, because of current practice, it is necessary to consider the case of data exchange using STL. This format implies a loss of information. STL loses geometry and connections.

As shown above, the STL format is a list of triangles, but in fact they could easily be any simply connected polygon specified as a list of vertex positions without changing the basic procedure. This makes the procedure adaptable

for reading other file formats such as DXF or VRML (if the vertices are not shared). The next section describes reading VRML files with shared vertices.

14.4.1 Reading non-connected facet formats

The object recreation algorithm involves creating separated faces and then joining them together. This works as follows:

1. The file is read. For each facet, a B-rep face is created and the corner vertices are placed in a list.
2. If there are less than a certain number of vertices in the input vertex list (10 in the current implementation), the vertices are compared and coincident vertices are merged, merging matching edges at the same time.
3. If the list size is greater than the limit, then the list is split into eight cells containing sublists (using an octree-type approach) and each sublist is processed in the same way.
4. Vertices close to the boundary of each cell are compared with the boundary vertices of neighbouring cells.

The process is summarised in figure 14.17. After the first step, all triangles are separate (figure 14.17a). Vertices are merged successively, joining the triangles (figure 14.17b). When, after a merge, there are two edges with the same start and end vertices (figure 14.17c), these are merged (figure 14.17d). This process is carried out until all coincident vertices have been merged.

When the file is read, each facet is created as a separate face surrounded by three edges and three vertices, making the model a patchwork of disjoint faces. The main problem with recreating a solid is to match the vertices of the facets with the vertices of neighbouring facets. As there are usually thousands of facets in the file, this is very time-consuming if done simply by comparing every vertex with every other vertex. Instead a method based on octree decomposition can be used. The octree subdivision is illustrated in figure 14.18. The vertices are put into a list. This list is subdivided based on which octant of the box they lie. This process is repeated recursively for the subboxes until only a few vertices remain in the box. These are then compared and joined if they lie approximately at the same position. Vertices lying close to the boundary of the box are compared with the vertices near the boundary of the neighbouring box.

The vertex merging procedure involves moving all edges of one vertex to the other vertex. If after this operation any pair of edges has the same start and end vertex, then these edges are merged. Normally the edges being merged should lie in opposite directions, but if the facets have opposed normals, then the edges have the same direction. Aligned edge pairs should be recorded for later error analysis. See figure 14.19.

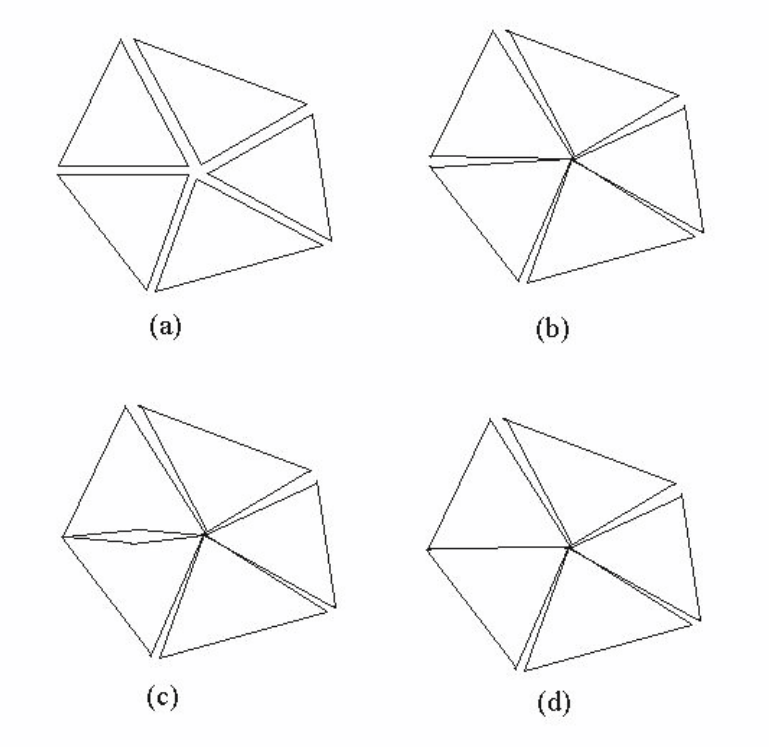


Figure 14.17: Model recreation from STL

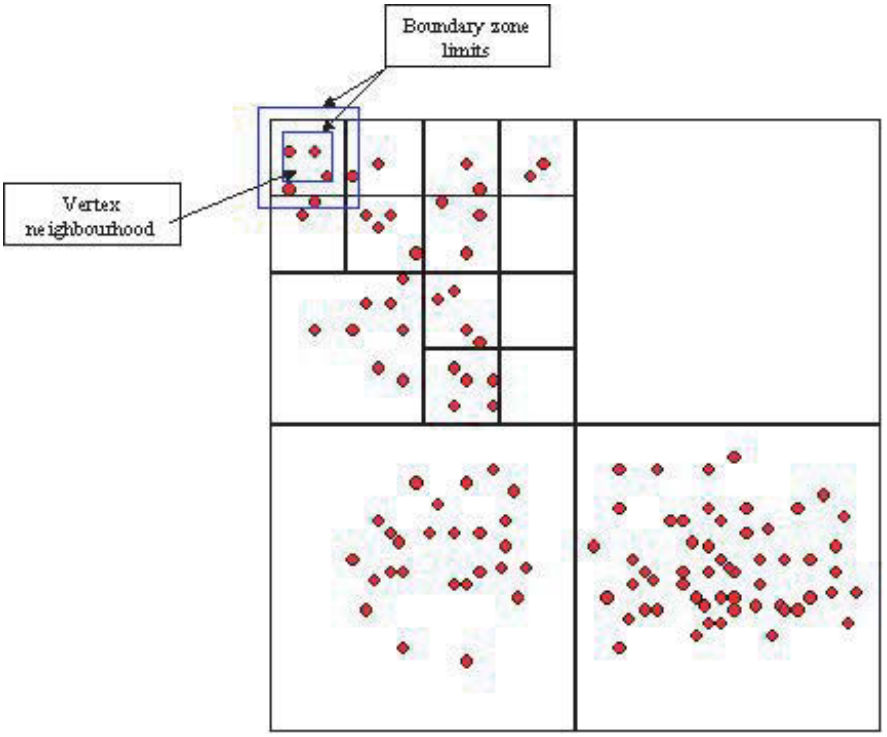


Figure 14.18: Octree spatial subdivision for vertex comparison

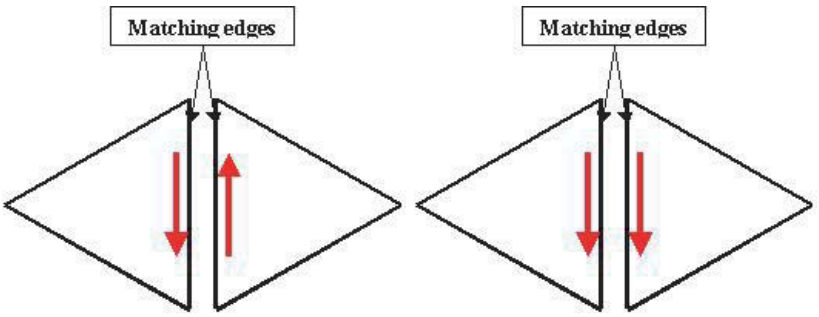


Figure 14.19: Opposed and aligned facet edges

Aligned edges are handled by keeping a record of the aligned pairs and post-processing. Once all normal edge-pairs have been joined, the list is accessed. If the edges are still aligned, that is, if they have not been handled, then one edge in the pair belongs to a negated object part and the other to a positive part. One of these object parts is chosen and negated. The edge pairs are then merged. When all aligned pairs have been treated, the object is checked to see whether it is negative; in which case, it is negated or positive.

Once all vertices have been joined, the structure can be analysed for errors. Edges with just one adjacent face indicate holes in the boundary. Edges with more than two adjacent faces also indicate problems. It may be that narrow facets have been generated incorrectly because the tolerance in the system that created the STL file is smaller than that in the receiving system. Another possibility is that there are internal walls in the object.

It is useful to keep a running record of the minimum and maximum x-, y-, and z- values during the recreation process.

14.4.2 Reading a shared vertex file

If a VRML file with shared vertices is to be read, then the whole procedure changes because vertices do not need to be matched. The file format consists, roughly, of a list of vertices followed by a list of faces specified as a list of corner vertex indices. When reading a face, it is necessary to check whether coincident vertices in the list are already connected by an edge or whether a new edge has to be created.

Imagine a file format such as

$$
\begin{array}{l}
points \\
(-10,-10,-10) \\
(10,-10,-10) \\
(-10,10,-10) \\
(10,10,-10) \\
(-10,-10,10) \\
(10,-10,10) \\
(10,10,10) \\
(-10,10,10) \\
facets \\
0\ \ 2\ \ 3\ \ 1 \\
4\ \ 5\ \ 6\ \ 7 \\
0\ \ 1\ \ 5\ \ 4 \\
1\ \ 3\ \ 6\ \ 5 \\
3\ \ 2\ \ 7\ \ 6 \\
2\ \ 0\ \ 4\ \ 7
\end{array}
$$

Writing such a format is relatively easy, but there are restrictions. It is suitable only for planar objects, so the object has first to be facetted, but the facets may have more than three edges. The facets need not be convex, but

they should not have inner loops. Wire and spur edges should be avoided, and the object should be manifold. None of these are impossible to cope with, but it makes things easier.

With these restrictions, the vertices of the object are numbered consecutively from 0 to $n-1$ (it could be 1 to n also; this is just a convention), where n is the number of vertices in the object. The vertices are then written, one per line, with just the coordinates. After this the faces are written, again one per line. For each face, the corner vertices are written, here in counter-clockwise order, using the vertex numbers mentioned above.

To read such a format, all coordinate positions are read and new vertices created, one for each set of coordinates. For the faces, the vertex numbers are read and edges are created between each pair, including the last to the first. However, each pair of vertices appears twice in the list, so it is necessary to check whether the vertices are already connected. If they are connected, then it is necessary to add an extra face reference to the edge.

Now look at the format again, this time with point and facet numbers. The numbers in the right-hand column are not part of the file format they are just there to make it easier to refer to in the text that follows:

points	
$(-10, -10, -10)$	*0*
$(10, -10, -10)$	*1*
$(-10, 10, -10)$	*2*
$(10, 10, -10)$	*3*
$(-10, -10, 10)$	*4*
$(10, -10, 10)$	*5*
$(10, 10, 10)$	*6*
$(-10, 10, 10)$	*7*
facets	
0 2 3 1	*0*
4 5 6 7	*1*
0 1 5 4	*2*
1 3 6 5	*3*
3 2 7 6	*4*
2 0 4 7	*5*

The eight vertices are read and created and put into an array or list from which they can be referenced by number. When the facets are found, the general procedure is as follows:

1. Create a face and an outer loop
2. Read the first vertex number and keep a reference to it
3. Read a vertex number and get the corresponding vertex from the list of all vertices

4. Check for an edge connecting the previous vertex with this vertex

5. If the vertices are connected, add a reference to the current loop as the right loop of the edge

6. If the vertices are not connected, create a new edge with the previous vertex as the start vertex, the current vertex as the end vertex, and the current loop as the left loop.

7. If this is the last vertex in the list, add an edge, if necessary, from the current vertex to the first vertex.

For facet 0, face 0 and loop 0 are created. 0 is read as the first vertex, and this is saved. 2 is read, and the vertex is found. There is no edge connecting vertex 0 to vertex 2, so edge 0 is created from vertex 0 to vertex 2 with loop 0 as its left loop. 3 is read, and because there is no edge from vertex 2 to vertex 3, edge 1 is created with vertex 2 as start vertex, vertex 3 as end vertex and loop 0 as its left loop. Similarly for the next pair, vertex 3 and vertex 1, edge 2 is created from vertex 3 to vertex 1 with loop 0 as its left loop. As vertex 1 is the last vertex, a new edge, edge 3, is created from vertex 1 to vertex 0 with loop 0 as its left loop. The edge order for face 0 is 0, 1, 2, and 3.

Similarly for the next facet, which is read as face 1 and loop 1, edges 4 (from vertex 4 to vertex 5), 5 (from vertex 5 to vertex 6), 6 (from vertex 6 to vertex 7), and 7 (from vertex 7 to vertex 4) are created with loop 1 as their left loop. The edge order for face 1 is 4, 5, 6, and 7.

For the next facet, face 2 and loop 2 are created. The first vertex is read, vertex 0, and remembered. With the next vertex, vertex 1, there is an edge, edge 3, already connecting the vertices. Therefore, loop 2 is added as the right loop of edge 3. There is no edge connecting vertex 1 to vertex 5, so a new edge (edge 8) is added. There is an edge connecting vertex 5 to vertex 4, so a reference to loop 2 is added as the right loop of this edge. There is no edge connecting vertex 4 to vertex 0, so edge 9 is created. The edge order for face 2 is 3, 8, 4, and 9.

The same is done for the other faces. Face 3 has edge order: 2, 10, 5, and 8. Face 4 has edge order: 1, 11, 6, and 10. Face 5 has edge order: 0, 9, 7, and 11.

Each vertex number pair (a, b) appears twice, once as (a, b) and once as (b, a).

Note that this is not a proper data exchange method because only part of the data is communicated.

Note, as well, that VRML can also be used to write out disjoint facets. In this case the same technique is used as for STL, that is, that each facet is created as a separate face and then these faces are joined using the octree method described.

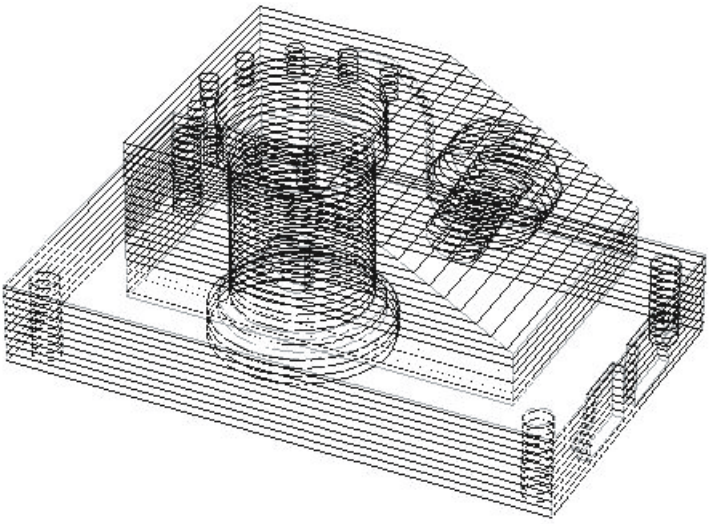


Figure 14.20: ANC101 object with slices

14.5 Rapid prototyping

Rapid prototyping techniques such as stereolithography (SLA) or selective laser sintering (SLS) build an object as a series of layers. A set of slices are determined from the shape to be made, as illustrated in figure 14.20.

The object is then made by solidifying material in layers to create the object (figure 14.21).

The company 3D Systems Inc. is generally credited with the development of the first rapid prototyping (RP) machine. The machine, for stereolithography, followed the methods of some of the first separate graphics machines in taking triangles as input. The procedure is to approximate a solid by triangles and then pass these through to a slice generator and finally to the controller, as illustrated in figure 14.22.

The contour generator slices the triangles in the STL file separately and then assembles them into contours. The contours are then used to control the movements of a laser beam. For SLA, the laser light solidifies a plastic liquid. For SLS, the laser heats powders locally, causing them to melt and merge with the layers underneath.

The process is sometimes called “freeform manufacturing” because it is generally easier to create objects with strange geometry than with conventional processes. However, there are limitations and it is a process that is continuing to be researched and developed. This section, though, deals with

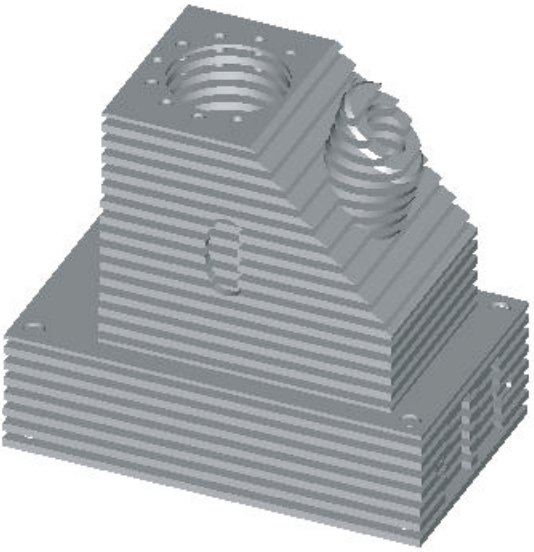


Figure 14.21: ANC101 object with separated slices

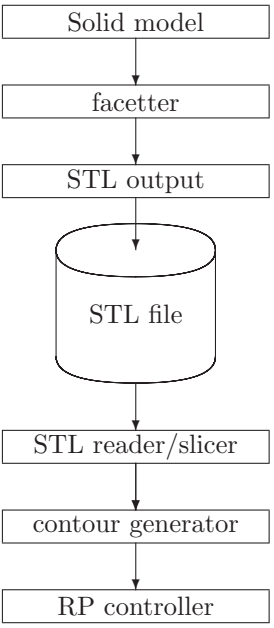


Figure 14.22: Rapid prototyping data flow (from [126])

some of the data exchange and geometric aspects.

First, the traditional approach. Each triangle can be sliced separately. To determine what kind of intersection results, the vertices of the triangle can be classified as −1 (below the plane), 0 (on the plane), and 1 (above the plane). The classifications 'below' or 'above' are made from the direction of the vertex from the plane compared with the plane normal. For simplicity these are shown ordered:

−1	−1	−1	Triangle below the plane, no intersection
−1	−1	0	One vertex on the plane, no intersection
−1	−1	1	Two vertices below the plane, one above, intersection
−1	0	0	One vertex below the plane, two on the plane, intersection
−1	0	1	One below, one on and one above the plane, intersection
−1	1	1	One vertex below, two above the plane, intersection
0	0	0	All vertices on the plane, ignore the triangle
0	0	1	Two vertices on, one above the plane, intersection line
0	1	1	One vertex on, two above the plane, no intersection
1	1	1	Triangle above the plane, no intersection

Only line intersections are considered here, and vertex intersections will not contribute to the contour. Where all vertices are on the plane, the whole triangle can be considered as within the contour. The individual line segments are created and then linked to create the contour.

The triangles used to communicate shape are useful for graphics because they are a standard form that can be handled simply. Although the same simplicity was useful for rapid prototyping, also, problems can arise for rapid prototyping. For graphics, triangle mis-orientation or incomplete models do not cause problems. However, such errors result in incomplete contours and hence can disrupt the process, possibly wasting many hours of machine time.

Although, when the process was first developed, computing power was limited, there has since been much development so that the process can be improved. The use of STEP as an alternative to STL has already been proposed by Carleberg [17] and Kumar and Dutta [73] have examined several exchange formats. Some of these formats are also two-dimensional rather than three-dimensional, so the slice generation can be done offline. Stroud and Xirouchakis [126] proposed extensions to STL to add surface data that, although far from ideal, would allow reconstruction of a solid from the STL file. This section follows the contents of [126]. A new proposed RP data exchange standard, including both 3D and 2D data exchange possibilities, is being worked on in the STEP-NC (ISO 14649) effort.

Assuming that exact models are available at the rapid prototyping machine level, the quality of fabrication can be improved in several ways. One of the simplest of these is to control shape generation in specific ways to avoid 'creep' due to random surface offsetting in the facetting process.

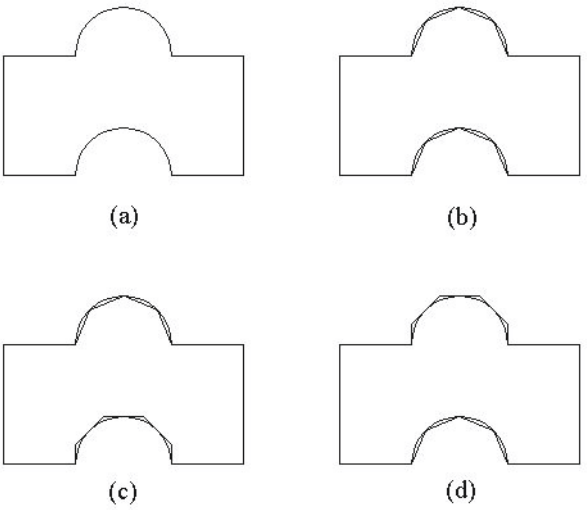


Figure 14.23: Contour approximations (from [126])

14.5.1 Slice approximation

If exact models are available at the RP level, it is possible to define some straightforward procedures for improving the quality of control data. Normally when the STL file is sliced, the resulting contour is an approximation to the real contour based on the approximation to the three-dimensional object in the facetting process, which produces the STL file. This approximation process is, in general, badly defined because it will typically lie inside convex surfaces and outside concave surfaces. For manufacturing purposes, the approximation should either completely enclose the object or be completely enclosed by it. This is difficult to achieve in three dimensions, but it is much easier to do in two dimensions. Figure 14.23 shows the approximations, the normal (mixed exterior and interior approximation) in figure 14.23a; the interior approximation is shown in figure 14.23b, and the exterior approximation is shown in figure 14.23c. (A related effect is described by Wozny [149] in three dimensions; this will be dealt with in the next section on layer approximation).

For the revised approximation method, first curves need to be subdivided if they have points of zero curvature between convex and concave sub-segments; then the contour should be approximated as usual (described in chapter 10). Exterior approximation convex edges or edge segments and interior approximation of concave edges or edge segments are the two cases that need to be handled; otherwise the points subdividing the edges are joined with straight lines. See figure 14.24.

In both cases, the approximation method is the same and involves finding an intermediate point at the intersection of the tangent directions at neigh-

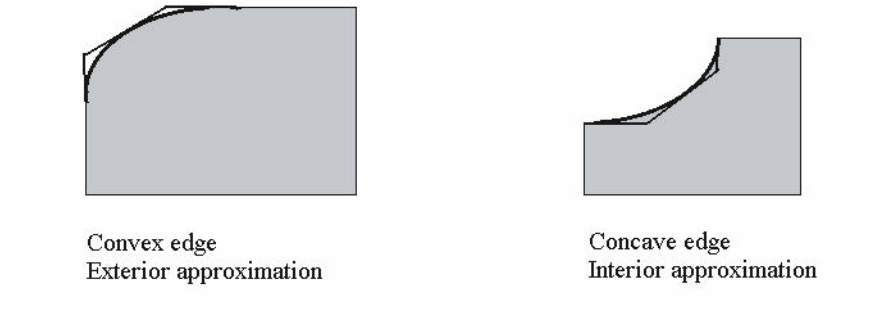


Figure 14.24: Convex exterior and concave interior approximation (from [126])

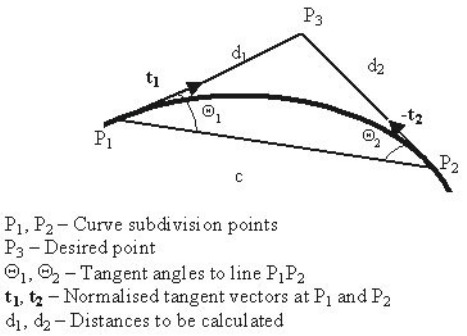


Figure 14.25: Approximating an arc (from [126])

bouring approximation points. This involves simple vector arithmetic. P_1 and P_2 are neighbouring approximation points, and P_3 is the desired point. The tangents to the curve at P_1 and P_2 in the direction of P_3 are t_1 and $-t_2$ respectively. The distances from P_1 and P_2 to P_3 are d_1 and d_2 respectively. The angles between the tangents at P_1 and P_2 and the line joining P_1 to P_2 are θ_1 and θ_2, respectively and the line has length c. See figure 14.25.

The following two relations are directly discernible: $d_1 cos\theta_1 + d_2 cos\theta_2 = c$ $d_1 sin\theta_1 = d_2 sin\theta_2$.

Therefore d_1 can be calculated as

$d_2 = d_1(sin\theta_1/sin\theta_2)$ and so
$d_1 cos\theta_1 + d_1 cos\theta_2(sin\theta_1/sin\theta_2) = c$, or

$d_1(cos\theta_1 sin\theta_2 + cos\theta_2 sin\theta_1)/sin\theta_2 = c$, or
$d_1 = csin\theta_2/(cos\theta_1 sin\theta_2 + cos\theta_2 sin\theta_1)$

where $cos\theta_1, sin\theta_1, cos\theta_2$, and $sin\theta_2$ can be computed from the dot and cross products of the tangent vectors t_1 and $-t_2$ with the normalised directions $(P_2 - P_1)$ and $(P_1 - P_2)$. Once d_1 has been calculated, the position of P_3 follows directly as

$P_3 = P_1 + d_1 t_1$

The desired approximation to the curve segment joins P_1 to P_3 and P_3 to P_2 instead of P_1 to P_2.

These results are not startling, just fairly obvious consequences of having exact geometry when slicing instead of being given an approximation.

14.5.2 Layer approximation

The next point to make about approximation for fabrication concerns the layers. Wozny [149] pointed out the problems caused by a naive use of contour information for building layers. To produce a consistent fabrication of an object, it is necessary to look at two slices when producing a layer, the current height slice, and the next slice. The contour to be filled is produced as a Boolean combination of the two slices.

Consider figure 14.26. The overlapping contours are shown in figure 14.23a. If the object to be produced is to be used to make a mould, then the object should be smaller than the real object so the contours should be combined using a Boolean intersection to produce the fabrication contour (figure 14.23b). If the positive object is to be made, then the fabrication should make a slightly larger object and so the layers should be combined with a Boolean addition operation to produce the fabrication contour (figure 14.23c). The resulting slice contour is then processed as described above to produce the approximation for fabrication.

14.6 Reverse engineering

Reverse engineering is another complex topic that is not covered fully here. There are whole books about reverse engineering, for example, that by Marshall and Martin [84], so the reader is, again, advised to look at one of these for more details. However, there are one or two parts of this that are solid modelling topics and these are described briefly here.

Reverse engineering is the name given to the process of constructing the computer model from an existing object. There are several reasons for doing this, namely:

1. Constructing an object from calculated points

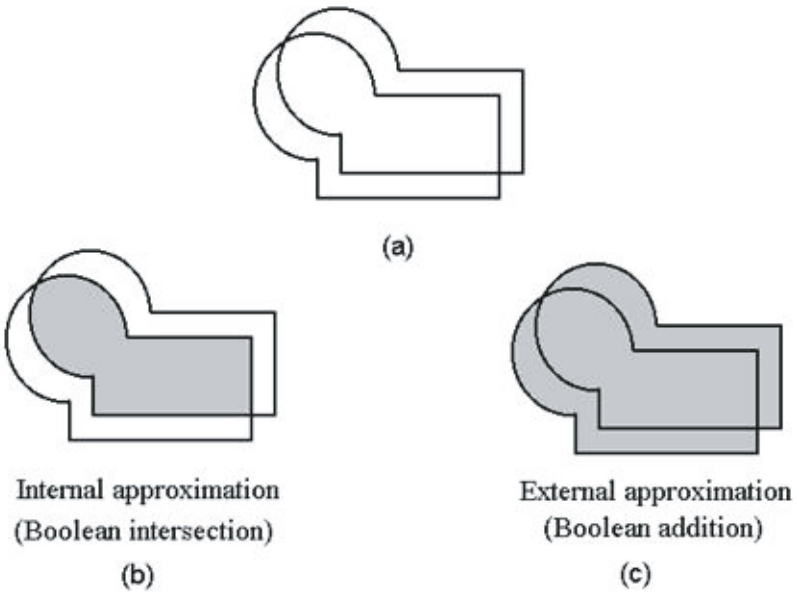


Figure 14.26: Slice combination for layer generation (from [126])

2. Recreating an object for which drawings do not exist
3. Creating a model from measured biological data
4. Measurement of museum objects
5. Model verification

Constructing an object from calculated points is an exercise that might occur when points can be calculated, but surface calculation is difficult, e.g., for a turbine blades or propeller surfaces.

Recreating an object for which drawings do not exist may be necessary if parts are needed to maintain machinery when the original supplier no longer exists, for example. Note that another use for this is to copy the products of another company, but this is ethically questionable; hence, it should not be done. The copying of products where a company no longer exists, though, is a different matter. The U.S. Department of Defense was reported as needing this because companies were sometimes created to produce only one particular weapon and subsequently disappeared. When spare parts were needed, the Department of Defense had no supplier and no CAD model from which to produce the required parts.

The third use of reverse engineering that is in common use is in the bio-engineering field. Prostheses can be made using measured data. For example, to make an artificial right leg, the left leg can be measured; then a reflection transformation is applied to the measured points and the reflected data used to create the prosthesis. This use, though, is different from the other two because there are different needs. Such prostheses are not items for mass production, and hence, there is not the same need for production efficiency. In some cases, it may be possible to use the measured data directly for machining (so-called direct machining) instead of generating toolpaths directly from a complete model or a surface model. Direct toolpath generation, or direct machining, can be done in the same way as for RP, by slicing measured data in the form of triangles and using the slice contour as a toolpath. Another difference is in the types of surface generated from the points. In general it is to be expected that the surfaces needed in bio-engineering will be free-form surfaces, not the simpler surface types common in mechanical parts. Also, it may be that there are fewer surfaces and perhaps that reconstruction can be done using more human interaction.

Measurement of museum objects is a specialised use of reverse engineering. These can be too precious or delicate to allow widespread access, so reverse engineering is used to create an electronic copy that can be distributed for research. In one example, measured data from Neanderthal skulls were used to provide the main shape, copying and transformations were then used to provide data to fill in gaps, and rapid prototyping techniques were finally used to produce a plastic copy for research. This, again, is a very specialised use that does not require the full range of manufacturing techniques but provides an interesting alternative illustration of the techniques.

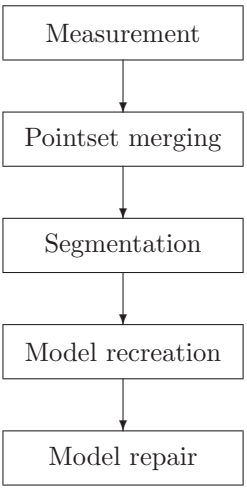


Figure 14.27: Simplified work flow for reverse engineering

Model verification is yet another application which uses the same types of technique in a different application area. This is a sort of automated quality control where the geometry of the physical model is compared with that of the CAD file. This sort of application can be useful to verify, for example, that the shape produced is acceptable. Even if the tool-paths used to produce the part are perfect, tool wear can mean that the deviations in the surfaces produced are incorrect.

These general observations are only meant as illustrations for applications of the techniques described later. The specialised applications described above use only parts of the general process of object reconstruction so that will be described here rather than the specialised applications.

A simplified diagrammatic view of reverse engineering activities is shown in figure 14.27.

14.6.1 Measurement

Point calculation methods will not be dealt with here.

There are two common measurement methods as follows:

1. Coordinate measuring machines
2. Optical scanners

These two methods are separated because the coordinate measuring machines use physical contact, whereas the optical scanners do not touch the surface. Optical scanning methods are sensitive to surface types. If the surface is reflective, then the sensing of the points can cause problems. Access

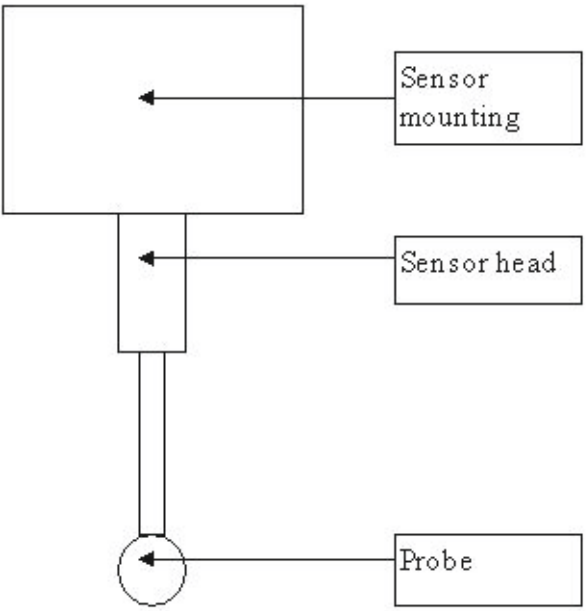


Figure 14.28: Schematic view of CMM head

possibilities and distances also have different consequences for different methods. Optical methods can be used to measure delicate objects, whereas CMM needs to apply force. Optical methods are quicker, but CMM is more accurate.

Coordinate measuring machines (CMM)

The diagrammatical form of a coordinate measuring machine is shown in figure 14.28.

One arrangement is that the sensor mounting is placed on a five-axis machine bed to give the required precision. The same requirements for machining accuracy also apply for measurement. However, instead of having the coordinates input, the position of the head is output. The X and Y coordinates are given from the position of the head above the bed. The Z coordinate is found by lowering the sensor head until contact is made. When contact is made with the probe tip, a point is determined. An option is additionally to touch the object at three points close to the first point. This information allows the measurement software to calculate the angle of the surface normal at the contact point. Surface normal vectors are important for surface reconstruction.

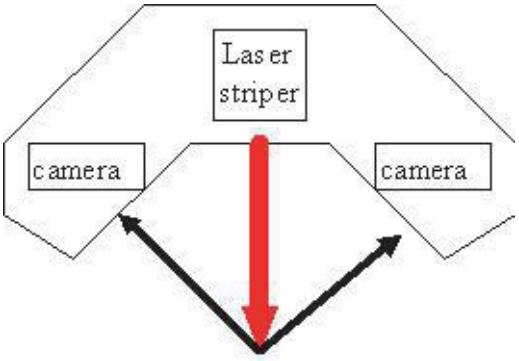


Figure 14.29: Schematic view of laser scanner head

Laser scanning

The general layout of a laser scanner head (based on the machine supplied by 3D Scanners Ltd.) is shown in figure 14.29.

The laser striper shines a beam of laser light down onto the base surface, and the position of points along it are sensed. The head is mounted on a three-axis machine that gives the X and Y coordinates; the Z coordinate is determined by triangulation. The cameras, though, have an optimal range, and hence, it is necessary to raise and lower the head to keep the distance between the head and the target object within this distance. One camera is the main camera, because only one is necessary for triangulation, but the other camera can be used when the main camera has difficulties because of reflection or because the point is obscured from one camera, say,

The head can be angled or rotated to give an alternative view for rescanning without moving the object. This is important because normally it is not possible to measure the whole of the object at one time, hence, the greater the measuring possibilities without moving the object the better.

Another possibility offered with this type of visual scan is to rotate an object while measuring it which gives a more complete measurement of the object at one step. Also, because the rotation is controlled by the measuring machine, the relative angles of the measured points are known and hence can be converted to a common coordinate system.

Photogrammetry

In one pattern measuring method, fixed patterns of lines are projected onto the object to be measured and then photographed using two cameras at known positions. The relative displacements of the lines in the images from the two cameras can be used to fix points. It is similar to the laser scanning method, but a greater area can be analysed at one time, saving time.

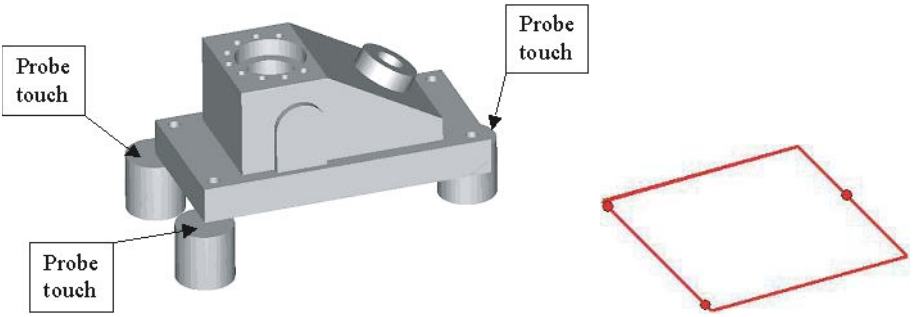


Figure 14.30: Finding a base plane

One system, developed at the University of Braunschweig and now commercialised, uses reference points attached to the object being measured to be able to relate measurements made from different positions. This is important for being able to relate multiple point sets (see later) to produce a complete measurement of an object.

Setting up the measurement

The different measuring methods have different characteristics and different demands to start measuring.

For the CMM machine, there is the question of attaching the correct probe, one that can access the parts of the object to be measured. This might require a longer probe or a thinner probe. Geometrically, it is necessary to set up a safety plane above the object where the head can be moved without colliding with the object. It is also necessary to select the coordinate system (origin and axes) in which the measured points will lie. For the machine in the laboratory, this is done by selecting the base plane, Z = 0, with three touches of the probe. See figure 14.30.

Then the X-axis, say, is determined with two touches of the probe. The two touches are projected onto the $Z = 0$ plane to give a line, the X-axis. See figure 14.31.

Finally, one touch of the probe defines the Y axis. The point is projected onto the plane, the perpendicular to the X axis defines the origin. See figure 14.32.

When the object is reoriented, a similar procedure is needed to identify the origin in the same manner, which means that the two sets of measurements have the same global coordinate systems to which the points are transformed so that they can be unified. See figures 14.33, 14.34, and 14.35.

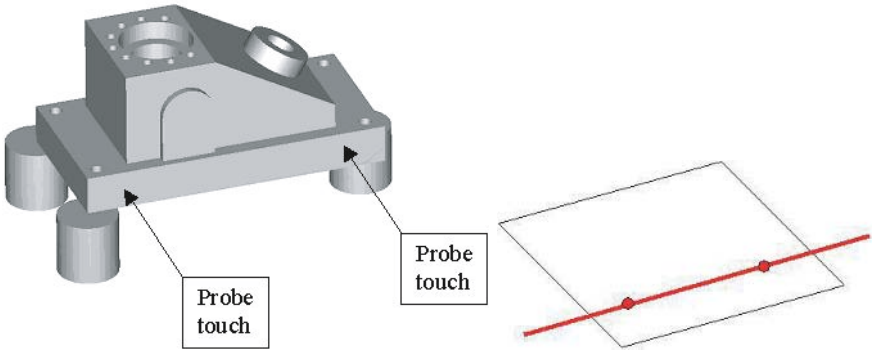


Figure 14.31: Finding the X-axis

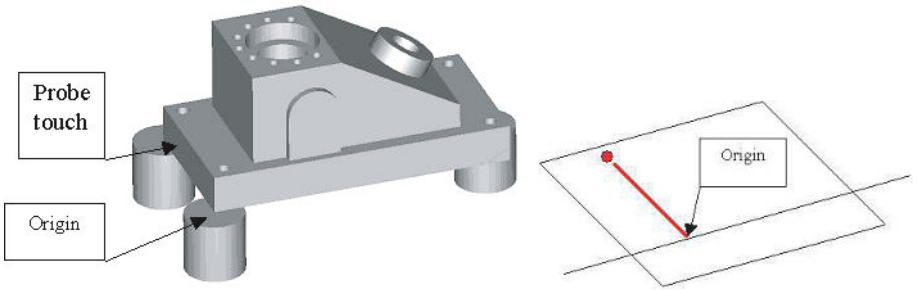


Figure 14.32: Finding the Y-axis

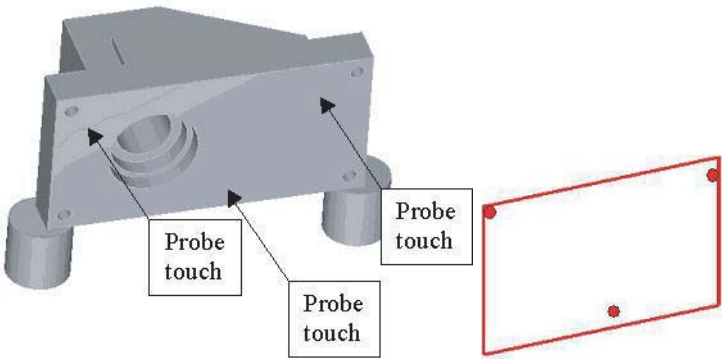


Figure 14.33: Finding a base plane

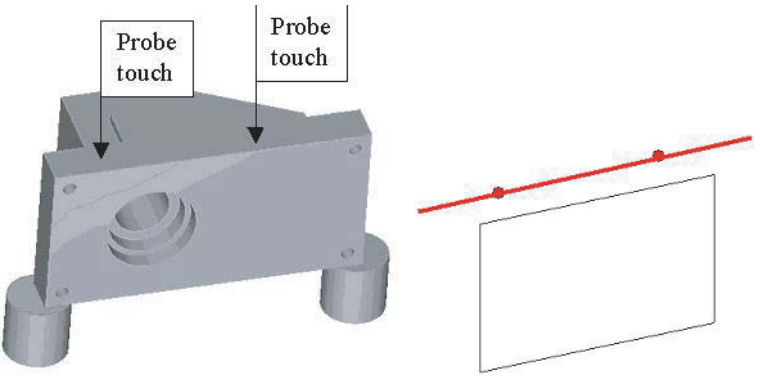


Figure 14.34: Finding the X-axis

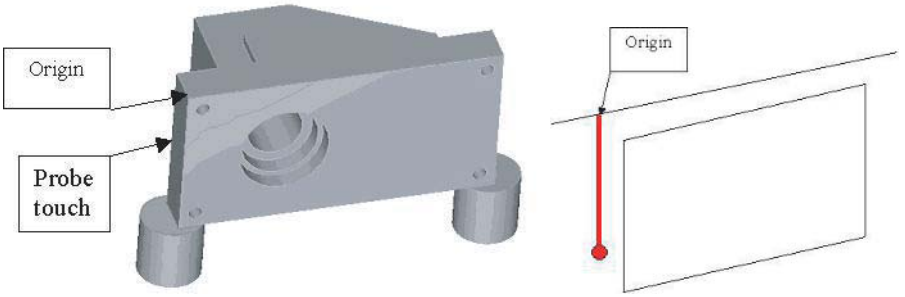


Figure 14.35: Finding the Y-axis

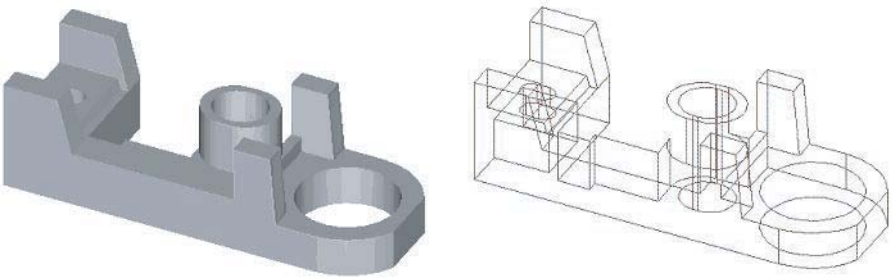


Figure 14.36: MBB object

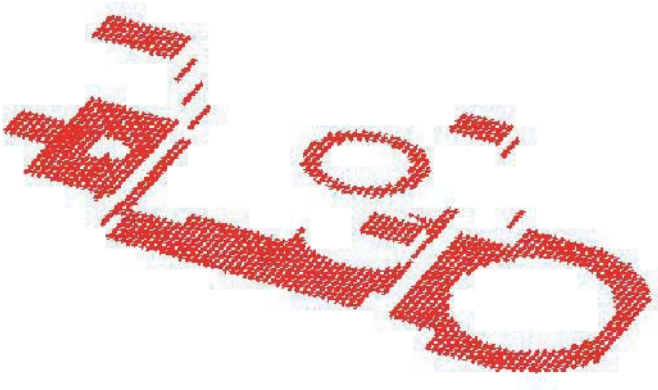


Figure 14.37: MBB object point set from above

14.6.2 Pointset merging

It should be fairly obvious that it is impossible to measure the whole object in one step. It is not (yet?) possible to have an object float in space and measure all around it. Even if it were possible, parts of the object are often 'shadowed', maybe requiring special measuring angles.

Consider the object in figure 14.36.

Figure 14.37 shows the point set is obtained by scanning the object from above.

If the object is scanned from the side, then the point set in figure 14.38 is obtained.

These two sets of points do not overlap because the scanning directions were orthogonal. Scanning from an oblique angle gives a third set of points shown in figure 14.39.

These point sets give several partial images that need to be combined.

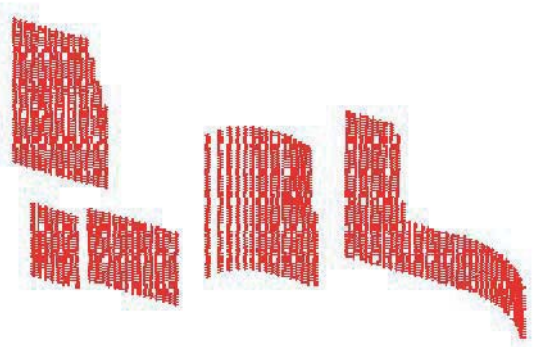


Figure 14.38: MBB object point set from side

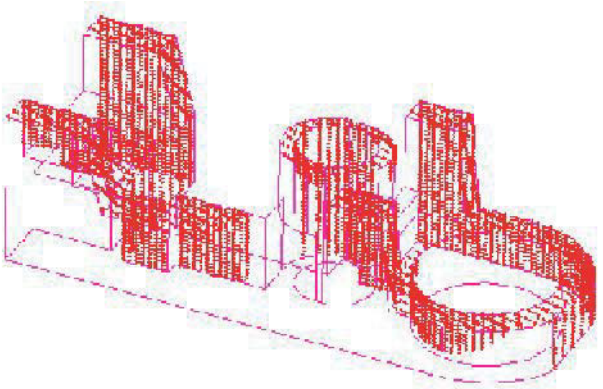


Figure 14.39: MBB object oblique point set

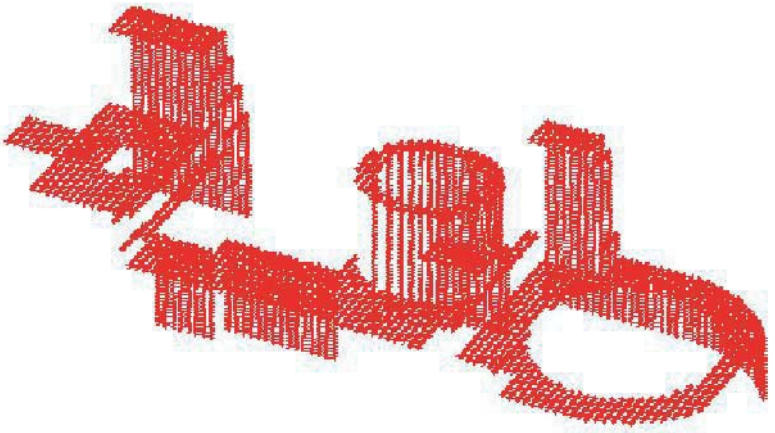


Figure 14.40: MBB object first combined point set

Combining the first two gives the point set in figure 14.40.

Adding the third set gives the set shown in figure 14.41.

This process fills in some of the gaps, but combining point sets can create other problems.

The point sets usually have a structure. For a laser scanner, the points are usually arranged in a grid pattern, although this can have holes. When several point sets are intermingled, this structure becomes disordered and it is not always easier to establish neighbours and hence structures for surface fitting. For a small number of points, it may be possible to resolve this problem by creating one structure and superimposing the other structures. An alternative is to filter the points by imposing a regular structure, a planar grid or cylindrical grid, say (figure 14.42). This method was developed by Martin et al. [85].

One point in each cell is chosen as a representative of that cell, thus performing data reduction as well as data organisation. See figure 14.43.

The choice of point within the cell can be done in several ways. One method is to choose the point closest to the centre. Another way is to choose the point that is at the modal depth. This works best, though, for data sets with almost a common measurement direction; hence, care should be taken in the choice of the grid plane.

It has been demonstrated experimentally that such filtering techniques can remove 70% to 80% of the data points without affecting the surface fitting process too adversely.

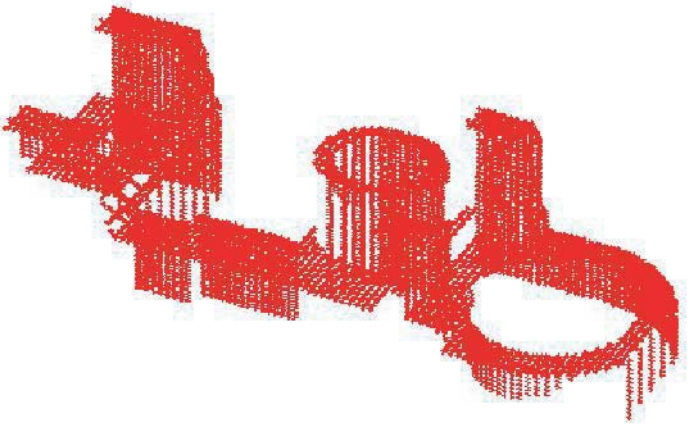


Figure 14.41: MBB object second combined point set

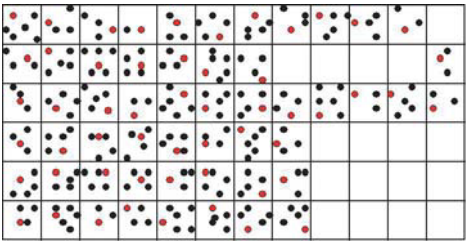


Figure 14.42: Original point set

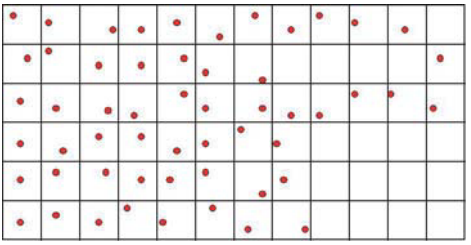


Figure 14.43: Filtered point set

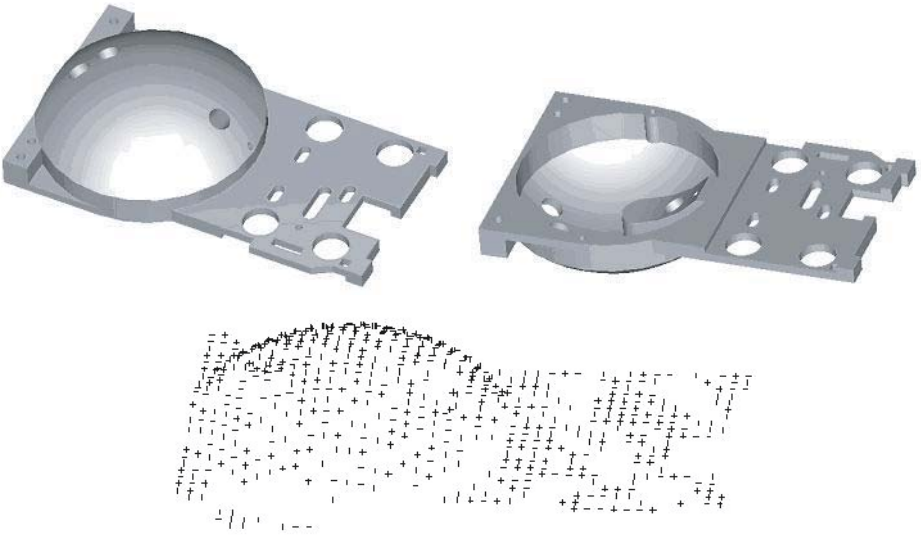


Figure 14.44: Printer head

14.6.3 Segmentation of surfaces and surface fitting

It is necessary to segment the points up into groups corresponding to faces in the measured object. This is not easy. The problem is twofold: Knowing which points belong together; and knowing to which surfaces they belong. These are called: 'Segmentation'; and 'Surface fitting' respectively, here.

Segmentation

One method of segmenting is called region growing. In this method seed points are established and neighbouring points are accreted to build up regions with specific characteristics. Another method involves trying to find edges in the data by estimating curvature. Both of these methods are similar in that they are trying to use the points to estimate surface properties and use these surface properties to divide up the point set.

Look at figure 14.44.

The object is shown at the top of the figure and a point set below. Although it is obvious to you the reader that there is a separation between a set of planar points and the points in the spherical surface, it is necessary to find the boundaries and fit a surface through these.

Unfortunately real-world objects are good at confusing such boundary finding schemes. Blends in objects introduce a level of smoothness making it more difficult to identify boundaries. Even without blends there can be critical cases, such as that shown in figure 14.45.

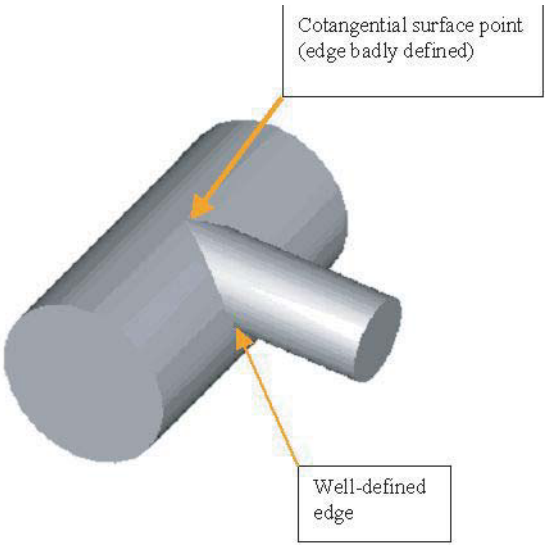


Figure 14.45: Difficult segmentation case

The curvature properties have to be estimated based on neighbouring points, which is a method developed by Besl [8]. Consider figure 14.46.

This figure shows a point, p_4, and its eight immediate neighbours. A numerical surface, a Bézier surface, for example, can be created that interpolates these points and the principle curvatures can be calculated from this interpolating surface.

The calculations for the control points, based on the Bézier surface patch equation:

$$\sum_{i=0}^{2}\sum_{j=0}^{2}\frac{2!}{i!(2-i)!}\frac{2!}{j!(2-j)!}(1-u)^{2-i}u^{i}(1-v)^{2-j}v^{j}b_{ij}$$

give:

$$\begin{aligned}
b_{00} &= p_0\\
b_{10} &= 2*p_1 - 0.5*(p_0+p_1)\\
b_{20} &= p_2\\
b_{01} &= 2*p_3 - 0.5*(p_0+p_6)\\
b_{11} &= 4*p_4 - p_1 - p_3 - p_5 - p_7 + 0.25*(p_0+p_2+p_6+p_8)\\
b_{21} &= 2*p_5 - 0.5*(p_2+p_8)\\
b_{02} &= p_6\\
b_{12} &= 2*p_7 - 0.5*(p_6+p_8)\\
b_{22} &= p_8
\end{aligned}$$

Besl [8] describes how the signs of the principle curvatures of this patch

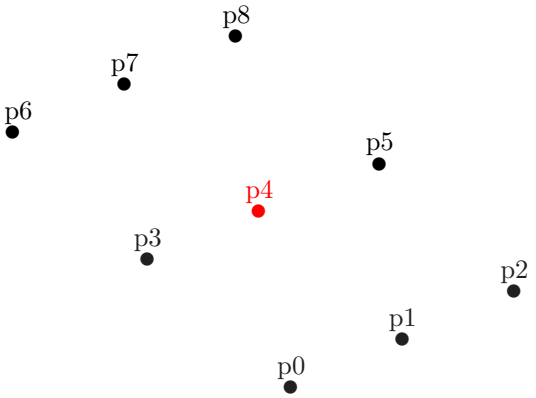


Figure 14.46: Local point group

evaluated at the point $u = v = 0.5$ can be used to classify the point p_4 in one of eight ways. When all points have been classified in this way, the points are then grouped locally according to their classifications and surfaces are fitted to the local point groups that have the same classification.

Surface fitting

Surface fitting is another of these complex topics that will not be dealt with in detail here. See Hoschek and Lasser [61], for example, for geometry fitting.

Surface fitting is complex for several reasons. One reason is because of the measurement errors and another reason is that the degree of the surface is unknown. One strategy is to use a filter method such as the one described earlier to partition the points into local sets and then fit surface patches to these. However, many common surface types, such as planes, cylinders, spheres, cones, and torii, would benefit from being recognised as such for later processing, so it would be better to check for these explicitly first. The characteristics of these surfaces, axes of cylinders, or plane normals, for examples, may be perturbed, though, by small errors. This increases the difficulty for object reconstruction later.

Now, as I have not worked with this topic, I have no special tricks or hints to impart. The subject has to be mentioned, but it would be better to look at the literature for more details.

14.6.4 Model recreation

There are two obvious ways to recreate the object, one using only the points and the surfaces from the segmentation step, and the other is based on recreating a triangulated object from STL.

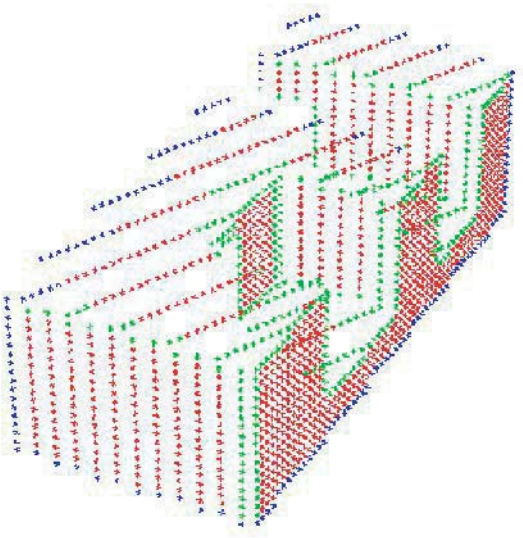


Figure 14.47: Segmented point set

For the first method, the surfaces are supplied during the segmentation step and it is necessary to recreate the topology from these. The topology creation uses the surfaces fitted in the segmentation step and the associated point sets to determine the topology of a model. Figure 14.47 shows three types of points.

The second method uses the possibility of having the measured point data in STL form. Recreating models from STL files is dealt with in section 14.4.

For the first method, the three types of points are:

1. Internal
2. Surface/surface separation
3. Surface/background separation

The internal points do not provide any extra information for the topology creation because they will be on the interior of the face. The surface/surface separation points are points between segmented regions, and the surface/background separation points are points bounding one segmented region. The separation points are needed to determine the curves bounding the surface segments.

The first step is to set approximate bounding curves for the surface regions based on the point extent. In the example shown in figure 14.48, the surface/surface separation points are associated with a curve calculated as the

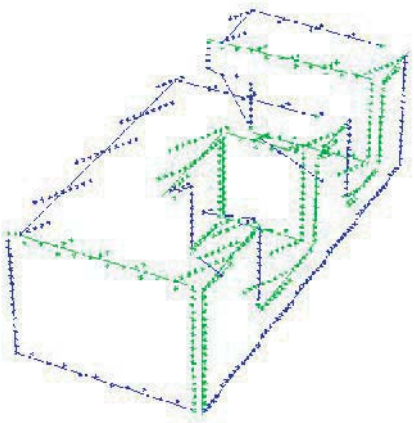


Figure 14.48: Edges from point sets

intersection of the two surfaces. The surface/background separation points are used to estimate an approximate boundary curve lying on the surface. The points are referred back to the curve to estimate the extent of the edge, and an isolated edge is created, associated with the face, or faces corresponding to the surfaces.

When this has been done, there are edges lying in curves that probably do not meet, nor is it likely that they match exactly, as shown in figure 14.49.

The final step is, therefore, to match the topological elements and to create a valid connected model, as shown in figure 14.50.

Models created in this way are partial models that need to be combined with models created from other views. The combination method is not trivial because it is possible, indeed likely, that the other partial models have been measured in different views and possibly by repositioning the part. A big problem remains, therefore, to find what matches what in the different models. This is a sort of feature recognition task, with the added twist that the models being recognised are incomplete. It is likely that a feature that is complete in one view is only partially seen in another view. It is, therefore, difficult to compare in geometric terms because only part of the geometry will be defined in both views. The same is true for the topology. Another possibility is to apply artificial intelligence techniques to try and pick out visual features from an image of the object in the chosen view.

The second method for object recreation uses the possibility of having the measured point data in STL form. Recreating models from STL files is dealt with in section 14.4. Unlike the method for joining simple triangular models read from STL files, it is possible to perform a post-processing step to simplify the objects.

It is necessary to segment and fit surfaces in the same way as described

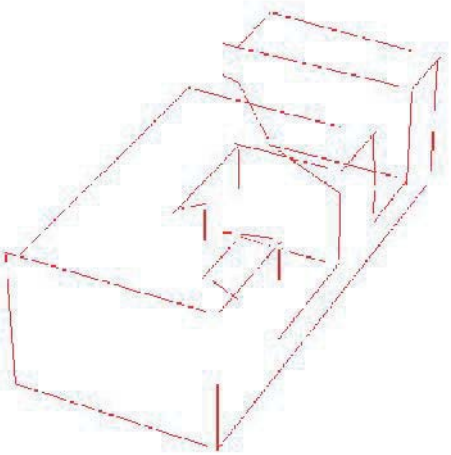


Figure 14.49: Merged topology

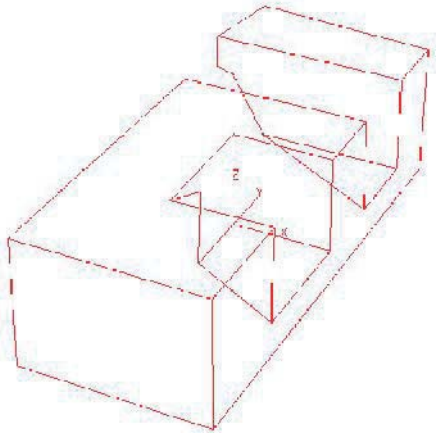


Figure 14.50: Corrected topology

earlier. Once this has been done, the surfaces are associated with the facets containing the points used to create them. The next step is like the 'makenice' operation described in section 14.2.5. This removes all internal edges between facets lying in the same surface. Once this has been done, the remaining edges lie between two different surfaces. It is necessary to intersect the two surfaces to produce the correct curve, which is then associated with the edge. When all edges have been processed, a second application of the makenice operation removes all superfluous vertices (vertices lying between edges in the same curve) to create the partial shape.

14.7 Volume calculation

The 'classical' volume calculation method uses triangulated models as approximations for volume calculation.

In pseudo-code form the algorithm looks like:

```
volsum = 0; facet body(b, tolerance);
for all facets in body(b, volsum = volsum + facet volume);
```

This description is a little simplified. The facet volume calculation finds the volume between the facet and a base plane. There is, however, a sign associated with this facet that depends on the direction of the facet normal with respect to the plane.

Consider the rectangular block shown figure 14.51.

Imagine that the facets are as shown in figure 14.52. This is arbitrary, but it does not matter for the method.

The facets can be written down in a list:

0	$(-68.3, -50, -18.3)(18.3, 50, -68.3)(18.3, -50, -68.3)(-0.5, 0, -0.866)$
1	$(-68.3, -50, -18.3)(-68.3, 50, -18.3)(18.3, 50, -68.3)(-0.5, 0, -0.866)$
2	$(-18.3, -50, 68.3)(68.3, -50, 18.3)(68.3, 50, 18.3)(0.5, 0, 0.866)$
3	$(-18.3, -50, 68.3)(68.3, 50, 18.3)(-18.3, 50, 68.3)(0.5, 0, 0.866)$
4	$(-68.3, -50, 18.3)(18.3, -50, -68.3)(68.3, -50, 18.3)(0, -1, 0)$
5	$(-68.3, -50, 18.3)(68.3, -50, 18.3)(-18.3, -50, 68.3)(0, -1, 0)$
6	$(68.3, -50, 18.3)(18.3, -50, -68.3)(18.3, 50, -68.3)(0.866, 0, -0.5)$
7	$(68.3, -50, 18.3)(18.3, 50, -68.3)(68.3, 50, 18.3)(0.866, 0, -0.5)$
8	$(68.3, 50, 18.3)(18.3, 50, -68.3)(-68.3, 50, -18.3)(0, 1, 0)$
9	$(68.3, 50, 18.3)(-68.3, 50, -18.3)(-18.3, 50, 68.3)(0, 1, 0)$
10	$(-18.3, 50, 68.3)(-68.3, -50, 18.3)(-18.3, -50, 68.3)(-0.866, 0, 0.5)$
11	$(-18.3, 50, 68.3)(-68.3, 50, -18.3)(-68.3, -50, 18.3)(-0.866, 0, 0.5)$

Next, the volumetric contribution of each facet is calculated and summed. To do this it is necessary to have a base plane for reference. It is convenient to have this below all facets in the object, so it is possible to choose the plane z=-68.3.

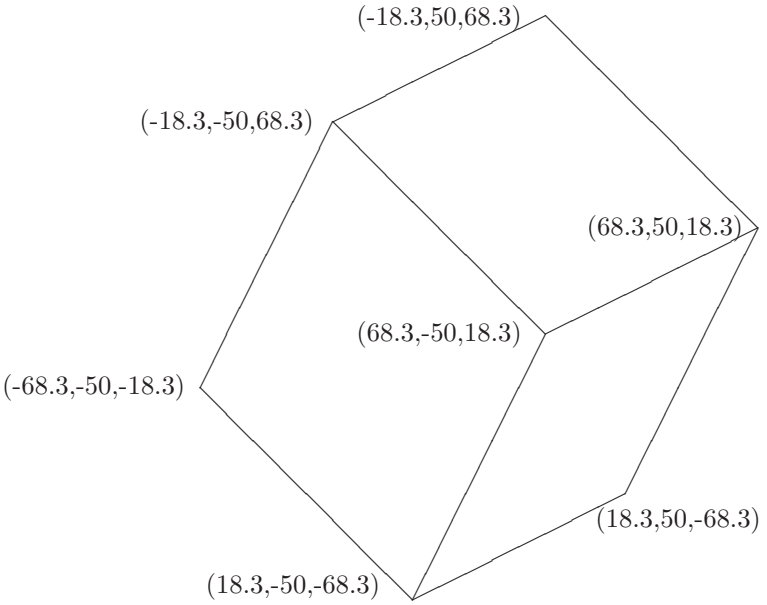


Figure 14.51: Tilted cube

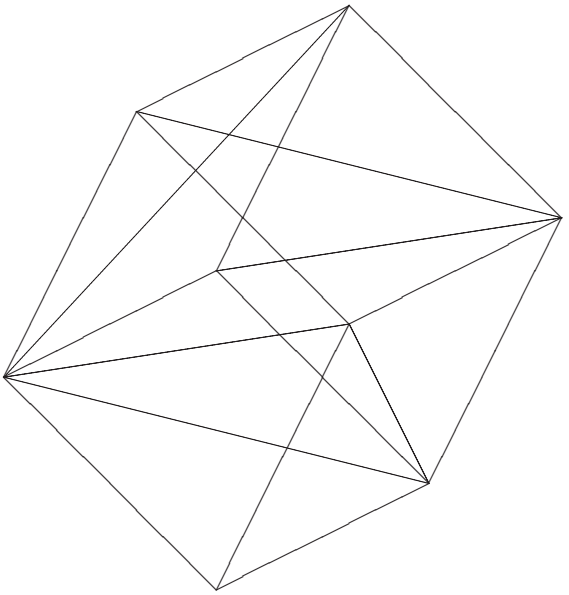


Figure 14.52: Tilted cube

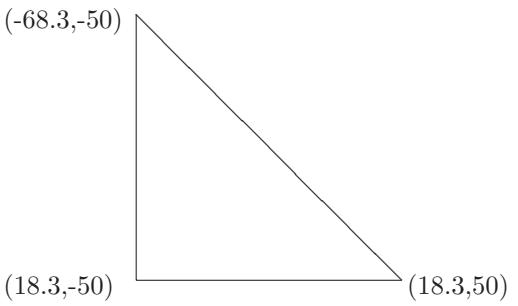


Figure 14.53: Projected facet

For the calculation, the volume under the facet is calculated as:

$$base \times height \times f_n \cdot bp_n \times zsum/6$$

where *base* and *height* are the base and height of the triangular projection of the facet onto the base plane. f_n is the facet normal, and bp_n is the base plane normal, in this case, $(0, 0, 1)$. zsum is the sum of the z-values of the heights of the corners above the base plane.

For the first facet, this works as follows. The corner points are:

$(-68.3, -50, -18.3)$, $(18.3, 50, -68.3)$, and $(18.3, -50, -68.3)$

with the normal as $(-0.5, 0, -0.866)$.

The base length is 100, the height of the triangle is 100, the cosine = 0.866 and the sum of the z coordinates above the plane $z = -68.3$ is 50. This gives the volumetric contribution as:

$$100 \times 100 \times 0.866 \times 50/6 = 72168.7836487$$

However, the normal points downwards, so this facet gives a negative contribution, -72168.7836487.

Summarising the contributions:

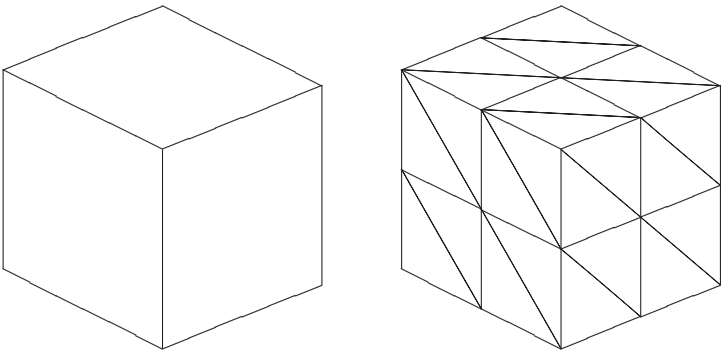


Figure 14.54: Cube and regular facetting

Facet	*volume*	*sum*
0	−72168.8	−72168.8
1	−144337.6	−216506.4
2	447155.7	230649.3
3	519322.3	749971.6
4	0	749971.6
5	0	749971.6
6	−72168.8	677802.8
7	−144337.6	533465.2
8	0	533443.2
9	0	533443.2
10	197168.8	730612.0
11	269337.6	999949.6

Facets 4, 5, 8, and 9 have zero contribution because the normals of these facets are perpendicular to the z direction.

Note, though, that this method is an approximative method and curved objects have to be broken into facets. This means that there is a tolerance to determine how close is the approximation.

14.8 Tetrahedral element decomposition

The initial step for this topic is related to the graphical facetting, but there are stricter criteria that need to be applied to the shape of the triangular facets. In general, the triangles should be more regular in shape.

A cube with regular facetting is shown in figure 14.54.

The process of removing tetrahedra starts with a corner vertex, a vertex where at least three planes meet. The removal of the first node from the cube is shown on the left of figure 14.55. The right of the figure shows the continuation of the process with eight corner nodes removed.

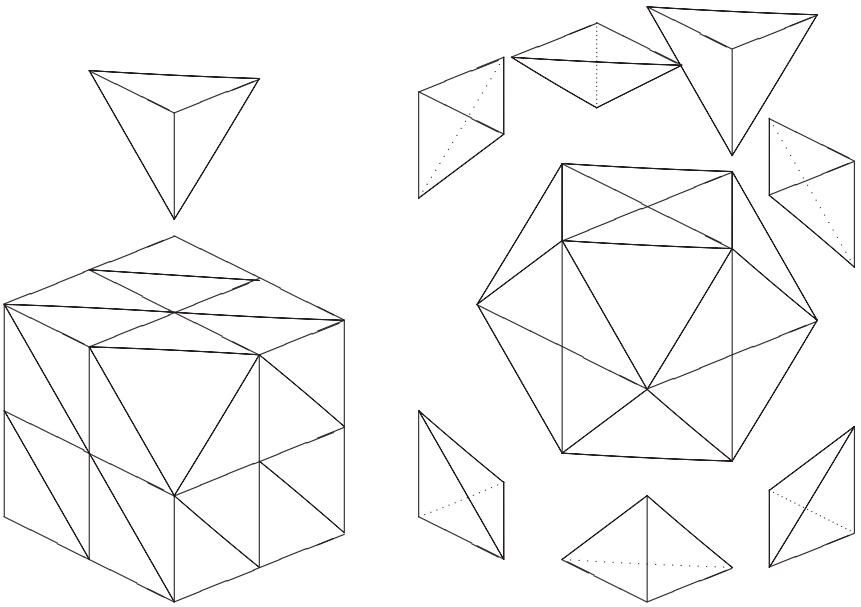


Figure 14.55: Facetted cube with extracted tetrahedra

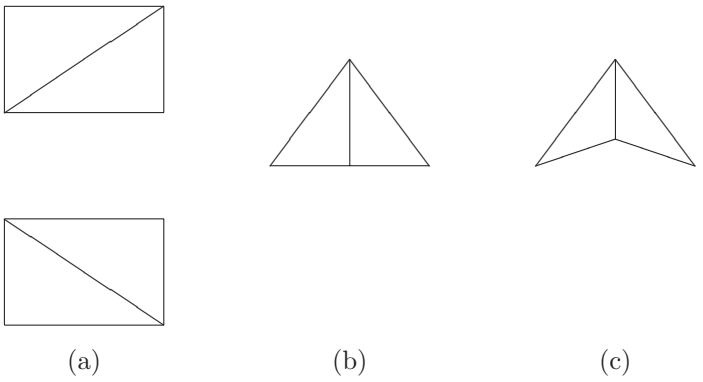


Figure 14.56: Diagonal swapping

It is obvious from comparison of the original facetting, shown in figure 14.54, with the result of removing the first tetrahedron, shown on the left of figure 14.55, that I have cheated again. One of the diagonals, on the left-hand facetted face, has been changed. The rule is that if there is a smooth edge at the vertex (i.e., an edge between two coplanar facets) then the edge can be switched, as shown in figure 14.56. If the angles at the vertices in the four-sided shape created by removing the edge are all less than 180 degrees then the diagonal can be swapped. See figure 14.56a. If, however, one angle is 180 degrees, then swapping the diagonal would create a degenerate facet, figure 14.56b, so this is not allowed. In figure 14.56c, swapping the diagonal would create a self-intersection, so this, too, is excluded.

Put simply, the decomposition method involves shrinking the object by removing connected facet pairs and repairing the holes left. When removing connected face pairs, three cases can be identified, as shown in figure 14.57.

In the left-hand column the vertex pair v0 and v2 has no common face; the four side edges e0, e1, e2, and e3 are sliced along with the four vertices v0, v1, v2, and v3, which creates a four-sided 'island', or inner contour, in a four-sided face. The inner contour is lifted out and made into the perimeter of a new face. This face and the other four-sided face are split with edges between v0 and v2 and their split image vertices. The tetrahedron is then separate.

In the middle column, there is one common face for edges e0 and e1, so this face becomes part of the new tetrahedron. Edges e2 and e3 are sliced, as before, as is vertex v2, creating a four-sided face. A new edge is created in this new face between vertex v0 and v1. Finally, these two vertices are split to separate the tetrahedron and the faces adjacent to the two edges between v0 and v1 are rearranged to separate the tetrahedron.

In the third column, there are two common faces for e0 and e1 and for e2 and e3. Vertices v0 and v1 need to be split, and the faces adjacent to the two

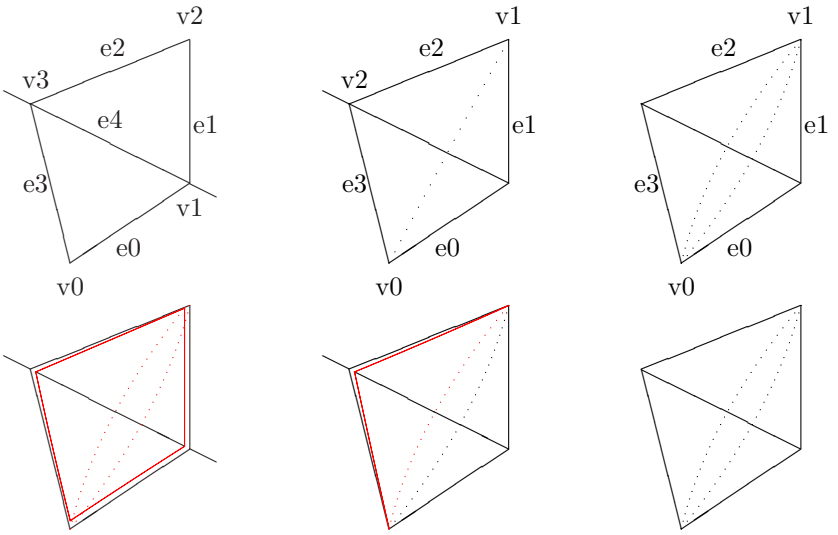


Figure 14.57: Cases for face pair removal

edges between v0 and v1 need to be rearranged, as before.

To perform the subdivision the first step is to find the edge that makes the smallest tetrahedron. This is a convex edge with the shortest distance between the two opposing vertices. For e4 at the top-left of figure 14.57, this would be v0 and v2. The four faces adjacent to the two faces extracted are added to a list of faces to be processed. Candidate faces are taken from this list, their adjacent faces are added to the 'to be processed' list, and then they are extracted. The edge of the face creating the smallest tetrahedron is chosen to find the face-pair to be extracted. If all edges in the face are smooth or concave, then the face is ignored.

There are some slight complications with this because some surrounding edges in figure 14.57 may be concave. If e0 and e1 are both concave on the left of figure 14.57, or edges e2 and e3 on the left and in the middle column are concave, then the procedure is the same as above.

If only one edge from the pair e2, e3 or from the pair e0, e1 is concave, and there is an adjacent coplanar face with the three tetrahedron vertices, then some topological adjustments need to be done. Figures 14.58 and 14.59 show two cases: on the top if e3 is concave and e2 convex, on the bottom if e2 is concave and e3 convex. The cases for e0 and e1 are similar.

When e3 is concave, the first step is to slice e3, second column. Vertex v3 is then split to create a large face (third column). Finally, two new edges are added, splitting this large face into triangular facets. The case for concave e2

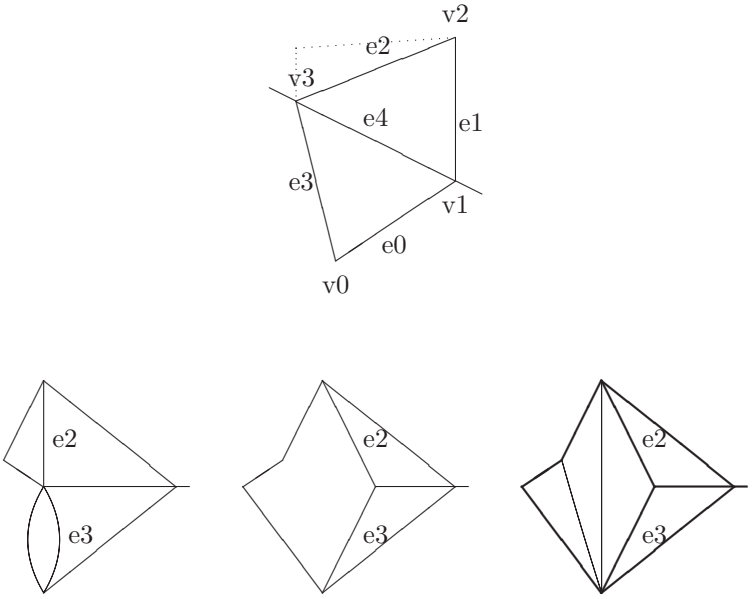


Figure 14.58: Handling mixed concave/convex edge pairs (part 1)

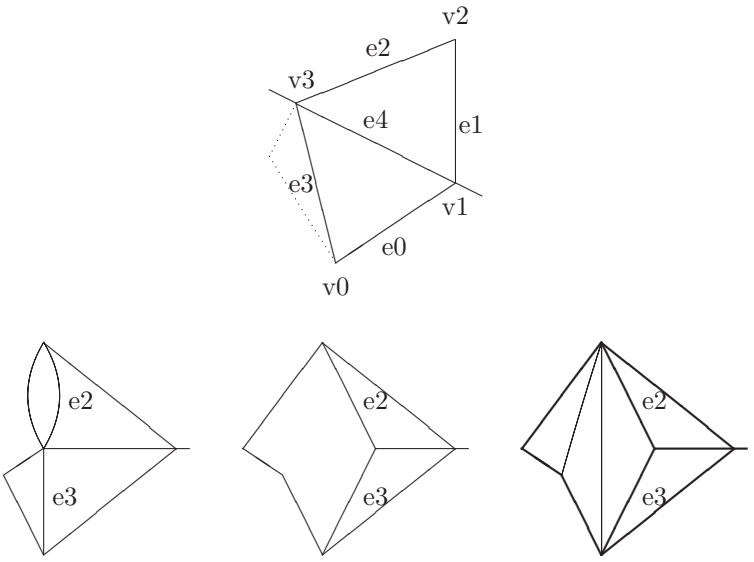


Figure 14.59: Handling mixed concave/convex edge pairs (part 2)

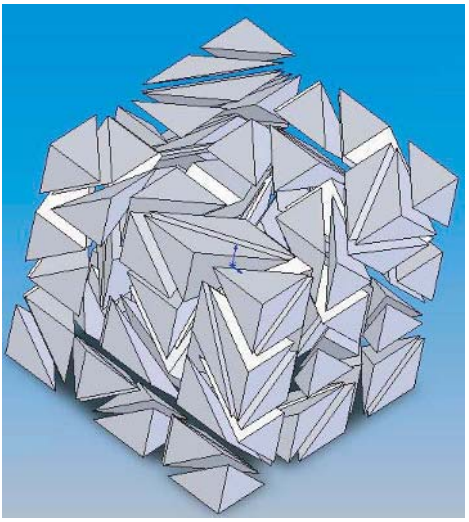


Figure 14.60: Tetrahedral decomposition of a cube

is symmetric.

A simple cube decomposition is shown in figure 14.60.

14.9 Patterns

The work described was published in Stroud and Xirouchakis [127].

Patterns are a type of space-filling mechanism that is not usual for mechanical parts and hence difficult to make using conventional CAD systems. The patterns described here are based on traditional Celtic patterns. Development of such patterns was described in articles by Glasner [44], [45], [46] based on the work of Bain [3]. However, it is necessary to point out that the implementation is not meant to be a complete Celtic art design tool. It produces what Bain terms the simplest class of patterns.

The method described by Bain [3], report by Glasner [44] starts by making a primary grid. The primary grid for a 3×4 pattern is shown in figure 14.61a. A secondary grid is then made by adding intermediate points, the points drawn striped in figure 14.61b. In the implemented code, a tertiary grid is then made from the remaining points, the striped points in figure 14.61c. The pattern creation technique uses a tiling technique, treating each tertiary grid point as the centre of a tile.

The corresponding tile boundaries are shown in figure 14.62.

Several tile types are used to fill this pattern. Figure 14.63 shows the main types, arranged into groups according to their symmetry. There is, in addition, one other important type, the EMPTY type that is used to mark

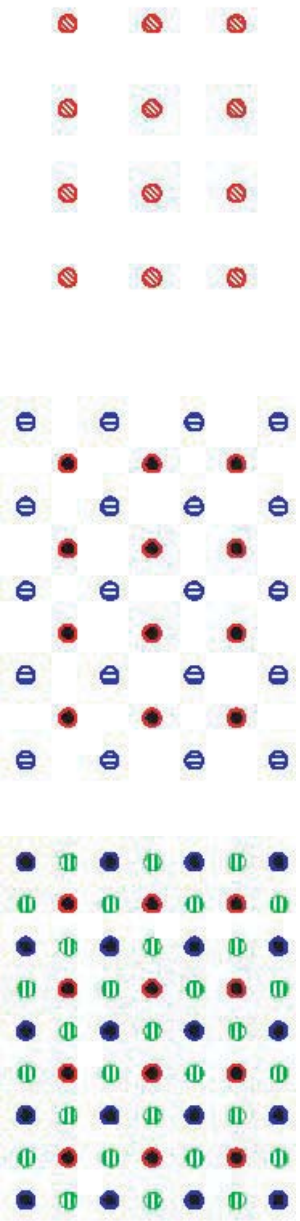


Figure 14.61: Primary, secondary, and tertiary grids for a 3×4 pattern (from [127])

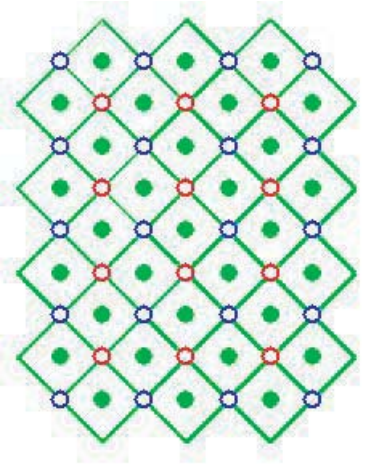


Figure 14.62: Tile boundaries for the 3×4 pattern (from [127])

gaps. To make a consistent pattern, all X-forms in the same row or same column must have the same type.

The way in which these are used to build up a pattern is shown in figure 14.64. The original pattern, an 8×12 block is shown in figure 14.64a. To create the patterned cross, first the corners are cleared, figure 14.64b, c, d, and e. Tiles adjacent to the clear region automatically become 'edge' tiles. The central portion is then cleared (figure 14.64f). A horizontal switch tile is then created (figure 14.64g), and finally two vertical switch tiles (figure 14.64h and i). The actual geometry is determined by the positions of the tile corner points so that geometrically they may not be in a linear sequence, but structurally they are in the same row or column of the tile matrix. This matrix also guides the joining process because tile neighbours can be recognised directly without searching.

The final shape and two 3D models created therefrom are shown in figure 14.65.

The pattern creation method is a two-dimensional graphic tool. Specifying a pattern size causes an array of tiles to be created. Only tiles in an even-numbered column and odd-numbered row or in an odd-numbered column and even-numbered column are used. Initially all boundary tiles are marked as edge types; all other tiles are marked as X types. The X tiles in the even columns are marked as X1 types; the tiles in the odd-numbered columns are marked as X2 types, where X1 and X2 correspond to the two X types shown in figure 14.63. Individual tiles can then be specified using their positions in the array and have their types changed to create the desired pattern.

Creation of a solid model from the pattern involves converting each separate tile and then joining coincident faces, a local operation which is more

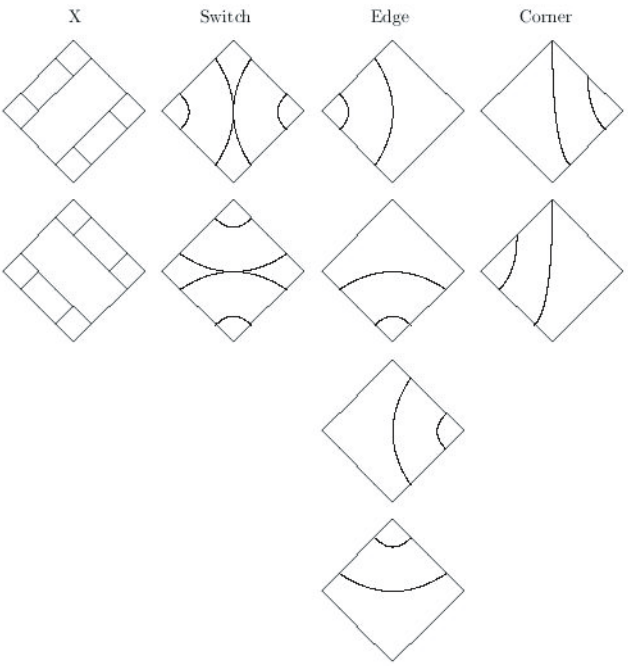


Figure 14.63: Celtic tile types (from [127])

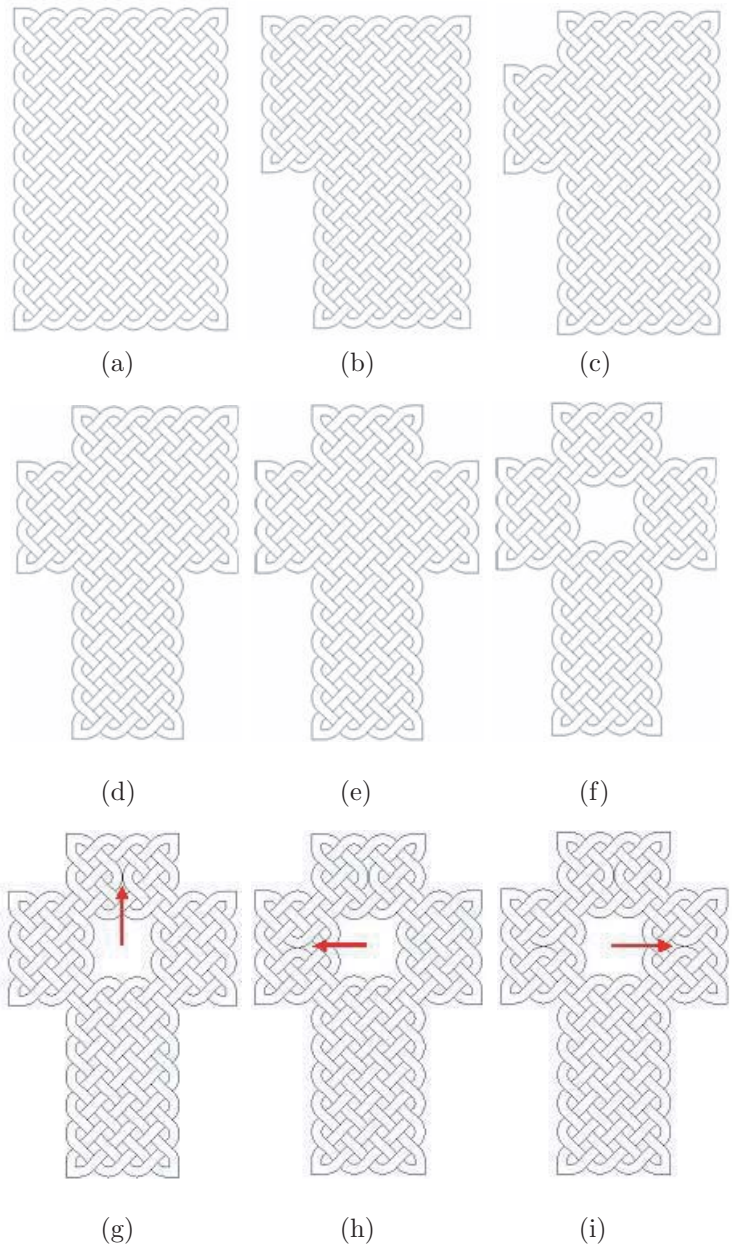


Figure 14.64: Cross-shape creation steps (from [127])

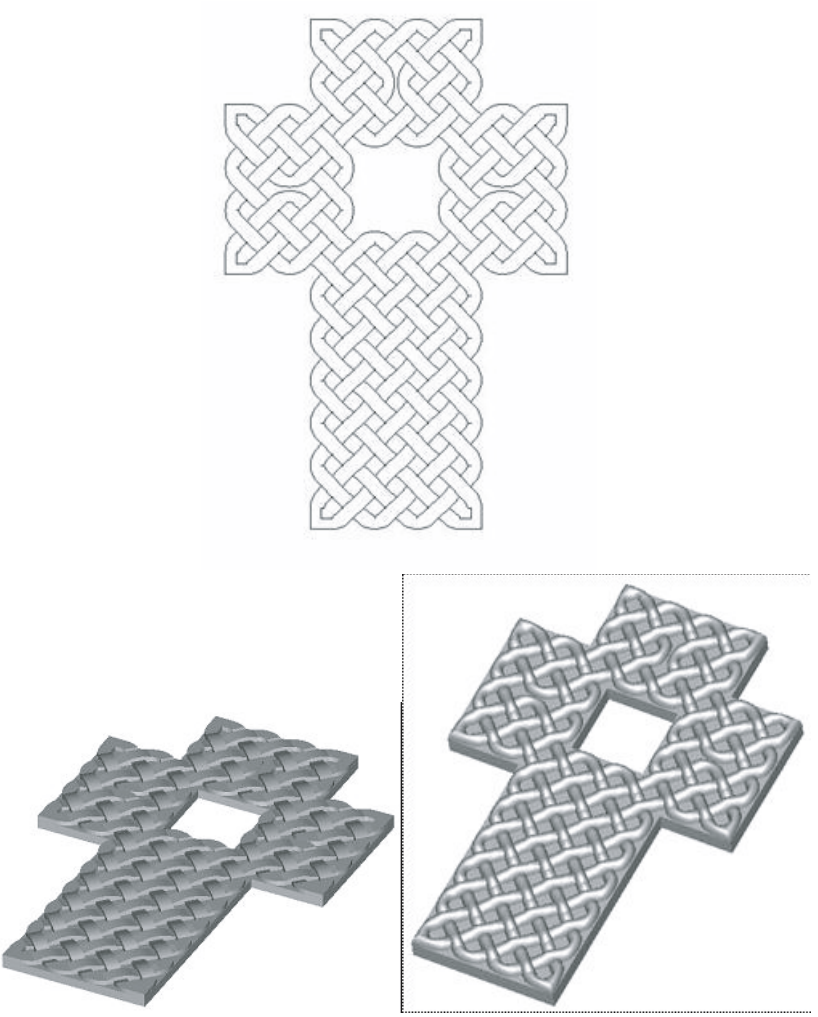


Figure 14.65: Celtic cross-shape and 3D models (from [127])

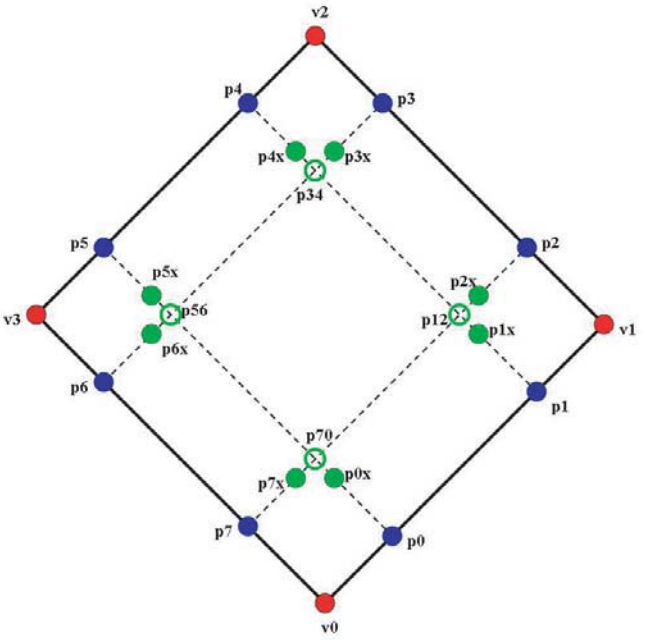


Figure 14.66: Basic point set for calculating tile geometry (from [127])

efficient than joining the tiles using a Boolean operation. Where this differs from traditional pattern making tools in CAD systems is that the tile shapes are not constant, but the neighbouring faces are identical. The resulting geometry is, therefore, more complex and would be difficult to produce using conventional operations.

The basic point set for calculating the geometry is shown in figure 14.66. The tile corner points are labelled v0 to v3; each side has two subdivision points, labelled p0 to p7. Neighbouring tiles with a common edge use the same points to ensure continuity.

There are four internal points, p12, p34, p56, and p70, and eight supplementary points used for generating the geometry, labelled p0x to p7x. Although, in the regular pattern, not all of these are necessary, in the general case, p0, p0x, p70, p56, p5x, and p5 need not be collinear.

For an X type tile, the topological structure is as shown on the left of figure 14.67. The X2 type is the same as the X1 type rotated through 90 degrees, so they are created in the same way. The other types are all similar to the topological structure on the right of the figure, although the switch types have a duplicate rounded shape, as well. The initial conversion method, used for creating the objects in figure 14.65 (bottom left) and figure 14.68, used extrusion to create the raised patterns. The current method uses spline geometry to create these shapes so that more complicated shapes can be made.

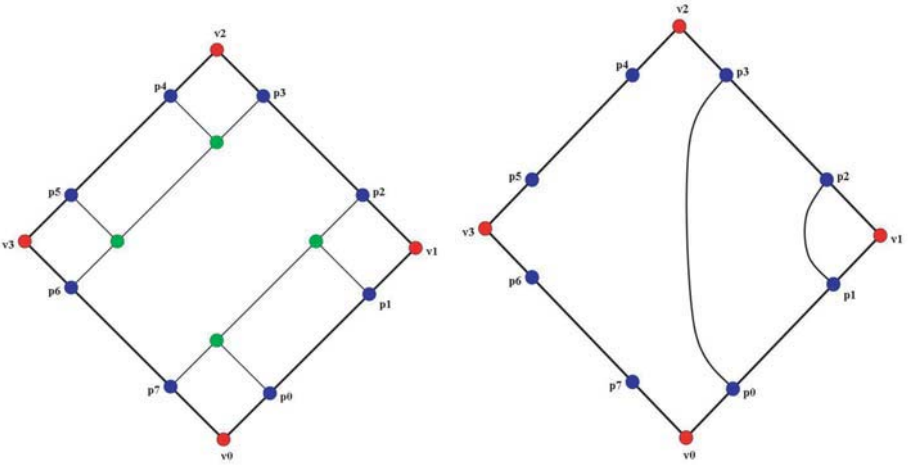


Figure 14.67: Internal topological structure for tiles (from [127])

Because the converted pattern is a volumetric object, it can also be combined with other operations to create more general shapes, such as that shown in figure 14.68.

An additional facility is to change the shape of the tiles and the weights of corners, giving extended patterns, as shown in figure 14.69. Changing the weight of the corner moves the bands closer to, or further from, a corner. This is essentially an 'offline' method, this would work differently if the pattern is created using interactive graphics, which would probably be the case for an interactive design tool.

With a minor extension to make them periodic, patterns may also be wrapped around cylinders or other surfaces so long as they can be mapped to a rectangular area, such as those shown in figure 14.70.

To be able to wrap patterns around like this, it is necessary to add an extra column to the end of the board, as in figure 14.71. The pattern looks as though it is incomplete at the left and right ends, but these elements are actually neighbours and join together.

This is shown in figure 14.72. When wrapped around, the corner marked A fits in the hole marked A', B fits with B', C with C', and D with D'. This means that it is necessary to note that these elements are neighbours.

The spherical case is a little different.

Here the pattern appears open at the top. In figure 14.73, edge a joins with edge h, b with c, d with e and f with g.

The geometry of the patterns becomes more extreme towards the 'poles' of the spheres. This problem is well known to map-makers, hence, the variety of map projection methods. In this simplified implementation, though, no attempt has yet been made to produce a 'pleasing' geometrical solution.

Figure 14.68: Celtic pattern integrated into solid model (from [127])

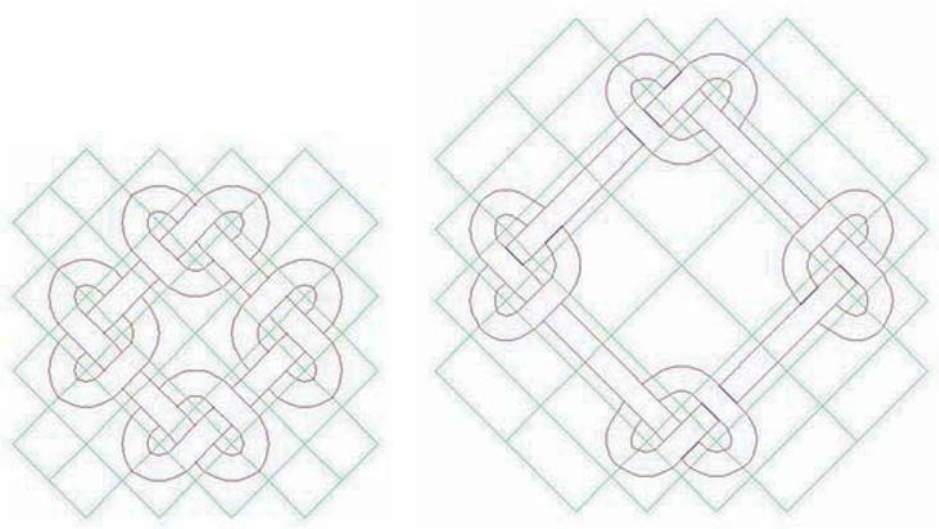


Figure 14.69: Celtic figure with changed tile sizes and corner weights (from [127])

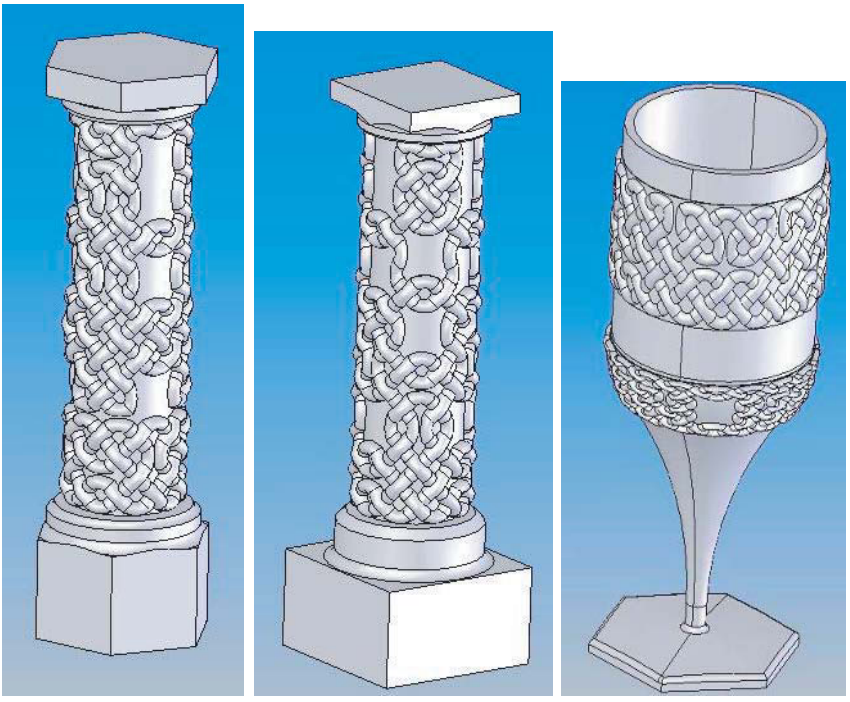


Figure 14.70: Cylindrical and spherical celtic patterns (from [127])

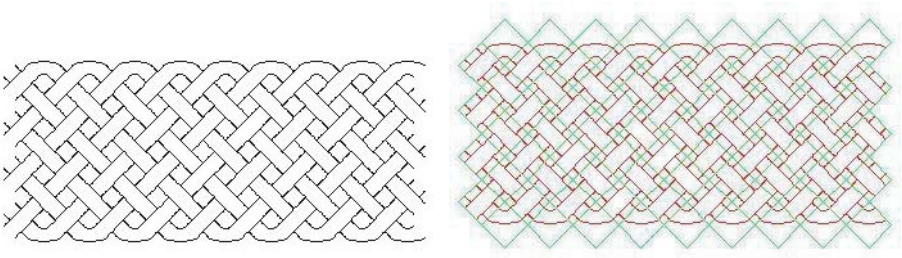


Figure 14.71: Cylindrical celtic pattern and board

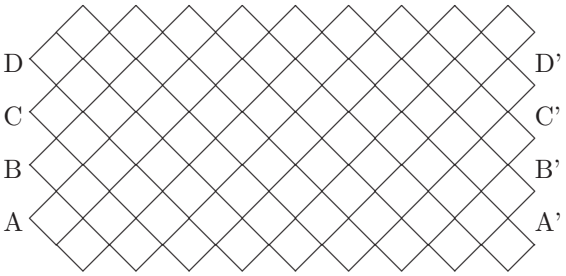


Figure 14.72: Cylindrical celtic pattern neighbours

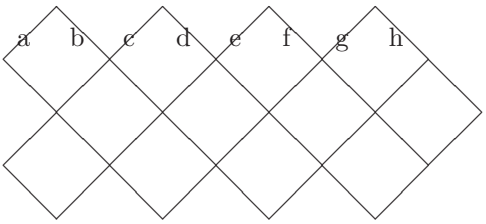


Figure 14.73: Cylindrical celtic pattern neighbours

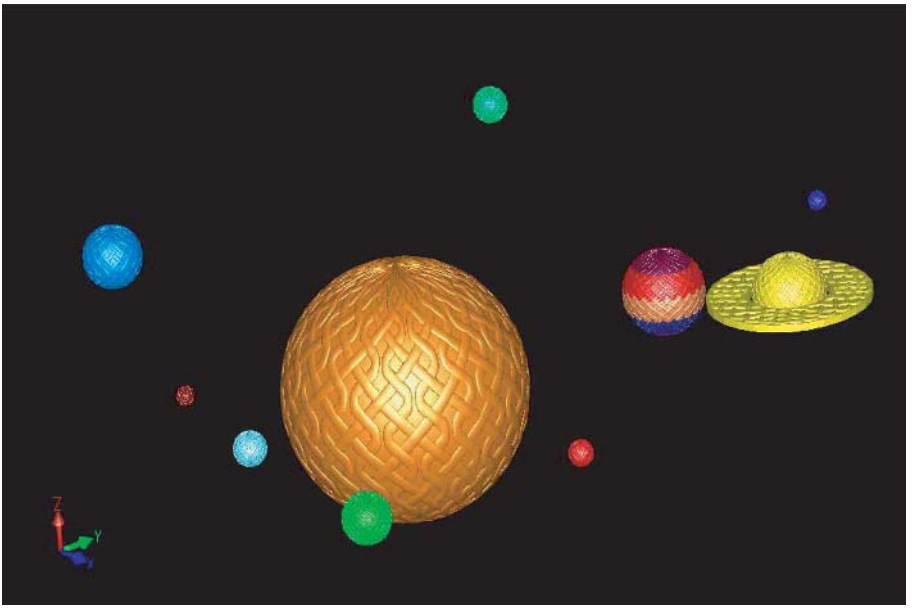


Figure 14.74: Celtic universe

Examples of spherical patterns are shown in figure 14.74.

Chapter 15

The Medial Axis Transform

The work described in this chapter was done jointly by Gabor Renner and Ian Stroud and is taken from various papers and reports written by them.

15.1 The medial axis transform

The medial surface (Medial Axis Transform surface or simply MAT) is a form of skeletal representation of an object. The MAT provides information that is missing from the boundary representation, by providing a notion of the thickness of the material at each point. The MAT offers a basis for several applications and is still the subject of research. It is perhaps a little early to make it part of a book because the calculation methods may well change. At present it is computationally very expensive and this makes it less attractive. However, the potential power of the tool means that it is important and needs to be researched further.

The literature survey below is mainly the work of Gábor Renner from [106] and [110] and is reproduced for completeness.

There are various methods that have been proposed to calculate the medial axis of solid models. A common method is based on cellular representation of an object. Hoffmann [59] published a method for the computation of the MAT of a CSG object. Another method was developed by Sherbrooke et al. [118] [119]. Reddy and Turkiyyah [105] calculated the MAT surface of boundary models using a dual space notion, which is the basis of section 15.5. Sheehy, Armstrong and Robinson [116] [117] describe a method based on Delaunay triangulation and tetrahedra. This section describes developments of the Reddy/Turkiyyah algorithm. The last method, described in section 15.6, also works in dual space, as does the Reddy/Turkiyyah algorithm, but it has

several differences.

There are many two-dimensional MAT algorithms. Among these are those developed by Montanari [87], Preparata [102], Srinivasan and Nackman [122]. Other algorithms, such as Gursoy's algorithm [52], can handle circular arcs as well as straight lines and points. So can the algorithm developed by Held, et al. [54]. Work is continuing as well in this area, as evidenced by the medial axis calculation methods described by Farouki and Ramamurthy [33] and Ramanathan and Gurumoorthy [103].

Hoffmann [59], Dutta and Hoffmann [29] proposed a method for the computation of the MAT of CSG solids. The method starts with computing the nearest bisecting points for each pair of boundary elements (points that are equidistant from both elements and have a minimum distance). The points are sorted by their distance from the boundary. Then an analysis is carried out to discover the topology of the MAT around the point, i.e., to identify whether the point is on a face, edge, or vertex of the skeleton. The analysis is based on the rank of the Jacobian of a system of equations that describes the equidistantial set belonging to the point. In the next step, points are processed by increasing distance and the MAT is constructed by tracing numerically the edges and faces arising.

Reddy and Turkiyyah [105] consider the MAT of a polyhedral object as a generalized Voronoi diagram of a set of mixed dimensional entities consisting of vertices, bounded edges, and bounded faces. In fact the dual of the Voronoi diagram, an abstract Delaunay triangulation, is computed, which reflects the structure of the MAT but ignores its geometry. As geometric entities, only the vertices of the MAT are determined, which is the most expensive part of the computations in the algorithm, because solutions of a set of non-linear equations are needed. If the polyhedral boundary is the result of an approximation of a smooth object, a lot of artefact facets in the MAT arise. These must be trimmed away to get a skeleton of the original object.

The method presented by Sherbrooke et al. [118],[119] starts from a "seam end" point and marches along an edge of the MAT to find the corresponding junction point. The marching is performed by integrating numerically the differential equation describing the seam, while a check is performed at each step to see whether any other boundary element than that determining the current seam (i.e., new boundary element) is to become active.

Etzion and Rappoport [31], introduced the Proximity Structure Diagram (PSD) which contains complete structural information for the MAT of an object, although not necessarily the exact geometry. They give an algorithm to compute the PSD of a planar polygon that uses only algebraic operations on linear geometric entities. Although for many practical applications, mainly when the evaluation of proximity relations is required, PSD is sufficient, parts of the MAT can be computed locally from the PSD in the areas of interest. In a further paper, Etzion and Rappoport [32] report the Proximity Structure Subdivision, which allows the separate computation of the symbolic and

geometric parts of the Voronoi diagram.

Choi and Seidel [22] describe the analysis medial axis surfaces using the Hausdorff distance as a means of avoiding instability due to variations in the boundaries of objects. Csabai and Xirouchakis [26] describe the computation of the medial axis for an important sub-class of objects as a means of obtaining volumetric information for collision detection and avoidance.

Spatial MAT structures can be determined by point sampling the boundary of the object, and then computing a (generalized) Delaunay triangulation. It can be proved that, under certain conditions, by increasing the point density (and using a reasonable sampling), the centre point of the simplexes of the Delaunay triangulation (in general tetrahedra) converges to the centre point of the maximal inscribed spheres, thus, to the skeleton points. Unfortunately the convergence is very slow, and the Delaunay triangulation is computationally intensive. Turkiyyah et al. [137] present a method by applying a constrained non-linear optimisation step (numerically) for each internal Delaunay tetrahedron. In this way a much sparser sampling is sufficient, and the geometry of the skeleton is improved by moving the vertices of the Delaunay tetrahedra and relocating their centres (as skeleton points).

Yet another approach is based on thinning cellular models of objects to get a digital skeleton; for example, see Sudhalkar et al. [129], Näf [90], Székely [132] or Foskey et al. [41]. While being relatively stable, there are also artefact problems due to the discretisation of the cells. Objects that are, in reality, symmetric may not have symmetric cellular representations. This is a different approach to that taken here and hence will not be discussed further.

15.2 MAT of a solid object

15.2.1 MAT terminology

The MAT algorithms described here work by calculating the critical points of a facetted approximation of the object. The critical points are the vertices of the MAT. The curves and surfaces of the MAT can be calculated as a post-processing step; as described in [106] once the dependencies have been identified.

Figure 15.1a shows a simple object and its MAT. There are four vertices, at the critical points, marked K, L, M, and N. There are four MAT edges, KL, LM, MN, and NK, and the MAT face is enclosed by these four edges. In addition there are eight wing edges, often trimmed: KA, KE, LB, LF, MC, MG, ND, and NH.

The critical points are the centres of what are termed 'Maximal Inscribed Spheres' in the object. The maximal inscribed sphere of an object is a sphere that is contained in the object but which is not a proper subset of (i.e., completely contained in) any other sphere inside the object. The medial axis (MA) or skeleton of a solid object is the locus of the centre points of

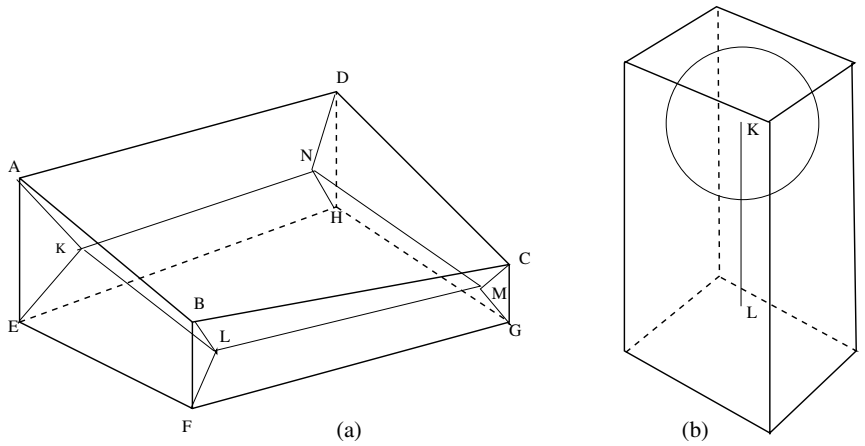


Figure 15.1: Medial surface (MAT) elements

all maximal spheres inside the object. Because the MA of a 3D object is a structure containing basically faces, it is also called the "Medial Surface". The radius function of the MA is a continuous function at the points of the MA and is equal to the radius of the corresponding maximal sphere. The MA together with the radius function is the MAT of the object.

For each critical point four unknowns are needed, that is, three unknowns for the coordinates of the centre point and one for the radius. In its simplest form, therefore, a vertex of the MAT is determined by four constraints, i.e., four touching points where the sphere with centre point at the vertex touches the boundary of the object (e.g., point K in figure 15.1a). For the calculation, curved objects are excluded, so this means that four different boundary elements are needed. If there are only three elements, then one unknown remains, which is equivalent to the sphere rolling along a curve and thus creating an edge of the MAT structure (KL). If there are only two touching points, i.e., the maximal sphere touches only two boundary elements, then the locus of the centre of the sphere forms a surface of the MAT (KLMN).

These are the regular cases for the topological elements of the MAT of a solid object. However, it is common that degeneracies occur such as the block with a square cross-section shown in figure 15.1b. Here the vertices of the MAT (points K and L in figure 15.1b) are determined by five elements. The position is determined by an end-surface and three side faces. The fourth side face is redundant for the calculation but provides a fourth touching point. The edge, KL, of the MAT is determined by the four side faces rather than just three. The MAT of a cube consists of a single vertex defined by all six faces together.

Because the algorithms deal with facetted objects, the relevant boundary elements for the calculation of the MAT are all faces, concave edges, and

concave vertices. It is clear that a sphere with a non-zero radius inside cannot touch a convex edge, and so these are excluded. The definition of a concave vertex is not easy to state simply, however. It is also obvious that a sphere with a non-zero radius cannot touch a vertex where only convex edges meet. Similarly, a sphere cannot touch a vertex where there is only one concave edge. A vertex that has only two concave edges does not contribute either because a sphere touching such a vertex also touches the two concave edges themselves. A vertex with only concave edges is definitely concave; a vertex with three or more concave edges has to be examined to determine whether it is needed. If a sphere can touch the vertex without touching two or more edges, then the vertex is needed.

15.3 Calculating critical points

The geometric calculations concern the determination of spheres touching four boundary elements selected from the set of all relevant boundary elements. The centre points of the spheres provide the vertices of the MAT structure; their radius is equal to the value of the radius function at a MAT vertex. The edges and faces of the object are finite portions of the corresponding infinite curve or surface, for the objects considered here, planes or straight lines. In the geometric calculations, the infinite geometric elements are used and the points where the spheres touch them are checked to see whether they lie in the finite bounding parts.

The descriptions here were originally developed and validated in conjunction with Tigyi [135]. Although work in the same vein, but for curves, has since been published by Culver et al. [27], the full point analysis is presented here because it was developed independently before that by Culver et al. and because of its importance to the MAT calculation algorithm presented later.

The critical point calculation is described by Renner and Stroud [110] based on Renner's work and the work done by Renner and Tigyi, reported in [135]. The following summarises the equations for spheres touching planes, lines and points.

- For a plane:
  $(C - P)n - r = 0$
  where n is the unit normal vector and P is a point of the plane.

- For a straight line:
  $(C - P)^2 - [(C - P)e]^2 - r^2 = 0$
  where e is the direction vector and P is a point of the straight line.

- For a point:
  $(C - P)^2 - r^2 = 0$
  where P is the point.

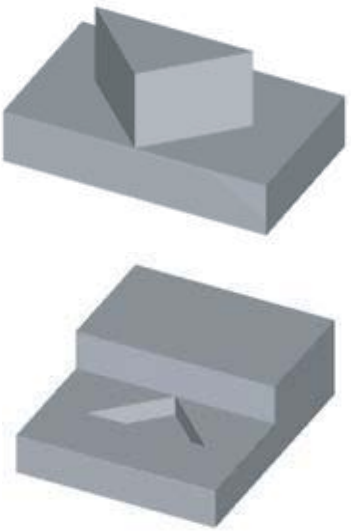


Figure 15.2: Collinear straight edges at a vertex (from [110])

To determine a critical point, it is necessary to use a set of four of these equations. There are fifteen combinations of these elements which can occur. These combinations lead to equations of different degrees for critical point calculation. However, there are also combinations of four elements where it will not be possible to determine critical points. One such is the side faces of the elongated figure with a square cross-section, as shown in figure 15.1. Another example is shown in figure 15.2, where there are two collinear straight edges meeting at a vertex. Similarly, if two coplanar faces meet at an edge then this, too, would lead to a degenerate case. However, it is assumed that edges between coplanar faces are removed as a pre-processing step. Collinear edges with a common vertex, however, cannot always be eliminated, as illustrated in the figure.

It should also be noted that planes have a direction. The convention is that the normal of the plane points outwards from the body. Renner and Tigyi analysed these cases as well, excluding combinations of elements with two planes with parallel normal vectors oriented in the same direction. The cases that they identify where a critical point cannot be determined are:

- Four coplanar points not lying on a circle
- Three parallel planes
- Two parallel planes with the same normal vectors
- Point(s) lying out of the directed regions of the plane(s)

- Edge(s) lying out of the directed regions of the plane(s)

- Three collinear points

- Two points lying on the same plane

- Two points lying on the same edge

- Point and edge lying on a plane but point does not lie on the edge

Once these cases have been excluded, Renner and Tigyi describe how a system of equations for a sphere touching four boundary elements is set up. They also investigated the structure of the equation systems determined from combinations of different elements. As a result they discovered various important properties and hence were able to simplify the calculations. This is important because it means that it is not always necessary to use numerical solvers, and so more accurate results can be obtained faster than using an iterative method.

In [110], Renner describes how the sphere radius r can always be eliminated, which reduces the number of equations to three and the total degree to eight. He gives the example of the equations for a sphere touching four planes:

$$(C - P_i)n_i = r \qquad i = 1, 2, 3, 4$$

subtracting the first equation from the second, third, and fourth, gives:

$$C(n_1 - n_i) = P_1 n_1 - P_i n_i \qquad i = 2, 3, 4$$

a system of three equations for the three coordinates of the centre point. Renner and Tigyi analysed the cases and found that most cases mean that the centre point and radius of the maximally inscribed sphere can be found using an analytic solution. In these cases, the solution is found by solving quadratic, cubic, or quartic polynomial equations. Only in a few cases is it necessary to use a non-linear system solver, and hence, the solution is more stable and quicker.

The solution table is given below:

	Case	Eqn. sys. Struc.	Cont. sols. Locus
1	4point	3linear	straight
2	4plane	3linear	straight
3	1plane, 3point	2linear, 1quadratic	no
4	2plane, 2point	2linear, 1quadratic	quadric
5	3plane, 1point	2linear, 1quadratic	no
6	3plane, 1straight	2linear, 1quadratic	straight
7	3point, 1straight	2linear, 1quadratic	straight
8	2point, 2straight	1linear, 2quadratic	straight / quadric
9	2plane, 2straight	1linear, 2quadratic	straight / quadric
10	2plane, 1point, 1straight	1linear, 2quadratic	quadric
11	1plane, 2point, 1straight	1linear, 2quadratic	quadric
12	4straight	3quadratic	straight / quartic
12s	4straight	1linear, 2quadratic	straight / quadric
13	1point, 3straight	3quadratic	straight / quartic
13s	1point, 3straight	1linear, 2quadratic	straight / quadric
14	1plane, 3straight	3quadratic	straight / quartic
14s	1plane, 3straight	1linear, 2quadratic	straight / quadric
15	1plane, 1point, 2straight	3quadratic	straight / quartic
15s	1plane, 1point, 2straight	1linear, 2quadratic	straight / quadric

(from [110])

15.4 Dual space

The multiple start point algorithm described in the next section derives from the algorithm of Reddy and Turkiyyah. Creating the dual of an object is described in section 6.10. Some examples are given in figures 15.3 to 15.7.

Dual space objects are not really useful directly for design, but the dual can be used as a background mapping object for different algorithms, such as the unfolding algorithm in chapter 5 or the feature finding method of Ansaldi et al. [1] mentioned in chapter 9. In the medial axis transform algorithms, each dual space node corresponds to a MAT vertex. Each face of the node corresponds to an edge or a wing of the MAT.

15.5 The multiple start point algorithm

The Renner/Stroud algorithm is an extension of the Reddy/Turkiyyah algorithm that copes with general objects with holes by looking for multiple pieces of MAT surface. Another development was to produce multiple start points, which means that the algorithm works in parallel and is faster, but does not really change how things work.

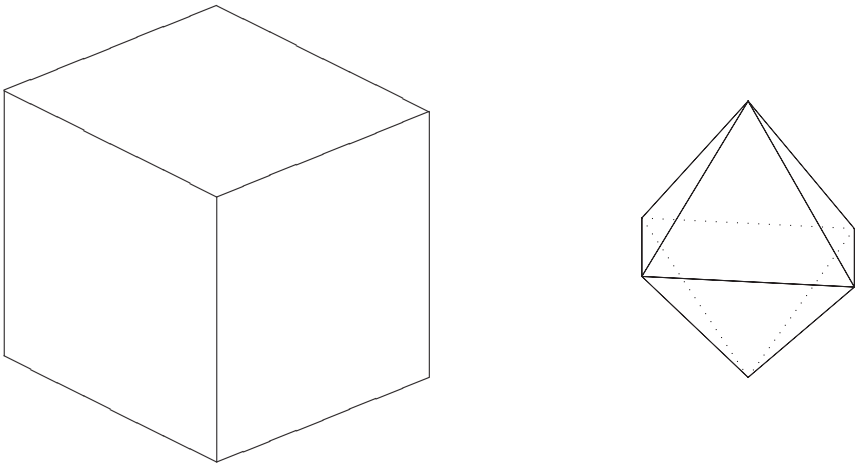


Figure 15.3: Cube and dual

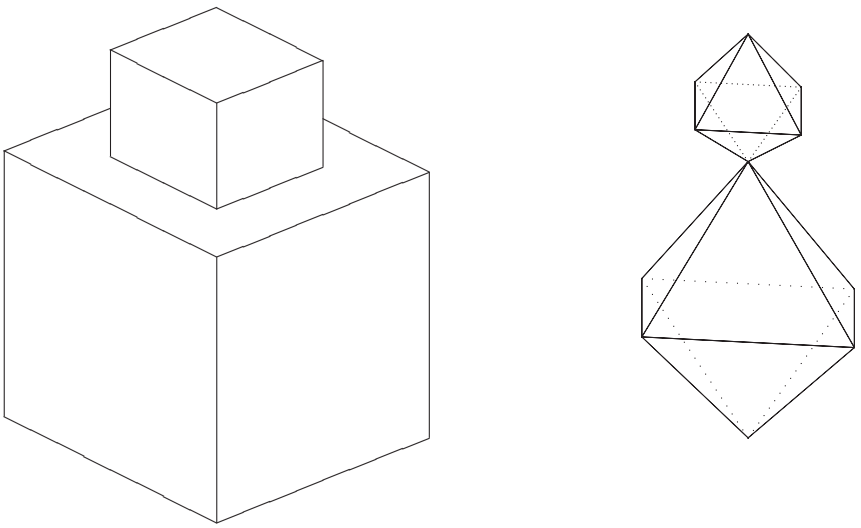


Figure 15.4: Cube with boss and dual

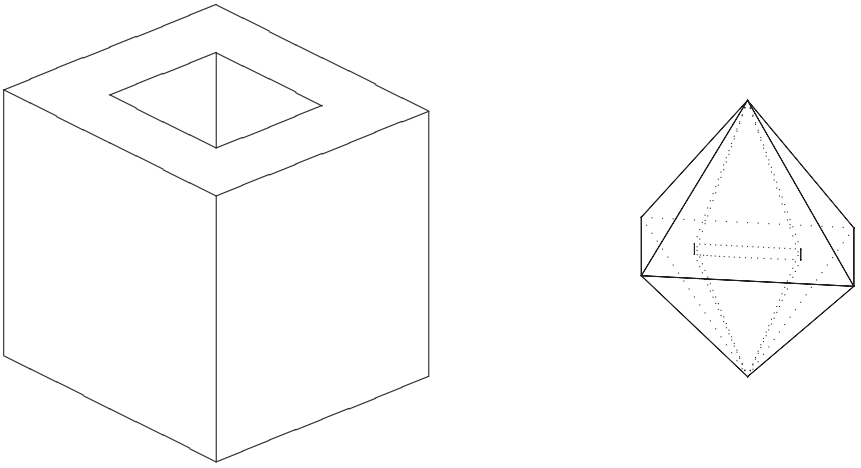


Figure 15.5: Cube with through hole and dual

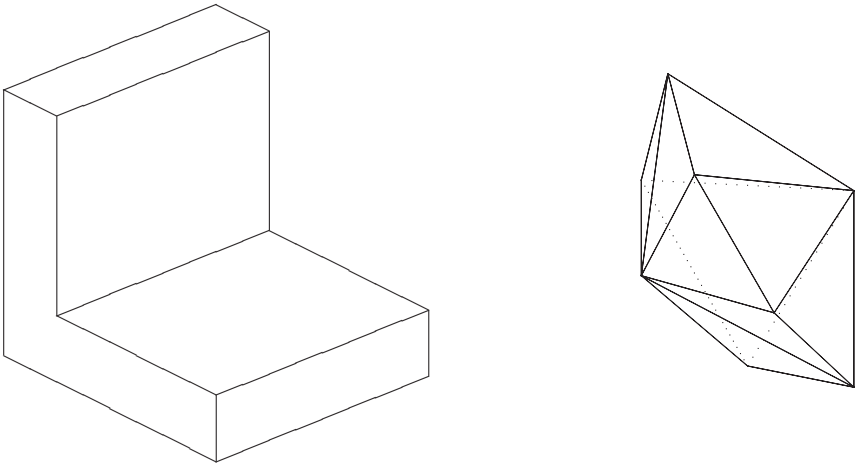


Figure 15.6: L-block and dual

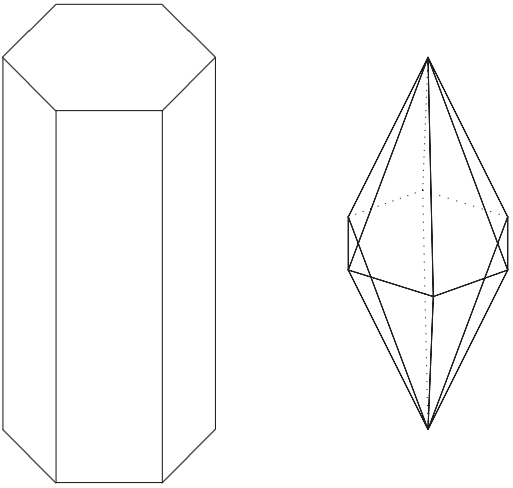


Figure 15.7: Hexagonal rod and dual

The basic single start point algorithm can be summarised as follows:

1. Make a list of all relevant units in the object. A unit is either a face, a concave edge, or a vertex where two or more concave edges meet.

2. Determine a starting point: in the original algorithm, a convex vertex where at least three faces meet along convex edges. These first three elements form a 'seed triangle' that is placed on a list of triangles to be processed

3. Pick a triangle from the list to be processed. If none exists, look for unused elements to make a new seed triangle; if all elements have been used, stop. Find a set of fourth elements that, together with the three defined by the triangle, bound spheres inside the object. If no candidate fourth elements are found, repeat this step.

4. Sort the list of possible elements in order of increasing distance of the sphere centre from the previous critical point.

5. Discard any candidates where other elements intersect the sphere; i.e., the distance to some element is less than the radius of the sphere bounded by the three seed elements and the candidate fourth element.

6. If any fourth elements remain, pick the closest one and form a new tetrahedron. If any triangles already exist, then merge the existing triangle into the new tetrahedron, if possible. Add any

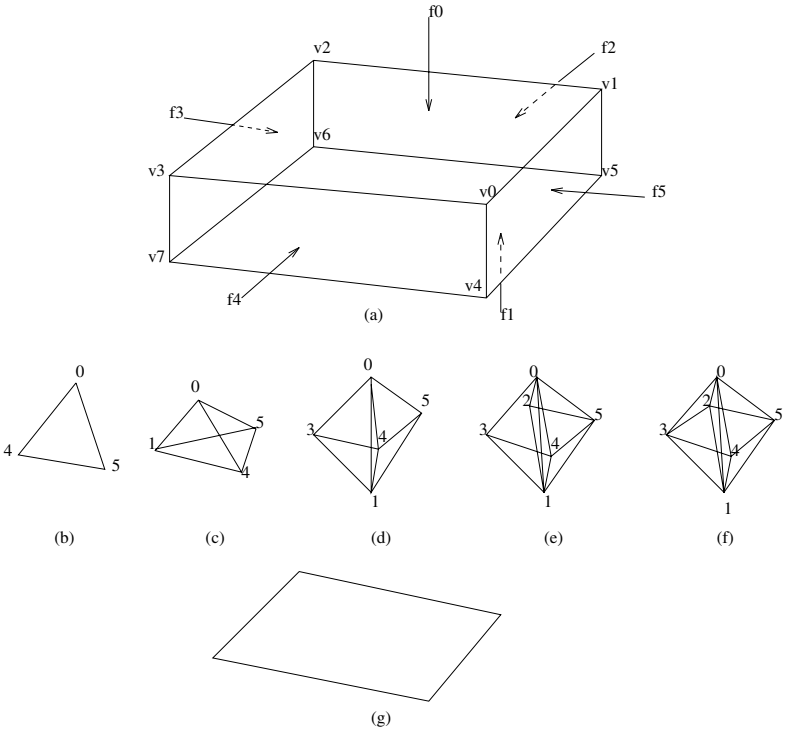


Figure 15.8: MAT calculation for a block (from [106])

new triangles bounding the tetrahedron to the list of triangles to be processed.

7. Repeat from Step 2.

The tetrahedral structure is built up during the process of calculating critical points. The actual MAT surface is the 'dual' of this structure.

For a simple $100 \times 70 \times 50$ block (shown in figure 15.8), the algorithm works as follows.

The relevant elements to define the MAT surface are the boundary faces, labelled f0...f5. A convex vertex (e.g., vertex v0) is chosen as the start point and three (here there are only three) faces at the vertex are chosen as defining the seed triangle, here faces 0, 4, and 5 (figure 15.8b). The triangle is placed on the list to be processed. This triangle is then selected as a potential base for a new tetrahedron, and a fourth element is sought. Each of the other three elements, faces 1, 2, and 3, are combined with faces 0, 4, and 5 to find the critical point, defined logically in the datastructure as a tetrahedron. Face 1 is chosen to make the tetrahedron (figure 15.8c) and the three new triangular triangles [0,1,4], [1,4,5], and [0,1,5] are placed on the list to

be processed. The next triangle [0,1,4] builds a tetrahedron with face 3 (figure 15.8d) giving the new triangles [0,3,4], [1,3,4], and [0,1,3], which are also placed on the list. The next triangle [1,4,5] is selected. Here no fourth element is found. Triangle [0,1,5] builds a tetrahedron with face 2 (figure 15.8e) giving new triangles [0,2,5], [1,2,5], and [0,1,2]. Triangles [0,3,4] and [1,3,4] do not build tetrahedra. Triangle [0,1,3] builds a tetrahedron with face 2 (figure 3f), forming new triangles [0,2,3] and [1,2,3]. The third triangle [0,1,2] already exists so tetrahedron 2 and tetrahedron 4 are connected through this triangle. Triangles [0,2,5] and [1,2,5] do not build tetrahedra. When triangle [0,1,2] is encountered on the list, it is noted that it has already been incorporated into a second tetrahedron so it is not processed. No other triangle builds a tetrahedron. The incomplete tetrahedra, i.e., where no fourth element is found, are preserved as they generate the so-called wings of the MAT.

After processing, the following complete tetrahedra have been defined:

- **Tetrah. 0: [f0,f1,f4,f5] critical point position (25,-10,0)**
- **Tetrah. 1: [f0,f1,f4,f3] critical point position (-25,-20,0)**
- **Tetrah. 2: [f0,f1,f2,f5] critical point position (25,10,0)**
- **Tetrah. 3: [f0,f1,f2,f3] critical point position (-25,10,0)**

And the following incomplete tetrahedra and their corresponding original object vertices have been defined:

Tetrah. 4: [f0,f4,f5,-1] vertex 0
Tetrah. 5: [f1,f4,f5,-1] vertex 4
Tetrah. 6: [f0,f2,f5,-1] vertex 1
Tetrah. 7: [f1,f2,f5,-1] vertex 5
Tetrah. 8: [f0,f2,f3,-1] vertex 2
Tetrah. 9: [f1,f2,f3,-1] vertex 6
Tetrah. 10: [f0,f3,f4,-1] vertex 3
Tetrah. 11: [f1,f3,f4,-1] vertex 7

The multiple start point algorithm is similar to the single start point algorithm except that the initial list of facets to be processed is longer. Instead of finding a single starting point, all possible starting points are found. This means that the triangles that have no fourth element are found straightaway. Multiple pieces are also found using this method. The algorithm also works in parallel and proved quicker than the original single point algorithm.

Some examples are shown in figures 15.9, 15.10, 15.11, 15.12, 15.13, and 15.14.

To represent the dual MAT structure some extra structures were created, the **FACET**, **TETRA**, and **STRUT**. An **eunit** is a face, edge, or vertex used for the MAT calculation. These are illustrated in figure 15.15.

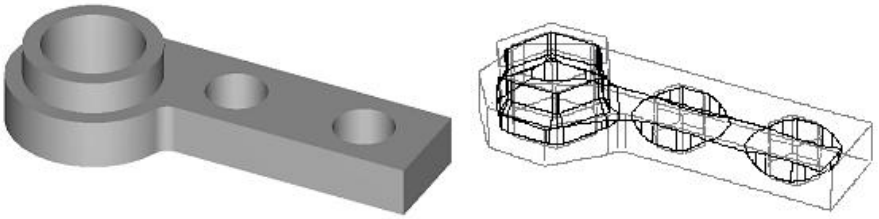


Figure 15.9: Bar and MAT (from [110])

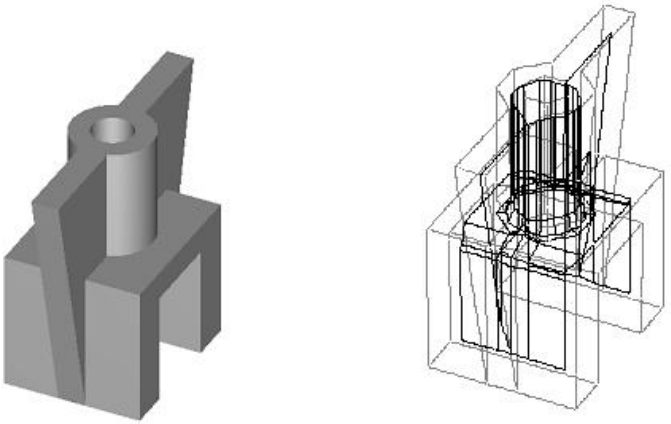


Figure 15.10: Winged object and MAT (from [110])

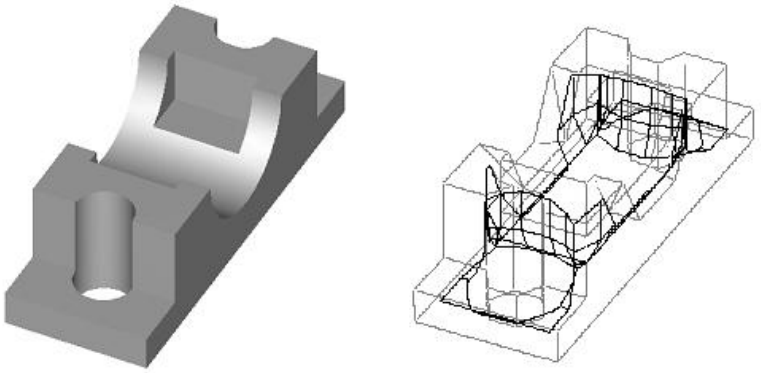


Figure 15.11: Bracket and MAT (from [110])

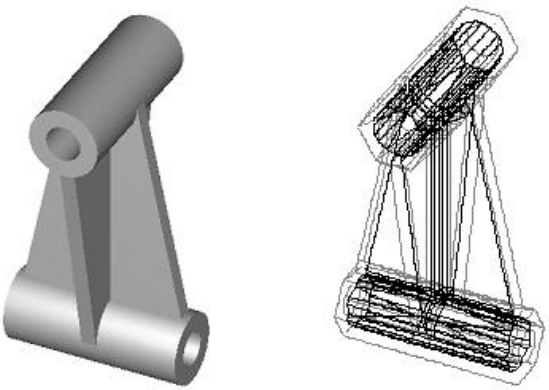


Figure 15.12: Perpendicular cylinder object and MAT (from [110])

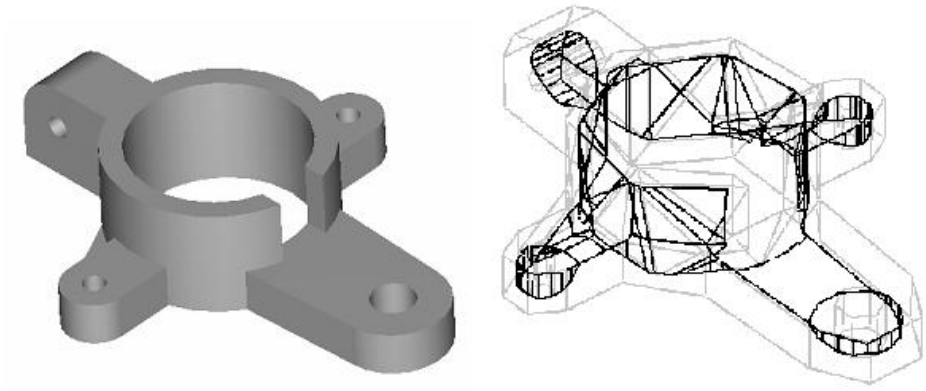


Figure 15.13: Cylindrical bracket and MAT (from [110])

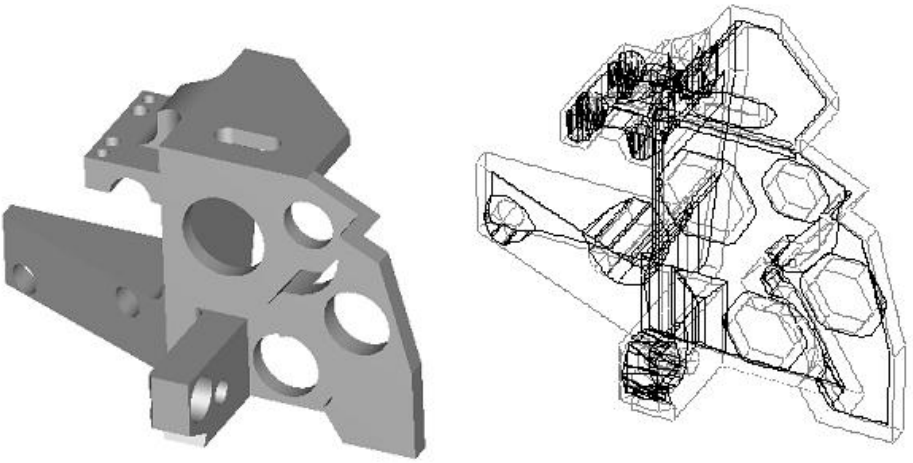


Figure 15.14: Braid's Ferranti part and MAT (from [110])

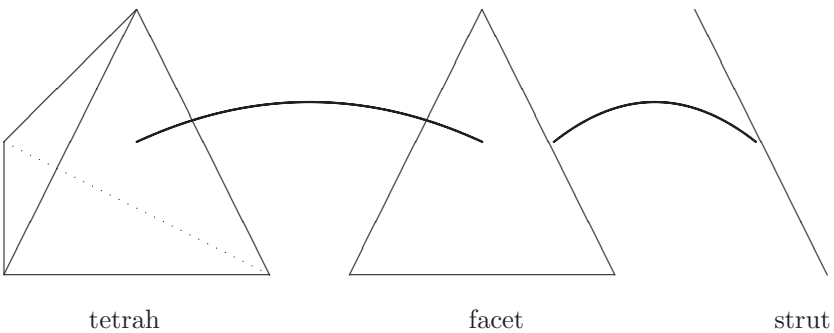


Figure 15.15: Tetrahedron (tetrah), facet, and strut

class strut

int	number;	The number of the strut, for debugging purposes
eunit	*start;	A pointer to the element corresponding to the start element of the strut
eunit	*end;	A pointer to the element corresponding to the end element of the strut
facet	*fer;	A pointer to one facet using the strut as an edge
loop	*loop;	A pointer to the loop of edges corresponding to the strut
strut	*next;	A pointer to the next strut in the list of all struts

class tetrah

int	number;	The number of the tetrhedron, for debugging purposes
facet	*basel;	The base facet of the tetrahedron
facet	*side1l;	The first extension face
facet	*side2l;	The second extension face
facet	*side3l;	The third extension face
eunit	*a0;	An entity defining the maximal sphere
eunit	*a1;	An entity defining the maximal sphere
eunit	*a2;	An entity defining the maximal sphere
eunit	*a3;	An entity defining the maximal sphere
vertex	*vvert;	A pointer to the vertex of the MAT corresponding to the tetrahedron
vector	centre;	The centre of the maximal sphere
double	radius;	The radius of the maximal sphere
tetrah	*next;	A pointer to the next tetrahedron in the list of all tetrahedra.

class facet

strut	*side0;	The first edge of the facet
strut	*side1;	The second edge of the facet
strut	*side2;	The third edge of the facet
facet	*s0prev;	A pointer to the previous facet around strut 0
facet	*s0next;	A pointer to the next facet around strut 0
facet	*s1prev;	A pointer to the previous facet around strut 1

Continued on next page

facet	*s1next;	A pointer to the next facet around strut 1
facet	*s2prev;	A pointer to the previous facet around strut 2
facet	*s2next;	A pointer to the next facet around strut 2
facet	*next;	A pointer to the next facet in the list of all facets
tetrah	*tetra;	A pointer to the tetrahedron of the facet
edge	*edge;	A pointer to the MAT edge to which the facet corresponds
curve	*curve;	A pointer to the curve of the MAT edge
int	type;	An indicator whether this is a real MAT edge or a wing edge
int	number;	The number of the facet, for debugging purposes
int	flag;	A general flag

The tetrah class corresponds to a vertex or node of the MAT. It is bounded by four facets and references the four elements that bound the maximal sphere. It also contains the maximal sphere data (centre and radius) and a pointer to the vertex of the mat to which it corresponds. A tetrah entity can be a degenerate tetrahedron as well, corresponding to a vertex of the original object. Building the MAT means creating a set of these tetrah entities. A post-processing step then builds the MAT structure using the normal faces, edges, and vertices of the solid modelling system. Geometrically, only the vertex positions are defined, the edges of the MAT are straight in the first step, and the faces have no surfaces. As described in [106], it is then possible to perform a geometric refinement process because the MAT construction process has defined the dependencies, three for a facet, two for a strut. The refinement process involves calculating a set of points equidistant from the three entities of the facet or the two entities of the strut. A curve or surface is then fitted to these points.

The facet entity corresponds to an edge of the MAT; a facet pair lies between two tetrah entities that are connected by the edge. The first step in creating the MAT is to look at all vertices in the original object and create facets with the elements that will form the bases of tetrahedra.

The strut entity corresponds to a loop of edges in the MAT. It is necessary to check whether more than one strut lies between the same two entities used for calculation of the critical points. If this is so, then one edge loop becomes a hole and the other the boundary of a face. Figure 15.16 illustrates such a medial surface.

15.6 The divide-and-conquer algorithm

It is easiest to explain the difference between the divide-and-conquer algorithm and the multiple start point using figures. Consider the object in figure 15.17.

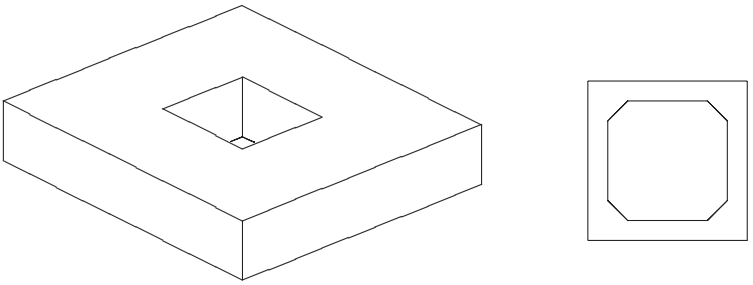


Figure 15.16: Block with hole and MAT

The MA is shown in figure 15.18. The dotted edges are the wings, which are usually suppressed.

It is possible to label the critical points with the entities that constrain them, as shown in figure 15.19.

The dual space decomposition is shown in figure 15.20.

The difference between the Reddy/Turkiyyah, the multiple-start-point, and the divide-and-conquer algorithms is that the Reddy/Turkiyyah algorithm and the multiple start point algorithm build the dual space node structure, whereas the divide and conquer subdivides the outside. The single start point type of algorithm is illustrated in the next sequences of figures.

In figure 15.21, vertex v2 is chosen as a start vertex because it is the first convex vertex found. The two adjacent elements, e1 and e2, form the base of a Delaunay triangle, and a third element is sought. Edge e3 is found to delimit a maximal sphere, completing the first Delaunay triangle.

Completing the Delaunay triangle introduces two new edges, each of which is potentially the base of a new Delaunay triangle so it has to be checked. The combination e2–e3 is checked, but there is no corresponding third element to delimit a new Delaunay triangle (e1 is excluded because it has already been used with e2 and e3). The combination corresponds to a vertex of the original object (v3). See figure 15.22

The combination e1–e3 forms the base of a new Delaunay triangle with the third element v1 delimiting a sphere within the original object (figure 15.23).

The combination v1–e1 does not delimit a circle with any other third element, and so it is not the base of a new Delaunay triangle. The pair v1–e3 does, however, form a new triangle with element e0 (figure 15.24).

The pair v1–e0 does not form the base of a new Delaunay triangle. The pair e0–e3 delimits a sphere with edge e7 (figure 15.25).

The pair e3–e7 forms a Delaunay triangle with vertex v4 (figure 15.26).

The pair e0–e7 forms a Delaunay triangle with vertex v0 (figure 15.27).

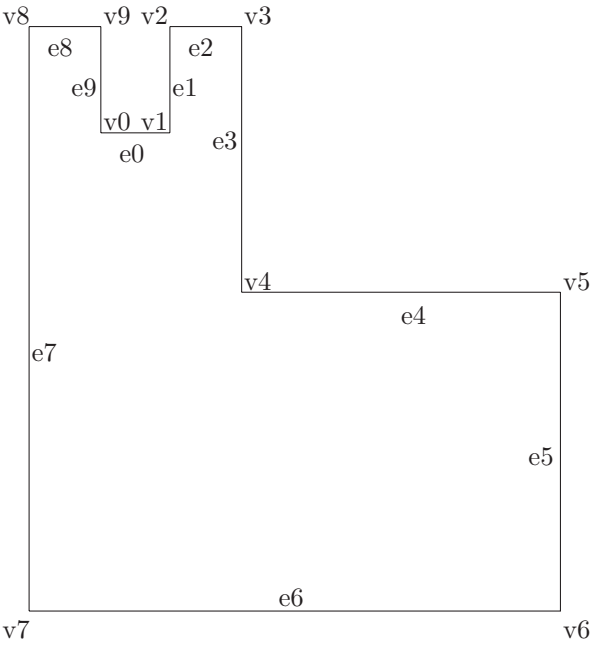


Figure 15.17: Simple figure (from [128])

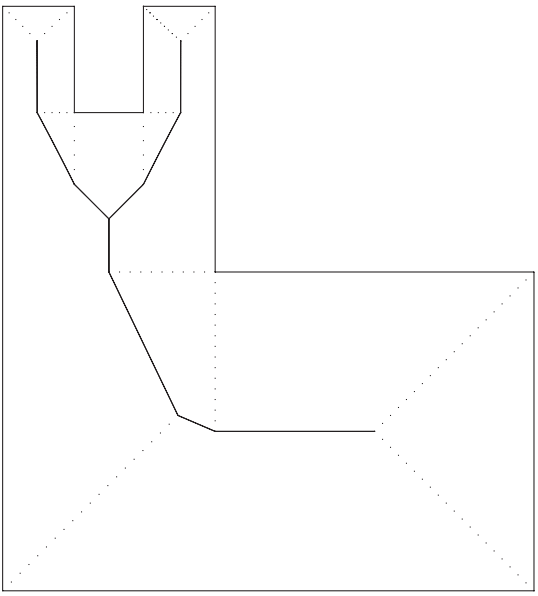


Figure 15.18: MA of simple figure (from [128])

The pair e3–v4 does not form a Delaunay triangle. The pair e7–v4 forms a Delaunay triangle with e6 (figure 15.28).

The pair e0–v0 does not form a new Delaunay triangle with any other node. The pair e7–v0 forms a Delaunay node with e9 (figure 15.29).

The pair e6–v4 forms a Delaunay triangle with e4. The pair e6–e7 is adjacent and there is no third element, this corresponds to vertex v7 (figure 15.30).

The pair e9–v0 does not form a Delaunay triangle with any element. The pair e7–e9 forms a Delaunay triangle with edge e8 (figure 15.31).

The pair e4–v4 does not form a Delaunay triangle. The pair e4–e6 forms a Delaunay triangle with edge e5 (figure 15.32).

None of the other candidate base lines form a Delaunay node because they all correspond to vertices. After processing these, the algorithm stops.

This method is reasonably stable because it does an exhaustive search for elements that delimit a critical point. However, not all nodes are delimited by three elements; hence, there can be multiple critical points. This problem will be explained more later. First, though, here is how the divide-and-conquer method works.

The divide-and-conquer method works in the opposite direction, starting with the outside and subdividing it, as in the following figures. The outside figure is a modified dual representation, which is shown in figure 15.34. This

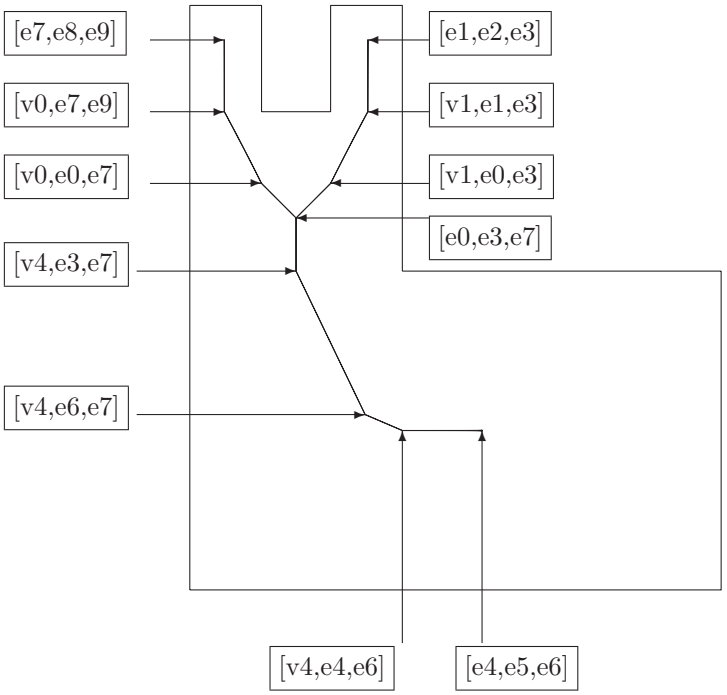


Figure 15.19: Labelled critical points of MA of simple figure (from [128])

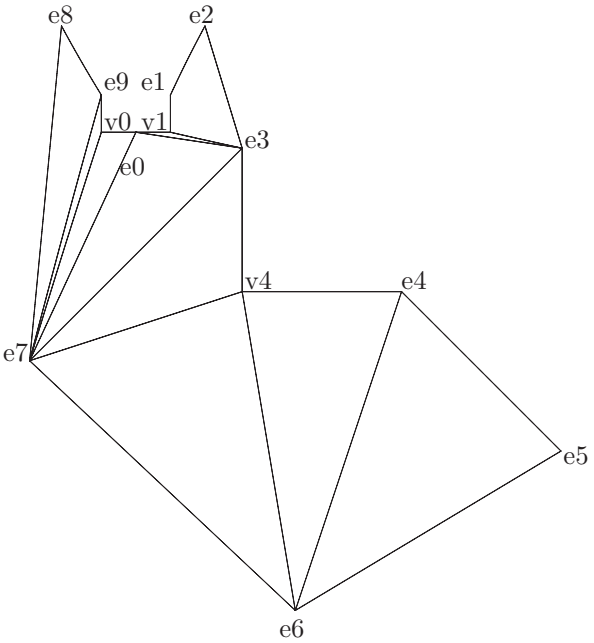


Figure 15.20: Dual space nodes for the MA of the simple figure (from [128])

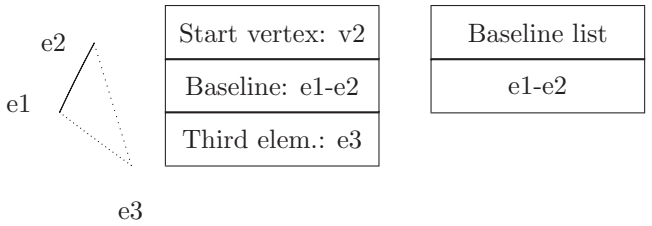


Figure 15.21: Dual space node 1 for the medial axis (from [128])

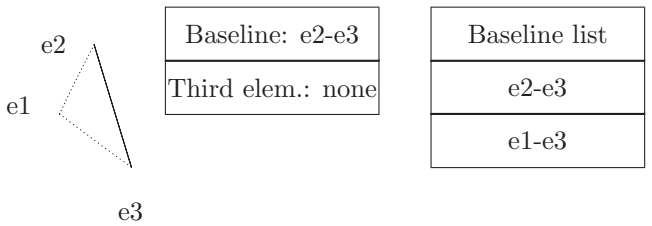


Figure 15.22: Dual space node 2 for the medial axis (from [128])

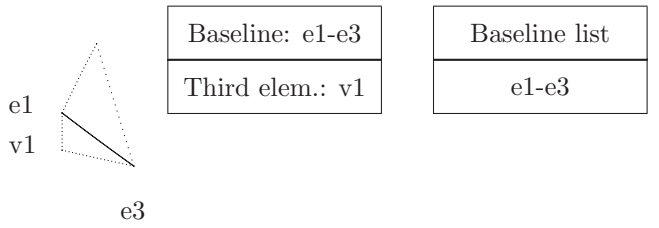


Figure 15.23: Dual space node 3 for the medial axis (from [128])

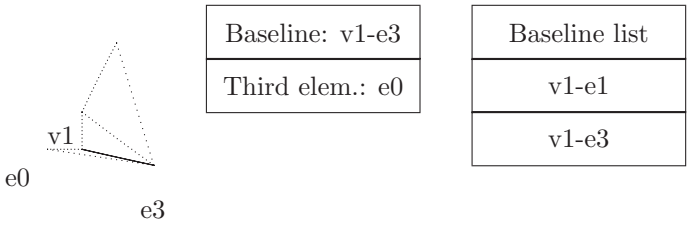


Figure 15.24: Dual space node 4 for the medial axis (from [128])

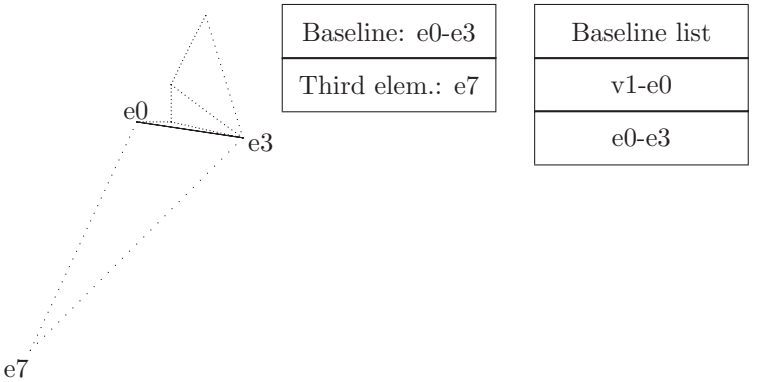


Figure 15.25: Dual space node 5 for the medial axis (from [128])

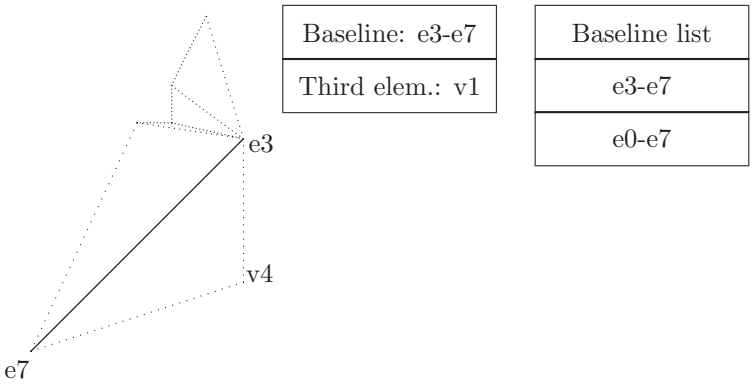


Figure 15.26: Dual space node 6 for the medial axis (from [128])

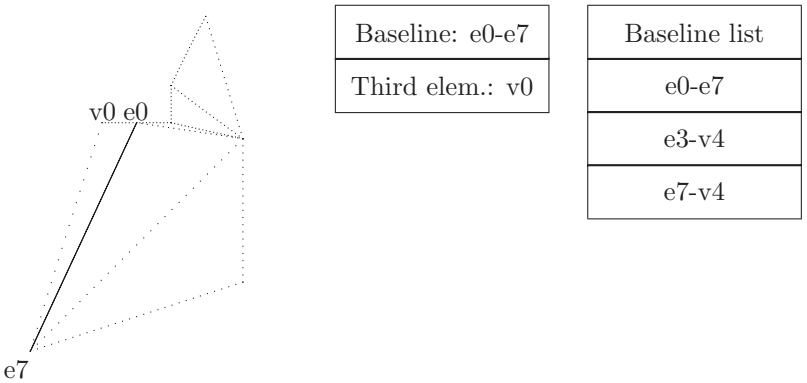
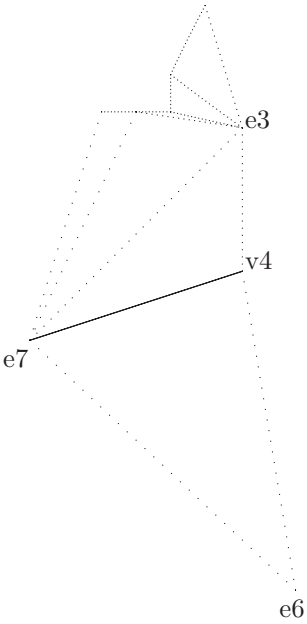


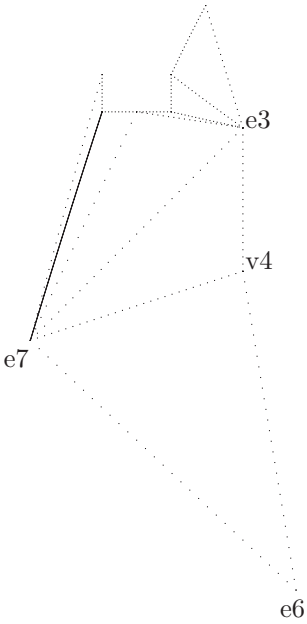
Figure 15.27: Dual space node 7 for the medial axis (from [128])



Baseline: e7-e4
Third elem.: e6

Baseline list
e3-e4
e7-v4
e0-v0
e7-v0

Figure 15.28: Dual space node 8 for the medial axis (from [128])



Baseline: e7-e4
Third elem.: e6

Baseline list
e0-v0
e7-v0
e6-v4
e6-e7

Figure 15.29: Dual space node 9 for the medial axis (from [128])

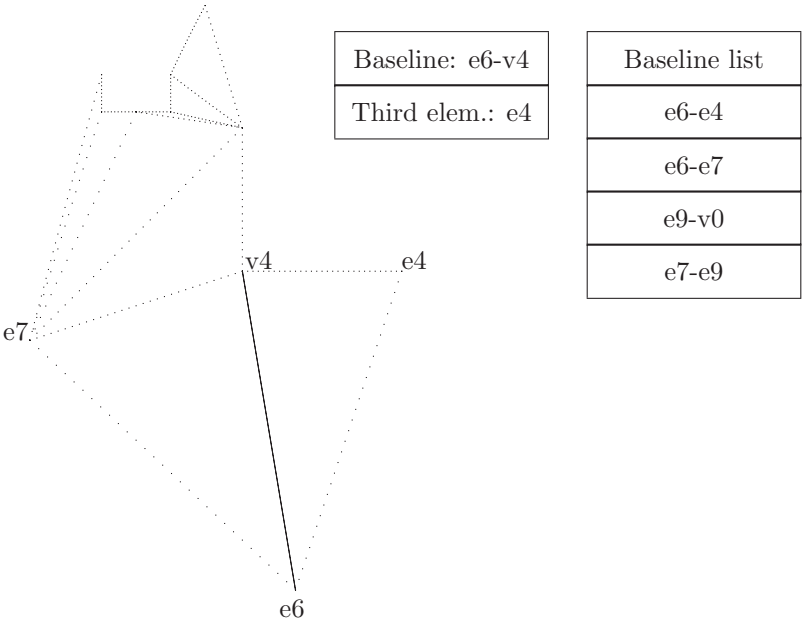
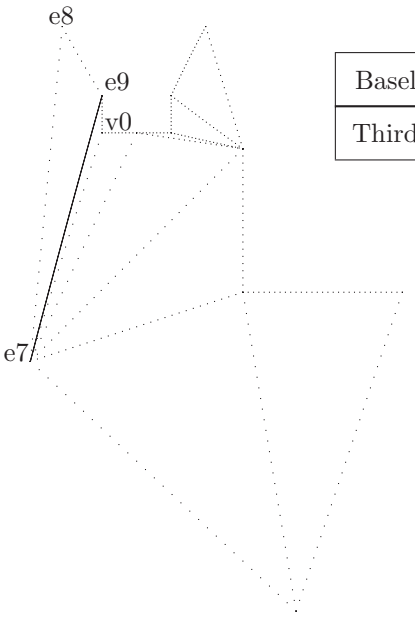


Figure 15.30: Dual space node 10 for the medial axis (from [128])



Baseline: e7-e9
Third elem.: e8

Baseline list
e9-v0
e7-e9
e4-v4
e4-e6

Figure 15.31: Dual space node 11 for the medial axis (from [128])

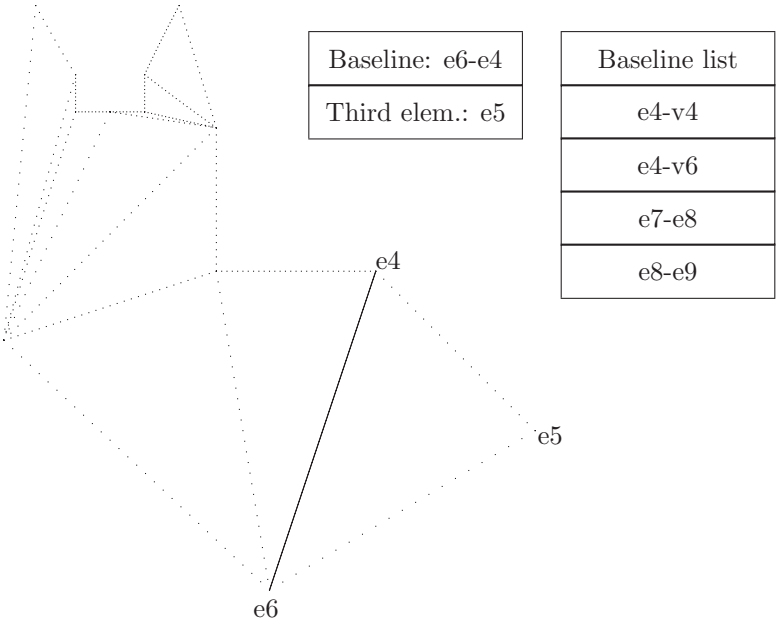


Figure 15.32: Dual space node 12 for the medial axis (from [128])

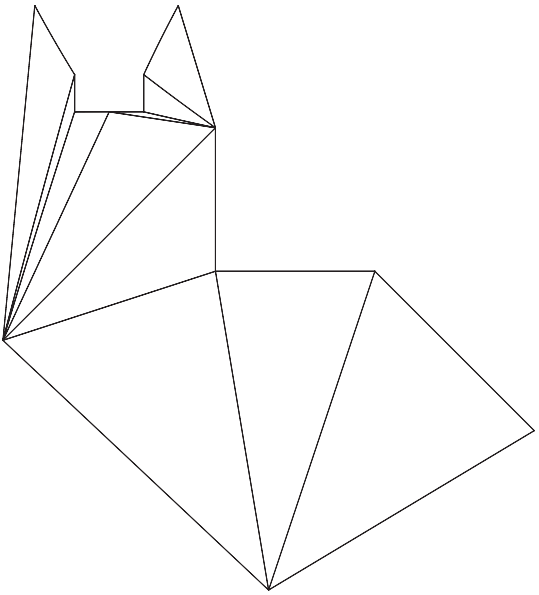


Figure 15.33: Dual space nodes for the medial axis (from [128])

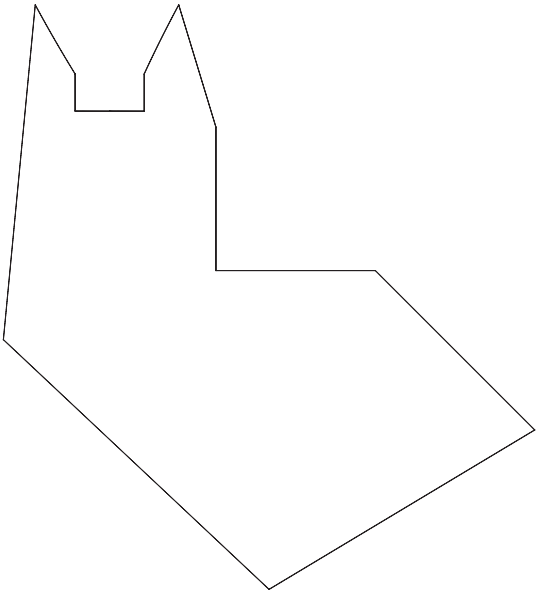


Figure 15.34: Modified dual for subdivision (from [128])

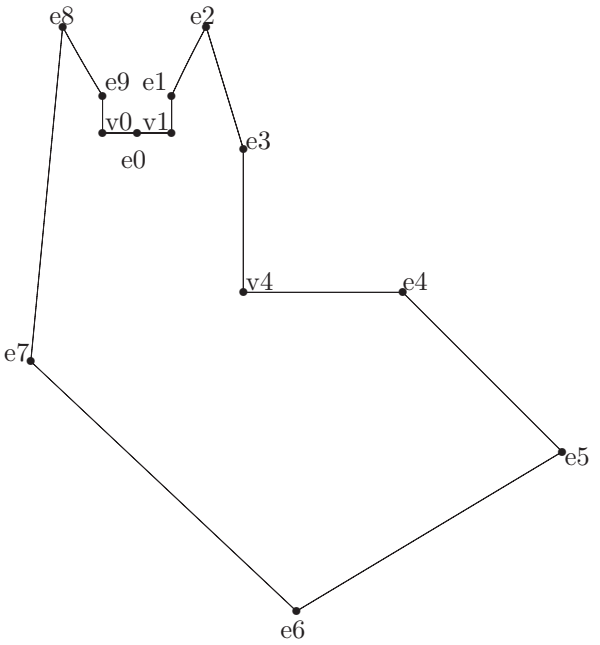
modified dual has all elements in the normal dual, plus extra vertices corresponding to the other elements needed for the MAT calculation. In the example, this means that the concave vertices are inserted into the figure as well.

The labelled modified dual is shown in figure 15.35. The modified dual is produced from the dual object, which defines the exterior of the union of Delaunay nodes. All bases of Delaunay nodes are defined at the beginning and put into a list for processing. The bases of the Delaunay nodes are the edges of the modified dual. The edges are ordered, with edges adjacent to a concave vertex appearing first.

All edges in the modified dual correspond to bases for Delaunay nodes. The edges with vertices corresponding to vertices in the original figure are put first in the list.

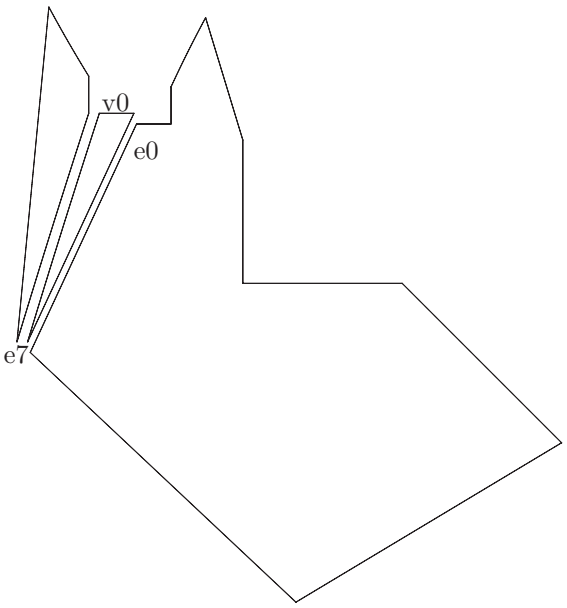
The pair v0–e0 forms a Delaunay triangle with edge e7. Separating the node from the dual graph causes the dual graph to separate into two pieces: one containing vertices v0, e7, e8, e9 and the other e0, v1, e1, e2, e3, v4, e4, e5, e6, e7 (figure 15.36).

The pair v0–e9 forms a Delaunay triangle with edge e7. Note that because of the split, it is only necessary to check edges e7 and e8 to see whether they build Delaunay triangles. None of the other edges can build a triangle because



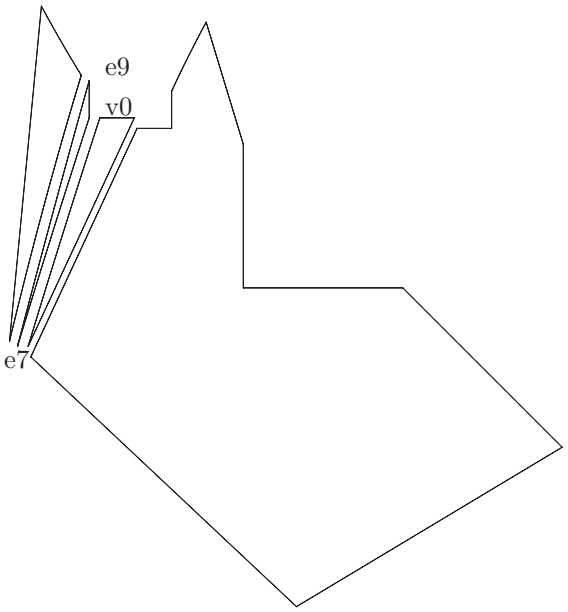
Baseline list
e0-v0
e9-v0
e0-v1
e1-v1
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.35: Modified dual for subdivision (from [128])



Baseline list
e0-v0
e9-v0
e0-v1
e1-v1
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.36: First node detached (from [128])



Baseline list
e9-v0
e0-v1
e1-v1
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.37: Second node detached (from [128])

they are all in the other part of the object; i.e., they are further away than edges e8 and e9 (figure 15.37).

The pair v1–e0 is examined next; it forms a Delaunay triangle with edge e3. Splitting the structures gives a new small piece with vertices v1,e1,e2,e3 and a larger piece with vertices e0,e3,v4,e4,e5,e6,e7. (figure 15.38).

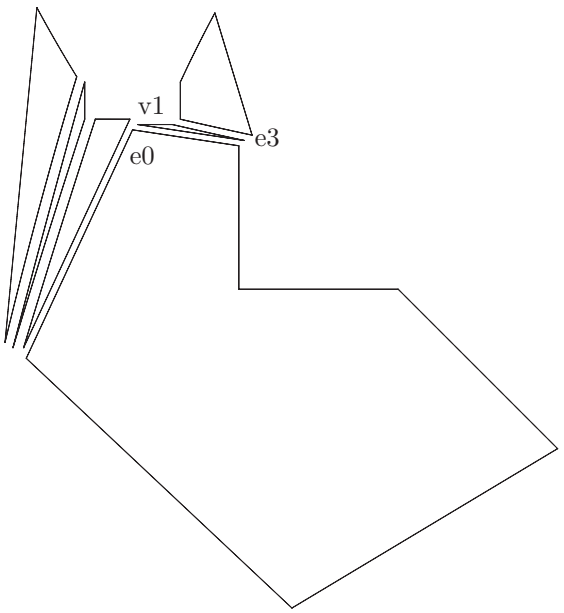
The pair v1–e1 forms a Delaunay triangle with edge e3. As before, only the subset e2, e3 needs be checked because all other vertices are further away (figure 15.39).

The pair v4–e3 forms a Delaunay triangle with edge e7, separating the structure into two new pieces: e0,e3,e7 and v4,e4,e5,e6,e7 (figure 15.40).

The pair v4–e4 forms a Delaunay node with e6 creating two new substructures: v4, e6, e7 and e4, e5, e6 (figure 15.41).

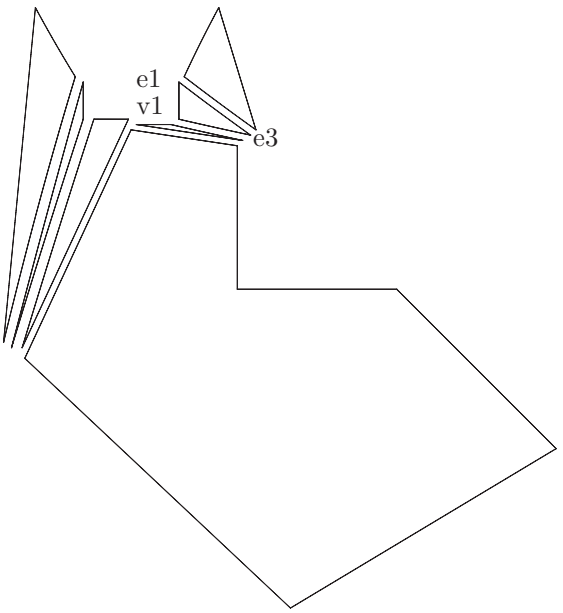
All other baselines are already parts of existing nodes.

These two-dimensional examples are intended to show one aspect of the divide-and-conquer algorithm, that is, how the structure decomposes. Every time a node is extracted the original shape is split, and if the object is not joined elsewhere, simpler subnodes are created. This is illustrated in fig-



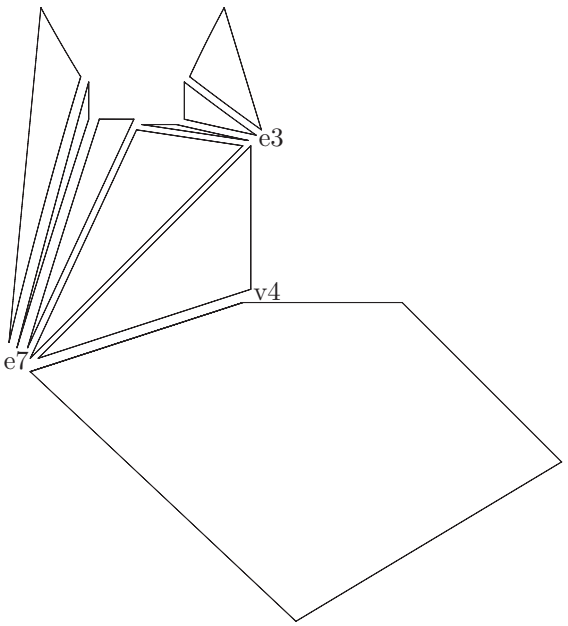
Baseline list
e0-v1
e1-v1
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.38: Third node separated (from [128])



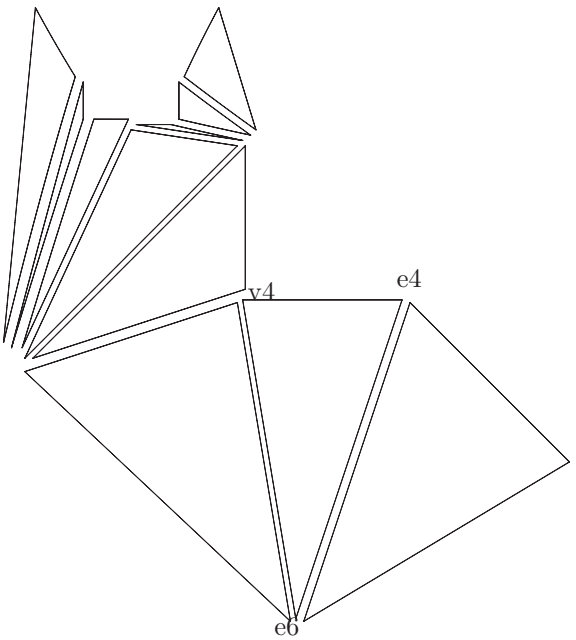
Baseline list
e1-v1
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.39: Fourth node separated (from [128])



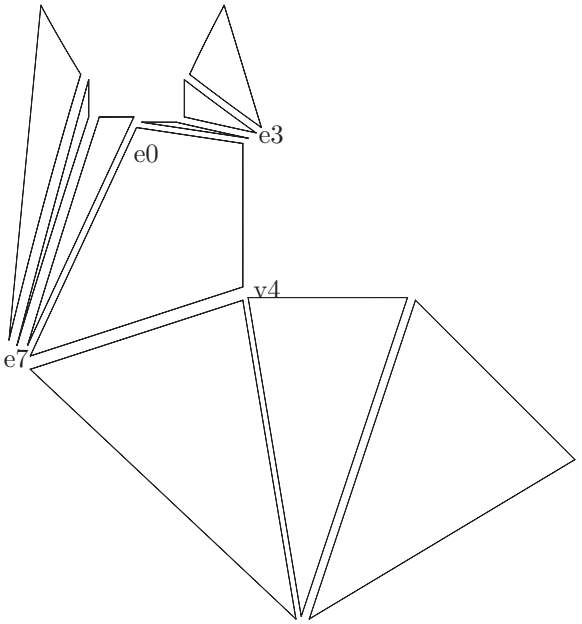
Baseline list
e3-v4
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.40: Fifth node separated (from [128])



Baseline list
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.41: Sixth node separated (from [128])



Baseline list
e4-v4
e1-e2
e2-e3
e4-e5
e5-e6
e6-e7
e7-e8
e8-e9

Figure 15.42: Node decomposition with non-triangular node

ures 15.37, 15.38, and 15.40. In each sub-node, there are fewer elements and hence it is quicker to identify the third elements to form the triangles. The method works in three dimensions as well, decomposing the modified dual into three-dimensional nodes. In their simplest form these three-dimensional nodes are tetrahedrae, but this is not always the case. With a slight change in the geometry of the two-dimensional object in figure 15.17, there would be a node defined by four elements, e1, e3, e7, and v4, as in figure 15.42.

This also works in three dimensions. A simple corner block and its MAT are shown in figure 15.43, and the Delaunay node decomposition is shown in figure 15.44. Note the nodes corresponding to the MAT vertices marked 1, 2, and 3 on the MAT in figure 15.43. These are pyramids with a square base in the node structure. The corners of the bases correspond to the vertex, two adjacent concave edges, and the common face of these edges.

No special structures are needed for this process because the modified dual and the nodes can all be represented with the standard structures, except that some way is needed to associate the maximally inscribed sphere centre points and radii with the nodes to which they correspond.

The generalised node structure is needed for computing the medial axis transforms of general objects. There are, however, two awkward steps in this procedure, namely creating the modified dual and then creating the nodes to be extracted.

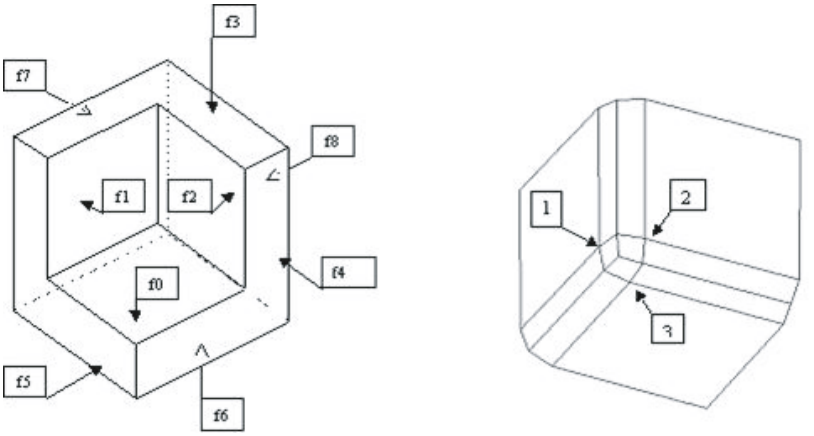


Figure 15.43: Corner block and MAT (from [128])

Figure 15.44: Corner block element node decomposition (from [128])

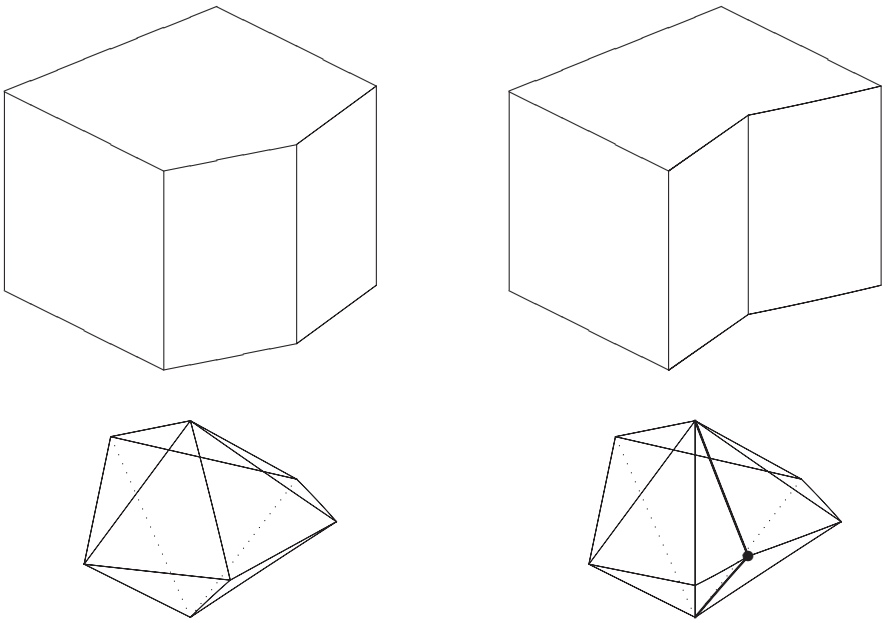


Figure 15.45: Objects, dual and modified dual

15.6.1 Creating the modified dual

The modified dual of an object is the 'standard' dual together with extra vertices and edges added according to geometric rules. To illustrate this, consider the object shown in figure 15.45.

Topologically, both objects at the top of the figure are the same, and so the normal topological dual, shown at the bottom left of the figure, is the same for both. However, the modified dual of the figure on the top right breaks the dual edge corresponding to the concave edge by inserting a vertex. This vertex is connected to the opposite vertices of the facet to produce the modified dual.

It is easiest to show some figures to show how this works. The case with a single concave edge has already been shown, above.

On the left of figure 15.46, there is a vertex with three concave edges, e0, e1, and e2. The normal dual is shown in the middle, and the modified dual is shown on the right. The vertex v0 corresponds to a face in the normal dual.

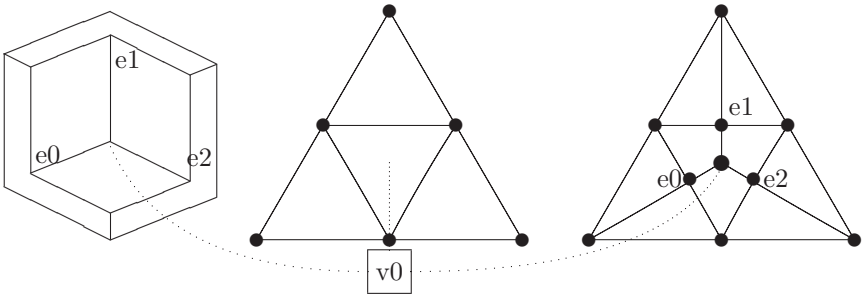


Figure 15.46: Concave vertex and modified dual

In the modified dual, however, the three concave edges are split and a new vertex corresponding to vertex v0 is inserted. This new vertex is connected to the three concave edge vertices, which are also connected to the vertices representing faces, as shown. This creates three four-sided faces that become non-tetrahedral Delaunay nodes with a fourth element.

In figure 15.47, the vertex v0 has two concave edges. The vertex is not a concave vertex because any sphere touching it touches either the base face or the two concave edges. The vertex makes, therefore, no contribution to the MAT.

At the end of the modified dual process, every element that is needed to create the MAT corresponds to a vertex of the modified dual.

The difficulty with creating the modified dual is to arrange the topology corresponding to vertices with mixed concave and convex edges. Simple empirical summaries of some of the cases are shown in figures 15.48, 15.49, and 15.50,

The summary figures are intended to show some examples of different concave edge configurations at a vertex. However, the configurations cannot be used as a stable basis for determining modified dual topology, which is why the last two elements of the right-hand column of figure 15.50 are blank. To determine the connection, it is necessary to perform a geometric test to determine the internal angle of the elements.

15.6.2 Extracting nodes

Once the modified dual has been created, the MAT creation method involves processing each face of the modified dual and searching for vertices that, together with the vertices around the face, forms a Delaunay node.

In the multiple start point algorithm, the search for the fourth element

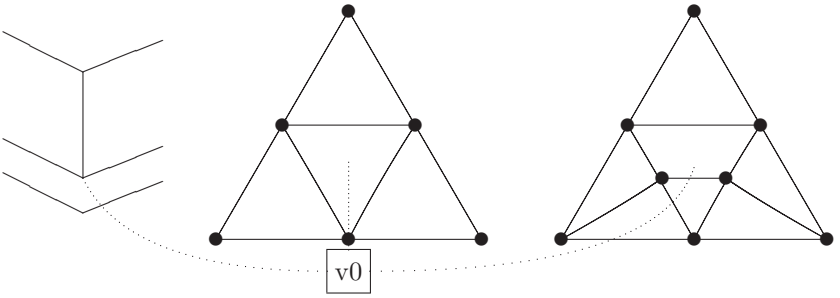


Figure 15.47: Vertex with two concave edges, dual and modified dual

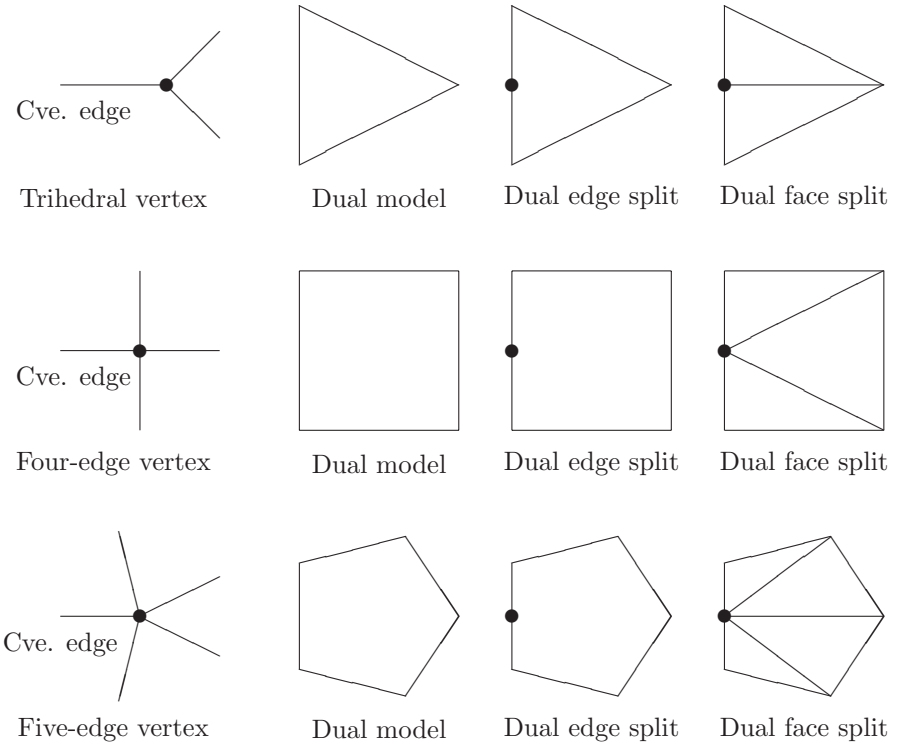


Figure 15.48: Single concave edge modification summary

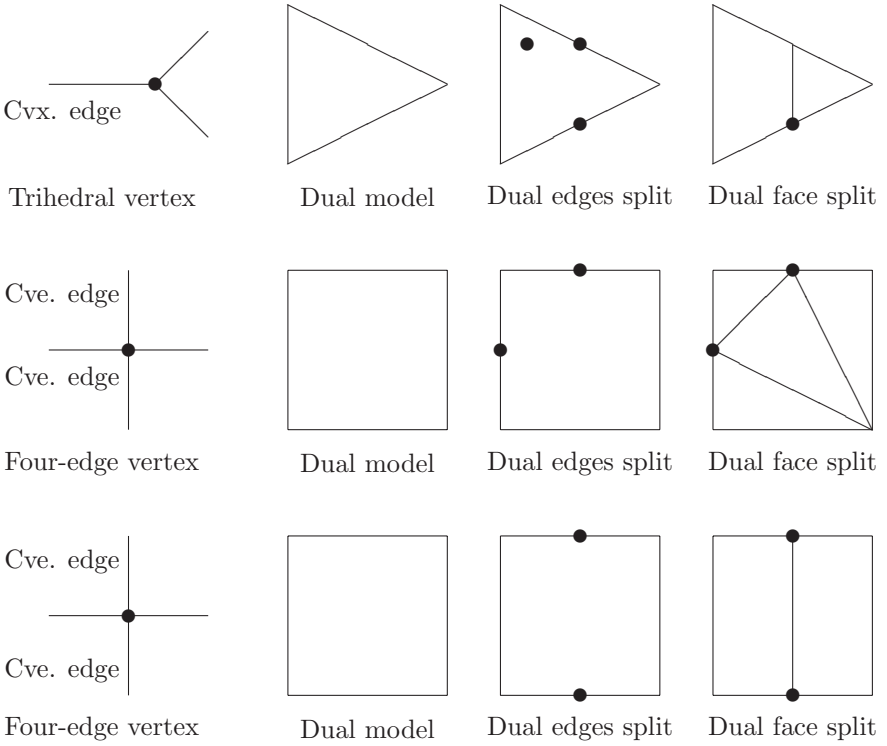


Figure 15.49: Double concave edge modification summary

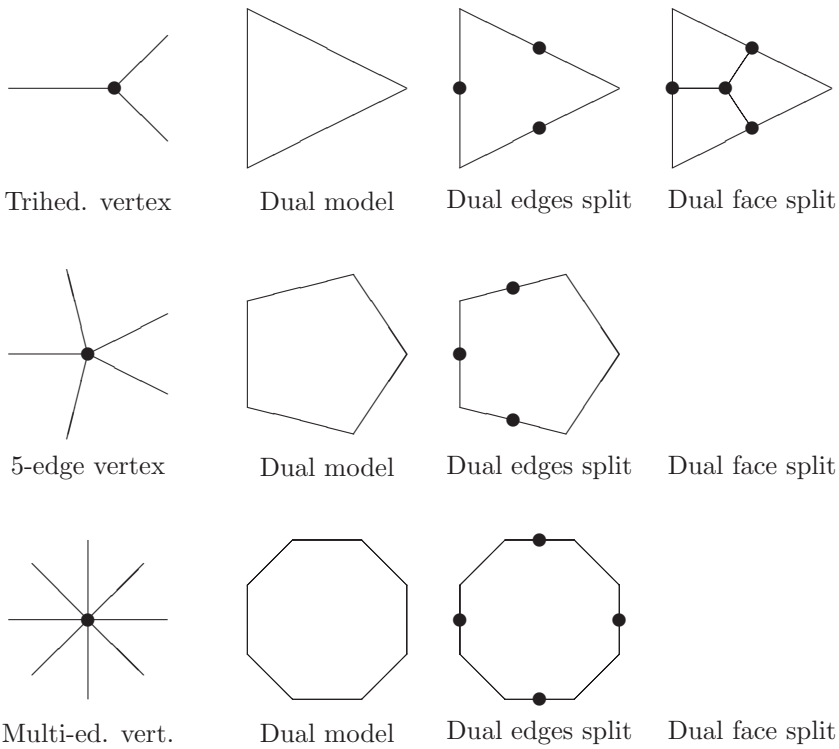


Figure 15.50: Multiple concave edge modification summary

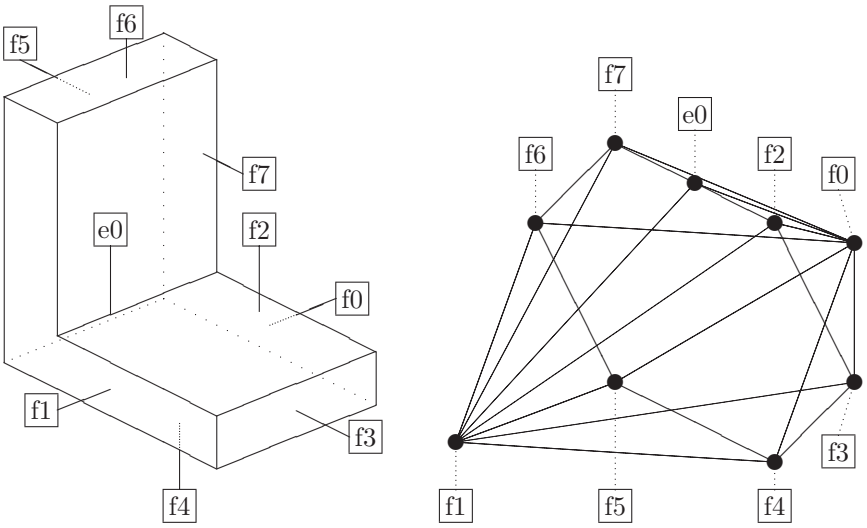


Figure 15.51: L-block and modified dual

to complete a tetrahedron is done using all available elements. In the divide-and-conquer algorithm, only the elements in the same local piece as the base face need be examined; hence, the search becomes simpler as the modified dual is broken into pieces. Note, though, that any shells in the original object have to be handled specially, so that they are included in the search.

The facets around concave edges should be chosen first to form the bases of nodes to be extracted. Choosing the facet (e0,f1,f2) first, as shown in figure 15.52, f4 is found as the vertex that completes a tetrahedral node.

To extract the facet, the base face is separated from the modified dual, making a flat object with two sides, and leaving a face to be partitioned in the modified dual, as shown at the top of figure 15.53. A new vertex corresponding to face f4 is inserted in the underside of the base face. This face is completed by adding edges connecting the three corner vertices to the new vertex. For the modified dual, a similar process is performed, but there is an existing edge between vertex f1 and vertex f4, so this is sliced. See the bottom of figure 15.53.

The process is then repeated for the next facet (e0, f0, f2). There is a slight difference, though, in the modified dual now because of the extracted node (figure 15.54). This has an effect when the second node is extracted.

To extract the facet, the base face is separated from the modified dual making a flat object with two sides. However, there is a neighbouring face that also belongs to the node, so this, too, is extracted (figure 15.55). The

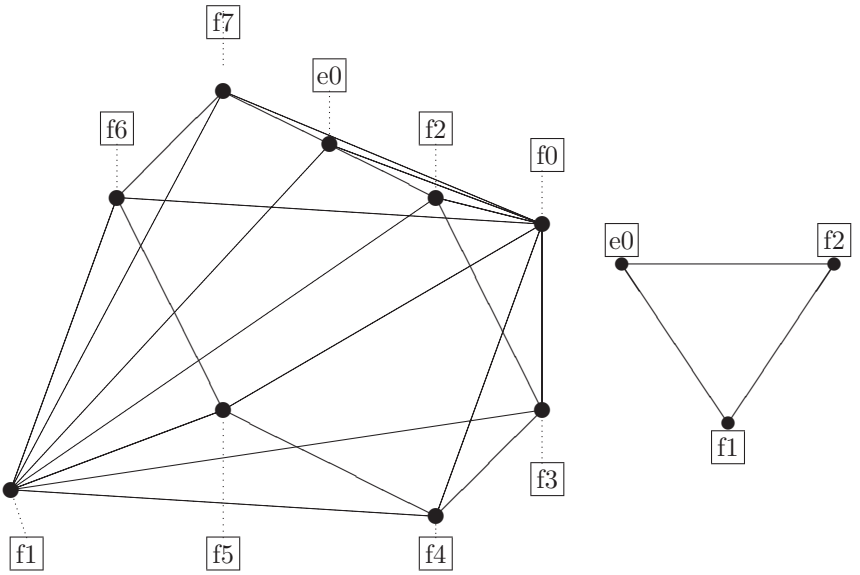


Figure 15.52: L-block, first node extraction

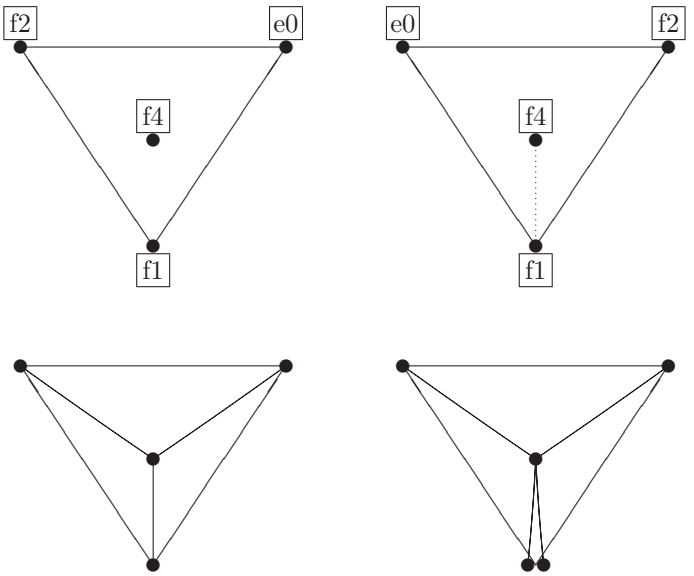


Figure 15.53: L-block, completing the first node

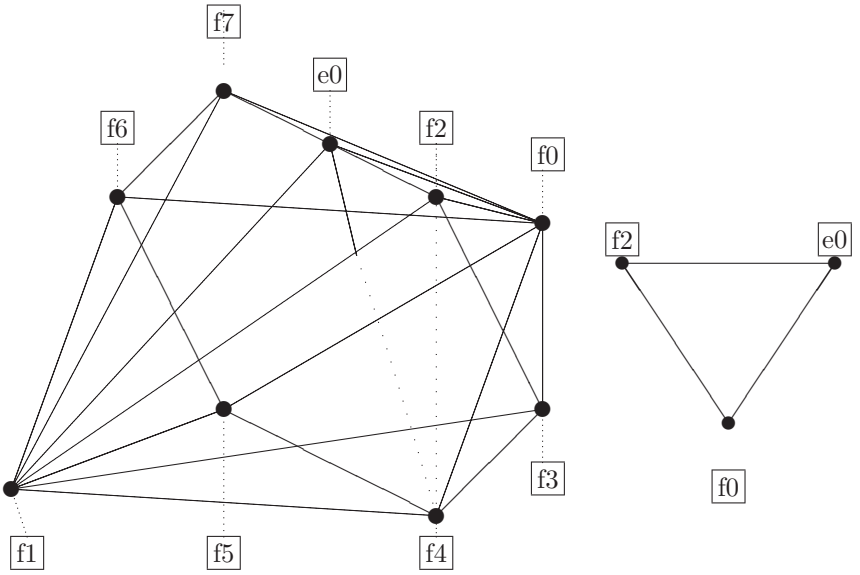


Figure 15.54: L-block, second node extraction

fourth node f4 is now part of the face set that has been removed, so all that is necessary to complete the topology of the node is to add an edge between the f0 vertex and the f4 vertex. For the modified dual, though, there is an existing edge that, when split, separates the modified dual into two pieces. See the bottom of figure 15.55.

The modified dual after the second node has been extracted is shown in figure 15.56. Vertices f0, f1, and f4 are duplicated in the two pieces. Vertices f2 and f3 belong only to the right-hand piece; vertices e0, f5, f6, and f7 belong only to the left-hand piece.

The next node to be extracted has base (e0,f1,f7), and the vertex which makes a tetrahedron is vertex f5. After this, the node with base (e0,f0,f7) is extracted and there is again a split giving the two parts shown in figure 15.57.

The three remaining compound pieces all separate about the middle line, creating the final division. The sequence is shown in full in [110].

The general procedure can be summarised as follows:

1. Put the base face in the list to be processed.
2. If there are no faces in the list to be processed, go to step 6.
3. Pick a face from the list to be processed and remove it from the list.
4. Look at the faces bounding the current face, and add any new faces bounded by vertices in the set of node vertices to the list to be processed.

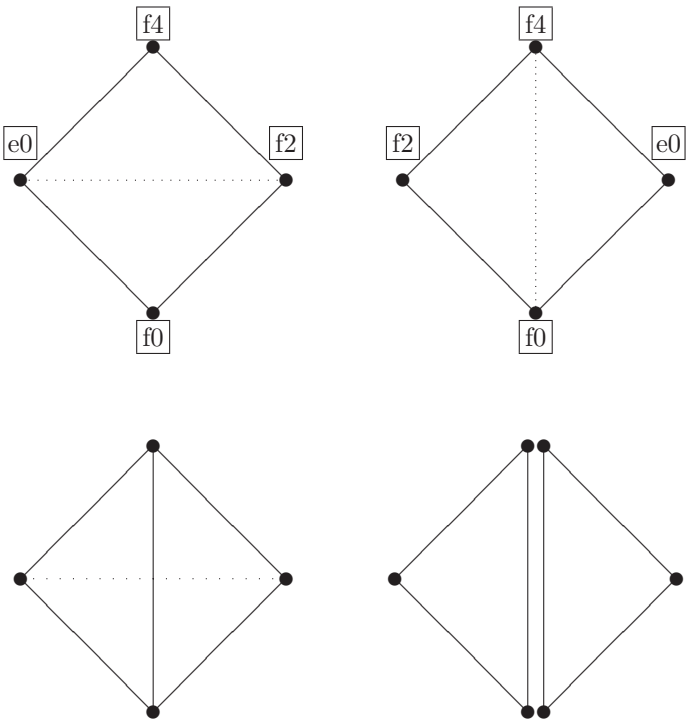


Figure 15.55: L-block, completing the second node

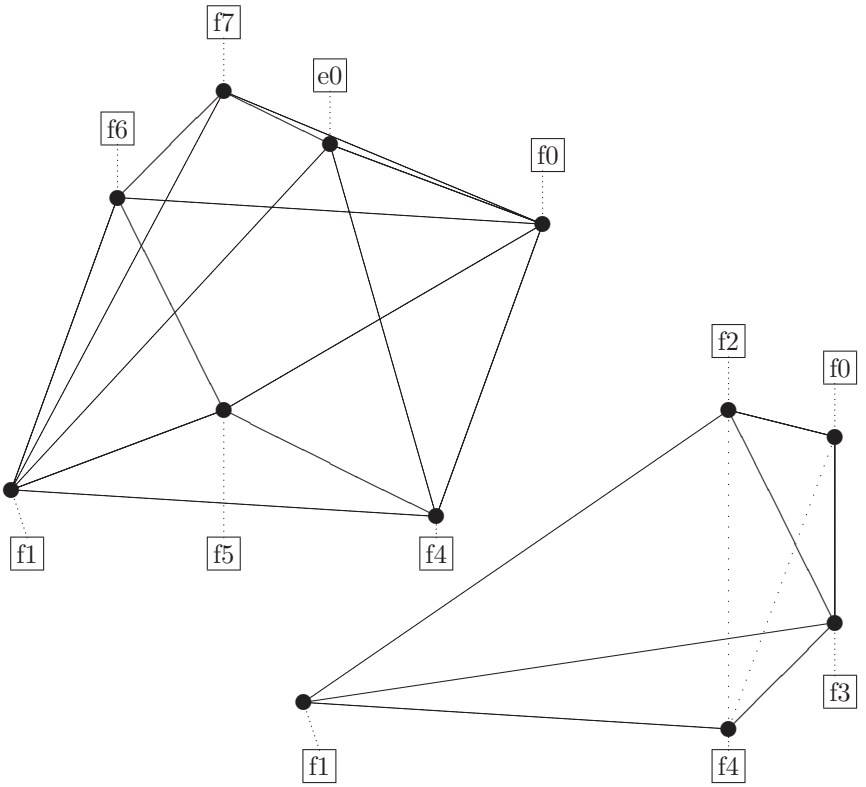


Figure 15.56: Modified dual after second node extraction

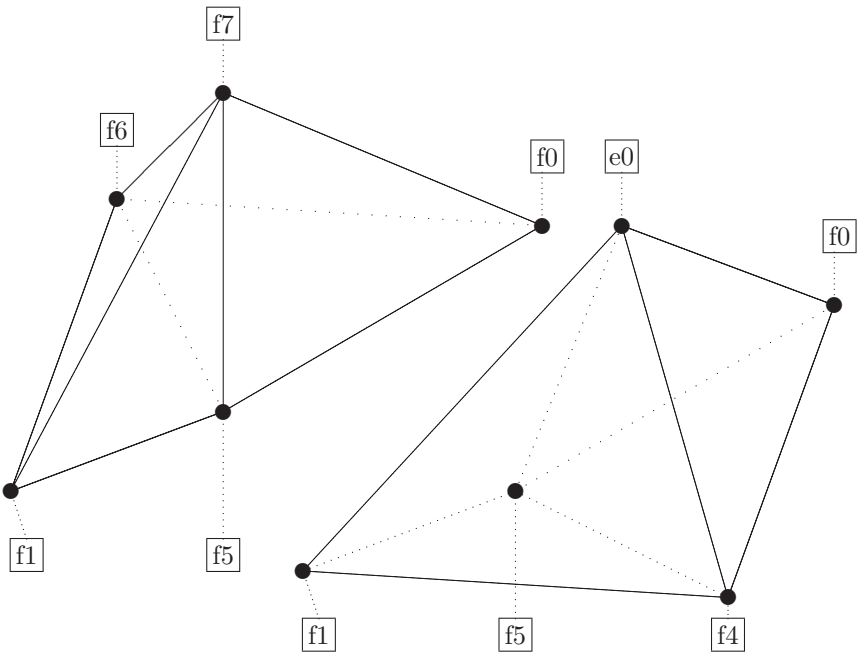


Figure 15.57: First subpiece after third and fourth node extraction

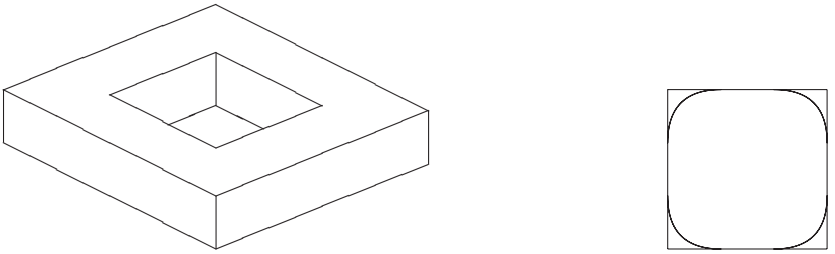


Figure 15.58: Square cross-section torus and MAT

5. Repeat from step 2.
6. For all connected faces, identify the boundary edges (edges separating the node face set from the rest of the piece).
7. If there are no boundary edges then stop (the piece is a complete node).
8. Slice all boundary edges.
9. Separate the base face and connected faces from the modified dual.
10. Add any unconnected vertices from the node vertex set as isolated vertices in the back face of the base face.
11. Divide up the back face, and connect the hanging vertices to the other vertices, creating matching edges in the modified dual (slicing existing edges).

The main problem with the node extraction phase is to get the node topology correct. As the nodes are part of dual space, their geometry is not really defined, the geometry shown here is only there for visualisation. This means that it is not possible to use the geometry of the nodes themselves for tests.

As an illustration of weird nodes, consider the example of a sort-of square torus shape, as shown in figure 15.58

Looking in detail at one corner, as shown in figure 15.59, the decomposition procedure gives four groups of nodes, as shown in figure 15.60.

The topology of the two large nodes is strange. There is one four-sided face, with corner vertices corresponding to the four faces in each group. There are two triangular faces corresponding to the start and end of the concave edge, e0. The remaining two triangular faces are bounded by the vertices corresponding to e0, f0, and f1, and one is bounded by f0, f1 together with either f3 or f5.

After the base face and attached faces (the two faces corresponding to the vertices of e0) have been extracted, you get a figure with the topology shown

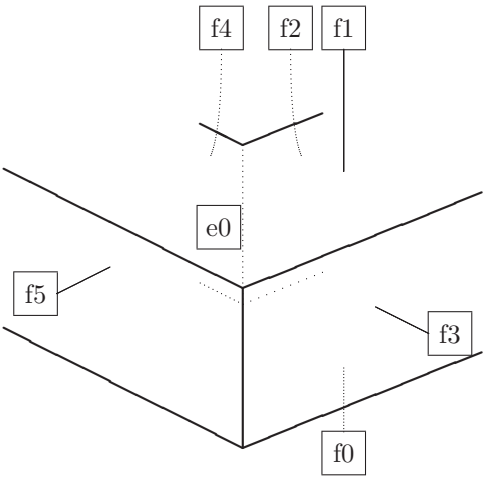


Figure 15.59: Square cross-section torus corner

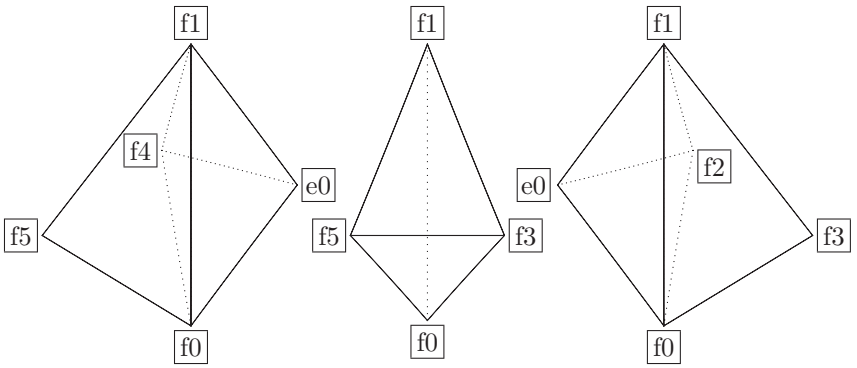


Figure 15.60: Square cross-section torus MAT corner node group

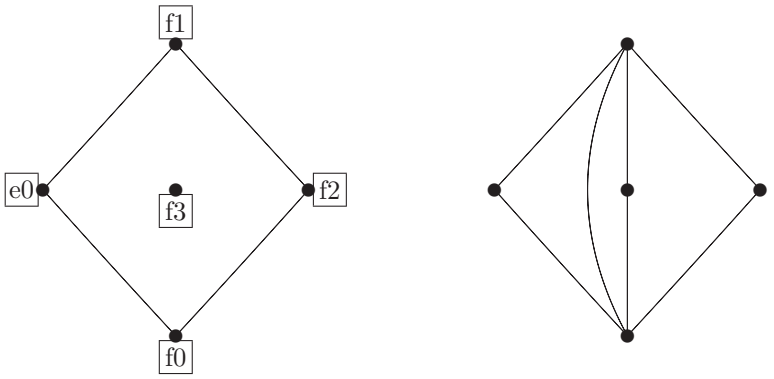


Figure 15.61: Back of existing face set and partition

on the left in figure 15.61. Connecting up the nodes gives the topological result shown on the right.

Determining where to put edges needs an analysis of how the vertex elements constrain the maximal spheres. In fact, each facet in the node corresponds to a curve of the MAT, so the corner elements need to leave one degree of freedom.

15.7 The negative MAT

The MAT creation methods described above can also be used to create the MAT of the exterior of the object, called here the "negative MAT". In fact, there is a slight trick, because the MAT is always applied to the interior of the object. However, if the object is negated, then what appears to be a object is, in fact, an object-shaped cavity in a universe full of material.

It is easiest, again, first to explain this in two dimensions (figure 15.62).

There is also the consideration that multiple objects have to be handled in the same way as multiple shell objects in the positive MAT.

The Delaunay node structure of the negative MAT is usually more complex than those of the positive MAT, but the principle is the same. The added complexity comes because the base nodes are often square, instead of triangular. Another different notion for the negative MAT is that there are infinite Delaunay nodes. In the positive MAT, every Delaunay node is bounded, but for the negative MAT, they can extend infinitely. There is, in this case, a notional completion node at infinity that is the same node for every infinite Delaunay node.

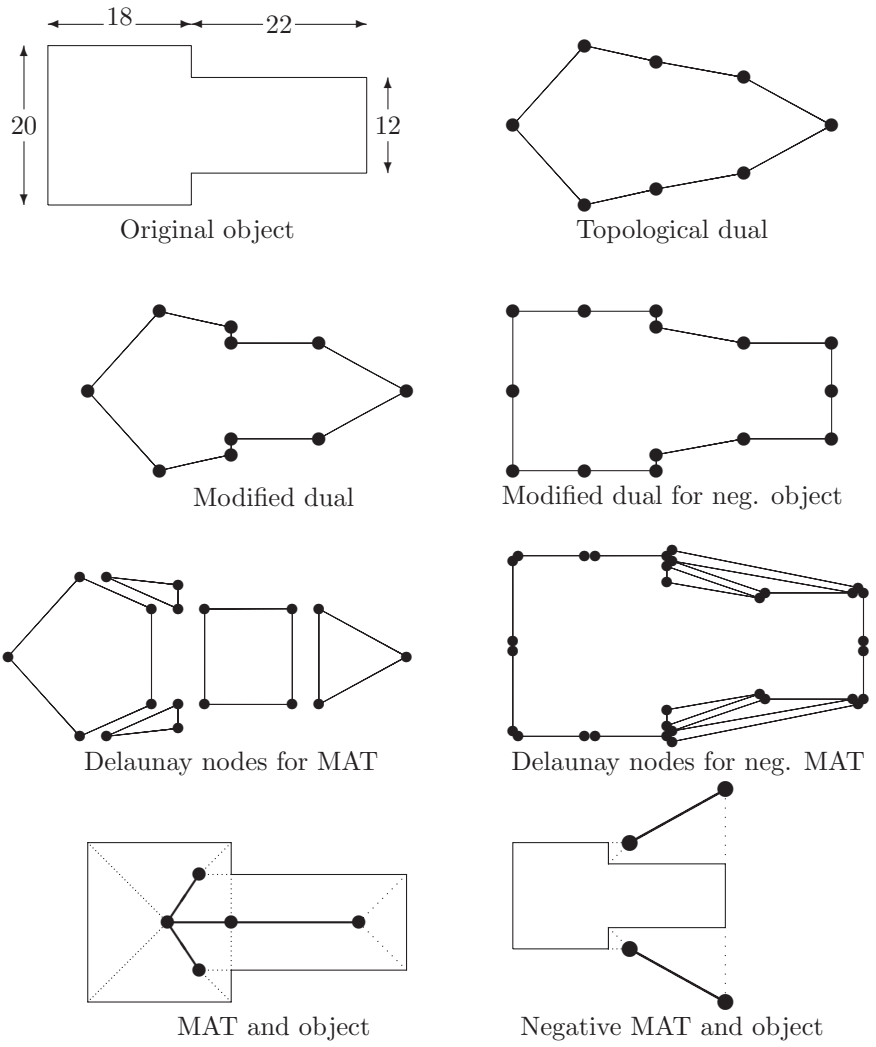


Figure 15.62: MAT and node structures (from[109])

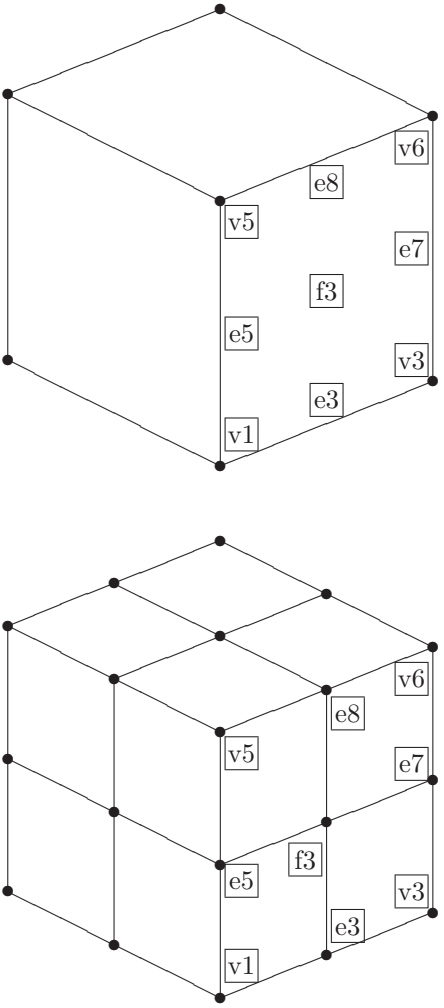


Figure 15.63: Cube and modified dual for negative MAT

For a cube, the object and modified dual are shown in figure 15.63, with one face labelled to show the relationship. Because the cube is convex, all of the 24 base faces are the bases of infinite nodes. For an object with at least one concave edge in the non-negated state, or multiple bodies, there will be a mixture of bounded nodes and infinite nodes. Otherwise, the MAT calculation method works in exactly the same way as for the positive MAT. The negative MAT has uses for toolpath and motion planning. Note, though, that if there are several separate objects, the negative MAT will have to cope with cavities. Each cavity is a separate object in normal space, but in negative space becomes a cavity in the universe of material. The notion of all nodes in a connected piece also has to cope with the cavities.

15.8 Subdividing and offsetting

One effect of having a correct Delaunay node decomposition of the modified dual is that it is very easy to do offsetting.

The basic procedure for offsetting is:

1. Pick a node.
2. If the radius of the maximal sphere of the node is greater than the offset, join it to all its neighbours to which it is not already joined.
3. Repeat from 1 until no more nodes are left.
4. Dual the resulting object.
5. Insert offset geometry.

This can be illustrated using the left-hand side of figure 15.62. The node division is shown in figure 15.64, labelled for convenience, together with the radii of the maximal spheres associated with the nodes.

The recombination procedure for offsetting is shown in figure 15.65. From the list of all nodes, it is possible to see that the critical radii are 4, 6, and 10, when the topology of the offset object changes.

The top row shows the 'active' nodes, that is, the nodes that have a radius greater than the offset distance. The middle row shows the joined nodes, and the bottom row shows the result of dualling the joined nodes and reassigning geometry. The first column shows what happens when the offset distance is between 0 and 4, with an offset of 2 for the final image. The middle column shows what happens when the offset is between 4 and 6, with an offset of 4 for the bottom image. The third column shows what happens with an offset between 6 and 10. The missing column is the boring one, which is where the offset is greater than 10, in which case, no nodes are active, there is no object to dual, and the result is nothing, but I will leave visualisation of that to your imagination.

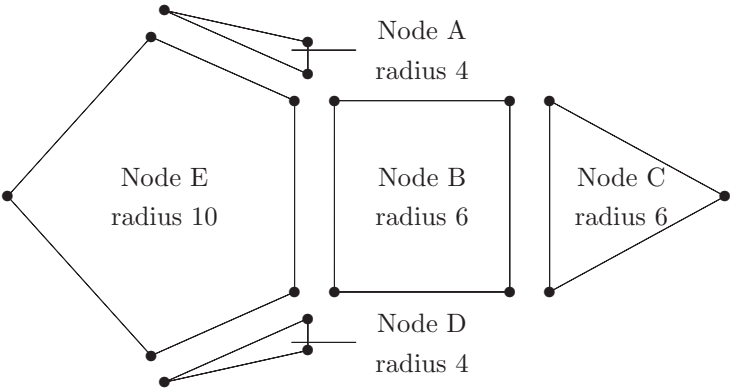


Figure 15.64: Delaunay nodes for recombination (from [109])

In the two-dimensional dual, every vertex will be replaced by an edge and every edge by a vertex. If the vertex corresponds to a concave vertex in the original object, then the edge that results from the final dual process will be part of a circle; otherwise, the edge will be straight. If the offset distance is 0, then the vertices corresponding to concave vertices in the original object will correspond to degenerate, zero radius circular edges in the final result and so need to be removed.

As some nodes may disappear, it is also possible that the joined nodes, and hence the final result, form several connected pieces. This can be checked for relatively easily in a post-processing step.

The radii of the maximal spheres for the simple figure shown in figure 15.17 are 3.375, 6.75, 10, 15, and 15.35894. Offsetting by these values gives the contours shown in figure 15.66.

It should come as no surprise that you can do the same thing with the negative MAT. One difference, though, is that there is no upper limit to the offset value, because some nodes are implicitly infinite.

Doing the same exercise as before with different offsets gives the results in figure 15.68

If the offset is between 0 and 4, then the offset method is as shown in the first column. If the offset is between 4 and 62.5, then the offset is as shown in the second column, with a slightly simpler topology. If the offset is 62.5 or greater, then the offset is shown in the third column (the figure is not to scale). Note that the topology of the joined node set is the same as the convex hull of the object, except that every edge between two convex vertices is split.

For applications, there may be interest in knowing the list of critical radii. For example, for milling, the list of radii from the negative MAT indicates limitations on the set of tools that can be used to mill a shape.

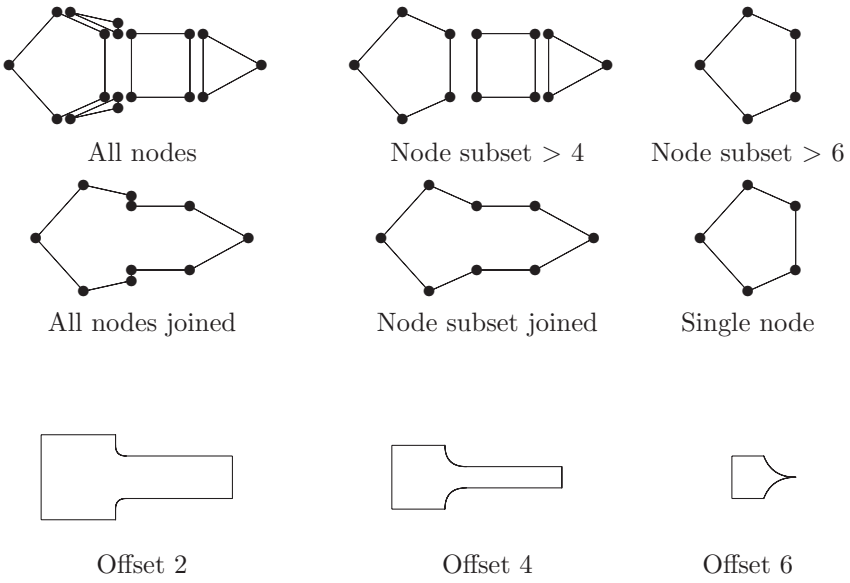


Figure 15.65: MAT and node structures (from [109])

Subdivision is another variant on this game with the Delaunay nodes. For subdivision there is no offsetting, and so all nodes are active, but some joins will not be done. In two dimensions the concave vertices and in three dimensions the concave edges act as blockers for stopping join operations. Consider the nodes of the simple figure from earlier in this chapter, shown in figure 15.69.

Essentially, it is forbidden to join edges where one of the end vertices corresponds to a concave vertex in the original figure. For the node set in figure 15.69, with the concave vertices marked, this gives the node sets in figure 15.70, where the joined edges are shown dotted.

To get back to normal space, it is necessary to dual the nodes and replace the geometry. If this is done, you get the result in figure 15.71 – almost. Here it is necessary to hope that the reader only scans lightly the text and figures and does not notice the problem. A more rigorous examination reveals two problems, or potential problems. The first is that the lower left-hand node in figure 15.70 is triangular and gives rise to a rectangular piece in figure 15.71. Just above this triangular node in figure 15.70 there is what appears to be a five-sided figure, which gives rise to a six-sided part.

Looking closer at the upper of the two nodes, as shown in figure 15.72, it is evident that the top horizontal edge is, in fact, two edges, and hence the node gives rise to a six-sided figure.

The lower triangular node is more difficult to explain. To begin with, it is

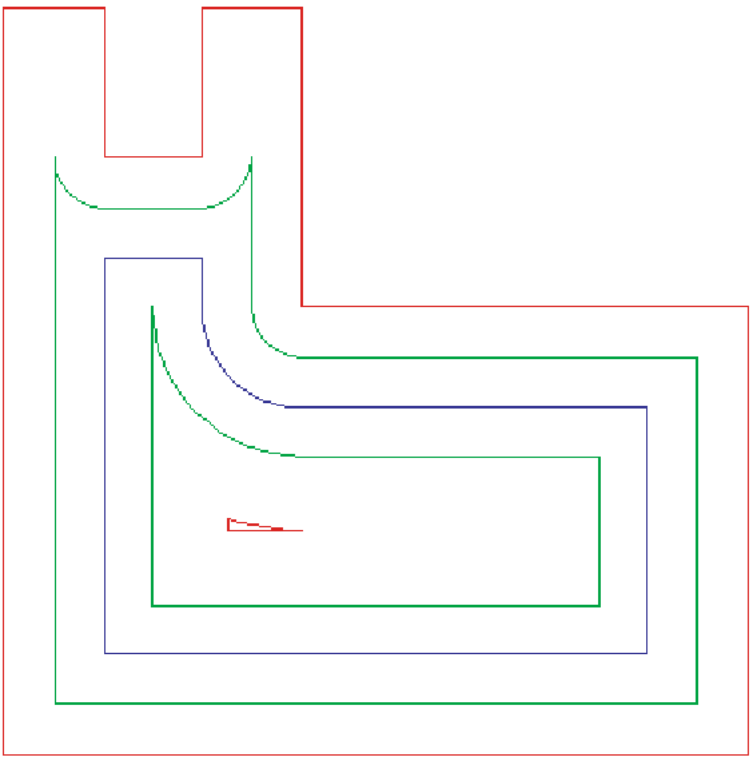


Figure 15.66: Simple figure and offsets

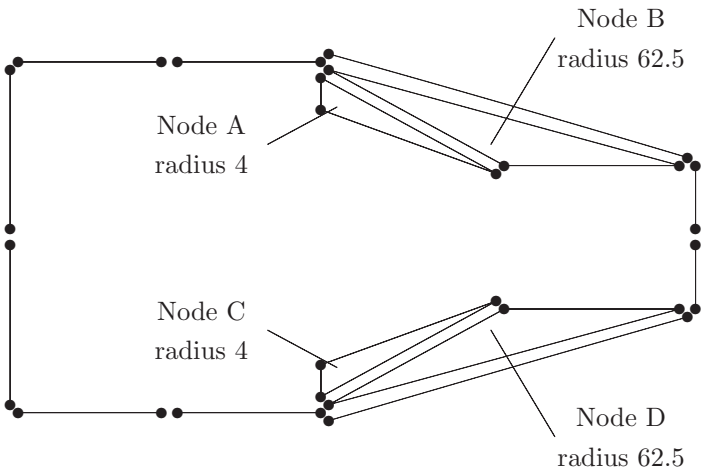


Figure 15.67: Negative MAT Delaunay nodes for recombination (from [109])

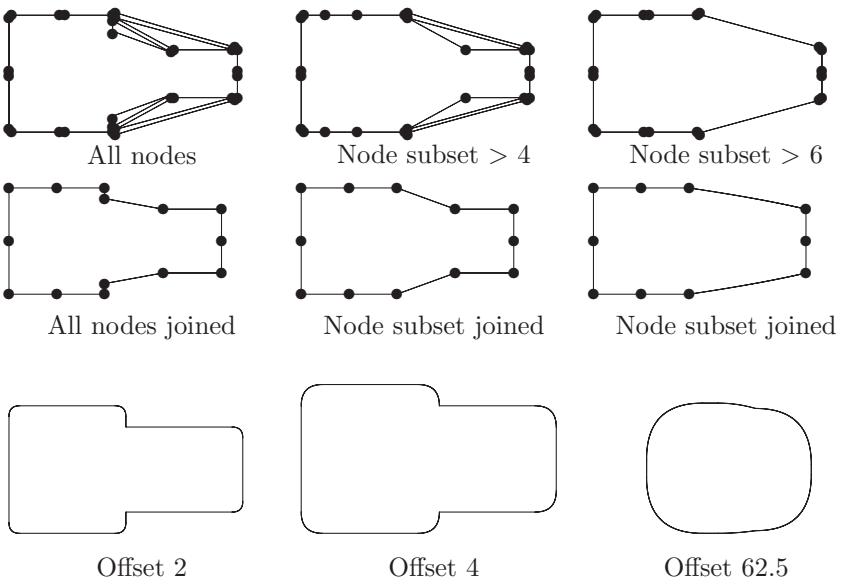


Figure 15.68: MAT and node structures (from [109])

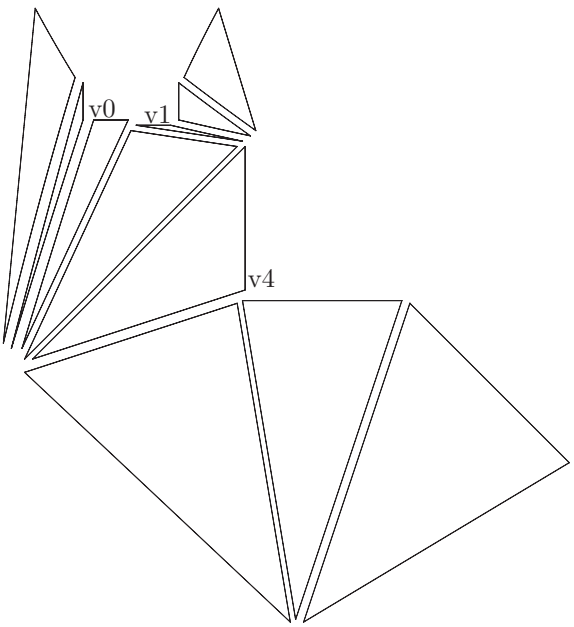


Figure 15.69: Node set from simple figure

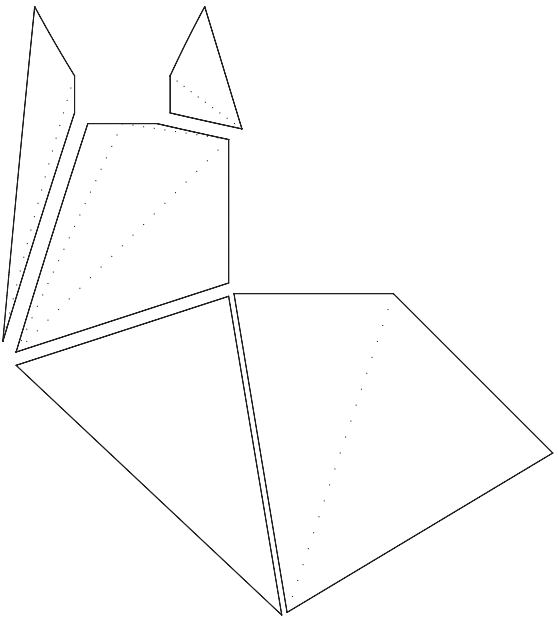


Figure 15.70: Node set from simple figure

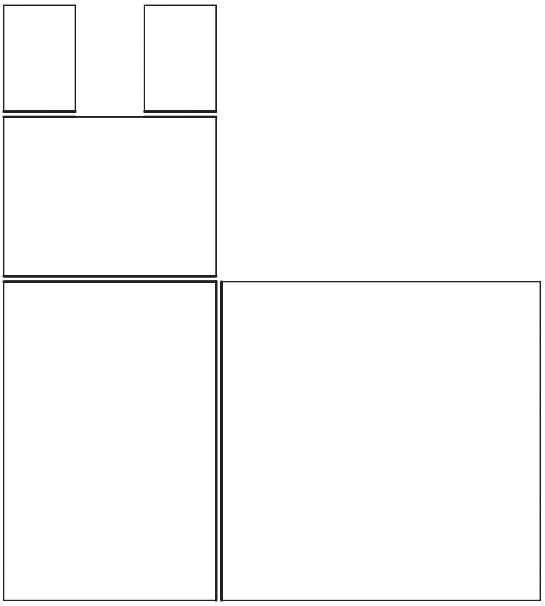


Figure 15.71: Dualled subdivided set

necessary to make an observation. As explained earlier, the vertices of the Delaunay nodes correspond to edges or vertices, in two dimensions, and the edges correspond to vertices. In figure 15.73 there are two vertices corresponding to edges e6 and e7. The edge running between these two vertices corresponds to their common vertex v7 in figure 15.17. The edges connecting the vertices corresponding to concave vertices in the original figure are different. They represent the projections of the concave vertices onto the other element, in the case of figure 15.73, the projections of v4 onto v6 and onto v7. The concave vertex corresponds to, therefore, an edge connecting the concave vertex of the original figure to the projected vertex. In figure 15.73, the dual vertex corresponding to v4 in figure 15.17 corresponds to an edge connecting two projections to the original vertex v4. This would mean that there is a single edge with a kink in it. As this book is not about kinky things, the vertex is split into two, as on the right of figure 15.73. This means that the zero-length edge corresponds to vertex v4, and the two coincident vertices correspond to the edges between v4 and its projections onto e6 and onto e7.

In three dimensions, the same sort of joining takes place, but with concave edges as the blockers. It is not so simple as with the two-dimensional case because it is not enough just to exclude joins between facets with vertices corresponding to concave edges. Maximal spheres are not allowed to roll around the edges but can roll along them, hence, an extra small geometric

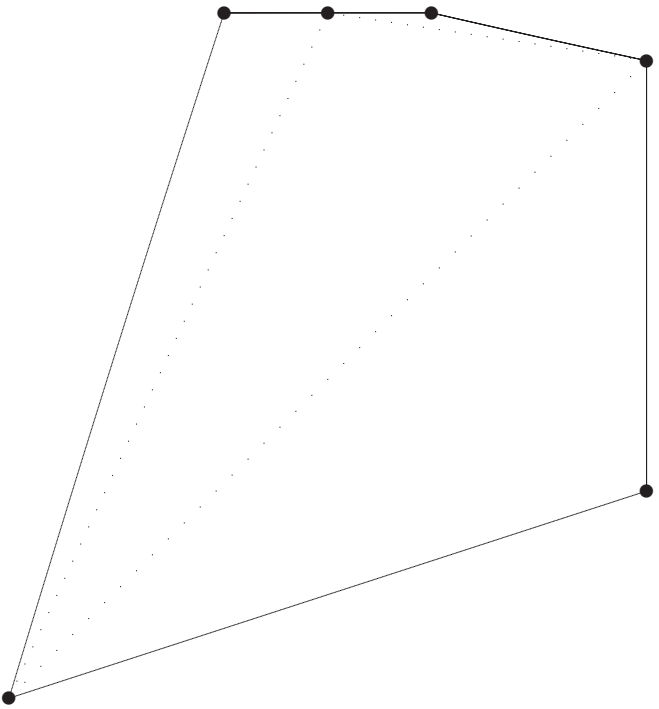


Figure 15.72: Upper hexahedral node

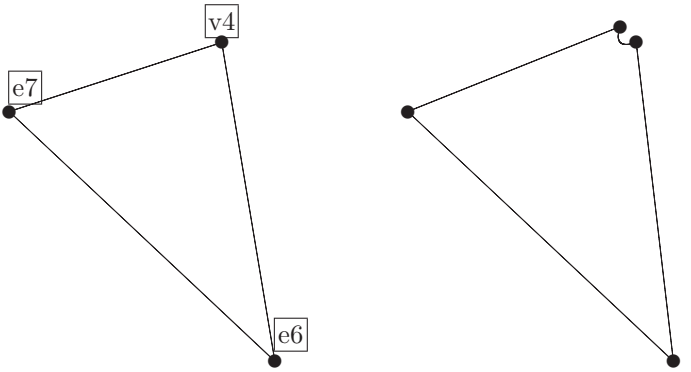


Figure 15.73: Triangular node and modification

check for this before accepting or excluding a join.

15.9 MAT with real geometry?

So, the MAT of a polygonal object is not good enough for you? The ideal would be to use the real geometry of the object rather than a polygonal approximation for the MAT calculation. This is preferable geometrically, but also practically, because the MAT calculation is heavily dependent on the number of model elements used for the calculation. It should be faster, therefore, to calculate the MAT from a real geometric model. The problem is, though, the calculation of the critical points becomes much more difficult.

Renner and Stroud [106] describe a method for geometric refinement of the MAT. Once the MAT has been calculated, the curves of the MAT should be equidistant from three elements and the surfaces from two elements, ignoring the degenerate cases. As these elements are known for each curve and surface, it is relatively easy to calculate a series of points along the entities and to 'relax' these so that they are equidistant from the elements. The correct geometry is then calculated by fitting a curve or a surface to these refined point positions.

Chapter 16

Miscellaneous aspects of modelling

16.1 Geometric tolerances

Computers do not work with exact values, so it is necessary to use tolerances when comparing values. Hans-Ulrich Pfeifer identified several of these very clearly for the GPM volume module, and what is written here is based on that:

- Machine tolerance (TOLABS)
- Length tolerance (TOLEPS)
- Angle tolerance (TOLANG)
- Geometric tolerance (TOLGEO)
- Relative tolerance (TOLREL)
- Polynomial tolerance (TOLPOL)

The machine tolerance is the smallest value that can be distinguished from 1. Pfeifer defined it as *epsilon*, where $1 + epsilon > 1$. This is determined by the precision of the machine.

The length tolerance is the shortest distance distinguishing two points.

The angle tolerance is the smallest angle that can be determined.

The geometric tolerance is the tolerance used for calculation of intersections and geometric properties.

The relative tolerance is used with respect to the sizes being compared.

The polynomial tolerance is used when working with polynomials for geometry.

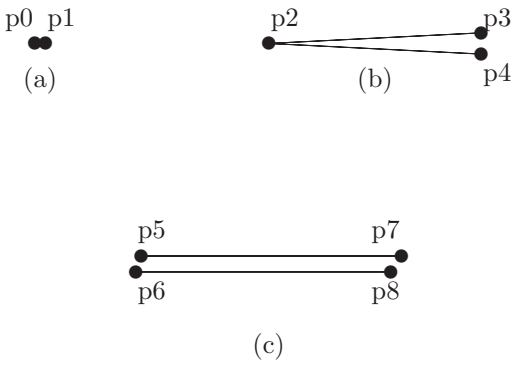


Figure 16.1: Triangular node and modification

When using the tolerances, it is important to be consistent. The distance between two points should not be compared with TOLABS, the angle between two lines should not be compared with TOLEPS, and so on.

Figure 16.1 is intended to illustrate this.

The distance between the two points, p0 and p1, in figure 16.1a is compared with TOLEPS. If less than this value, then the points are considered to be at the same position. In figure 16.1b, there are two lines at an angle to each other. If you compare the angle between them, then the tolerance TOLANG should be used. However, if the lines have the same length, then it would also be possible to compare the positions of points p3 and p4 with TOLEPS. If they are closer than this, then the angle is implicitly zero. Finally, in figure 16.1c, the lengths of two lines can be compared with TOLREL, or the positions of point pairs p5, p6 and p7, p8 could be compared with TOLEPS to check for coincidence. The choice of which tolerance to use depends on how the elements are compared.

The last point to mention is that tolerances need not stay the same. This is why it is important to distinguish between ways to compare entities. In some cases, it may be necessary to be more lax about, say, geometry calculation tolerances. If the use of tolerances is mixed, then changing one tolerance may lead to strange effects elsewhere.

Finally note, again, the need to record tolerances in disc files. This is part of the environment in which models have been created and hence is needed if there are errors in recreation.

Although the use of these is fairly obvious, it may be necessary to adjust the values to make work more efficient. For example, surface–surface intersections are sometimes calculated by marching and finding matching points on surfaces. If the tolerances are small, then this process may be very slow.

16.2 Debugging

Debugging is an important part of modeller development as sometimes it is hard to know what is going on. Many languages now have debuggers that can be used to examine the status of programs during operation. However, this is not always a convenient method of gathering information. In general it can be useful to have both debuggers for examining the values of variables and single values together with another mechanism for outputting more complete information such as model listings.

The BUILD system used a global set of debug words, the individual bits of which could be set and sensed to turn on debugging. This means that you could write something like:

```
if ( debug1 BIT 7 ) print(body);
```

This could be useful because a lot of bits are available for turning on debugging in individual routines. In BUILD, the disadvantage, though, was keeping track of the use of individual bits. A list of uses had to be kept and maintained to avoid having debug information from different sources.

There are other solutions to this problem, as well. The choice of debugging method is, of course, a matter for the software developer to decide. This brief account is only meant to point out why the mechanism is needed and to indicate some possible solutions. The choice is whether to build into the code or to rely on language debuggers. A language debugger is a useful tool for examining individual values in a flexible way. If, however, datastructure elements are to be examined, then access to printing or even drawing functions may be preferable.

16.3 Error handling

In general, modelling systems are large and are usually developed by several people who use each others' code. Error messages should be as clear as possible. If the error occurs in a low-level system function, then this is probably not where the real error occurred, just where it was found. It is important, then, to be able to trace the error back through the calling code. This is done through the use of return codes and traceback functions.

16.3.1 Error messages

This, and the next two sections, are related. If a special, or an unexpected condition arises then this should be reported as an error which can be identified properly. An error message such as "Internal error E3" illustrates the type of anonymous error message that should be avoided. In general, error messages should identify the procedure in which they occur and provide more

information about the condition that has arisen. It may be possible to use codes and check the error code with the program source code, but this is not always possible for users of modelling systems.

In modelling functions, various special conditions can be identified and checked for. One of these is when a NULL pointer is given. This should be checked for rather than trying to access an invalid memory location causing a drastic failure. Dividing by zero or trying to find the square root of a negative number are two other common problems in geometric routines.

There are two types of error. The first is a hard error, a real error that should halt an operation. The other is a warning, a soft error that indicates that some unexpected condition has arisen. It may be possible to use default values or get around these and carry on. In service routines, warning messages may have a useful annoyance value that will cause the higher level routines using these to be corrected. If there is no warning, then the special condition may not be noticed.

One philosophy for the actual messages is to group the actual text messages together into one file so that language can be changed. If the messages are spread out throughout the code, this is more difficult. This reason, though, is probably no longer very important, but might be a consideration.

16.3.2 Return codes and tracebacks

The error messages are one element in error handling. Another is how to handle these in the software. One commonly used method is to have return codes from routines. If an operation is successful, then the return code may be zero. Warnings might be, say, positive numbers and hard errors negative numbers. This implies that these return codes should be checked after every call. This was done in the GPM volume module and was tedious. The benefit, though, was that the calling sequence could be traced back by reporting non-zero codes from the calling routines.

The BUILD system used a sophisticated method that exploited the traceback facilities of Algol68c. The names of calling routines were recorded as part of the run-time system, so when an error occurred, it was possible to use the system traceback rather than having to do this manually.

Tracing error conditions back to their origins is an important part of debugging. This may be partly superseded by the use of debuggers, though.

Chapter 17

Acknowledgements

It is important to point out that the work described in this book is the product of many years of work while the author was at various institutes. The work involved was done as part of different teams, to whom I would like to express my gratitude for their help and forbearance. Rather than try and name them all individually, risking forgetting some people, I hope they will forgive me for naming them as groups. The groups with whom I have worked have been:

1977 – 1980: The CAD Group at the Cambridge University Computer Laboratory.

1980 – 1985: The Computer-Aids for Production Group at the Royal Institute of Technology, Stockholm, Sweden (and colleagues at VTT, Espoo, Finland with whom I worked on the GPM project)

1986 – 1989: The BUILD Research Group in The Department of Applied Computing and Mathematics at Cranfield Institute of Technology, Cranfield, Bedfordshire, U.K.

1989 – 1996: The Geometric Modelling Laboratory, Computer and Automation Research Institute, Hungarian Academy of Sciences, Hungary

1997 – present: LICP, EPFL, Switzerland

I would also like to thank Graham Jared for considerable help and many useful comments during the years.

Thanks to Mikó Péter and Frank Ganz for help with LaTeX.

Finally I would like to thank my wife, Hildegarde, for her patience and the rest of my family for moral support.

Appendix A

Data definitions

The following list provides possible datastructure and geometry definitions for solid modelling. These definitions are based on existing modelling systems, principally the BUILD system and the GPM volume module, with some extensions. The definitions are described in terms of their fields, pointers, integers, reals, etc. Each of the datastructure definitions defines a pointer type.

Not all of these are necessary, and not all fields need necessarily be included. For example, the FACEGROUP entity may be useful but also adds complexity to the operations. The choice of datastructures is part of basic modeller design and cannot be prescribed. This Appendix is intended to illustrate various options.

A.1 Datastructure definitions

A.1.1 VERTEX

A vertex is a node of the datastructure, lying at a point in space. There are three variants: the first is for a manifold vertex, the second is for the vertex-edge link variant, and the third supports non-manifold vertices using the 'bundle' mechanism of Luo and Lukacs.

Direct pointer and loop-edge link variant

This variant is the commonly used one. Some implementations have a list of edges instead of one to support non-manifold vertices instead of the 'friend' pointer.

edge	EDGE ptr.	Pointer to one edge at the vertex.
loop	LOOP ptr.	Pointer to the loop owning the vertex if there are no attached edges (should be set to NIL if there are any attached edges).

Continued on next page

position	POINT ptr.	Pointer to the geometry of the vertex, i.e., the point containing its coordinates.
next	VERTEX ptr.	Pointer to the next vertex in the 'all vertices in body' list.
friend	VERTEX ptr.	For associating non-manifold vertices.
cogeom	V/SG ptr.	Pointer to a vertex or supplementary geometry entity using the same point.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the vertex.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the vertex.
marker	BITS	Sixteen marker bits for use by the modelling operations.
number	INTEGER	Integer vertex number.
name	NAME ptr.	Pointer to the name of the vertex.

Vertex-edge link variant

This is an alternative to using the loop-edge link edge variant.

link	VE-LINK ptr.	Pointer to one edge link at the vertex.
position	POINT ptr.	Pointer to the geometry of the vertex, i.e., the point containing its coordinates.
next	VERTEX ptr.	Pointer to the next vertex in the 'all vertices in body' list.
friend	VERTEX ptr.	For associating non-manifold vertices.
cogeom	V/SG ptr.	Pointer to a vertex or supplementary geometry entity using the same point.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the vertex.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the vertex.
marker	BITS	Sixteen marker bits for use by the modelling operations.
number	INTEGER	Integer vertex number.
name	NAME ptr.	Pointer to the name of the vertex.

Bundle variant

bundle	BUNDLE ptr.	Pointer to one bundle at the vertex.
position	POINT ptr.	Pointer to the geometry of the vertex, i.e., the point containing its coordinates.
next	VERTEX ptr.	Pointer to the next vertex in the 'all vertices in body' list.
friend	VERTEX ptr.	For associating non-manifold vertices.
cogeom	V/SG ptr.	Pointer to a vertex or supplementary geometry entity using the same point.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the vertex.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the vertex.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer vertex number.
name	NAME ptr.	Pointer to the name of the vertex.

A.1.2 EDGE

An edge is a segment of a curve, running between two vertices. In an Eulerian model (i.e., not a wireframe model), edges lie in two loops, or possibly occur twice in the same loop.

Direct pointer variant

rcw	EDGE ptr.	Pointer to the Right ClockWise edge, i.e., the edge clockwise from this edge around the right loop of the edge.
rcc	EDGE ptr.	Pointer to the Right Counter-Clockwise edge.
lcw	EDGE ptr.	Pointer to the Left ClockWise edge.
lcc	EDGE ptr.	Pointer to the Left Counter-Clockwise edge.
rloop	LOOP ptr.	Pointer to the right loop of the edge.
lloop	LOOP ptr.	Pointer to the left loop of the edge.
start	VERTEX ptr.	Pointer to the start vertex of the edge.
end	VERTEX ptr.	Pointer to the end vertex of the edge.
next	EDGE ptr.	Pointer to the next edge in the 'all edges in body chain'.
prev	EDGE ptr.	Pointer to the previous edge in the 'all edges in body chain'.

Continued on next page

owner	object ptr.	Pointer to the owner object.
curve	CURVE ptr.	Pointer to the geometric definition of the edge.
friend	EDGE ptr.	For associating non-manifold edges.
cogeom	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the same curve.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the edge.
info	INFO.EL ptr.	Pointer to a list of information elements.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer edge number.
name	NAME ptr.	Pointer to the name of the edge.

Loop-edge link variant

The use of loop edge links facilitates loop traversal. There are several variants. This one assumes that non-manifold edges are multiply defined and have links between them. Each edge has only two loop-edge links. Contrast with the star variant and the wedge variant.

rlink	L-E LINK ptr.	Pointer to a LOOP-EDGE link on the right of the edge.
llink	L-E LINK ptr.	Pointer to a LOOP-EDGE link on the left of the edge.
start	VERTEX ptr.	Pointer to the start vertex of the edge.
end	VERTEX ptr.	Pointer to the end vertex of the edge.
next	EDGE ptr.	Pointer to the next edge in the 'all edges in body chain'.
prev	EDGE ptr.	Pointer to the previous edge in the 'all edges in body chain'.
owner	object ptr.	Pointer to a VOLUME, SHELL or WIREFRAME object.
curve	CURVE ptr.	Pointer to the geometric definition of the edge.
friend	EDGE ptr.	For associating non-manifold edges.
cogeom	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the same curve.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.

Continued on next page

group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the edge.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer edge number.
name	NAME ptr.	Pointer to the name of the edge.

Vertex-edge link variant

rloop	LOOP ptr.	Pointer to the right loop of the edge.
lloop	LOOP ptr.	Pointer to the left loop of the edge.
start	V-E LINK ptr.	Pointer to the VERTEX-EDGE link to the start vertex of the edge.
end	V-E LINK ptr.	Pointer to the VERTEX-EDGE link to the end vertex of the edge.
next	EDGE ptr.	Pointer to the next edge in the 'all edges in body chain'.
prev	EDGE ptr.	Pointer to the previous edge in the 'all edges in body chain'.
owner	object ptr.	Pointer to a VOLUME, SHELL or WIRE-FRAME object.
curve	CURVE ptr.	Pointer to the geometric definition of the edge.
friend	EDGE ptr.	For associating non-manifold edges.
cogeom	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the same curve.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the edge.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer edge number.
name	NAME ptr.	Pointer to the name of the edge.

'Star' variant

The star variant is currently popular but there is the inherent ambiguity already mentioned. This can be improved in several ways, if necessary.

link	L-E LINK ptr.	Pointer to one LOOP-EDGE link around the edge.
start	VERTEX ptr.	Pointer to the start vertex of the edge.
end	VERTEX ptr.	Pointer to the end vertex of the edge.
next	EDGE ptr.	Pointer to the next edge in the 'all edges in body chain'.
prev	EDGE ptr.	Pointer to the previous edge in the 'all edges in body chain'.
owner	object ptr.	Pointer to a VOLUME, SHELL or WIRE-FRAME object.
curve	CURVE ptr.	Pointer to the geometric definition of the edge.
friend	EDGE ptr.	For associating non-manifold edges.
cogeom	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the same curve.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the edge.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer edge number.
name	NAME ptr.	Pointer to the name of the edge.

Wedge variant

wedge	WEDGE ptr.	Pointer to one WEDGE at the edge.
start	VERTEX ptr.	Pointer to the start vertex of the edge.
end	VERTEX ptr.	Pointer to the end vertex of the edge.
next	EDGE ptr.	Pointer to the next edge in the 'all edges in body chain'.
prev	EDGE ptr.	Pointer to the previous edge in the 'all edges in body chain'.
owner	object ptr.	Pointer to a VOLUME, SHELL or WIRE-FRAME object.

Continued on next page

curve	CURVE ptr.	Pointer to the geometric definition of the edge.
friend	EDGE ptr.	For associating non-manifold edges.
cogeom	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the same curve.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the edge.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer edge number.
name	NAME ptr.	Pointer to the name of the edge.

A.1.3 LOOP-EDGE LINK

A link between an edge and a loop. The edge-loop link is used to chain all edges around the loop in either clockwise or counter-clockwise order. Edge adjacency is determined via these links. Note, again, that the first variant does not support 'star'-type non-manifold edges, which are handled in another way (see chapter 5). The 'star' variant is commonly used, but there is an inherent ambiguity about how bodies are connected. This can be solved by duplicating edges or using Luo and Lukacs' wedges.

Multiple edge variant

edge	EDGE ptr.	Pointer to the edge owning the link.
loop	LOOP ptr.	Pointer to the loop owning the link.
next	L-E LINK ptr.	Pointer to the next link around the loop.
prev	L-E LINK ptr.	Pointer to the previous link around the loop.

Star variant

edge	EDGE ptr.	Pointer to the edge owning the link.
loop	LOOP ptr.	Pointer to the loop owning the link.
next	L-E LINK ptr.	Pointer to the next link around the loop.

Continued on next page

prev	L-E LINK ptr.	Pointer to the previous link around the loop.
friend	L-E-LINK ptr.	Pointer to the next L-E LINK around the edge.

A.1.4 VERTEX-EDGE LINK

A link between an edge and a vertex. The edge-vertex link is used to chain all edges around the vertex in either clockwise or counter-clockwise order. Edge adjacency is determined via these links.

edge	EDGE ptr.	Pointer to the edge owning the link.
loop	LOOP ptr.	Pointer to the loop owning the vertex if there are no attached edges (should be set to NIL if there are any attached edges).
vertex	VERTEX ptr.	Pointer to the vertex owning the link.
next	V-E LINK ptr.	Pointer to the next link around the vertex.
prev	V-E LINK ptr.	Pointer to the previous link around the vertex.

A.1.5 BUNDLE

edge	EDGE ptr.	Pointer to the edge owning the link.
loop	LOOP ptr.	Pointer to the loop owning the vertex if there are no attached edges (should be set to NIL if there are any attached edges).
vertex	VERTEX ptr.	Pointer to the vertex owning the bundle.
next	BUNDLE ptr.	Pointer to the next link around the vertex.

A.1.6 WEDGE

edge	EDGE ptr.	Pointer to the edge owning the wedge.
rlink	L-E LINK ptr.	Pointer to a LOOP-EDGE link on the right of the edge.
llink	L-E LINK ptr.	Pointer to a LOOP-EDGE link on the left of the edge.
next	WEDGE ptr.	Pointer to the next wedge around the edge.

A.1.7 FACE

Faces are portions of surfaces. Faces are bounded by loops, which are ordered sets of edges.

loop	LOOP ptr.	Pointer to the first loop in the face.
surface	SURFACE ptr.	Pointer to the surface of the face (if any).
owner	FGROUP ptr.	Pointer to the facegroup owning the face.
next	FACE ptr.	Pointer to the next face in the facegroup.
feature	FEAT-LNK ptr.	Pointer to the head of a chain of feature links to features in which the face is included.
friend	FACE ptr.	For associating faces.
cogeom	F/FGR/SG ptr.	Pointer to a face, facegroup, or supplementary geometry entity using the same surface.
orient	BOOLEAN	Optional flag to indicate the orientation of the face w.r.t. the surface.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the face.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer face number.
name	NAME ptr.	Pointer to the name of the face.
access	ACCESS ptr.	A pointer to information about who created the face and its access status (for multi-user modelling).

A.1.8 LOOP

In its simplest form, the model datastructure consists only of faces, edges, and vertices. However, this does not allow multipli-connected faces, where there is an outer boundary and one or more inner boundaries, or where there are multiple outer boundaries. To allow for this kind of model, the edges bounding a face are divided into closed circuits of edges, called Loops.

Direct pointer and vertex-edge link variant

face	FACE ptr.	Pointer to the face owning the loop.
edge	EDGE ptr.	Pointer to the first edge in the loop (the others can be found using edge connectivity).
vertex	VERTEX ptr.	Pointer a vertex if there are no edges attached to the vertex (should be set to NIL if there are any attached edges).
next	LOOP ptr.	Pointer to the next loop in the face.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the loop.
marker	BITS	Marker bits for use by the modelling operations.
holeloop	LOGICAL	TRUE if this is a holeloop; FALSE if it is a perimeter loop.
number	INTEGER	Integer loop number.

Loop-edge link variant

face	FACE ptr.	Pointer to the face owning the loop.
link	W-E LINK ptr.	Pointer to the link of the first edge in the loop.
vertex	VERTEX ptr.	Pointer a vertex if there are no edges attached to the vertex (should be set to NIL if there are any attached edges).
next	LOOP ptr.	Pointer to the next loop in the face.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the loop.
marker	BITS	Marker bits for use by the modelling operations.
holeloop	LOGICAL	TRUE if this is a holeloop; FALSE if it is a perimeter loop.
number	INTEGER	Integer loop number.

A.1.9 FACEGROUP

Faces can be grouped together as logical units called Facegroups. Each facegroup forms a sub-unit according to some modelling criteria, e.g., that the faces in the facegroup have a common surface, or that they were produced in the same basic operation. Facegroups are for use in the modeller, not for a user. The feature structures, defined later, are more suitable for public use.

face	FACE ptr.	Pointer to the first face in the facegroup.
facegroup	FGRP ptr.	Pointer to the first facegroup in the facegroup.
next	FGRP ptr.	Pointer to the next facegroup at the same level in the owner.
owner	SH/FGRP ptr.	A pointer to the shell or facegroup that owns the facegroup.
surface	SURFACE ptr.	Pointer to the surface of the facegroup (if any).
cogeom	F/FG/SG ptr.	Pointer to a face, facegroup, or supplementary geometry entity using the same surface.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the facegroup.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer facegroup number.
name	NAME ptr.	Pointer to the name of the facegroup.

A.1.10 SHELL

Each closed set of faces in the object forms a Shell. It is useful to represent these shells explicitly in some way in a model, rather than having to retrieve the information by traversing a faceset to see whether it is closed.

owner	object ptr.	Pointer to the VOLUME or SHEET-OBJECT, which owns the shell.
next	SHELL ptr.	Pointer to the next shell in the object.
facegroup	FGRP ptr.	Pointer to the first FACEGROUP in the shell.

Continued on next page

info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the shell.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer shell number.
name	NAME ptr.	Pointer to the name of the shell.

A.1.11 WIREFRAME_OBJECT

Wireframe objects are collections of edges and vertices only, face and loop information being ignored.

edge	EDGE ptr.	Pointer to the first edge, the head of the 'all edges in body' list in the wireframe object.
vertex	VERTEX ptr.	Pointer to the first vertex, the head of the 'all vertices in body' list of the wireframe object.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the object.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the object is referenced.
xused	INTEGER	Integer indicating the number of times the object is referenced externally.
number	INTEGER	Integer object number.
name	NAME ptr.	Pointer to the name of the wireframe object.

A.1.12 SHEET_OBJECT

Sheet objects are degenerate models of thin plate objects, for example, with the small side faces represented by edges. They can also be used for various

other purposes. They are considered in this dissertation as being Eulerian (see section 3.3) and locally manifold.

edge	EDGE ptr.	Pointer to the first edge, the head of the 'all edges in body' list in the sheet object.
vertex	VERTEX ptr.	Pointer to the first vertex, the head of the 'all vertices in body' list of the sheet object.
shell	SHELL ptr.	Pointer to the first shell in the sheet object (sheet objects should really only have a single shell, except possibly for temporary results, but multiple shell sheet objects need not be excluded).
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the object.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the object is referenced internally.
xused	INTEGER	Integer indicating the number of times the object is referenced externally.
number	INTEGER	Integer object number.
name	NAME ptr.	Pointer to the name of the sheet object.

A.1.13 VOLUME_OBJECT

Volume objects are 'complete' solid models, with closed sets of faces bounding volumes of space.

edge	EDGE ptr.	Pointer to the first edge, the head of the 'all edges in body' list in the volume object.
vertex	VERTEX ptr.	Pointer to the first vertex, the head of the 'all vertices in body' list of the volume object.
shell	SHELL ptr.	Pointer to the first shell in the volume.
feature	FEATURE ptr.	Pointer to the head of a chain of feature records.

Continued on next page

info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the object.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the object is referenced.
xused	INTEGER	Integer indicating the number of times the object is referenced externally.
number	INTEGER	Integer object number.
name	NAME ptr.	Pointer to the name of the volume object.

A.1.14 POINT

A zero-dimensional entity, a position in 3D Euclidean space defining the position of a vertex or a supplementary geometric entity.

position	VECTOR	Position data of the POINT.
user	V/SG ptr.	Pointer to a vertex or supplementary geometry entity using the point.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the point.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the point is referenced.
number	INTEGER	Integer point number.

A.1.15 CURVE

A one-dimensional entity defining the shape of an edge.

type	CRVTYP ptr.	Pointer to a curve type, specifically an entity containing the type of the curve and the associated geometric data.
user	ED/SG ptr.	Pointer to an edge or supplementary geometry entity using the curve.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the curve.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the curve is referenced.
number	INTEGER	Integer curve number.

A.1.16 CURVETYPE

This is a united mode of all allowed curve types in the modeller. The exact format of the geometric types depends on basic decisions about the modeller design; hence, are not fixed. Various geometric types are suggested in section 2 of this appendix. With the curve types defined there, the CURVETYPE mode is defined (in Algol68 form) as:
UNION(STRAIGHT, CIRCLE, ELLIPSE, PARABOLA, HYPERBOLA, FFCURVE)
where FFCURVE stands for some kind of free-form curve.

If the UNION construct is not supported, then this can be replaced by an integer type and having an anonymous pointer.

A.1.17 SURFACE

A two-dimensional entity defining the shape of a face.

type	SRFTYP ptr.	A pointer to the specific geometric data of the surface.
user	F/FGR/SG ptr.	Pointer to a face, facegroup, or supplementary geometry entity using the same surface.

Continued on next page

info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the surface.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the surface is referenced.
number	INTEGER	Integer surface number.

A.1.18 SURFACETYPE

This is a united mode of all allowed surface types in the modeller. As with curves, the exact format of the geometric types depends on basic decisions about the modeller design. Various geometric types are suggested in section 2 of this appendix. With the types defined there, the SURFACETYPE mode is defined (in Algol68 form) as:

UNION(PLANE, SPHERE, CYLINDER, CONE, QUADRIC, TOROID, FF-SURF)
where FFSURF stands for some kind of free-form surface.

As for curves, if the UNION construct is not supported, then this can be replaced by an integer type and having an anonymous pointer.

A.1.19 INSTANCE

An example of a prototype model (single object or group of objects) with an associated transformation.

object	obj./group ptr.	Pointer to a single object (wireframe-, sheet-, or volume-object) or to a group-of-objects instanced by this entity.
owner	GR.OF.OBJ ptr.	Pointer to the group of objects owning this instance.
next	INSTANCE ptr.	Pointer to the next instance in the group-of-objects.
trans	TRANSF. ptr.	Pointer to the transformation data of the instance.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.

Continued on next page

group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the instance.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer instance number.

A.1.20 GROUP-OF-OBJECTS

An assembly or collection of instances creating a single logical unit.

inst	INSTANCE ptr.	Pointer to the first instance in the group of objects.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the group-of-objects.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
used	INTEGER	Integer indicating the number of times the group-of-objects is referenced.
xused	INTEGER	Integer indicating the number of times the object is referenced externally.
number	INTEGER	Integer group-of-objects number.
name	NAME ptr.	Pointer to the name of the group of objects.

A.1.21 FEATURE

As well as the facegroup structures for grouping faces, it is advisable to have an extra facegrouping mechanism for preserving feature interpretations in the model. These differ from the facegroup mechanism in that a face can belong to several different feature facesets, either because the same face is allocated as part of two different features, or because there are parallel feature interpretations of a model. Ideally these feature datastructures should be in the form of 'frames' (Minsky [86]) to preserve the interpretations of the roles of the various faces grouped into a feature, or as simple groups of faces.

However, if features are represented as frames, then there has to be a frame for each feature, so two possible types of structure are described, frames and uninterpreted facegroups.

Frame alternative

The frame alternative means that there are separate interpretations of various features.

next	FEATURE ptr.	A pointer to the next feature at the same level as the feature record.
child	FEATURE ptr.	A pointer to the head of a chain of subordinate feature records.
owner	FEATURE ptr.	A pointer to the feature on which this feature is dependent.
frametype	FRMTYP ptr.	Pointer to the structural data, and feature-specific data, of the feature.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.
group	GRP-LNK ptr.	Pointer to the head of a chain of group links to the groups containing the feature.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer feature number.
name	NAME ptr.	Pointer to the name of the feature.

Facegroup alternative

The facegroup alternative means that faces are grouped together without necessarily defining their roles in the features.

next	FEATURE ptr.	A pointer to the next feature at the same level as the feature record.
child	FEATURE ptr.	A pointer to the head of a chain of subordinate feature records.
owner	FEATURE ptr.	A pointer to the feature on which this feature is dependent.
face	FEAT-LNK ptr.	Pointer to the first link to the faces contained in the feature.
info	INFO.EL ptr.	Pointer to a list of information elements associated with the edge.

Continued on next page

group	GRP-LNK ptr.	Pointer to the head of a chain of group links to groups containing the feature.
space	SPACE ptr.	A pointer to a box, sphere, or some such spatial occupancy measure.
marker	BITS	Marker bits for use by the modelling operations.
number	INTEGER	Integer feature number.
name	NAME ptr.	Pointer to the name of the feature.

A.1.22 FRAMETYPE

This is analogous to the CURVETYPE and SURFACETYPE. The FEATURE entity contains a pointer that can be to any feature type that the modeller knows about. The feature data, in terms of the face structure and any other data relevant for the feature, varies for each type of feature. Possible feature formats are contained in section A.3. The definition of frametype is a united mode, for example (based on Kyprianou's classifications):
UNION(BOSS, POCKET, SLOT, PARTIAL SLOT, BRIDGE,
REENTRANT, RAIL, THROUGH-HOLE, UNCLASSIFIED, ROOT)

A.1.23 FEATURE-LINK

This is simply a link between features and faces so that faces can participate in several features. It is useful both for the feature as facegroup structure, above, and for feature frames, the feature frame holding lists of FEATURE-LINKS to faces in the feature.

feature	FEATURE ptr.	Pointer to the feature in which the face participates.
featnext	FEAT.-LINK ptr.	Pointer to the next FEATURE-LINK in the feature.
facenext	FEAT.-LINK ptr.	Pointer to the next FEATURE-LINK connecting a face to features in which it participates.
face	FACE ptr.	Pointer to the face which is contained in the feature.

A.1.24 MECHANISM_CONSTRAINT

For defining relationships between sub-models in a group-of-objects, or between sub-models and supplementary geometry. Please note, the following is only a simplified suggestion. The exact nature of these extra items is variable,

because they are largely user controlled. The following is a suggestion and should not be taken as definitive.

part1	OBJECT ptr.	Pointer to one object to be linked.
part2	OBJ/GEOM ptr.	Pointer to a second object or to a supplementary geometry item to be linked to the first object.
mechm	GRP-OF-OB ptr.	Pointer to the assembly containing the linked parts.
origin	VECTOR	Origin of the mechanism local coordinate system.
x-axis	VECTOR	X-axis of the mechanism local coordinate system.
y-axis	VECTOR	Y-axis of the mechanism local coordinate system.
type	INTEGER	The constraint type, e.g. 'fixed', 'rotational', 'slide', etc.
limit1	VECTOR	Limit data in some form.
limit2	VECTOR	Limit data in some form.
info	PASS._DATA ptr.	Pointer to passive_data to provide supplementary information about the mechanism.
name	NAME ptr.	The name of the mechanism.
number	INTEGER	The integer mechanism constraint number.

A.1.25 SHAPE_MODIFIER

For defining implicit modifications to model elements, e.g., implicit blends on edges, or screw-threads for circular holes or projections.

item	topology ptr.	Pointer to the item whose shape is to be modified, i.e., a FACEGROUP, FACE, EDGE, VERTEX, or FEATURE.
value	REAL	Real shape modification parameter (probably more data will be needed).
type	INTEGER	Integer modification type, e.g. 'blend', 'chamfer', 'thread', 'weld-line', etc.
info	PASS_DATA ptr.	Pointer to passive data, supplementary information about the shape modifier.
name	NAME ptr.	Pointer to the name of the shape modifier.
number	INTEGER	Integer shape modifier number.

A.1.26 CONSTRAINT_DATA

For defining relationships such as distance or angle between elements in the model.

item	topology ptr.	Pointer to the constrained item (FACE, EDGE, VERTEX, or FEATURE).
base	topology ptr.	Pointer to the item taken as a datum for the constrained item (FACE, EDGE, VERTEX, FEATURE, or SUPPLEMENTARY GEOMETRY).
type	INTEGER	Integer constraint type, e.g. 'Distance', 'Angle', 'Surface finish', etc.
value	REAL	Constraint value.
info	PASS_DATA ptr.	Pointer to passive data, supplementary information about the constraint.
name	NAME ptr.	Pointer to the name of the constraint.
number	INTEGER	Integer constraint number.

A.1.27 SUPPLEMENTARY_GEOMETRY

Supplementary geometry is used as a design aid and for helping specify constraints between instances in a group-of-objects.

geometry	geometry ptr.	Pointer to a SURFACE, CURVE, or POINT.
cogeom	top./SG ptr.	Pointer to a FACEGROUP, FACE, EDGE, VERTEX, or SUPPLEMENTARY GEOMETRY entity using the same geometry.
name	NAME ptr.	Pointer to the name of the geometry.
number	INTEGER	Integer supplementary geometry number.

A.1.28 PASSIVE_DATA

Information intended to be communicated with the model, for example, material, colour, and price.

text	STRING	Character data.
ints	[1..n] INTEGER	List of integers associated with the data.
reals	[1..n] REAL	List of real data.

A.1.29 NAME

Name entities are for users to assign names to items. The effect of modelling operations on named items should be clear, if possible, but the onus of name handling really lies with the user, not the system developer. Names are a convenience to allow selection.

text	STRING	Name of the item.
global	LOGICAL	Logical to indicate whether the name is available to other users.
status	INTEGER	Status code for the name (fixed, unique, etc.)

A.1.30 INFORMATION_ELEMENT

Used to make lists of information elements associated with an entity.

info	info. ptr.	Pointer to a MECHANISM_CONSTRAINT, a SHAPE_MODIFIER, a CONSTRAINT_DATA, or a PASSIVE_DATA element.
next	INFO.EL. ptr.	Pointer to the next information element link in the list associated with some entity.

A.1.31 TRANSFORMATION

Transformation data describes how a model is to be modified, reflected, scaled, translated, or rotated. Other transformation data are possible, e.g., shearing, but are generally complicated to apply in practice. For assemblies, it is advisable to limit the transformations to rotations and translations only. Transformations should always be invertable; hence, zero scaling in any direction should be disallowed. Valid transformations affect the geometry, not the topology, and hence the task of handling them belongs to the functional interface to the low-level modeller.

matrix	4x4 REAL matrix	Transformation data.
negative	LOGICAL	Logical indicating whether the matrix is negative.

A.1.32 GENERAL_GROUP

A useful general mechanism for associating elements for some purpose.

list	GRP_LINK ptr.	Pointer to the link to the first element in the group.
info	INFO.EL ptr.	Pointer to information associated with the group.
name	NAME ptr.	Pointer to the name of the group.
number	INTEGER	Integer group number.

A.1.33 GENERAL_GROUP LINK

The link between groups and the items contained in them.

group	GROUP ptr.	Pointer to the group owning the link.
grpnext	G.G.LINK ptr.	Pointer to the next group link in the group.
entnext	G.G.LINK ptr.	Pointer to the next group link associated with an entity.
entity	entity ptr.	Pointer to an entity in the group.

A.1.34 SPACE

Spatial occupancy measure or measures. Possible formats for three types are described here: spheres (also called bubbles), orthogonal boxes, and fat sectors.

Sphere alternative

centre	VECTOR	Coordinates of the centre of the sphere (bubble).
radius	REAL	Radius of the sphere.

Box alternative

xmin	REAL	Minimum x value of the box.
ymin	REAL	Minimum y value of the box.
zmin	REAL	Minimum z value of the box.
xmax	REAL	Maximum x value of the box.
ymax	REAL	Maximum y value of the box.
zmax	REAL	Maximum z value of the box.

Fat sector alternative

centre	VECTOR	Centre of the fat sector.
inrad	REAL	Inner radius of the fat sector.
outrad	REAL	Outer radius of the fat sector.
n	INTEGER	Number of sides for the bounding pyramid.
nactual	INTEGER	Size of the fat sector data array.
ndata	[1..nactual]VEC.	Data array (normals to the sides of the pyramid) for the pyramid.

A.1.35 ACCESS

Access data for model elements for use in multi-user modelling. This entity is speculative, because it concerns facilities for allowing multiple access to modelling systems for collaborative design. The idea is that certain parts of a design can be fixed, the interfaces between sub-parts in an assembly for example, and modelling operations that affect these should be forbidden unless specifically allowed by the designer responsible for the assembly.

ownerid	STRING	Identifier of the person who created the entity.
password	STRING	User password.
access	STRING	Access code for the entity.

A.2 Geometry formats

Once the set of geometry for the modeller has been identified, the question of how this geometry is to be represented arises. A possible set of surfaces, identified in chapter 3, is:
PLANES
SPHERES
CYLINDERS
CONES
GENERAL QUADRICS
TOROIDS
FREE-FORM SURFACES

and the set of curves:
STRAIGHT LINES
CIRCLES
ELLIPSES
PARABOLAS

HYPERBOLAS
FREE-FORM CURVES

It should be pointed out that the exact format and type of geometry can vary a lot. There is no one 'fixed' set, and no set that is necessarily better because the way of judging the quality of a set of geometry is, to some extent, subjective. The important thing is the information content of the set, which should be similar for any geometry set, and the stability of the set under modelling operations. It may be advantageous to include extra information to facilitate calculations, but this information has to be consistent.

The definitions here follow Pfeifer's [99] philosophy of using points as the basis for surface forms, as far as possible. There are also specific geometry types rather than having a single general representation; other formats may be found in general geometry textbooks. Generally the geometry formats follow the GPM volume module (as devised by Fjällström, Lindholm, and Pfeifer) or of BUILD/FFSolid (Braid, Smith, and Solomon). Possible formats for these are as follows.

A.2.1 PLANE

This differs from the point-based plane definition (which would consist of three non-collinear points).

matrix	[1..4] REAL	Normal and minimum distance to the origin. The sum of the squares of the first three elements should be 1. If the elements of the matrix are named A, B, C, and D, then the set of points (x, y, z) on the surface are related by the equation $Ax + By + Cz + D = 0$.
xdir	*VECTOR*	*Optional x-direction for parametrisation*

A.2.2 PLANE (alternative)

The first plane definition is a minimal definition; an alternative is to provide a point and a normal vector.

origin	VECTOR	Origin point on the plane.
normal	VECTOR	Normal vector to the plane.
xdir	*VECTOR*	*Optional x-direction for parametrisation*

A.2.3 SPHERE

The sphere definition here follows the point-based philosophy of Pfeifer. It is also possible to define the sphere as a centre-point and a radius. This second alternative is not described here.

centre	VECTOR	Centre of the sphere.
srfpnt	VECTOR	Point on the surface of the sphere.
negative	LOGICAL	Logical value to determine whether the surface is negated.

A.2.4 CYLINDER

Again the point-based version. It is possible to represent the cylinder as a point, a direction and a radius, but this is not described here.

axpnt1	VECTOR	One point on the cylinder axis.
axpnt2	VECTOR	Second point on the cylinder axis.
srfpnt	VECTOR	Point on the cylindrical surface.
negative	LOGICAL	Logical value to determine whether the surface is negated.

A.2.5 CONE

This can be used to define a half cone, and ignore the half of the cone lying on the other side of the apex from the surface point.

axpnt1	VECTOR	Apex of the cone.
axpnt2	VECTOR	Point on the cone axis.
srfpnt	VECTOR	Point on the cone surface.
negative	LOGICAL	Logical value to determine whether the surface is negated.

A.2.6 GENERAL QUADRIC

Like the planar surface definition, this does not use points

matrix	4x4 REAL matrix	General quadric surface definition.

A.2.7 TOROID

Lines drawn from the basepnt to the ringpnt and surfpnt in the definition should, if possible, be at right angles to each other to ensure a numerically more stable definition.

basepnt	VECTOR	Point at the centre of the toroid ring.
ringpnt	VECTOR	Point on the ring (circle, radius: torus major axis).
surfpnt	VECTOR	Point on the toroidal surface lying in the plane of the torus ring.
negative	LOGICAL	To indicate whether the torus is negated.

A.2.8 FREE-FORM SURFACE

The exact form of free-form surfaces is also variable because there are several types. Some sort of free-form geometry is necessary to provide general shape facilities in the modeller.

m	INTEGER	Number of segments in the u direction.
n	INTEGER	Number of segments in the v direction.
data	[0..m][0..n]VECTOR	Control points for the surface.
uweights	[0..m]REAL	*Possible u weights for a rational surface form.*
vweights	[0..n]REAL	*Possible v weights for a rational surface form.*
knots	[0..m]REAL	Knot vector in the u direction for a non-uniform form.
knots	[0..n]REAL	Knot vector in the v direction for a non-uniform form.
negative	LOGICAL	Logical value to determine whether the surface is negated.

A.2.9 STRAIGHT LINE

There are basically two variants to this: two points or point and direction vector.

p1	VECTOR	Start point of the line.
p2	VECTOR	End point of the line.

The other form is:

p	VECTOR	Start point of the line.
dir	VECTOR	Direction of the line (usually normalised).

The second form is more stable than the first form for short lines. The first form can also be used to represent a line segment. The second form, though, is probably marginally preferable to the first (a personal opinion that you can feel free to ignore).

A.2.10 STRAIGHT LINE – short form

This was used in BUILD/FFSolid and in the GPM volume module as a kind of null definition. There is no geometric data, the positions of the start and end vertices of the edge to which the curve is attached define the curve. However, this form of the curve is dependent on the edge, so it cannot be used independently of the edge. Therefore, the full straight line form is necessary for communication to the geometric utilities, for example.

A.2.11 CIRCLE

Because the circle is a closed curve, there has to be some way of determining which portion of it defines the shape of an edge. The curve has, therefore, an implicit direction from p1 to arcpt to p2.

p1	VECTOR	The first point on the curve.
arcpt	VECTOR	A second point on the curve.
p2	VECTOR	A third point on the curve.

A.2.12 CIRCULAR ARC – short form

This is Malcolm Sabin's circular arc form, also known as a k-curve. The curve data consist of the circle normal and a 'bulge factor' to define the curvature of the curve. The information needed to define the complete shape of the curve is supplied by the positions of the start and end vertices of the edge. Note that if the vector from the start vertex of the edge to the end vertex of the edge is not orthogonal to the curve normal, then the geometric information is inconsistent. This form of circle definition cannot be used to define a complete circle as the bulge factor tends to infinity and the form generally becomes unstable. The convention in BUILD/FFSolid was that this short form of circular arc should not be used to describe curves subtending an angle of

more than 180 degrees at their centre; hence, the bulge factor was always between +1 and -1.

normal	VECTOR	Circle normal.
bulge	REAL	'Bulge' factor of the curve; if t denotes the angle of the arc at its centre then the bulge factor has a value of $tan(t/4)$.

A.2.13 ELLIPSE

The point format of the ellipse uses three points at three extreme points of the ellipse, i.e., at two ends of the minor axis and one end of the major axis, or at two ends of the major axis and one end of the minor axis, the first and last points being on the same axis. The order of the points determines the direction of the curve. The centre of the ellipse is determined by the average of the first and third points; hence, the other axis can be determined from the second point and the ellipse centre. Obviously the axes should be orthogonal to each other.

p1	VECTOR	First axis extremity.
p2	VECTOR	Second axis extremity.
p3	VECTOR	Third axis extremity.

A.2.14 PARABOLA

The placing of the points is illustrated in figure A.1.

p1	VECTOR	Focus.
p2	VECTOR	Parallel axis extremity.
p3	VECTOR	Point on curve.

A.2.15 HYPERBOLA

The placing of the points is illustrated in figure A.2.

p1	VECTOR	Origin.
p2	VECTOR	Curve extremity.
p3	VECTOR	Axis point.

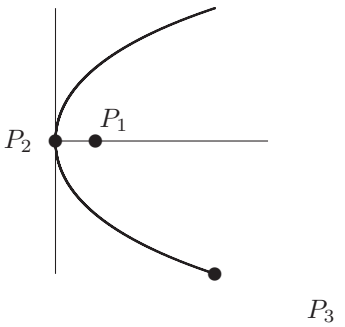


Figure A.1: Parabola definition points

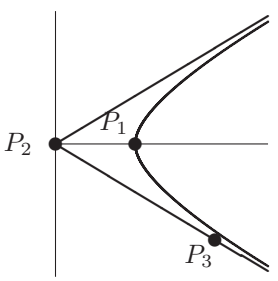


Figure A.2: Hyperbola definition points

A.2.16 FREE-FORM CURVE

There are several of these, but the most popular at present is the NURBS (Non-Uniform Rational B-Spline) because of its flexibility.

n	INTEGER	Number of segments.
data	[0..n]VECTOR	Control points for the curve.
weights	[0..n]REAL	*Possible weights for a rational curve form.*
knots	[0..n]REAL	*Possible knot vector for a non-uniform form.*

A.3 Feature frames

The feature descriptions here are not intended as a definitive set but as examples. There has been a large amount of work, for example, in the ISO 10303 AP214 and AP224 (STEP) effort, to create a unified set of features. This, however, is aimed at manufacturing features rather than features in general.

A.3.1 Boss

A boss is defined as a base, or top face with convex edges surrounded by primary faces (or blends between the boss top and a primary face) each with one concave edge to a face not surrounding the boss. The main piece of information is the boss, the facegroup contains the surrounding faces, but this should be verified before use in a feature-based application.

top	FACE ptr.	Pointer to the top face of the boss.
sides	FACEGROUP ptr.	The set of faces surrounding the top face of the boss.

A.3.2 Pocket

A pocket can be defined as a base face with concave edges surrounded by primary faces, or blends between the base face and the primary faces, each with one convex edge to a face not a side face of the pocket. This is analogous to the boss feature structure.

base	FACE ptr.	Base face of the pocket.
sides	FACEGROUP ptr.	The set of side faces around the pocket.

A.3.3 Slot

A slot is defined as a base face with two sequences of concave edges, or concave blend faces separated by one or more convex edges

base	FACE ptr.	Pointer to the slot base.
side1	FACEGROUP ptr.	Pointer to the faces along one side of the slot.
side2	FACEGROUP ptr.	Pointer to the faces along the other side of the slot.

A.3.4 Partial slot

A partial slot is defined as a base face with one connected set of concave edges separated by one or more convex edges

base	FACE ptr.	Pointer to the slot base.
sides	FACEGROUP ptr.	Pointer to the faces along the sides and end of the slot.

A.3.5 Bridge

A bridge is defined as two rings of concave edges bounding a connected set of bridge-faces. A bridge-face is either a closed convex curved face or a face with at least two neighbours in the set separated by concave edges.

sides	FACEGROUP ptr.	The set of side faces around the bridge.

A.3.6 Reentrant

The reentrant feature is a feature that has two exits to the outside of an object, like a through hole, but the geometry is not two-and-a-half dimensional.

sides	FACEGROUP ptr.	The set of side faces around the reentrant feature.

A.3.7 Rail

A rail is defined as a base face with two sets of neighbouring primary faces. This is the extrusive feature that corresponds to the intrusive slot.

top	FACE ptr.	Pointer to the rail top.
side1	FACEGROUP ptr.	Pointer to the faces along one side of the rail.
side2	FACEGROUP ptr.	Pointer to the faces along the other side of the rail.

A.3.8 Through hole

A through hole is defined as two rings of convex edges bounding a connected set of through-faces. A through-face is either a closed concave curved face or a face with at least two neighbours in the set separated by convex edges.

sides	FACEGROUP ptr.	The set of side faces around the through hole

A.3.9 Unclassified

faces	FACEGROUP ptr.	The set of the faces that constitute the feature

Appendix B

Datastructure traversals

Three kinds of traversals are presented in this appendix: multiple element traversals, edge element traversals, and single element traversals.

B.1 Multiple element traversals

The traversals provide a convenient way of accessing a model datastructure without having to know the exact way that entities are connected. As such they form part of a 'standard functional interface' to a model. Obviously, because they are directly linked to the way that the datastructure is implemented, it is impossible to be exact about how these work. As mentioned in Chapter 3, the extra links included in the datastructure can provide more efficient traversal for certain purposes at the expense of extra housekeeping. An example is that of edges (or vertices) in a body. These could be accessed via the face structure rather than having them in separate chains. However, such access can be tedious in some circumstances, hence, the provision of the extra chains for direct access. What is important is the functionality rather than, exactly, the means of achieving this functionality. This appendix contains examples to illustrate how they work based on the datastructure described in Appendix A. It should be fairly clear how these examples can be adapted for different datastructures.

There are two basic strategies: 1) to traverse the structure applying a user-supplied function to each entity or 2) to build a separate list and traverse the list, applying the user-function to each element in the list afterwards.

The first method, which was implemented in the BUILD system, for example, in effect treats the elements in the datastructure as though they were linked in a dynamic list. It is more efficient, in storage terms, than building a separate structure, but it has inherent problems if the datastructure is changed while traversing it.

Building a separate list of entities that are then traversed is less efficient

but a more stable method if the traversal tools are to be provided for use by people other than the modeller developers, applications programmers for example. This kind of technique was included in the CAM-I Applications Programming Interface, for example, with the "Identify connnected entities" class of utilities [12]. There are two variations on the technique: 1) returning a list of entities and 2) building the list and applying a user-supplied function to the elements.

The first of these variations is exemplified by the CAM-I API "Identify connected elements" methods, as mentioned above. It assumes that the user will be responsible for cleaning up the extra structure. The second variation means that the user need never know the exact method or deal with the extra datastructures, even though it is not an onerous task.

Which of these basic methods or variations to choose is largely a matter of taste. Building a separate list of entities seems to be better from the point of view of stability, and hiding that list from the user also seems slightly preferable, from the authors' point of view, so that is the philosophy that will be illustrated here. Modification to perform the first strategy, traversing the datastructure directly, or the variation where the user is given a list to handle, is straightforward.

The set of traversals identified in chapter 3 is as follows:

Owner to single level structure following:
vertices in body
edges in body
loops in face
faces in facegroup containing only faces
instances in a group of objects
notes in entity

Shared entity traversal:
surfaces in object
curves in object
single objects in a group of objects

Tree-structure traversal:
facegroups under object or facegroup
faces in body
faces in facegroup where the facegroup contains facegroups
instances in an assembly

Topological set traversal:
edges in vertex
edges in loop

faces, edges, and vertices in a shell

B.1.1 Owner to single level structure following

The traversals in this section include:
vertices in body
edges in body
loops in face
faces in facegroup containing only faces
instances in a group of objects
notes in entity

These can be connected in open (i.e., ending with a NIL pointer) or closed chains. Generally it is always desirable to be able to reach the head of a chain, by having a pointer back to the owner of the chain, or by having a doubly linked chain or a circular chain. The general form of the traversal, which deals with both properly circular lists and open lists, is:

```
list := empty_list;
current := start := first entity referred to from owner;
UNTIL ( current = NIL )
BEGIN
add_to_list(current, list);
current = next OF current;
IF ( current = start ) current = NIL
END;
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
if ( NOT empty(list) ) delete(list);
```

B.1.2 Shared entity traversal

surfaces in object
curves in object
single objects in a group of objects

The shared entity traversals are for accessing once entities that may be referenced from several places in a datastructure. Typical examples are those

shown above. Curves and surfaces may, potentially, be referenced from several edges or faces (respectively) in an object. Also, the same object may be referenced from several places in an assembly or group of objects. The surfaces, curves, or single objects are referenced by traversing an overlying structure.

It should be noted that short form curves and straight lines were included in the BUILD system to save space. Several edges could share the same curve, straight lines or circular arcs, rather than have their own unique geometry, the geometry being completely defined geometrically together with the start and end vertex positions of the edge. There is now a tendency to have complete geometry for each edge, because storage space is no longer as crucial as in the early days of solid modelling.

Surfaces and curves may or may not be shared, according to the design and working of the modeller. The description below is intended as a general guide to outline the principles; the actual needs are determined from the design of a modeller.

A simple way of checking whether entities have already been processed is to use the marker bit mechanism described in chapter 3.

For traversing all surfaces in a body:

```
 list := empty_list;
for all faces in body(b, mark(surface of face));
for all faces in body(b, (FACE f):
if ( marked(surface of f) )
BEGIN
add_to_list(surface, list);
unmark(surface of face)
END)
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
 if ( NOT empty(list) ) delete(list);
```

B.1.3 Tree-structure traversal

facegroups under object or facegroup
faces in body
faces in facegroup where the facegroup contains facegroups
instances in an assembly

```
list := empty_list;
cur := instance OF assembly;
 UNTIL ( cur IS NIL )
BEGIN
add_to_list(cur, list);
if ( instance refers to an assembly )
cur = first instance OF assembly
else BEGIN
cur = next instance OF assembly
if ( cur IS NIL )
BEGIN done = FALSE; while ( (NOT done) AND (cur IS NIL) )
BEGIN
cur = parent instance OF parent assembly OF cur;
if ( cur IS NIL ) done = TRUE;
else if ( next instance OF cur ISNT NIL )
cur = next instance OF cur; END
list := tail(list);
done := user_function(current)
END;
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
if ( NOT empty(list) ) delete(list);
```

B.1.4 Topological set traversal

Traversing manifold and non-manifold topological structures are similar but slightly different because of the different ways of relating entities, so they are treated differently here.
edges in vertex
edges in loop
faces, edges and vertices in a shell

```
 list := empty_list;
flist := empty_list;
for all faces in body(b, (FACE f): mark(f));
for all edges in body(b, (EDGE e): mark(e));
for all vertices in body(b, (VERTEX v): mark(v));
add_to_list(first face OF b, flist); nf = 1; i = 0;
```

```
while ( i < nf )
BEGIN
f = flist MEMBER i;
add_to_list(f, list);
for all loops in face(f, (LOOP l)
BEGIN
v = vertex OF l;
if ( v ISNT NIL ) add_to_list(v, list); else for all edges in loop(l, (EDGE e)
BEGIN
if ( marked(e) )
BEGIN add_to_list(e, list); unmark(e); END;
v = vcwel(e, l);
if ( marked(v) )
BEGIN add_to_list(v, list); unmark(v); END;
of = face OF lopel(e, l);
if ( marked(of) )
BEGIN add_to_list(of, flist); unmark(of); nf = nf + 1; END
END;
END))
i = i + 1;
END
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
 if ( NOT empty(list) ) delete(list);
```

Another interesting traversal is the 'all edges in loop' using the winged-edge pointers. For this it is necessary to have a trailing vertex so that wire edges can be traversed properly.

```
list := empty_list;
current := start := edge OF loop;
if ( wire(current) ) vertex = end OF current;
else vertex = vccel(current, loop)
UNTIL ( current = NIL )
BEGIN
IF ( NOT wire(current) OR NOT in_list(current, list) )
add_to_list(current, list);
IF ( wire(current) ) current = ecwev(current, vertex);
ELSE current = eccel(current, loop);
vertex = vopev(current, vertex);
```

```
IF ( current IS start )
BEGIN
IF ( NOT wire(current) OR (vertex IS end OF current) )
current = NIL
ELSE BEGIN
current = ecwev(current, vertex);
vertex = vopev(current, vertex);
END;
END;
END;
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
if ( NOT empty(list) ) delete(list);
```

With the loop-edge links, this becomes a simple list following process, with the same necessity of checking wire edges so that they appear in the list only once. The basic process is similar to that for assigning a new loop pointer to a closed set of edges, as described in Appendix C.

Note also, in the above traversal, the duality between eccel and ecwev. They both give the next edge counter-clockwise around a loop. This can also be used for setting wireframe edges into a surface.

B.1.5 Manifold datastructure traversals

The set of traversals was mainly worked out for manifold modelling. It is not necessary to have a special set of manifold operations, but some details may need to be changed to ensure a manifold behaviour for a non-manifold datastructure. In particular, the topological set traversals may need to be modified. For non-manifold vertices, there are several distinct edge sets. One variant on the vertex structure has a list of start edges instead of the 'friend' links in the structure in Appendix A. With this, traversing all edges at the vertex requires a start edge in the desired edge set. Traversing all faces, edges and vertices in a connected set also needs to cope with multiple faces at an edge. This is ambiguous if there is a simple ring of loop-edge links around the edge as in the star representation described in chapter 5. Having a linked edge set with just two faces at each vertex makes it clearer to distinguish between the cases where the edges represent material with zero thickness or a gap with zero thickness. Non-manifold cases are presented in the next section.

B.1.6 Non-manifold datastructure traversals

A set of non-manifold traversals is as follows:
all faces round edge
all edges at a vertex

The exact nature of these depends on how the non-manifold structure is represented. In the star structure, vertices have a list of edges, one in each separate edge set meeting at the vertex. Also, edges have a ring of loop-edge links around the edge. This makes it easy to traverse, but much more difficult to distinguish between the different conditions, i.e., if objects just touch each other or just miss. In the degenerate type, there are multiple edges or vertices at the same place, linked by a 'friend' pointer. This disambiguates the cases, but it duplicates information. Luo and Lukacs [78] proposed extra structures, named bundles and wedges, which are more-or-less equivalent to loops but for edges and vertices. The description below is for the degenerate description, for the popular star representation getting the faces at the edge is a simple list following traversal.

```
 list := empty_list;
current := edge;
UNTIL ( current = NIL )
BEGIN
rlink = rlink OF current; llink = llink OF current;
add_to_list(face OF loop OF rlink, list);
add_to_list(face OF loop OF llink, list);
current = friend OF current;
IF ( current = edge ) current = NIL
END;
 done := FALSE;
UNTIL ( empty(list) OR done )
BEGIN
current := head(list);
list := tail(list);
done := user_function(current)
END;
 if ( NOT empty(list) ) delete(list);
```

B.2 Edge element traversals

The aim of the functions in this section is to provide functional access to the datastructure regardless of the way that it is implemented. Obviously it is slightly wasteful to use functions instead of direct pointers, but if the functions are used instead, code becomes independent of datastructure details. The details are handled by these functions.

Note that these are essentially for manifold edges. If the edge is non-manifold, then there are several possibilities for the right and left loops and neighbouring edges.

B.2.1 rloop

This finds the right loop of the edge, either using a direct pointer or via links.

For the winged-edge or vertex-edge link cases, this might be:

```
PROC rloop(EDGE e)
BEGIN
IF e IS NULL THEN return NULL;
ELSE return rloop OF e
FI
END
```

For winged-edge (or loop-edge) links, this is more complicated:

```
PROC rloop(EDGE e)
BEGIN
LOOP l;
L-E LINK rlink;
IF e IS NULL THEN rlink = NULL; ELSE rlink = rlink OF e FI;
IF rlink IS NULL THEN l = NULL; ELSE l = loop OF rlink; FI;
return l
END
```

The above assumes that the links are directly referenced. Some systems do not have two link references but instead have a pointer to one link in a ring. In this case, you might have:

```
PROC rloop(EDGE e)
BEGIN
LOOP l;
L-E LINK llink, rlink;
IF e IS NULL THEN rlink = NULL; ELSE rlink = link OF e FI;
IF (rlink ISNT NULL) AND leftlink(rlink)
THEN llink = rlink; rlink = friend OF llink;
FI
IF rlink IS NULL THEN l = NULL; ELSE l = loop OF rlink; FI
return l
END
```

The operation ‘leftlink’ simply tests whether the L-E LINK is a left link. This is conveniently done using flags.

All these assume that the edge is manifold. In the structures in Appendix A where edges are duplicated to represent non-manifold topological conditions, then this is not a problem. If, as in the the last example, the link is simply one of a ring of links, then the operation is ambiguous. An option is to check for this and provide a warning, as follows:

```
PROC rloop(EDGE e)
BEGIN
LOOP l;
L-E LINK llink, rlink;
IF e IS NULL THEN rlink = NULL; ELSE rlink = link OF e FI;
IF rlink IS NULL THEN llink = NULL; ELSE IF leftlink(rlink)
THEN llink = rlink; rlink = friend OF llink
ELSE llink = friend OF rlink; FI
IF (llink ISNT NULL) AND (friend OF llink ISNT rlink)
THEN warning("non-manifold edge");
IF rlink IS NULL THEN l = NULL; ELSE l = loop OF rlink; FI
return l
END
```

B.2.2 lloop

The left loop. These functions are similar to the above set of right loops, so they will not be given explicitly.

B.2.3 svert

The start vertex function. For winged-edge and winged-edge link structures where there is a direct pointer this is simply:

```
PROC svert(EDGE e)
BEGIN
IF e IS NULL THEN return NULL
ELSE return svert OF e
FI
END
```

Otherwise you have the more complex version:

```
PROC svert(EDGE e)
BEGIN
VERTEX v;
V-E LINK vlink;
IF e IS NULL THEN vlink = NULL; ELSE vlink = start OF e FI;
```

```
IF vlink IS NULL THEN v = NULL; ELSE v = vertex OF vlink; FI
return v
END
```

B.2.4 evert

The end vertex function. This is very similar to the svert function, so it will not be repeated here.

B.2.5 rcwe

The right clockwise edge. With the direct pointer variant, this is trivial:

```
PROC rcwe(EDGE e)
BEGIN
IF e IS NULL THEN return NULL
ELSE return rcwe OF e
FI
END
```

Otherwise you have the indirect traversal via loop-edge links (assuming that they are in counter-clockwise order):

```
PROC rcwe(EDGE e)
BEGIN
EDGE rese = NULL;
L-E LINK rlink, xlink;

IF e IS NULL THEN rlink = NULL ELSE rlink = rlink OF e;
IF rlink ISNT NULL
THEN
xlink = prev OF rlink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

Alternatively there is the version with vertex-edge links (assuming that the links are arranged counter-clockwise around the vertex):

```
PROC rcwe(EDGE e)
BEGIN
EDGE rese = NULL;
```

```
V-E LINK elink, xlink;

IF e IS NULL THEN elink = NULL ELSE elink = end OF e;
IF elink ISNT NULL
THEN
xlink = next OF rlink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

B.2.6 rcce

The right counter-clockwise edge. With the direct pointer variant, this is:

```
PROC rcce(EDGE e)
BEGIN
IF e IS NULL THEN return NULL
ELSE return rcce OF e
FI
END
```

The indirect traversal via loop-edge links (assuming that they are in counter-clockwise order) is:

```
PROC rcce(EDGE e)
BEGIN
EDGE rese = NULL;
L-E LINK rlink, xlink;

IF e IS NULL THEN rlink = NULL ELSE rlink = rlink OF e;
IF rlink ISNT NULL
THEN
xlink = next OF rlink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

Or the version with vertex-edge links (assuming that the links are arranged counter-clockwise around the vertex) is:

```
PROC rcce(EDGE e)
BEGIN
EDGE rese = NULL;
V-E LINK elink, xlink;

IF e IS NULL THEN elink = NULL ELSE elink = start OF e;
IF elink ISNT NULL
THEN
xlink = prev OF elink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

B.2.7 lcwe

The left clockwise edge. With the direct pointer variant, this is:

```
PROC lcwe(EDGE e)
BEGIN
IF e IS NULL THEN return NULL
ELSE return lcwe OF e
FI
END
```

The indirect traversal via loop-edge links (assuming that they are in counter-clockwise order) is:

```
PROC lcwe(EDGE e)
BEGIN
EDGE rese = NULL;
L-E LINK llink, xlink;

IF e IS NULL THEN llink = NULL ELSE llink = llink OF e;
IF llink ISNT NULL
THEN
xlink = prev OF llink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

Or the version with vertex-edge links (assuming that the links are arranged counter-clockwise around the vertex) is:

```
PROC lcwe(EDGE e)
BEGIN
EDGE rese = NULL;
V-E LINK elink, xlink;

IF e IS NULL THEN elink = NULL ELSE elink = start OF e;
IF elink ISNT NULL
THEN
xlink = next OF elink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

B.2.8 lcce

The left counter-clockwise edge. With the direct pointer variant, this is:

```
PROC lcce(EDGE e)
BEGIN
IF e IS NULL THEN return NULL
ELSE return lcce OF e
FI
END
```

The indirect traversal via loop-edge links (assuming that they are in counter-clockwise order):

```
PROC lcce(EDGE e)
BEGIN
EDGE rese = NULL;
L-E LINK llink, xlink;
IF e IS NULL THEN llink = NULL ELSE llink = llink OF e;
IF llink ISNT NULL
THEN
xlink = next OF llink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI
```

```
return rese
END
```

Or the version with vertex-edge links (assuming that the links are arranged counter-clockwise around the vertex):

```
PROC lcce(EDGE e)
BEGIN
EDGE rese = NULL;
V-E LINK elink, xlink;

IF e IS NULL THEN elink = NULL ELSE elink = end OF e;
IF elink ISNT NULL
THEN
xlink = prev OF elink;
IF xlink IS NULL THEN rese = NULL ELSE rese = edge OF xlink FI;
FI

return rese
END
```

B.3 Single element traversals

B.3.1 vopev

Find the vertex opposite a given vertex along a given edge.

```
PROC vopev(EDGE e, VERTEX v)
BEGIN
VERTEX sv, ev, resv = NULL;

IF e IS NULL
THEN sv = ev = NULL;
ELSE sv = svert(e); ev = evert(e);
FI;
IF v IS sv THEN resv = ev;
ELSE IF v IS ev THEN resv = sv;
ELSE error("vopev - unconnected edge and vertex");
FI;

return resv;
END;
```

B.3.2 lopel

Find the loop opposite a given loop across a given edge.

```
PROC lopel(EDGE e, LOOP l)
BEGIN
LOOP rl, ll, resl = NULL;

IF e IS NULL
THEN rl = ll = NULL;
ELSE rl = rloop(e); ll = lloop(e);
FI;
IF l IS rl THEN resl = ll;
ELSE IF l IS ll THEN resl = rl;
ELSE error("lopel - unconnected edge and loop");
FI;

return resl;
END;
```

B.3.3 ecwel

Find the edge clockwise around a given loop from a given edge.

```
PROC ecwel(EDGE e, LOOP l)
BEGIN
EDGE rese = NULL;

IF (e IS NULL) OR (l IS NULL)
THEN rese = NULL;
ELSE IF rloop(e) IS l THEN rese = rcwe(e);
ELSE IF lloop(e) IS l THEN rese = lcwe(e);
ELSE error("ecwel - unconnected edge and loop");
FI;

return rese
END
```

B.3.4 eccel

Find the edge counter-clockwise around a given loop from a given edge.

```
PROC eccel(EDGE e, LOOP l)
BEGIN
```

```
EDGE rese = NULL;

IF (e IS NULL) OR (l IS NULL)
THEN rese = NULL;
ELSE IF rloop(e) IS l THEN rese = rcce(e);
ELSE IF lloop(e) IS l THEN rese = lcce(e);
ELSE error("eccel - unconnected edge and loop");
FI;

return rese
END
```

B.3.5 vcwel

Find the vertex clockwise around a given loop from a given edge.

```
PROC vcwel(EDGE e, LOOP l)
BEGIN
VERTEX resv = NULL;

IF (e IS NULL) OR (l IS NULL)
THEN rese = NULL;
ELSE IF rloop(e) IS l THEN resv = evert(e);
ELSE IF lloop(e) IS l THEN resv = svert(e);
ELSE error("vcwel - unconnected edge and loop");
FI;

return rese
END
```

B.3.6 vccel

Find the vertex counter-clockwise around a given loop from a given edge.

```
PROC vccel(EDGE e, LOOP l)
BEGIN
VERTEX resv = NULL;

IF (e IS NULL) OR (l IS NULL)
THEN rese = NULL;
ELSE IF rloop(e) IS l THEN resv = svert(e);
ELSE IF lloop(e) IS l THEN resv = evert(e);
ELSE error("vccel - unconnected edge and loop");
```

```
FI;

return rese
END
```

B.3.7 ecwev

Find the edge clockwise around a given vertex from a given edge.

```
PROC ecwev(EDGE e, VERTEX v)
BEGIN
EDGE rese = NULL;

IF (e IS NULL) OR (v IS NULL)
THEN rese = NULL;
ELSE IF svert(e) IS v THEN rese = rcce(e);
ELSE IF evert(e) IS v THEN rese = lcce(e);
ELSE error("ecwev - unconnected edge and vertex");
FI;

return rese
END
```

B.3.8 eccev

Find the edge counter-clockwise around a given vertex from a given edge.

```
PROC eccev(EDGE e, VERTEX v)
BEGIN
EDGE rese = NULL;

IF (e IS NULL) OR (v IS NULL)
THEN rese = NULL;
ELSE IF svert(e) IS v THEN rese = lcwe(e);
ELSE IF evert(e) IS v THEN rese = rcwe(e);
ELSE error("eccev - unconnected edge and vertex");
FI;

return rese
END
```

B.3.9 lcwev

Find the loop such that the given edge is clockwise around that loop from the given vertex.

```
PROC lcwev(EDGE e, VERTEX v)
BEGIN
LOOP resl = NULL;

IF (e IS NULL) OR (v IS NULL)
THEN rese = NULL;
ELSE IF svert(e) IS v THEN resl = rloop(e);
ELSE IF evert(e) IS v THEN resl = lloop(e);
ELSE error("lcwev - unconnected edge and vertex");
FI;

return resl
END
```

B.3.10 lccev

Find the loop such that the given edge is counter-clockwise around that loop from the given vertex.

```
PROC lccev(EDGE e, VERTEX v)
BEGIN
LOOP resl = NULL;

IF (e IS NULL) OR (v IS NULL)
THEN rese = NULL;
ELSE IF svert(e) IS v THEN resl = lloop(e);
ELSE IF evert(e) IS v THEN resl = rloop(e);
ELSE error("lcwev - unconnected edge and vertex");
FI;

return resl
END
```

B.3.11 ecwlv

Find the edge clockwise from a given vertex around a given loop.

```
PROC ecwlv(LOOP l, VERTEX v)
BEGIN
```

```
LOOP rese = NULL;

    PROC cwe(EDGE e)
IF lcwev(e, v) IS l
THEN rese = e; return TRUE
ELSE return FALSE
FI

all_edges_in_vertex(v, cwe);

return rese
END
```

B.3.12 ecclv

Find the edge counter-clockwise from a given vertex around a given loop.

```
PROC ecclv(LOOP l, VERTEX v)
BEGIN
LOOP rese = NULL;

    PROC cce(EDGE e)
IF lccev(e, v) IS l
THEN rese = e; return TRUE
ELSE return FALSE
FI

all_edges_in_vertex(v, cce);

return rese
END
```

B.3.13 eclev

Find the edge clockwise or counter-clockwise from a given edge around a loop in the direction of a given vertex.

```
PROC eclev(LOOP l, EDGE e, VERTEX v)
BEGIN
VERTEX resv = NULL;

IF (e IS NULL) OR (l IS NULL) OR (v IS NULL)
THEN rese = NULL;
```

```
ELSE IF vcwel(e, l) IS v THEN rese = ecwel(e, l);
ELSE IF vccel(e, l) IS v THEN rese = eccel(e, l);
ELSE error("eclev - unconnected edge, vertex and loop");
FI;

return rese
END
```

B.3.14 vve

Find the edge connecting two given vertices (if any).

```
PROC vve(VERTEX v1, VERTEX v2)
BEGIN
LOOP rese = NULL;

    PROC opv(EDGE e)
IF vopev(e, v1) IS v2
THEN rese = e; return TRUE
ELSE return FALSE
FI

all_edges_in_vertex(v1, opv);

return rese
END
```

Appendix C

Topological utilities

C.1 Creating and deleting entities

Exactly how entities are chained together in the datastructure is not fixed because of the variable nature of the connections. When creating entities, it is useful to add them to a structure so that they are not left hanging. For example, new edges in an object can be created and immediately added to the 'all edges in body' chain. The topological connections are made elsewhere, by the Euler operators (described in section 3.3) or by the modelling operators. Another decision that has to be made concerns how dynamic allocation is handled. If a modeller includes a memory handling mechanism using 'free-lists', or lists of deleted entities, then these have to be handled by such creation and deletion utilities.

The general form of creation is something like:

```
EDGE make_edge(OBJECT b)
BEGIN
EDGE rese, fe, le;
IF ( edge_free_list IS NIL )
THEN rese = new EDGE;
ELSE BEGIN
rese = edge_free_list; edge_free_list = next OF rese;
END
number OF rese = edge_number; edge_number = edge_number + 1;
 fe = edge OF b;


IF ( fe IS NIL )
THEN BEGIN
edge OF b = rese; next OF rese = prev OF rese = rese;
END
```

```
ELSE BEGIN
le = prev OF fe;
prev OF rese = le; next OF le = rese;
next OF rese = fe; prev OF fe = rese;
END



return rese; END
```

Where 'OBJECT' covers the three forms: VOLUME, SHELL, or WIRE-FRAME.

The general form of deletion is something like:

```
void kill_edge(EDGE e)
BEGIN
EDGE ne, pe;
OBJECT b;

ne = next OF e; pe = prev OF e; b = owner OF e;
IF ( (ne IS e) AND (pe IS e) ) edge OF b = NIL;
ELSE BEGIN next OF pe = ne; prev OF ne = pe END

next OF e = edge_free_list; edge_free_list = e;
END
```

This is roughly the same for the other creation routines that are linked into lists, like the vertices or faces in a shell. There is, though, a slight weakness in the kill_vertex routine because there is no direct pointer from the vertex to the object so this has to be provided as a parameter. There is, therefore, a potential source of error, but it should not be critical.

C.2 Moving entities to new owners

These perform the same sort of 'housekeeping' operation as the above, except for moving them between owners.

The routine to move an edge to a new object might be:

```
void move_edge(EDGE e, OBJECT nb)
BEGIN
OBJECT ob = owner OF e;
EDGE ne, pe;

ne = next OF e; pe = prev OF e; b = owner OF e;
IF ( (ne IS e) AND (pe IS e) ) edge OF b = NIL;
```

```
ELSE BEGIN next OF pe = ne; prev OF ne = pe END

ne = edge OF nb;
IF ( ne IS NIL )
THEN BEGIN
edge OF nb = e; next OF e = prev OF e = e;
END
ELSE BEGIN
pe = prev OF ne;
prev OF e = pe; next OF pe = e;
next OF e = ne; prev OF ne = e;
END
END
```

Again there is the same slight weakness in the move_vertex routine that the object from which the vertex is to be moved has to be given explicitly.

Some other move operations might be:

- Face to shell
- Loop to face
- Edge to vertex
- Shell to object

It is also useful to have some operations to simply disconnect elements so that they can be separated from the chains.

- Face from shell
- Loop from face
- Edge from vertex
- Shell from object

C.3 Separate sequence of edges into new loop

This is used for partitioning portions of loops after a new edge has been added. It is necessary to traverse one portion, adjusting the loop pointers of the connected edges to point to the new loop. This process traverses the new loop portion in the same way as the 'for all edges in loop' traversal above.

There are different ways to do this according to the connection method chosen. It is easier with loop-edge links than with winged-edge pointers.

The starting point for the operation is the new edge, because it is an edge that lies between the two loops and can be used as a starting point for both

loops. If winged-edge pointers are used, then the traversal is similar to that given in Appendix B, but care has to be taken about handling the new edge, which appears as a wire edge.

```
current = start = rlink OF edge;
UNTIL ( current = NIL )
BEGIN
loop OF current = newloop;
current = next OF current;
IF ( current IS start ) current = NIL
END;
```

C.4 Separate a sequence of edges belonging to a new vertex

This is used for separating edges around a vertex when splitting the vertex, which is analogous to the way edges are separated into new loops above. It uses the same sort of traversal as the 'for all edges in vertex' traversal above.

The first edge to be moved is a parameter, 'start edge' and the last edge, 'end edge' to be moved is also necessary, as are the old vertex (overt) and the new vertex (nvert).

```
current = start edge;
UNTIL ( current = NIL )
BEGIN
next = eccev(current, overt);
IF ( start OF current IS overt ) THEN start OF current = nvert;
ELSE IF ( end OF current IS overt ) THEN end OF current = nvert;
ELSE error!
current = next;
IF ( current IS end edge ) current = NIL
END;
```

C.5 Separate a set of adjacent faces, edges, and vertices into a new shell

This process is used to separate portions of objects once they have been topologically separated. It is needed to pull apart the lists of entities in an object. It uses the 'for all face edges vertices in shell' traversal in section B.1.4 to access the connected entities and the entity moving routines.

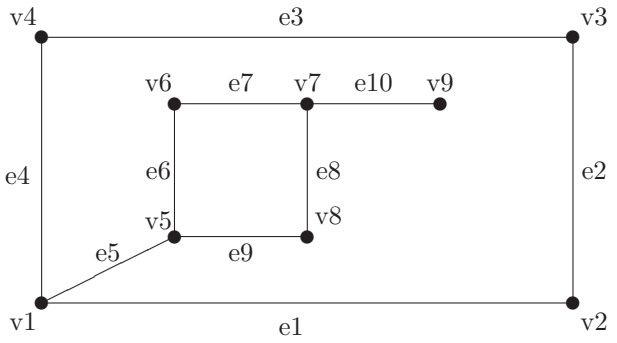


Figure C.1: Illustration of special topological elements (wires, spurs)

C.6 Check whether an edge is a wire edge

This process is for checking edges in Eulerian objects. The condition for an edge to be a wire edge is that it refers to the same loop as both left and right loop. Edge 5 in figure C.1 is a wire edge. It is important to know whether edges are wire-edges, for traversing edge sequences, for the Euler operators, and for the modelling algorithms.

The check is relatively simple:

IF (rloop(e) IS lloop(e)) THEN wire

C.7 Check whether an edge or vertex is a spur

A spur vertex is a vertex with only one edge attached. A spur edge is one where one or both end vertices are spur vertices, i.e., the edge is the only edge at its start or its end vertex. In consequence, the winged-edge pointers of the edge refer to the edge itself. Edge 10 in figure C.1 is a spur edge, and vertex 9 is a spur vertex.

IF (wire(edge) AND
(ecwev(edge, start OF edge) IS edge) OR
(ecwev(edge, end OF edge) IS edge)) THEN spur edge

IF (ecwev(edge OF vertex, vertex) IS edge OF vertex)
THEN spur vertex

C.8 Check whether a loop is a hole-loop

Checks whether the loop is a hole loop in the datastructure. This is not necessarily a geometric check. If the status is recorded in the datastructure then it is merely a check of the status of the loop. If this is not recorded, then it is necessary to check the relative sizes of the loops in the face. The process of checking is described in chapter 14. Recapitulating briefly, the loops of a face are sorted in descending order of size and measured using a bounding box. The largest is taken as the exterior loop; others are taken as interior loops. It is possible to perform a full test by intersecting all edges in the face to check for overlaps, or just testing a point on the loop to see if it is in the face. Note the special case of cylinders and cones, for example, where a face without fake edges may have two outer boundaries.

C.9 Checks for special object types

Two interesting special object types are minimum Eulerian objects and objects with no faces, edges, or vertices. A minimum Eulerian object is one with a single vertex, a single face, and no edges. An object with no faces, edges, or vertices can be a degenerate result that should be deleted.

C.10 Copy an object

This process is used, for example, when reflecting an object and in Booleans for copying multiply instanced objects, and it is useful as a user tool. The method for copying an object is similar to that used to write out the object to a discfile, as described in chapter 11.

First of all, all elements in the datastructure are counted and put into lists of each entity type. Then, another set of lists is created with the same size as the corresponding lists of the original elements, and these are filled with new entities of the corresponding types.

The general procedure for copying an entity, in this case, a face with a slightly simplified datastructure, is shown in figure C.2.

The structure is shown at the top left. It is used to interpret the elements of the face. An example of an old face is shown at the bottom left. The first value, 27, is an integer representing the number of the face. It is an identity number and is not copied. The identity number should be assigned automatically when an entity is created and not changed, except, possibly, temporarily in the copying and disc-writing operation described in chapter 11.

The next value is a shell pointer. The address of this shell in the list of shells is found, in this case 0. It is treated as a logical pointer and used to refer to the new lists. The address of the shell in location 0 of the list of new shells is written into the corresponding element of the new face. The next

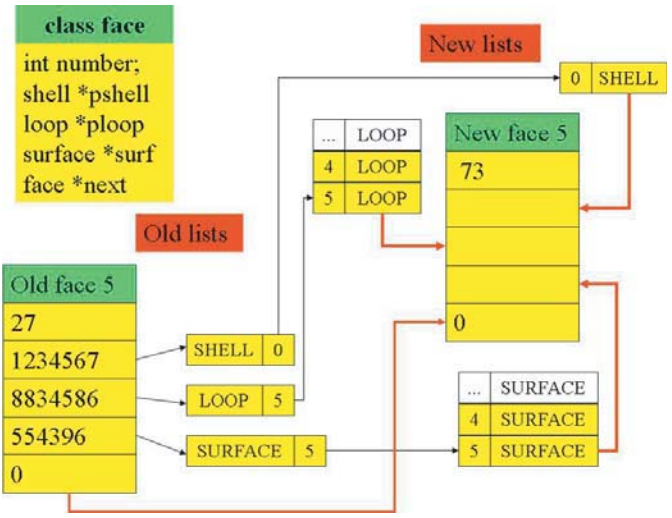


Figure C.2: Copying an entity

element is a loop pointer. This loop is loop 5 in the old lists, so a pointer to the fifth loop in the list of new loops is written into the third element of the new face. The same method is used to deal with the surface pointer. The final element, the next pointer, is NIL, so a NIL pointer is written into the corresponding element of the new face.

With slight variations, this is also the basic method for creating the dual of an object, except that some switching around is needed.

C.11 Modify an object with a transformation

This process uses the geometry modification utility described in the next section but is responsible for coordinating the changes for the whole object. As described in section 3.2.1, curve, surface (and possibly coordinate) geometry can be shared within an object. This utility is responsible for modifying each geometric entity once only and for handling the spatial approximation measures, bubbles, face boxes, and so on.

Appendix D

Geometrical utilities

The following appendix provides an overview of some aspects of the geometric utilities needed for a solid modeller. Note that this appendix is intended to give a brief outline to the work of the geometric part of the modeller. Creating a geometric package is a complex task, so it is only dealt with sketchily here.

D.1 Geometric intersection package

To intersect two complete geometric entities of the general classes: Point, Curve or Surface. This is a much used facility, necessary for many basic modelling operations. The six possible combinations are as follows:

Point–point intersection	Effectively, "check if two points are coincident"
Point–curve intersection	Check whether a point lies on a curve. This could be combined with tool 5 (see later) to return the parameter value (or values) of the point if it lies on the curve. This is a practical question, so the tools are differentiated here.
Point–surface intersection	Check whether a point lies on a surface. As above, this could return the result in the form of the parameter values of the point if it lies on the surface, but the tools are differentiated here.
Curve–curve intersection	Check whether two curves cut each other at points, are coincident or are partially coincident.

Continued on next page

Curve–surface intersection	Check whether a curve lies on a surface, cuts through it or is partially coincident with it.
Surface–surface intersection	Check whether two surfaces cut each other along a curve, touch at a point, and are coincident or partially coincident.

The interface sorts out which intersections to use from the format of the geometry. It looks at the geometry types and decides whether, say, a cylinder–cylinder intersection is required, a circle–plane intersection, and so on. A delimiting measure (sphere or box, as mentioned in section 3.1.2) is needed when approximations to infinite curves need to be calculated.

In what form the results are returned is not fixed. They could be given as a set of geometry, points, curves, or surfaces, but extra information can be provided as well. A convenient result set is as follows:

- COINCIDENCE (same orientation)
- COINCIDENCE (reverse orientation)
- POINT
- CURVE
- SELF-INTERSECTING CURVE
- SURFACE

or a set of these. Partial coincidence can be handled by returning a new geometric entity (curve or surface) representing the coincident subpart of the curve or surface.

The first task of the intersection package is to separate the different geometric types. If, say, the set of surfaces is:

PLANE
CYLINDER
CONE
SPHERE
TORUS
BSPATCH

and a set of curves:

STRAIGHT
CIRCLE
ELLIPSE
PARABOLA
HYPERBOLA
BSPLINE

it is necessary to create intersection routines for each pair.

The pseudo-code for surface–surface intersection looks something like:

```
intersect_surfaces(SURFACE s1, SURFACE s2)
BEGIN
CASE (type OF s1) IN
(PLANE p1)
CASE (type OF s2) IN
(PLANE p2) pla_pla_int(p1, p2)
(CYLINDER c2) pla_cyl_int(p1, c2)
(CONE c2) pla_con_int(p1, c2)
(SPHERE s2) pla_sph_int(p1, s2)
(TORUS t2) pla_tor_int(p1, t2)
(BSPATCH b2) pla_bsp_int(p1, b2)
ESAC
(CYLINDER c1)
CASE (type OF s2)IN
(PLANE p2) pla_cyl_int(p2, c1)
(CYLINDER c2) cyl_cyl_int(c1, c2)
(CONE c2) cyl_con_int(c1, c2)
(SPHERE s2) cyl_sph_int(c1, s2)
(TORUS t2) cyl_tor_int(c1, t2)
(BSPATCH b2) cyl_bsp_int(c1, b2)
ESAC
(CONE c1)
CASE (type OF s2) IN
(PLANE p2) pla_con_int(p2, c1)
(CYLINDER c2) cyl_con_int(c2, c1)
(CONE c2) con_con_int(c1, c2)
(SPHERE s2) con_sph_int(c1, s2)
(TORUS t2) con_tor_int(c1, t2)
(BSPATCH b2) con_bsp_int(c1, b2)
ESAC
(SPHERE s1)
CASE (type OF s2) IN
(PLANE p2) pla_sph_int(p2, s1)
(CYLINDER c2) cyl_sph_int(c2, s1)
(CONE c2) con_sph_int(c2, s1)
```

```
(SPHERE s2) sph_sph_int(s1, s2)
(TORUS t2) sph_tor_int(s1, t2)
(BSPATCH b2) sph_bsp_int(s1, b2)
ESAC
(TORUS t1)
CASE (type OF s2) IN
(PLANE p2) pla_tor_int(p2, t1)
(CYLINDER c2) cyl_tor_int(c2, t1)
(CONE c2) con_tor_int(c2, t1)
(SPHERE s2) sph_tor_int(s2, t1)
(TORUS t2) tor_tor_int(t1, t2)
(BSPATCH b2) tor_bsp_int(t1, b2)
ESAC
(BSPATCH b1)
CASE (type OF s2) IN
(PLANE p2) pla_bsp_int(p2, b1)
(CYLINDER c2) cyl_bsp_int(c2, b1)
(CONE c2) con_bsp_int(c2, b1)
(SPHERE s2) sph_bsp_int(s2, b1)
(TORUS t2) tor_bsp_int(t2, b1)
(BSPATCH b2) bsp_bsp_int(b1, b2)
ESAC
ESAC
END
```

This separates the generic type, surface, into specific cases that can be handled numerically. The same process applies for the curve–surface and curve–curve intersections.

It is obvious that having many surface types increases the work of actually writing the routines. However, the big advantage of having so many different routines is that special cases can easily be identified. This was the approach developed by Pfeifer for the GPM volume module. These special cases may also be numerically unstable, so having analytic solutions instead of relying on numeric procedures can add stability, as for the medial axis point calculation described in chapter 15.

Another advantage of using explicit geometry is that some infinite curves can be calculated explicitly. If a marching method is used, then there has to be a method of limiting the domain of interest. This is done using bounding boxes or bubbles, for example, to limit the area of interest for the intersection routines. Such limiting entities should always be included as input parameters to intersection routines.

With the routines above several special conditions can be identified. First, though, a small divergence, because the line–plane intersection and line–line intersection operations are also used for surface–surface intersections.

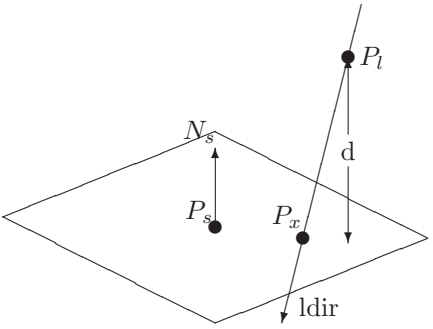


Figure D.1: Line–plane intersection

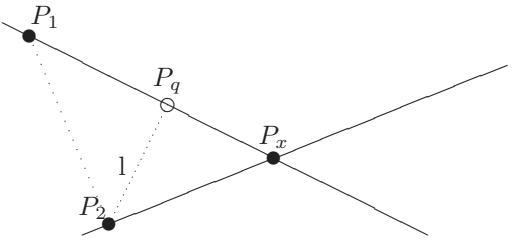


Figure D.2: Line–line intersection

The line–plane intersection (figure D.1) is:

$$P_x = P_l + ldir * (d/(N_s \bullet ldir))$$
$$d = (P_s - P_l) \bullet N_s$$

where P_x is the intersection point of the line and the plane. This is for planes and lines where the line is not perpendicular to the plane normal.

The line–line intersection (figure D.1) is:

$$P_q = P_1 + ((P_2 - P_1) \bullet d_1) * d_1$$

where d_1 is the normalised direction of line 1.
$P_x = P_2 + d_2 * l/((P_q - P_2) \bullet d_2)$

where d_2 is the normalised direction of line 2.

A check needs to be made that the lines are not parallel and that P_2 is not on the first line.

The method to find the closest points on two lines is related to the line–line intersection and is also useful for intersection operations.

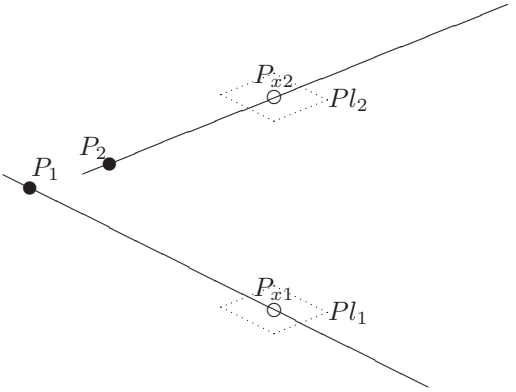


Figure D.3: Line–line closest points

Assuming that the directions of the lines are d_1 and d_2, the cross product, $d_1 \times d_2$ gives the normal of the parallel planes, the distance between which is the distance between the lines. If the length of this vector is zero, then the lines are parallel or coincident. The closest point pair could be, say, P_1 and the closest point to that on the second line.

If the normal vector is zero, then the lines are parallel. If the normal vector is non-zero, then there are two planes, Pl_1 containing line 1 and Pl_2 containing line 2. The intersection of line 1 and the projection of line 2 into plane 1 gives the point on line 1 closest to line 2, and the projection of this point into plane 2 gives the point on line 2, which is the closest to line 1.

D.1.1 Planar intersections

Most special cases come with planar intersections because the analytic curve types mentioned above are planar. These are summarised in this section.

pla_pla_int(p1, p2)

If the normal vectors of the two planes are parallel and the distances of the point defining the first plane lies in the second plane, then the planes are coincident. If the normal vectors are in the opposite direction, then the planes are reverse coincident.

Otherwise the intersection of the two planes gives a straight line. Calculation of this line is relatively straightforward. The cross product of the two normals gives the direction of the line. The cross product of this line direction with the normal N_2 gives the direction of a line in plane 2. The point of the result line is the intersection of the line in plane 2 with the first plane.

Try this as an alternative explanation:

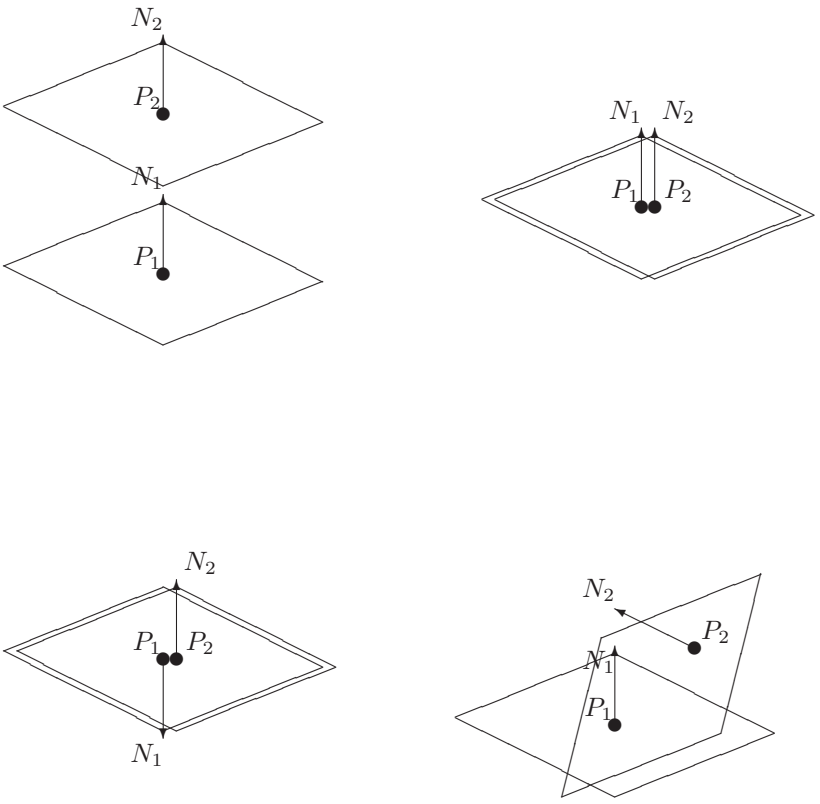


Figure D.4: Plane–plane intersection

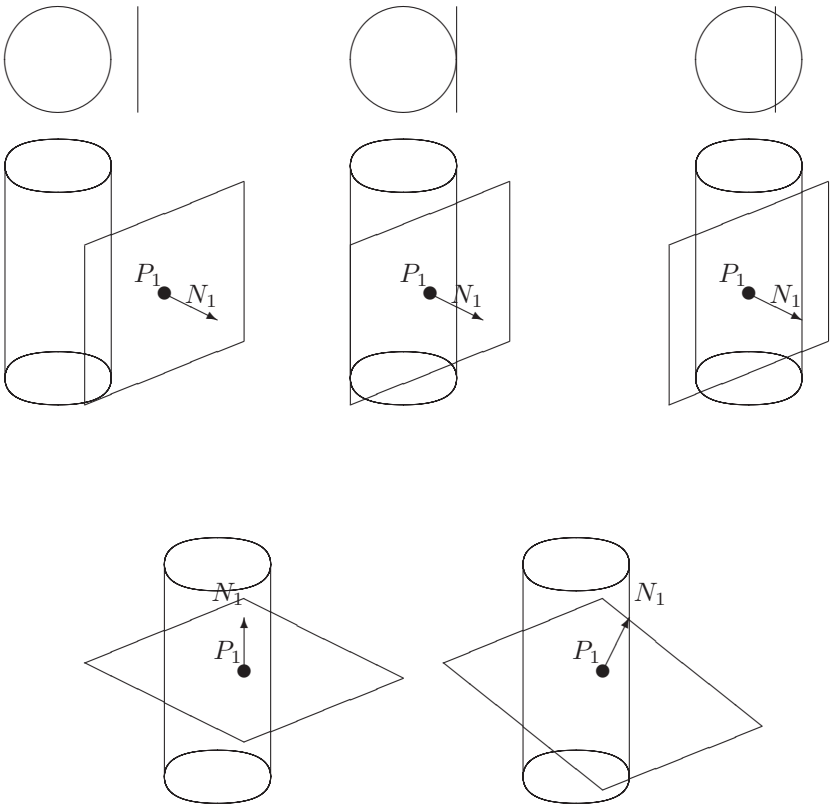


Figure D.5: Plane–cylinder intersection

$resdir = N_1 \times N_2$
$lxdir = N_2 \times resdir$
$respnt = int\_lin\_plan(P_2, lxdir, P_1, N_1)$

pla_cyl_int(p1, c2)

The first special case is if the plane normal is perpendicular to the cylinder axis. In this case, if the distance of the axis to the plane is more than the radius of the cylinder, then there is no intersection. If the distance of the axis to the plane is equal to the radius of the cylinder, then the intersection is a single straight line. Otherwise the intersection result is two straight lines.

If the plane normal is not perpendicular to the cylinder axis, then the plane cuts the cylinder somewhere. The intersection of the axis with the plane gives the centre of an ellipse or circle. The result of the intersection is a circle, with a radius equal to that of the cylinder, if the plane normal vector is parallel

to the cylinder axis. If the result is an ellipse, then the the short axis of the ellipse is given by $normXCA$ where norm is the plane normal and CA is the cylinder axis. The short axis length is the same as the cylinder diameter, and the long-axis length is the cylinder diameter divided by the cosine of the angle between the cylinder axis.

pla_con_int(p1, c2)

The first point to make is that a cone extends above and below the crossing point along its axis. It may be preferable, though, to restrict the cone to a half shape. The following assumes a half cone.

If the cone apex is on the plane, then the result depends on the angle between the normal and the cone axis. If 90 degrees minus this angle is less than the cone half angle, then the result is two lines. If 90 degrees minus this angle is the same as the cone half angle, then the plane is tangent to the cone and the result is one line. Otherwise the result is the apex point of the curve.

If the cone apex does not lie on the plane, then the angle between the cone axis and the plane normal is again used to distinguish between the different cases. If the angle is greater than the cone half angle, then the result is a parabola, the focus being the intersection point of the cone axis and the plane. If the angle is equal to the half angle, then the result is a hyperbola.

If the angle between the plane normal vector and the cone axis is less than the cone half angle, then the plane may not cut a half cone. It is first necessary to check the intersection point of the plane and the cone axis to see whether this is above the cone apex. Otherwise, if a double cone is used or if the intersection point is below the cone apex, then a circle or ellipse is the result. The result is a circle if the plane normal vector is parallel to the cone axis.

pla_sph_int(p1, s2)

There are three cases to check, depending on the distance of the plane from the sphere centre. If the distance of the sphere centre to the plane is greater than the sphere radius, then there is no intersection. If the distance is equal to the sphere radius, then a point is the result. Otherwise a circle is the result.

pla_tor_int(p1, t2)

The first thing to note about a torus is that there are three types: the normal torus, the lemon torus, and the apple torus (Solomon's classification). These types are illustrated in cross-section in figure D.7. In the normal torus, the major radius is greater than the minor radius. For the apple torus and lemon torus, the minor radius is greater than the major radius. The difference between them is which part is used and this can be determined by a flag, for example.

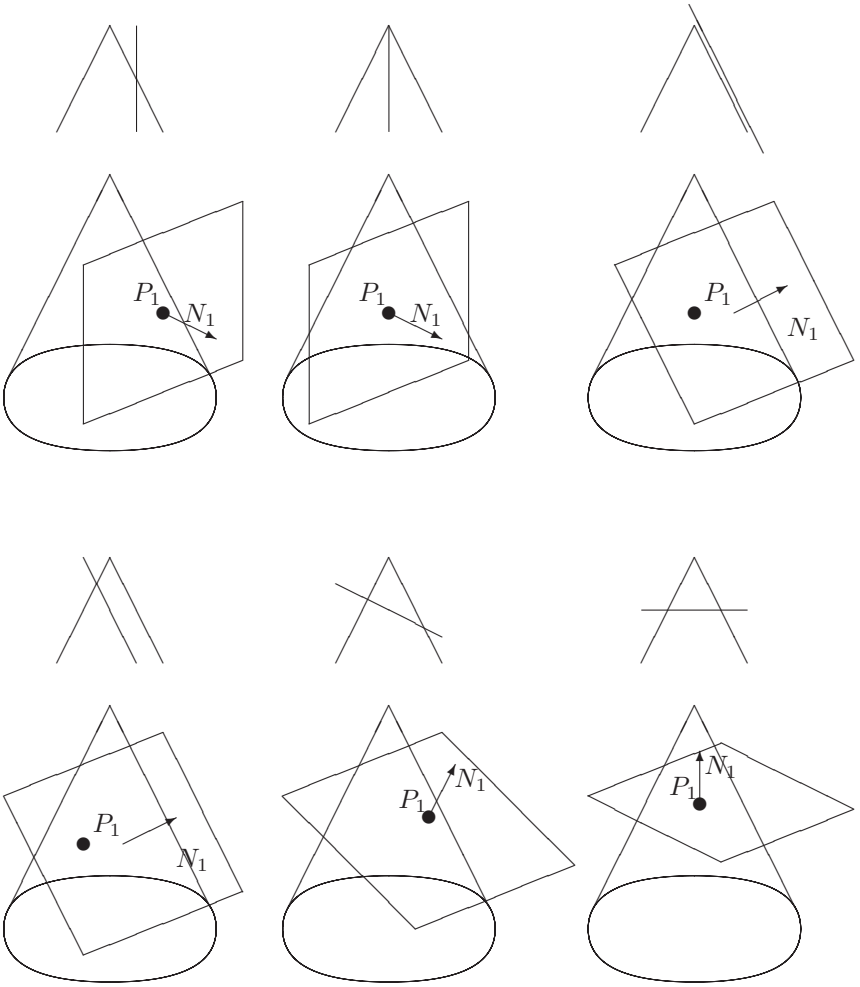


Figure D.6: Plane–cone intersection

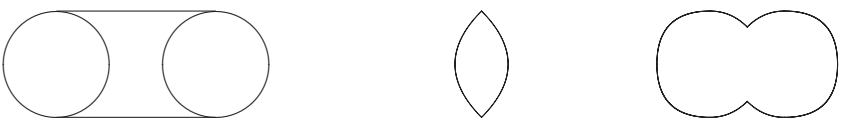


Figure D.7: Torus types

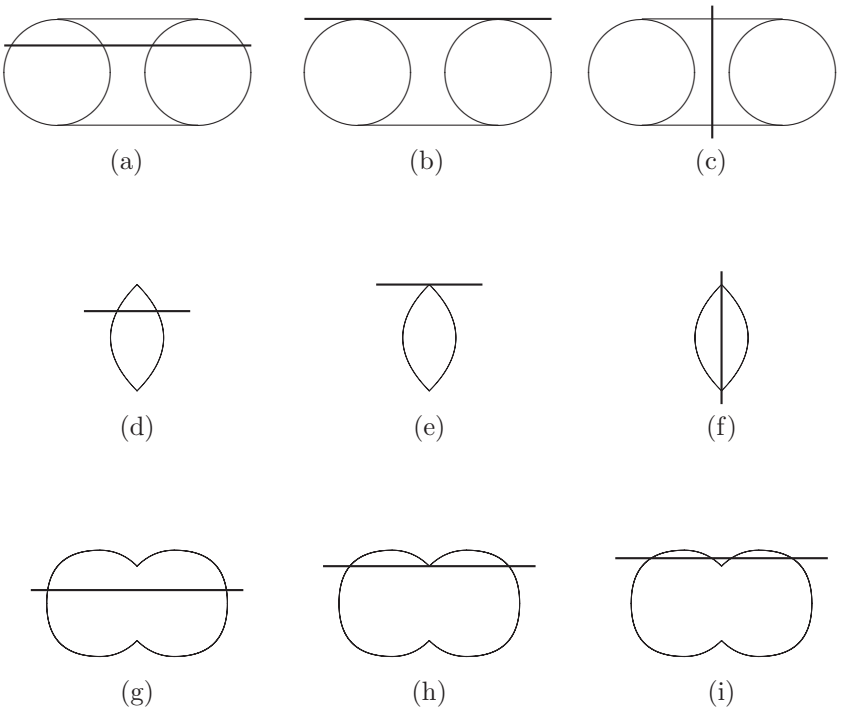


Figure D.8: Torus intersections

There are fewer special cases of which to take advantage. These are summarised in figure D.8

In figure D.8a, there are two horizontal rings inside each other, centred where the torus axis intersects the plane.

In figure D.8b, there is one horizontal ring with radius 'major_radius' of the torus.

In figure D.8c, there are two vertical rings with radius 'minor_radius' of the torus.

In figure D.8d, there is one horizontal ring.

In figure D.8e, there is a single point.

In figure D.8f, there are two circular arcs.

In figure D.8g, there is one horizontal ring.

In figure D.8h, there is a horizontal ring and a point.

In figure D.8i, there are again two horizontal rings inside each other.

For the apple torus, there are the other two cases, where there is a single ring and where there are two circular arcs.

Other intersections use the general solution where a start point is calculated, or start points are calculated, and successive points are calculated.

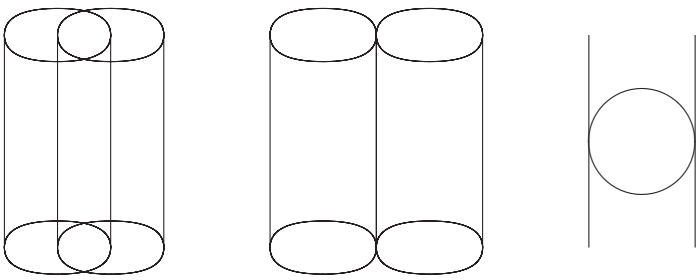


Figure D.9: Cylinder–cylinder intersection

Finally, one or more results are created by fitting a numerical curve or curves through the point set(s).

pla_bsp_int(p1, b2)

If all control points of the patch lie in the plane, then the plane and patch are coincident. It is necessary to check the normals to see whether they are in the same or reversed direction.

D.1.2 Cylinder intersections

This section summarises the special cases that can be identified with cylinder intersections.

cyl_cyl_int(c1, c2)

The cylinder–cylinder special cases are summarised in figure D.9.

In the left hand column, the cylinder–cylinder intersection results in two lines. In the middle column, the cylinders just touch and the intersection result is a single line. In the final special case, where the cylinders have the same radii and the axes are perpendicular and intersect, then the result is two ellipses. If the distance between the two axes is greater than the sum of the radii of the two cylinders, then there is no intersection. If the distance between the axes is exactly the sum of the radii and the axes are not parallel, then the intersection result is a point. Otherwise a general intersection procedure is used.

The general intersection procedure is to find a point on the two surfaces and then to determine a set of points on the intersection curve or curves by marching. A closest point can be found by 'relaxation'. First the closest points on the axes are found. The initial approximation to a point can be a point on one cylinder, in a direction perpendicular to the line joining the two

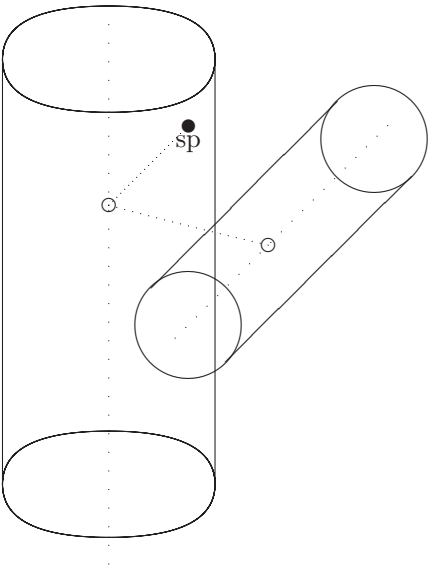


Figure D.10: Cylinder–cylinder intersection

closest axis points (figure D.10). This point is then projected onto the second cylinder for a second level approximation. This second level point is projected back to the first cylinder, and so on until a point is found that lies on both cylinders.

Each successive point on the intersection curve is then found by determining a nearby approximation point and then applying the same relaxation procedure to get the point on the intersection curve.

An analysis can also be used to help guide the general intersection procedure. There are some critical cases, though, which are summarised in figure D.11.

If the cylinders are sufficiently close to being tangential, then it is possible that the marching method may leap onto a different track. Analysing the cases can help to direct the marching method to choose the correct topology of the result curve.

cyl_con_int(c1, c2)

If the cylinder and cone axes are aligned, then there is a circular intersection result. If the cone is a half cone, then another special case is where the apex touches the cylinder, in which case, one result will be a point.

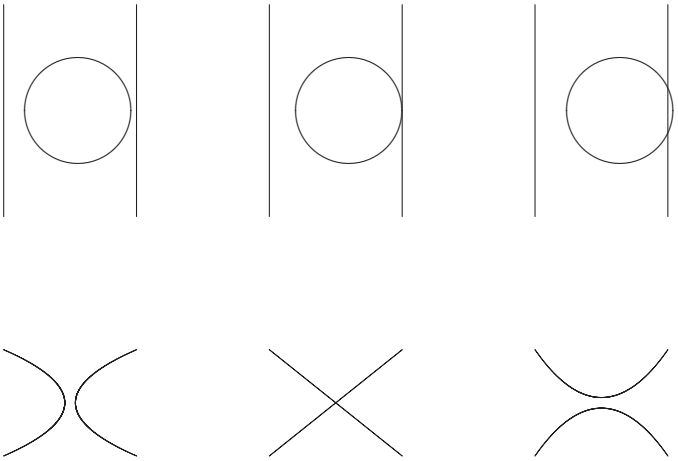


Figure D.11: Cylinder–cylinder intersection spline cases

cyl_sph_int(c1, s2)

If the sphere centre is on the cylinder axis, then the intersection will be a circle if the radius of the sphere is equal to or greater than the radius of the cylinder.

If the distance of the sphere centre from the cylinder axis is greater than the sum of the radii of the sphere and the cylinder, then there is no intersection. If the sum is equal to the distance between the sphere centre and the cylinder axis, then the result is a point. If the sphere radius is smaller than that of the cylinder and the distance from the cylinder axis to the sphere centre plus the radius of the sphere is equal to the radius of the cylinder, then a point is the result.

cyl_tor_int(c1, t2)

Some cases are shown in figure D.12.

If the cylinder axis is perpendicular to the torus axis and the distance between the cylinder axis and the centre of the torus is equal to the cylinder radius plus the minor radius of a normal torus or apple torus, then the cylinder and the torus share a common tangent plane. The torus profile in this plane is a circle and the cylinder profile a line. Intersecting the line and the circle gives the result.

A cylinder may touch the top of a lemon torus or the outside of a torus to give a point. It may also touch the inside of a normal torus.

There are the obvious checks for non-intersection, if the distance between the torus axis and the cylinder axis is greater that the sum of the minor radius, major radius, and cylinder radius. The torus will be inside the cylinder if the

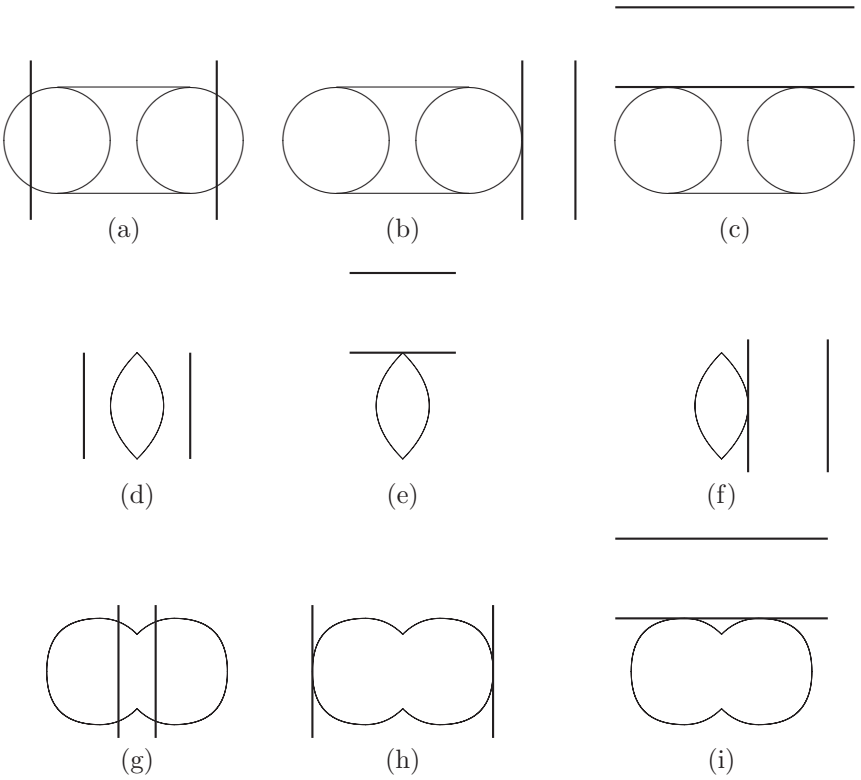


Figure D.12: Torus–cylinder intersections

axes are parallel and the distance of the torus centre plus the major and minor radii is less than the radius of the cylinder.

Figure D.12 summarises some special cases.

cyl_bsp_int(c1, b2)

No special cases to note. Using rational forms of numerical geometry it is possible to represent a cylinder, or other simple geometric forms, exactly so that there may be simplifications. However, finding out that a particular surface patch represents a simple geometric form is difficult. One possibility is to record the operations from which it is derived. For example, if a simple surface is changed to numerical form so that non-uniform scaling can be applied, then it would be possible to attach pointers to the original geometry and to the transformation matrix. In this way, as the transformation is changed, so the surface could be re-derived if necessary. This is not within the scope of this appendix, though.

D.1.3 Cone intersections

The cone intersections are summarised in this section.

con_con_int(c1, c2)

Some cone–cone intersections for special cases for half cones are shown in figure D.13. In the first column, there is no intersection, and in the middle column the cones intersect at a point. In the right-hand column, the cones intersect in a circle.

Two cones may also share a tangent plane; in which case, the result is a line. If the cones have the same vertex point, the same half-angle, and the same axis, then they are coincident.

con_sph_int(c1, s2)

If the sphere centre lies on the cone axis, then there are three cases, as summarised in figure D.14.

con_tor_int(c1, t2)

The cone torus intersection cases bear a certain similarity to the cylinder torus cases. The cone may share a tangent plane with the top of a normal or an apple torus, giving zero, one, or two points as a result. A cone may also touch a torus at a point.

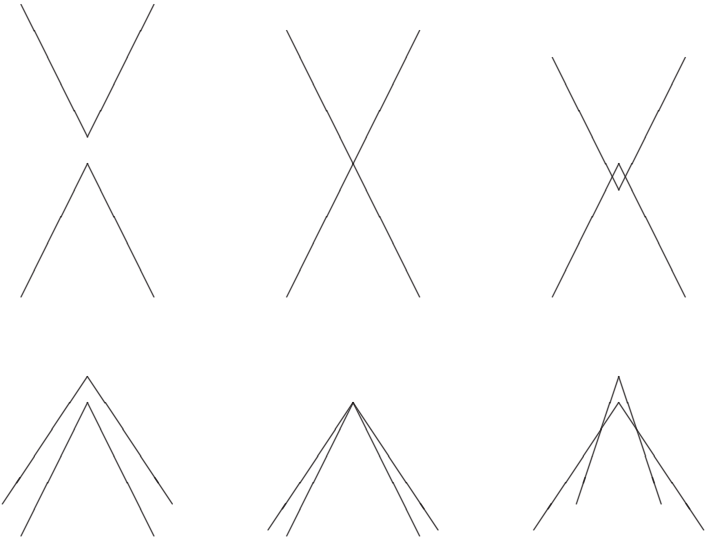


Figure D.13: Cone–cone intersection cases

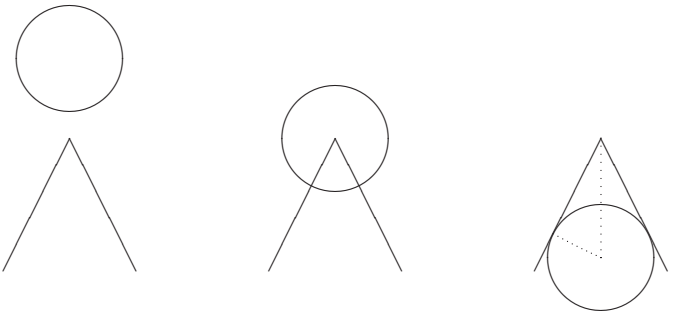


Figure D.14: Cone–sphere intersection cases

con_bsp_int(c1, b2)

No special cases to be identified. Note also, that, as with the cylinder numeric patch intersections, it is possible that a numeric patch represents exactly the same cone as that with which it is being intersected.

D.1.4 Sphere intersections

sph_sph_int(s1, s2)

If the distance between the sphere centres is greater than the sum of their radii, then no intersection results. If the sum is the same as the sum of the radii, then the spheres touch at a point. If the difference between the centres plus the smaller sphere radius is equal to the larger sphere radius, then the spheres also touch at a point. It is simple to test whether the spheres are coincident. Otherwise the spheres intersect in a circle.

sph_tor_int(s1, t2)

A sphere may touch a torus at a point. A sphere may also touch a torus along a circle.

sph_bsp_int(s1, b2)

No special cases to identify.

D.1.5 Torus intersections

Tori are complex, and there are not too many special cases to identify.

tor_tor_int(t1, t2)

Tori may be coincident. A lemon torus and an apple torus may touch at two points. Tori may touch at single points.

tor_bsp_int(t1, b2)

This is too complex for the scope of this appendix.

D.1.6 Numeric patch intersections

Numerical geometry is the catch-all, used to model all geometry types not explicitly represented. This makes it important but, at the same time, very complex. This is a specialised subject that is outside the scope of this appendix. See Hoschek and Lasser [61], for example, or other specialised texts.

D.2 Curve tangent

This calculates the tangent direction at a point or parameter value on a curve. A basic textbook such as that of Faux and Pratt [34] gives a comprehensive description of these basic geometric calculations.

Note that it is useful if the tangent vector is explicitly made to be in the direction of the curve at the calculation point as this aids determination of operations such as when intersecting a face with a curve.

Strictly speaking, if a point is given, then that point should be on the curve. One option for this function is to return the closest point on the curve and the tangent at that closest point. If a parameter value is given, then the point corresponding to that parameter value would be returned.

Again, the use of explicit geometry helps to divide the cases into simple cases.

Line tangent

This can be given directly from the line geometry regardless of the point or parameter value. If the line is specified by a point and a direction, then the tangent is just the normalised direction. If the line is specified by two points, then the tangent is just the normalised difference of the second minus the first.

The closest point P_q to a given point P_2 on a line specified by a point P_1 and a normalised direction d is given by:
$P_q = P_1 + ((P_2 - P_1) \bullet d) * d$

Circle tangent

For a point, the requirement is that the point is in the same plane but not at the circle centre. If the point is not on the plane then it can be projected onto the plane. Once the point is on the same plane as the circle, it is possible to project the point onto the circle and use the projected point for the tangent calculation.

The circle must have a direction. If a parameter is given, then it is necessary to have an appropriate circle parametrisation. The format in appendix A follows the GPM method and has three points: p_1, $arcpt$, and p_2. The direction of parametrisation is from p_1 through $arcpt$ to p_2. The first and last points are not allowed to be identical. The first point is taken as the zero parameter point. The parameter range is 0 to 2π.

The tangent is the vector, perpendicular to the vector from the circle centre point to the given point in the direction of the circle. Taking a parametrisation such as $x = rcos\theta$ and $y = rsin\theta$, the derivative of this gives $x = -rsin\theta$ and $y = rcos\theta$, where θ is the parameter varying from 0 to 2π.

This is for a circle centred at the origin so there is an implicit transformation to get the correct geometry. Using cpt as the centre point, found from the three points in the datastructure, the x-axis direction is given by $p_1 - cpt$, the normalisation of this vector is $xdir$, and its length is the radius r. The normalised vector $ydir$ is given by $cnormXxdir$, where cnorm is

the normal to the plane of the circle. This gives the tangent vector *tvec* as $tvec = -xdir * rsin\theta + ydir * rcos\theta$

The values of $cos\theta$ and $sin\theta$ can be calculated from the scalar and vector products of the x-direction and the vector from the circle centre to the given point.

Ellipse tangent

The ellipse calculation is similar to that of the circle. The three points in the format are endpoints of the major or minor axes. The first and third points define one axis. The closest point on this axis to the second point defines the centre of the ellipse.

The first and third points should be equidistant from the ellipse centre; if not, then the data are not as defined. Actually the third point need only be on the axis and the first point at the endpoint; this is not the definition, however. The parametrisation starts at the first point and proceeds in the direction of the second and third points.

Taking a parametrisation such as $x = acos\theta$ and $y = bsin\theta$ the derivative of this gives $x = -asin\theta$ and $y = bcos\theta$, where θ is the parameter varying from 0 to 2π.

As for the circle, using *cpt* as the centre point, $(p_1 + p_3)/2$, the x-axis direction is given by $p_1 - cpt$, the normalisation of this vector is $xdir$, and its length is a. The other length b is found from the distance of p_2 from the line from p_1 to p_3. The normalised vector $ydir$ is given by $enormXxdir$, where enorm is the normal to the plane of the ellipse. This gives the tangent vector, *tvec* as $tvec = -xdir * asin\theta + ydir * bcos\theta$.

The values of $cos\theta$ and $sin\theta$ can be calculated from the scalar and vector products of the x-direction and the vector from the ellipse centre to the given point.

Parabola tangent

As before, the point definition of the curve defines a plane on which the curve lies. A simplified version of a parabolic equation is $y^2 = 4ax$. A parametric version of this is given by $x(t) = at^2$ and $y(t) = 2at$. Differentiating with respect to t gives $x'(t) = 2at$ and $y'(t) = 2a$.

Taking the distance between p_1 and p_2 as a, the normalised vector from p_2 to p_1 is $xdir$. The three points are used to calculate the normal to the plane of the curve, which gives $ydir = normXxdir$ The tangent then becomes $2at * xdir + 2a * ydir$.

Hyperbola tangent

A simplified hyperbolic equation is $x^2/a^2 - y^2/b^2 = 1$.

One solution for the parametrisation is $x(t) = a(1 + t^2)/(1 - t^2)$, $y(t) = 2bt/(1 - t^2)$ for $-1 < t < 1$. This is an inconvenient parametrisation, though, because as t tends to 1 or -1, so the values of x and y tend to infinity.

Differentiating with respect to t gives $x'(t) = 2at/(1-t^2)+2at(1+t^2)/(1-t^2)^2$ and $y'(t) = 2b/(1-t^2)+4bt^2/(1-t^2)^2$. Simplifying gives $x'(t) = 4at/(1-t^2)^2$ and $y'(t) = 2b(1+t^2)/(1-t^2)^2$.

The tangent is, therefore, $xdir * 4at/(1-t^2)^2 + ydir * 2b(1+t^2)/(1-t^2)^2$, where $xdir$ is the difference between p_1 and p_2, the normal to the plane of the curve, $zdir$ is given by $(p_1 - p_2)X(p_3 - p_2)$, and $ydir = zdir \times xdir$. All vectors are normalised, of course.

Free-form curve tangent

It is relatively easy to determine the tangent to a free-form curve provided that a parameter value, t, is given. This is simply the derivative of the curve. For a cubic Bézier, for example, you get:

$$P'(t) = -3(1-t)^2B_0 + 3((1-t)^2 - 2t(1-t))B_1 + 3(2t(1-t) - t^2)B_2 + 3t^2B_3$$

It is also possible to use the matrix forms

$$P(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

and

$$P'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix}$$

D.3 Surface normal

The surface equivalent to the above utility. Again, the different surface types can be used to subdivide the problem.

Plane normal

As with the tangent to the straight line, the normal is given explicitly in the plane format. Finding the projection of any given point onto the plane is easy. Assuming that the plane is defined by a point, P_p and normalised normal vector n, then the projection of a point P_x is just $P_x+((P_p-P_x)\bullet n)*n$.

Cylinder normal

The first step is to find the point, closest to the given point, on the cylinder axis.

As mentioned before, when considering straight line tangents, the closest point P_x to a given point P_q on a line specified by a point P_1 and a normalised direction d is given by:

$P_x = P_1 + ((P_q - P_1) \bullet d) * d$

Here the line direction is given by the normalised difference between the two axis points. The normalised vector difference $P_q - P_x$ gives the normal direction. Obviously, P_q is not allowed to be on the axis. If it is, then a normal vector with zero magnitude could be returned as result, as well as an error code. The surface point is $P_x + r \times n$, where n is the surface normal and r is the cylinder radius.

Cone normal

Note, first of all, that the apex of the cone is a critical point where the normal is not defined.

The calculation is relatively simple in vector terms. Given that $axpnt1$ is the apex of the cone, $axpnt2$ is a point on the axis of the cone and p_q is the given point, you get:

$(axpnt2 - axpnt1) \times (p_q - axpnt1)$ gives the normal to the plane, $tdir$, of the required normal, and $tdir \times (p_q - axpnt1)$ gives the normal. The direction needs to be changed for negative surfaces.

Sphere normal

The normal is simply the normalised difference vector of the given point and the sphere centre. The point on the sphere is the sphere centre plus the sphere radius times the normal vector. Again, obviously, the given point is not allowed to be at the sphere centre.

Torus normal

The plane in which the normal lies is determined by the given point and the axis of the torus. The point is not allowed to be on the axis of the torus.

The plane containing the normal vector is defined by the torus axis and the given point. This plane cuts the central circle of the torus in two points that are the centres of two circular cross sections. The given point should lie on one of these cross sections, and the normal is given by the vector from the centre of this circle to the given point.

Free-form surface normal

Free-form surfaces have a parametrisation that allows calculation of the normal from two partial derivatives. Given $S(u, v) = F(u, v)$, the normal is given by $\frac{dF(u,v)}{du} \times \frac{dF(u,v)}{dv}$.

D.4 Parameter value of point on curve

Parameter information may be supplied directly, if, say, a parametric geometric form was given to the intersection package, or might have to be calculated specifically for non-parametrised forms. Parameter values are used, say, for

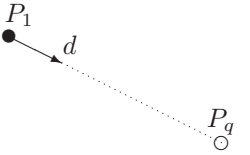


Figure D.15: Line parameter definition

sorting intersection points along curves, so the function is necessary for this if they are not supplied from intersection utilities.

This is needed both for analytic and numeric geometry. A straight line, for example, has a convenient zero-parameter point as the start point. Using a normalised direction vector, the parameter value of any point on the line can be taken as the distance from the start point using the direction to determine whether this is positive or negative. Closed curves can be parametrised from 0 to 2π, for example, with one start point. An alternative is to divide the curve into two pieces parametrised from 0 to 1.

This procedure is, however, not straightforward for numerical curves.

As with the function to find the tangent at a point, there is the option to find the parameter value of the projection of the given point onto the curve. Otherwise, if the point is not on the curve, an error should be flagged.

Straight line

See figure D.15. The closest point P_q to a given point P_x on a line specified by a point P_1 and a normalised direction d is given by

$$P_q = P_1 + ((P_x - P_1) \bullet d) * d$$

The parameter value is just $(P_q - P_1) \bullet d$

Circle

See figure D.16. If the given point is not in the plane of the circle, then the projection of the point onto the plane p_p should be used. The closest point is then simply $P_{centre} + radius * pdir$, where $pdir$ is the normalised vector from P_{centre} to P_p.

The angular parametrisation was described above for finding the circle tangent. The three definition points for the circle are used to find the circle centre p_{centre}. The normalised vector from p_{centre} to p_1 defines the x-axis direction $xdir$. The three points define the normal vector. The scalar product of $pdir$ and $xdir$ gives the sin of the parameter, θ. The vector product of $pdir$ and $xdir$ defines $cos\theta$ with the sign found by comparing the vector with the circle normal vector.

Ellipse

See figure D.17. The case for the ellipse is similar to that for the circle. The

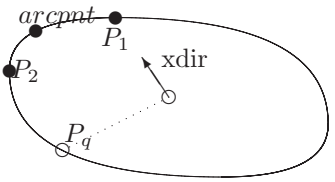


Figure D.16: Circle parameter definition

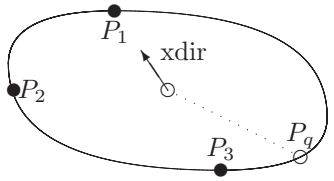


Figure D.17: Ellipse parameter definition

given point is projected onto the ellipse plane if necessary. The normalised vector from the ellipse centre to the given point defines $pdir$. The x-direction $xdir$ is defined as the normalised direction of the first axis and the value of θ is calculated in the same way.

Parabola

See figure D.18. The calculations are based on the distance between p_1 and p_2, denoted by a, and the normalised vector from p_2 to p_1, denoted by $xdir$. The three points are used to calculate the normal to the plane of the curve, which gives $ydir = normXxdir$. The tangent then becomes $2at * xdir + 2a * ydir$. Using the parametrisation $x(t) = at^2$ and $y(t) = 2at$, the magnitude of $(p_q - p_2) \bullet xdir$ gives the value of x. Dividing by a and taking the square root gives the value of the parameter t.

Hyperbola

See figure D.19. The parametrisation defined previously was $x(t) = a(1 + t^2)/(1 - t^2)$, $y(t) = 2bt/(1 - t^2)$ for $-1 < t < 1$. The distance a is given by the distance between p_2 and p_1 and the normalised vector $p_2 - p_1$ is the x-direction $xdir$. The X coordinate of the hyperbola is given by $(x - p_1) \bullet xdir/a$ and the value of the parameter $t = sqrt((X - 1)/(X + 1)$.

Free-form curve

This is difficult. For a Bézier cubic curve, for example, you have the relation: $(1 - t)^3 B_0 + 3t(1 - t)^2 B_1 + 3t^2(1 - t)B_2 + t^3 B_3 = P$. Taking just the X-coordinate, say, you get the polynomial: $(B_3^x - 3B_2^x + 3B_1^x - B_0^x)t^3 + (3B_2^x - 6B_1^x + 3B_0^x)t^2 + (3B_1^x - 3B_0^x)t + B_0^x - P^x = 0$, which is solved to find

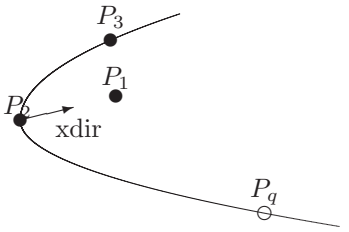


Figure D.18: Parabola parameter definition

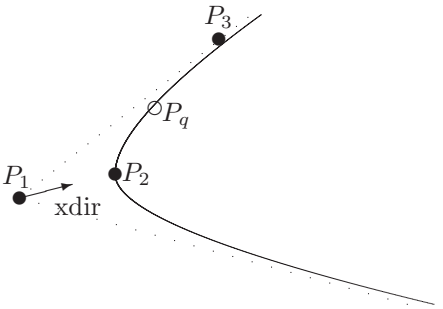


Figure D.19: Hyperbola parameter definition

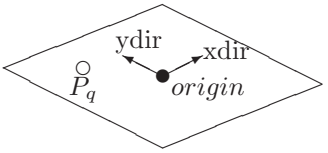


Figure D.20: Plane parameter definition

the value of t. Similarly for y and z because a curve lying in the plane $x = 0$, for example, would not yield a result. However, this is a complicated topic. Piegl and Tiller [100] describe a general method.

D.5 Parameter values of a point on a surface

For calculating parameter values on parametric surfaces. Numerical surfaces have a defined parameter space. Analytic surfaces need to have a parameter space definition assigned. For a plane, an x-direction and a y-direction have to be defined for the points. Analogously to a circle, a cylinder needs to have a start 'seam' so that the parameter values of points around the cylindrical surface can be determined. The other direction can be determined from a start point on the axis and the axis direction.

Plane

See figure D.20. The plane data of a normal and distance from the origin do not provide sufficient information in themselves for a consistent parametrisation. For this reason, it may be necessary to store an extra vector (the x-direction, *xdir*, for example) with the plane, but such a vector should be perpendicular to the normal. If such an x-direction is not given, then *xdir* can be defined as the projection of the global x-axis onto the plane. If the plane normal is parallel to the x-axis, then the projection of the negative z-axis can be used. However, the parameter values of a point may change according to the orientation of the surface, which may be undesirable.

Anyway, assuming that *xdir* is defined, the vector product of the plane normal with *xdir* gives the y-direction *ydir*. The origin of the plane is either defined implicitly, with the minimal plane representation, or explicitly. If the point is defined implicitly (the $Ax+By+Cz+D$ form), then the origin point is $(-DA, -DB, -DC)$.

The parameter values u, v of a point P_q are given by $u = (P_q - origin) \bullet xdir$ and $v = (P_q - origin) \bullet ydir$.

Cylinder

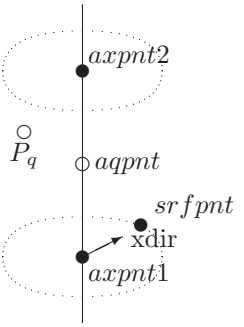


Figure D.21: Cylinder parameter definition

See figure D.21. The cylinder data points define two axes, with the third given by the vector product of these two.

The point $axpnt1$ defines the origin. The normalised vector $axpnt2 - axpnt1$ defines the z-direction $zdir$. The surface point $srfpnt$ is projected onto the axis as $aspnt$. The normalised vector $srfpnt - aspnt$ gives the x-direction $xdir$. If the query point is p_q and the projection of this onto the axis is $aqpnt$, the normalised vector $p_q - aqpnt$ gives the query vector $qvec$. $xdir \bullet qdir$ gives the cosine of the angle around the axis; $(xdirXqdir) \bullet zdir$ gives the sine of the angle, from which the u parameter is calculated. The v parameter is simply $(aqpnt - axpnt1) \bullet zdir$.

Cone

See figure D.22. The cone calculation is similar to that for the cylinder.

Sphere

See figure D.23. The sphere datastructure does not provide enough information for an orientable parametrisation, but it is possible to use a global parametrisation, which means that if the parameter values of a point are found in one position, and then the point and surface are transformed, the transformed point may not have the same parameter values on the transformed surface. If this is not desired, then it is necessary to store extra parameter information with the surface.

With the global parametrisation, the x-direction $xdir$ is the vector (1,0,0), the y-direction $ydir$ is the vector (0,1,0), and the z-direction $zdir$ is the vector (0,0,1). If the vector from the circle centre to the query point P_q is $qdir$, then the projection of $qdir$ onto the plane with normal $zdir$ compared with $xdir$ gives one angle parameter. The angle of $qdir$ about $ydir$ gives the second angle parameter.

Torus

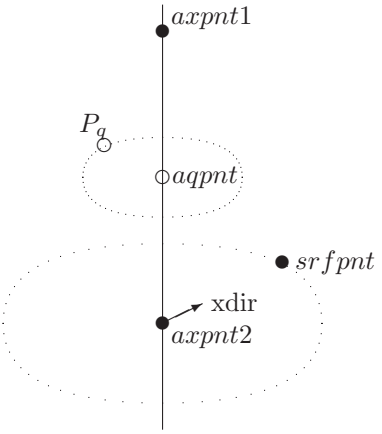


Figure D.22: Cone parameter definition

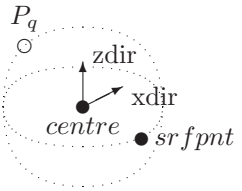


Figure D.23: Sphere parameter definition

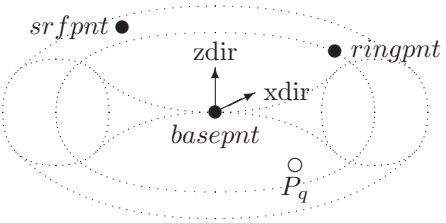


Figure D.24: Sphere parameter definition

See figure D.24. The query point together with the torus axis defines a cutting plane. The intersection of the torus with this plane gives two circles, on one of which lies the point. The normalised vector from the torus centre *basepnt* to the ring point *ringpnt* defines the x-direction *xdir*. The z-direction *zdir* is defined from $zdir = xdirX(srfpnt - basepnt)$. The centre of the circular cross-section on which the point lies is termed *qcpnt*. Comparing the vector $qcpnt - basepnt$ with *xdir* gives the first parameter. The direction around the circle is defined by the torus normal calculated from the three tour definition points. Comparing the vector $qpnt - qcpnt$ with the vector $basepnt - qcpnt$ defines the second parameter value.

Free-form surface

As with the free-form curve, this is complicated.

D.6 Curve coordinates from parameter value

The inverse function to number 4, above. The function is useful, among other things, for calculating intermediate points on a curve between known points and for drawing curves. This is much more straightforward than the inverse function, but for analytic curves, the parametrisation definition needs to be stored.

Straight line

Given a line specified by a point P_1 and a normalised direction d, the desired point at parameter t is simply given by

$$P_q = P_1 + t * d.$$

Circle

The circle centre *cpnt* is found from the three points. The vector $xdir$ is the normalised direction vector of p_1 from *cpnt*. The normal to the plane containing the circle *cnorm* is calculated from the three given points. It is possible to calculate the vector $ydir$ as $ydir = cnorm \bullet xdir$. The desired point corresponding to the input parameter t is then simply:

$$P_q = cpnt + xdir * cos(t) + ydir * sin(t)$$

Ellipse

The ellipse centre *epnt* is found from $(p_1 + p_2) * 0.5$. The vector $xdir$ is the normalised direction vector of p_1 from *cpnt*. The distance of p_1 from *cpnt* is the measure a. The distance of p_2 from the line from p_1 to p_3 gives the value b, and the direction vector from *cpnt* to p_2 gives $ydir$. Note that this should be perpendicular to $xdir$. The desired point corresponding to the input parameter t is then simply:

$$P_q = cpnt + a * xdir * cos(t) + b * ydir * sin(t)$$

Parabola

The point is found by substituting the parameter value t into the formula: $P(t) = xdir * at^2 + ydir * 2at$. The normalised direction of the vector $p_1 - p_2$ in the data definition defines $xdir$, and the magnitude of this vector is the value a. The normal to the plane of the parabola *norm* is the normalised vector in the direction calculated from the three points $(p_1 - p_2)X(p_3 - p_2)$. The vector $ydir = normXxdir$.

Hyperbola

The point is found from the formula: $P(t) = xdir \times a(1 + t^2)/(1 - t^2) + ydir \times 2bt/(1 - t^2)$. The value of the parameter t is restricted to be $-1 < t < 1$. The three points in the data definition define the normal *norm* to the plane of the hyperbola. The magnitude of the vector $p_2 - p_1$ defines the value of a. The direction of this vector defines the x-direction $xdir$ and the y-direction $ydir = norm \times xdir$.

Free-form curve

The point is found directly from substitution of the curve parameter into the formula for the type of free-form curve. For example, for a cubic Bézier curve, you have:

$$P(t) = (1 - t)^3 B_0 + 3 * t * (1 - t)^2 B_1 + 3 * t^2 * (1 - t) B_2 + t^3 B_3$$

D.7 Surface coordinates from parameter values

The inverse function to number 5, above. It is again necessary to divide the surface into different types.

Plane

The position is found from the formula: $P(t) = origin + u \times xdir + v \times ydir$. The origin is either given explicitly in the data definition or implicitly, with the plane form: $Ax + By + Cz + D = 0$. With the implicit form, the origin is the point $(-AD, -BD, -CD)$. A method for determining the x-direction $xdir$ was given when defining the parameter position of a point. However, if calculating points from parameters, it is advisable to give the x-direction explicitly as part of the data definition.

Cylinder

The parameters are interpreted with u as an angle and v as a height, as $P(u, v) = axpnt1 + xdir * cos(u) + ydir * sin(u) + zdir * v$. The vector $zdir$ is the normalised vector with direction $axpnt2 - axpnt1$. The vector $xdir$ is the vector from the projection of $srfpnt$ onto the cylinder axis to the surface point $srfpnt$. The vector $ydir$ is calculated as $zdirXxdir$.

Cone

As for the cylinder, the u parameter is interpreted as an angle around the axis and the v parameter as a height. The formula is the same $P(u, v) = axpnt1 + xdir * cos(u) + ydir * sin(u) + zdir * v$. The definitions of $xdir$, $ydir$, and $zdir$ are also as for the cylinder.

Sphere

The parameters are interpreted as two angles. The first step is to interpret u, giving a first vector: $uvec = (1, 0, 0) * cos(u) + (0, 1, 0) * sin(u)$. This gives an x-direction $xdir$ defined as the normalised direction of $uvec$ and $P(u, v) = xdir * cos(v) + (0, 0, 1) * sin(v)$.

Torus

As with the sphere, the two parameters are interpreted as angles. The u parameter is interpreted as an angle around the central ring of the torus. The v parameter is interpreted as an angle around the circular cross-section centred on the point c_u defined by the u parameter. $c_u = xdir * cos(u) + ydir * sin(u)$. The direction $xdir$ is the normalised vector in the direction $ringpnt - basepnt$. The vector $ydir$ is defined as $zdir \times xdir$, where $zdir$ is the normalised vector in the direction $xdir \times sdir$, where $sdir$ is the normalised vector in the direction $(srfpnt - basepnt)$. The final point is given by $P(u, v) = (-sdir) * cos(v) + zdir * sin(v)$.

Free-form surface

This is obtained by substituting the values of u and v into the free-form surface formula. For example, for a Bézier surface, this is:

$$\sum_{i=0}^{n}\sum_{j=0}^{m}\frac{n!}{i!(n-i)!}\frac{m!}{j!(m-j)!}(1-u)^{n-i}u^{i}(1-v)^{m-j}v^{j}B_{ij}$$

D.8 Create a surface by sweeping an edge

This is the main geometric creation utility for sweeping. It uses the curve of the edge and the sweep direction to create the surface shape and the orientation of the edge in the loop to determine the orientation. The main cases are for straight and circular sweeping, but extensions can also be made for sweeping along a curve.

Straight line

For sweeping in a straight line, this will generate a plane. The vector product of the line direction and the sweep direction gives the normal to the plane and the line definition point, or one of them, if two are used, is a point on the plane.

There are several alternatives if sweeping in a circle. Note, for all circular sweeps, the sweep is considered to be in a complete circle. The topology then defines which part of this surface will be used. If the line is parallel to the axis, then a cylinder is produced. If the line is perpendicular to the axis, then a plane is produced. If the line intersects the axis of rotation, then a cone is produced. These are the usual cases. If the line is not in the same plane as the axis, then a quadric surface is produced that could be represented by the general quadric. However, it may be easier to produce a free-form surface directly. If free-form surfaces are produced for circular sweeping, then the rational form should be used; otherwise, the surface has to be of high degree or be composed of several patches to obtain a sufficiently accurate form. One way of creating the surface is to calculate the positions of the line endpoints at various places around the axis and then interpolate these. The surface will be a ruled-surface so the curves interpolating these points define the surface.

Circle

If the sweep direction is the same as the normal to the plane containing the circle, then a circular cylinder will be generated; otherwise, an elliptic cylinder will be generated. Now comes the 'BUT'; if the sweep direction is perpendicular to the normal of the circle, then a plane could be generated. This is probably an error condition, though, so an option is to let this function squeal and return an error. The alternative is to return a plane and let the function that uses this sort out the problem, if there is one. Note

that if an elliptic cylinder is not specifically included in the surface types, then another type of surface is needed. A general quadric surface could be used or a free-form surface. The surface can be calculated by calculating the transformed position of the curve after the extrusion. The surface will be a ruled surface interpolating these two curves. Note, as well, that the extrusion extent is needed for this because a free-form surface is implicitly limited by the parameter range.

If a circle is being swept in a circle, then the result will be a torus if the axis lies in the plane of the circle. If the axis of rotation is perpendicular to the circle plane, then the result is a plane. Otherwise it is easiest to create a free-form surface.

Ellipse

This is similar to the circle case. A linear sweep at an angle that is not parallel to the normal of the ellipse plan may create a cylindrical surface. This is a special case that can be checked. Another special case is one in which a rotational sweep can create a torus. However, it may be easier to handle this and the other cases by creating free-form surfaces.

Parabola

For both linear and circular sweeping, this can be handled by creating free-form surfaces in the same way as outlined above. If the extrusion is linear and the direction is perpendicular to the normal of the parabola plane, then a plane will be created. Similarly with circular sweeping where the sweep axis is parallel to the parabola plane normal, then a plane will be created.

Hyperbola

As for parabolas.

Free-form curve

Extrusions of free-form curves are handled in the way explained above, by translating the curve and creating a surface interpolating the original and transformed curves.

D.9 Modify geometry

As well as being used to change the shape or position of complete objects, without changing the topology, geometry modification is also used for such operations as sweeping or swinging and bending. This may even change the form of some geometry, say changing a circle to an ellipse, with non-uniform scaling.

The geometric entities described in Appendix A are largely point based, with one or two vectors. Assuming transformations specified by a 4×4 matrix,

to transform a point (x,y,z), you have the matrix multiplication:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

or for vectors:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 0 \end{bmatrix}$$

Reflection, or symmetry, matrices are also possible. The slight difficulty, as mentioned above, comes with non-uniform scaling. If this is allowed, the simplest method is to convert the simple geometry into a numeric form, spline curve, or surface and then to modify the control points. Other matrix possibilities such as shear transformations should be treated with care. Note, though, that if geometry is converted to a numeric form, then it is difficult to get back to the simple form.

D.10 Reparametrise a curve

Reparametrisation is necessary to ensure that consistency is maintained if the parametrisation of a curve is linked to an edge, and the start and end positions of the edge are changed. Also, with some parametrisations, two distinct points that are close to each other but distant from a parametrisation origin may have parameter values within tolerance of each other.

An important topic is the reparametrisation of free-form curves. This is described by Farin [35] or Piegl and Tiller [100], for example.

D.11 Offset curve from a given curve

This is necessary for producing offset geometry when giving a sheet object thickness (see section 4.7). The difficulty is that not all offset curves can be represented by the explicit curve set, so quickly it becomes necessary to use free-form curves. The input to such a function can be the curve to be offset, a point on the given curve, and the corresponding point through which the offset curve passes. If the curve is planar and such a point is given, then this, together with the normal of plane of the original curve, defines the plane of the new offset curve.

Line

This can be calculated simply with the given offset point as a point on the line and the same line direction as the original line.

Circle

This, like the line, is relatively simple. The normal to the original circle defines an axis that is the same as for the offset. The distance of the given offset point from the axis defines the radius of the new offset curve, and its projected position on the axis defines the centre. The three new definition points are calculated from the old definition points. Each new point is in the same direction as its corresponding point from the old centre, but at a distance of the new radius from the new centre.

Ellipse, Parabola, and Hyperbola

The easiest way of handling these is to generate a set of offset points and then to produce a curve interpolating these. The normal to the plane containing these curves and the given offset point define the plane of the new curve. Points on the original curve are projected onto the new plane and then offset the right amount in the appropriate direction. The 'right amount' is calculated by projecting the original curve point onto the new curve plane and computing the distance to the given offset point. The 'appropriate direction' is a vector perpendicular to the tangent vector of the curve at the point being offset.

Free-form curve

This is difficult because there is no obvious direction for the offset, as for the planar curves. One representation for free-form curves associates two surfaces with the curve if it is an intersection curve between the two surfaces. If such information is available, then the surfaces can be offset and the new curve produced as the intersection of the new offset surfaces. The alternative is to use a marching procedure calculating the offset points successively.

D.12 Offset surface from a given surface

As above. This is used when giving sheet objects thickness.

Plane

The new offset plane is a plane with the same normal as the original plane, passing through the given point.

Cylinder

An offset cylinder has the same axis points as the original cylinder and the given point as surface point.

Cone

The cone is similar to that of the cylinder. The axis is the same for the offset surface as for original surface, and the given offset point is the new surface point. However, the cone apex, which is $axpnt1$ in the data definition, needs to be recalculated. If the given points on the original surface and the offset surface are the apices of the cone, then the offset point cannot be used as the surface point but can be used directly as the apex point of the new surface and a new surface point has to be calculated. Otherwise the offset apex point is calculated by offsetting the original apex along the axis.

Sphere

The offset surface of a sphere is another sphere with the same centre and the given point as the surface point.

Torus

The offset of the torus is a torus with the same base point and ring point but with the given point as the surface point.

Free-form surface

A set of points on the original surface are determined, and the offset points are calculated by offsetting these along the surface normal vectors by the offset distance. These points are then interpolated to create the new offset surface.

Appendix E

General utilities

This appendix describes some general utilities which are used in the modelling algorithms.

E.1 Marker bits

The marker bit mechanism is very heavily used in the algorithms described in chapter 6 for recording special conditions. The marker bits are a set of bits associated with each entity. The use of these bits needs to be listed somewhere to avoid conflicting uses.

There are three basic operations: SETBIT, CLEARBIT, and BIT (to sense the value of a bit).

E.2 Matrix and vector packages

As both matrices and vectors are used for geometry representation, it is important to have general packages for handling them. Common vector operations needed are adding, subtracting, scalar, and cross products and normalisation. Matrix operations include multiplication, inversion, and determinants. Commonly 4×4 matrices are needed, but for interpolation, higher degree matrices are also needed.

Note that it is possible to distinguish between vectors used to define positions and vectors used to define directions.

E.3 Point in body

Determine whether a point is inside or outside an object. This utility is used, for example, for determining how to handle objects in Boolean operations where the boundaries do not intersect.

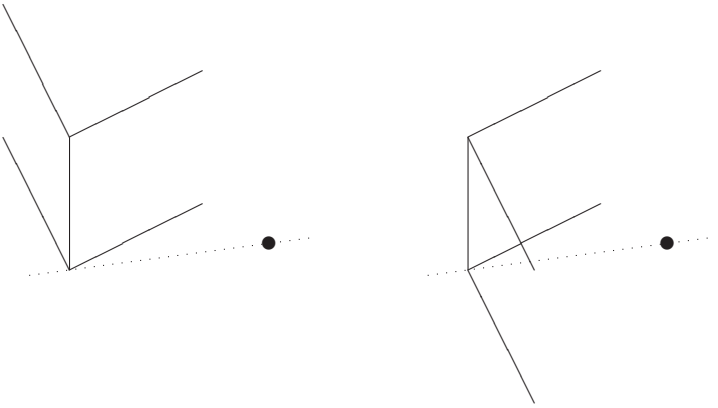


Figure E.1: Determining rays for point-in-body tests

It is very tempting to start off by checking whether the point is inside or outside the box or bubble surrounding the object, and if outside, conclude that the point lies outside the object. Unfortunately this is not enough. With a negative object, a point lying outside the object box or bubble will lie inside the object. In fact, the simple test for whether a body is positive or negative is to take a point lying outside the object box or bubble and to check whether it lies inside or outside the object. If the point lies inside the object, then the object is negative.

The test can be thought of as firing a ray from the point to infinity and counting the number of intersections with the body. If there are an even number of intersections, then the point lies outside the object; an odd number indicates the point lies inside the object. (See Braid [11] for a description of ray tests in faces.) An alternative method is to generate a straight line passing through the point and find a segment of this containing the point and lying inside or outside the object. This latter algorithm is described below.

First of all, it is necessary to determine a ray for the test. This may not be easy. One choice is the centre of the box of the object, but there is no guarantee that the ray joining the point being tested and this point will cut the object. The ray has to cut through some part of the object to be able to classify the interaction. Another choice is a vertex on the object, but not all vertices are valid. Figure E.1 shows two cases. On the left the ray just grazes the object; on the right it cuts the object. The condition for a vertex to be acceptable is that all face normals of the faces around the vertex are in the opposite sense to the direction from the point to the vertex. There still may not be such a vertex in some extreme conditions: if the ray is in the tangent plane to a cylinder, for example. In such cases, it may be necessary to fiddle and find some internal point within the object by some special means.

Once the ray has been determined, then it is intersected with the object

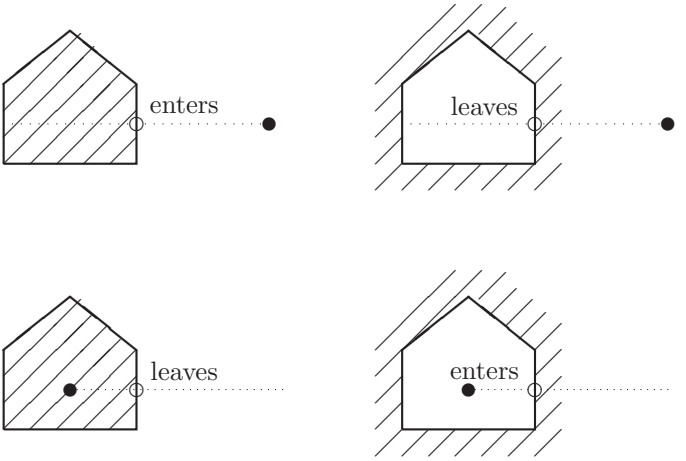


Figure E.2: Enters and leaves ray classification

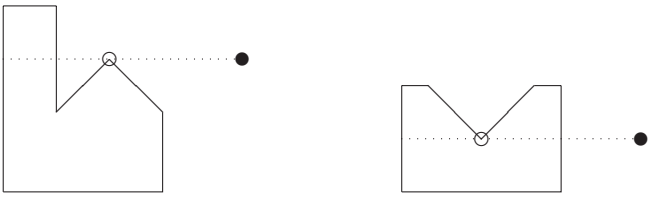


Figure E.3: External and internal grazing rays

and each intersection point is classified. If the first real intersection point is an 'enters' point, then the point is outside the body. If the first real point is classified as a 'leaves' point, then the point is inside the body. These are shown in figure E.2 for positive and negative figures.

The term 'real intersection point' is included above because there are some cases in which the ray just grazes the body without cutting it (figure E.3). In these cases, the grazing point is ignored.

A check for the point being on the boundary of the body should be included.

E.4 Point in face

This is important for testing to which faces internal loops belong, for example. The general method is to use a ray test, but care is needed to cope with special

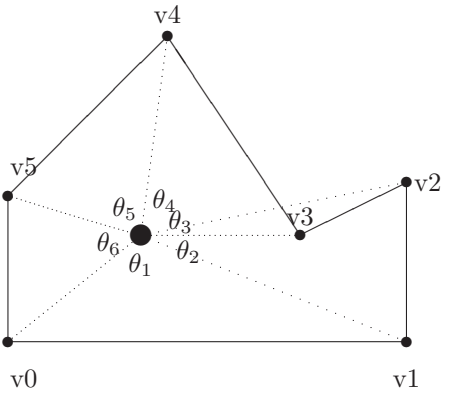


Figure E.4: Angle counting point-in-face

cases if no fake edges (see section 3.1.4) are included in the model.

First, though, a slight divergence to examine an original and interesting method by Jared.

The end vertices of each edge together with the point being tested form a triangle. The sum of the angles at the point being tested is 360 degrees if the point is inside the face, and 0 degrees if the point is outside the face. Note that in figure E.4, angle θ_3 is actually negative. A special check needs to be made to see whether the point being tested is on the boundary.

This method is worth mentioning because it is a simple robust method for testing points in planar faces. It is used, for example, in rapid prototyping to test polygonal planar faces. The method also works for slightly curved faces, although the angle is not 360 degrees. This is one reason that BUILD did not allow faces extending through more than 180 degrees.

Another method, which can be used with a wider range of geometry, is the ray test. To test a point, a plane (containing the normal to the face at the point) is intersected with the surface of the face to produce a curve lying in the face. This intersection curve is then intersected back with the face to produce a set of intersection points along the curve. If the first one after the point being tested is classified as 'leaves', then the point is inside the face; otherwise, it is outside. The parametrisation of the curve, with the point being tested as the zero point, determines the first point. Note the grazing point in figure E.5, which also needs to be ignored here, as it was in the point-in-body test.

However, there can be special cases with curved geometry. A naive approach can also lead to problems. In figure E.6, it is possible that the intersection curve does not cut any edge if there are no fake edges around the cylinder. One approach is to add one fake edge so that one edge will always be cut. This edge will be a wire edge, though. Another solution is to make the

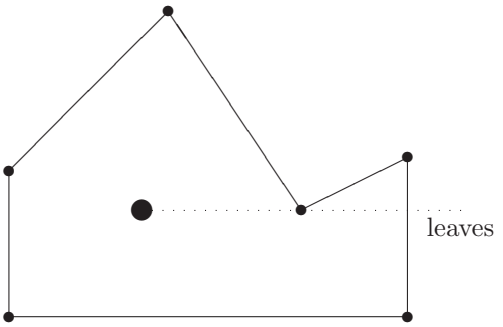


Figure E.5: Ray firing point-in-face

intersection curve so that it always cuts at least one edge, analogously to the way that a ray was chosen for the point-in-body test. However, the minimal representation is a face with a solitary vertex and no edges. It is necessary, therefore, to check for this special case.

E.5 Loop in face

This is important for correct partitioning of hole-loops when a face has been divided. In its simplest form, it may only require that a point on the loop be tested to see whether it lies in a face. If more complicated conditions can occur, then it may be necessary to return more than a Boolean, indicating, perhaps, whether a loop intersects or coincides with the boundary of a face, as well as testing whether the loop lies completely inside or completely outside a face.

E.6 Intersect a face with a curve

Basically this involves intersecting the curve with all edges bounding a face, i.e., for the exterior loop and all hole-loops, and sorting out the results (figure E.7). Once all intersection points have been found, they are sorted, basically by parameter value but with extra checks for coincidence. In the algorithms described in chapter 6, this is used for Boolean operations, sectioning, imprinting, and bending. The same basic process is also useful for drawing hatching lines and silhouette curves for faces.

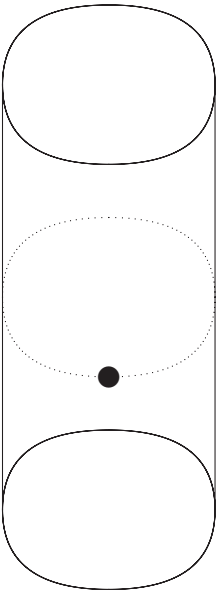


Figure E.6: Ray-firing around cylinder

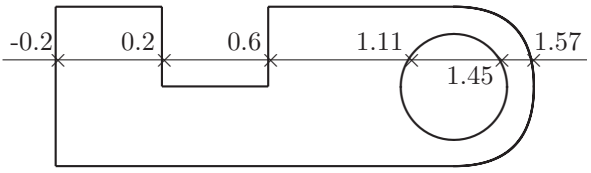


Figure E.7: Intersecting a face with a curve

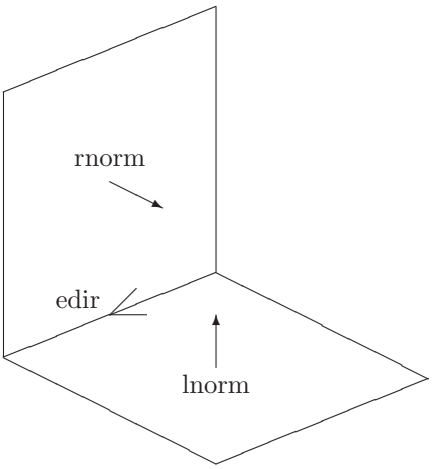


Figure E.8: Concave, convex, smooth test

E.7 Curve-edge orientation

Determine whether a curve lies between two edges. This is used to determine geometrically the order of edges around a vertex. It is used by the imprinting operation when more than two edges at a vertex are adjacent to the face being imprinted. A similar test is used in the Booleans. It is also useful for user-friendly Euler operations to avoid requiring orientation edges from a user to determine where a new edge should lie.

E.8 Convex, concave, smooth edge

Check whether an edge is convex, concave, or smooth at a point. This check is partly topological, because it refers to edges, and partly geometrical, because it involves the surfaces meeting at the edge. The test uses the face normals on either side of the edge to perform the check. As edges can change character along their length, it is necessary to specify a point on the edge for the test.

The value of $(lnorm \times rnorm) \bullet edir$ is positive if the edge is convex, negative if the edge is concave, and zero if the edge is smooth. To classify an edge, it is not enough to test a single point. Figure E.9 shows an example by Alan Smith of the BUILD group showing two cylindrical surfaces intersecting in an elliptic edge. The edge is smooth at the start and endpoints but convex in the middle.

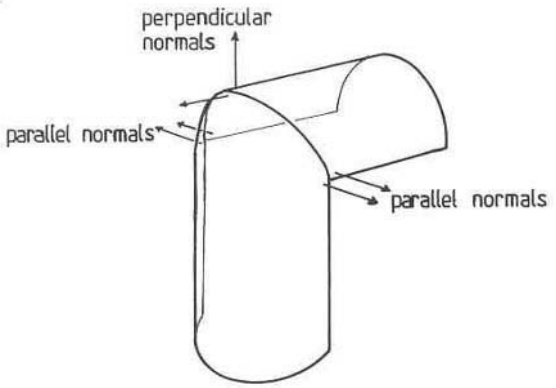


Figure E.9: Convex and smooth edge (after Alan Smith of the former BUILD group)

E.9 Loop area and orientation

Calculate the orientation and approximate area of a loop. This operation, named "polyareavec" in BUILD, is very useful for checking the orientation of faces and individual loops. It can be used for checking orientations of loops in low-level Euler operations as well as for higher level operations. This is only suitable for faces extending through less than 180 degrees.

The method is similar to the angle counting method for the point-in-face test. The difference is that there is no internal point that can be used as a base. Instead, one of the vertices is used as a start point, here vertex v0. The area of the triangle formed by vertices v0, v1, v2 gives the first area element. The triangle v0, v2, v3 gives a small negative contribution. Triangle v0, v3, v4 is then added and finally the area of triangle v0, v4, v5.

The area of each triangular element is found using the cross product of two sides. The sum of the vector directions gives an approximation to the orientation of the face. This is only valid, though, for planar faces.

An alternative method is to facet the face and sum the contributions of the facets to find the approximate area of the face. This method can be used also with curved geometry.

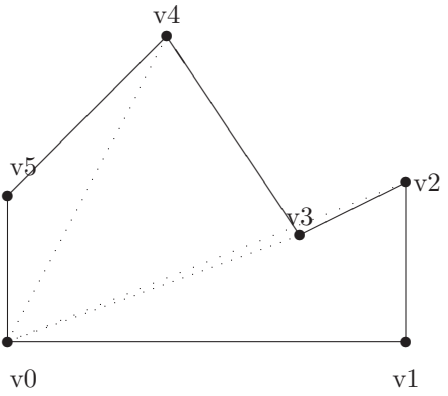


Figure E.10: Polyareavec

Appendix F

Euler operators

As stated in chapter 4 the numbers of elements in the datastructure for a valid object or objects are related by a series of rules concerning the numbers of vertices, edges, faces, inner loops, the genus, and the multiplicity of the object. The objects can be represented by the nodes of a five-dimensional hyper-net in the six-dimensional space of these elements. The operators to traverse this network are called Euler operators, and any change made by them must comply with the formula:

$$v - e + f - h = 2(b - g) \text{ [the Euler–Poincaré formula.]}$$

where:
v is the number of vertices
e is the number of edges
f is the number of faces
h is the number of inner-, or hole-loops
g is the genus
b is the multiplicity or number of shells

Table 1 lists all possible Euler operators that change the number of any element by at most 1.

Table F.1: Set of Euler operators

1. (-1,-1,-1,-1,-1,-1)	kvefhgb	51. (+1,+1, 0, 0, 0, 0)	mev
2. (0, 0,-1,-1,-1,-1)	kfhgb	52. (-1, 0,+1, 0, 0, 0)	mfkv
3. (+1,+1,-1,-1,-1,-1)	mvekfhgb	53. (0,+1,+1, 0, 0, 0)	mfe
4. (-1, 0, 0,-1,-1,-1)	kvhgb	54. (+1,-1,-1,+1, 0, 0)	mvhkef
5. (0,+1, 0,-1,-1,-1)	mekhgb	55. (0,-1, 0,+1, 0, 0)	mhke
6. (-1,+1,+1,-1,-1,-1)	mefkvhgb	56. (+1, 0, 0,+1, 0, 0)	mvh
7. (0,-1,-1, 0,-1,-1)	kefgb	57. (-1,-1,+1,+1, 0, 0)	mfhkve
8. (+1, 0,-1, 0,-1,-1)	mvkfgb	58. (0, 0,+1,+1, 0, 0)	mfh

Continued on next page

9. (-1,-1, 0, 0,-1,-1)	kvegb	59. (+1,+1,+1,+1, 0, 0)	mvefh
10. (0, 0, 0, 0,-1,-1)	kgb	60. (-1,+1,-1,-1,+1, 0)	megkvfh
11. (+1,+1, 0, 0,-1,-1)	mvekgb	61. (-1, 0,-1, 0,+1, 0)	mgkvf
12. (-1, 0,+1, 0,-1,-1)	mfkvgb	62. (0,+1,-1, 0,+1, 0)	megkf
13. (0,+1,+1, 0,-1,-1)	mefkgb	63. (-1,+1, 0, 0,+1, 0)	megkv
14. (+1,-1,-1,+1,-1,-1)	mvhkefgb	64. (-1,-1,-1,+1,+1, 0)	mhgkvef
15. (0,-1, 0,+1,-1,-1)	mhkegb	65. (0, 0,-1,+1,+1, 0)	mhgkf
16. (+1, 0, 0,+1,-1,-1)	mvhkgb	66. (+1,+1,-1,+1,+1, 0)	mvehgkf
17. (-1,-1,+1,+1,-1,-1)	mfhkvegb	67. (-1, 0, 0,+1,+1, 0)	mhgkv
18. (0, 0,+1,+1,-1,-1)	mfhkgb	68. (0,+1, 0,+1,+1, 0)	mehg
19. (+1,+1,+1,+1,-1,-1)	mvefhkgb	69. (-1,+1,+1,+1,+1, 0)	mefhgkv
20. (-1,+1,-1,-1, 0,-1)	mekvfhb	70. (+1,-1,+1,-1,-1,+1)	mvfbkehg
21. (-1, 0,-1, 0, 0,-1)	kvfb	71. (+1,-1,-1,-1, 0,+1)	mvbkefh
22. (0,+1,-1, 0, 0,-1)	mekfb	72. (0,-1, 0,-1, 0,+1)	mbkeh
23. (-1,+1, 0, 0, 0,-1)	mekvb	73. (+1, 0, 0,-1, 0,+1)	mvbkh
24. (-1,-1,-1,+1, 0,-1)	mhkvefb	74. (-1,-1,+1,-1, 0,+1)	mfbkveh
25. (0, 0,-1,+1, 0,-1)	mhkfb	75. (0, 0,+1,-1, 0,+1)	mfbkh
26. (+1,+1,-1,+1, 0,-1)	mvehkfb	76. (+1,+1,+1,-1, 0,+1)	mvefbkh
27. (-1, 0, 0,+1, 0,-1)	mhkvb	77. (+1,-1, 0, 0, 0,+1)	mvbke
28. (0,+1, 0,+1, 0,-1)	mehkb	78. (0,-1,+1, 0, 0,+1)	mfbke
29. (-1,+1,+1,+1, 0,-1)	mefhkvb	79. (+1, 0,+1, 0, 0,+1)	mbfv
30. (-1,+1,-1,+1,+1,-1)	mehgkvfb	80. (+1,-1,+1,+1, 0,+1)	mvfhbke
31. (+1,-1,-1,-1,-1, 0)	mvkefhg	81. (-1,-1,-1,-1,+1,+1)	mgbkvefh
32. (0,-1, 0,-1,-1, 0)	kehg	82. (0, 0,-1,-1,+1,+1)	mgbkfh
33. (+1, 0, 0,-1,-1, 0)	mvkhg	83. (+1,+1,-1,-1,+1,+1)	mvegbkfh
34. (-1,-1,+1,-1,-1, 0)	mfkvehg	84. (-1, 0, 0,-1,+1,+1)	mgbkvh
35. (0, 0,+1,-1,-1, 0)	mfkhg	85. (0,+1, 0,-1,+1,+1)	megbkh
36. (+1,+1,+1,-1,-1, 0)	mvefkhg	86. (-1,+1,+1,-1,+1,+1)	mefgbkvh
37. (+1,-1, 0, 0,-1, 0)	mvkeg	87. (0,-1,-1, 0,+1,+1)	mgbkef
38. (0,-1,+1, 0,-1, 0)	mfkeg	88. (+1, 0,-1, 0,+1,+1)	mvgbkf
39. (+1, 0,+1, 0,-1, 0)	mvfkg	89. (-1,-1, 0, 0,+1,+1)	mgbkve
40. (+1,-1,+1,+1,-1, 0)	mvfhkeg	90. (0, 0, 0, 0,+1,+1)	mgb
41. (-1,-1,-1,-1, 0, 0)	kvefh	91. (+1,+1, 0, 0,+1,+1)	mvegb
42. (0, 0,-1,-1, 0, 0)	kfh	92. (-1, 0,+1, 0,+1,+1)	mfgbkv
43. (+1,+1,-1,-1, 0, 0)	mvekfh	93. (0,+1,+1, 0,+1,+1)	mefgb
44. (-1, 0, 0,-1, 0, 0)	kvh	94. (+1,-1,-1,+1,+1,+1)	mvhgbkef
45. (0,+1, 0,-1, 0, 0)	mekh	95. (0,-1, 0,+1,+1,+1)	mhgbke
46. (-1,+1,+1,-1, 0, 0)	mefkvh	96. (+1, 0, 0,+1,+1,+1)	mvhgb
47. (0,-1,-1, 0, 0, 0)	kfe	97. (-1,-1,+1,+1,+1,+1)	mfhgbkve
48. (+1, 0,-1, 0, 0, 0)	mvkf	98. (0, 0,+1,+1,+1,+1)	mfhgb
49. (-1,-1, 0, 0, 0, 0)	kev	99. (+1,+1,+1,+1,+1,+1)	mvefhgb
50. (0, 0, 0, 0, 0, 0)	Null op.		

A restricted set (after Pavey) of these is shown in Table 2. The set comprises only those operators that change two or three elements.

Table F.2: Reduced set of Euler operators

1	(0, 0, 0, 0,-1,-1)	kgb	21	(+1,+1, 0, 0, 0, 0)	mev
2	(-1, 0,-1, 0, 0,-1)	kvfb	22	(-1, 0,+1, 0, 0, 0)	mfkv
3	(0,+1,-1, 0, 0,-1)	mekfb	23	(0,+1,+1, 0, 0, 0)	mfe
4	(-1,+1, 0, 0, 0,-1)	mekvb	24	(0,-1, 0,+1, 0, 0)	mhke
5	(0, 0,-1,+1, 0,-1)	mhkfb	25	(+1, 0, 0,+1, 0, 0)	mvh
6	(-1, 0, 0,+1, 0,-1)	mhkvb	26	(0, 0,+1,+1, 0, 0)	mfh
7	(0,+1, 0,+1, 0,-1)	mehkb	27	(-1, 0,-1, 0,+1, 0)	mgkvf
8	(0,-1, 0,-1,-1, 0)	kehg	28	(0,+1,-1, 0,+1, 0)	megkf
9	(+1, 0, 0,-1,-1, 0)	mvkhg	29	(-1,+1, 0, 0,+1, 0)	megkv
10	(0, 0,+1,-1,-1, 0)	mfkhg	30	(0, 0,-1,+1,+1, 0)	mhgkf
11	(+1,-1, 0, 0,-1, 0)	mvkeg	31	(-1, 0, 0,+1,+1, 0)	mhgkv
12	(0,-1,+1, 0,-1, 0)	mfkeg	32	(0,+1, 0,+1,+1, 0)	mehg
13	(+1, 0,+1, 0,-1, 0)	mvfkg	33	(0,-1, 0,-1, 0,+1)	mbkeh
14	(0, 0,-1,-1, 0, 0)	kfh	34	(+1, 0, 0,-1, 0,+1)	mvbkh
15	(-1, 0, 0,-1, 0, 0)	kvh	35	(0, 0,+1,-1, 0,+1)	mfbkh
16	(0,+1, 0,-1, 0, 0)	mekh	36	(+1,-1, 0, 0, 0,+1)	mvbke
17	(0,-1,-1, 0, 0, 0)	kfe	37	(0,-1,+1, 0, 0,+1)	mfbke
18	(+1, 0,-1, 0, 0, 0)	mvkf	38	(+1, 0,+1, 0, 0,+1)	mbfv
19	(-1,-1, 0, 0, 0, 0)	kev	39	(0, 0, 0, 0,+1,+1)	mgb
20	(0, 0, 0, 0, 0, 0)	Null op.			

This Appendix describes implementation details of a selected set of these Euler operators, providing comparisons between topological implementations based on direct winged-edge pointers, winged-edge links (also called half-edges), and vertex links (similar to winged-edge links, but arranged around vertices instead of around loops).

According to vector space theory, it is possible to select a set of just five of the Euler operators and their inverses with which any change in an Eulerian object can be made (see chapter 4). The Euler operator spanning set can be thought of as prescribing how the network of valid objects should be traversed. Although the minimal spanning set is sufficient for making any change, or to build up any model, extra Euler operators can also be useful for taking 'short cuts', that is, making the modification more efficient. For this reason, algorithms for more than just the basic set of Euler operators are included.

Also, because the number of elements that the Euler operators change can be mapped onto different topological changes, the modelling operations that make the changes sometimes correspond to several Euler operators, or several modelling operations correspond to one Euler operator. For example, the Euler change: (1,1,0,0,0,0), Make an Edge and a Vertex, can be interpreted in (at least) three ways. It can add a spur edge and vertex (figures F.1, top three), it can split an edge (figure F.1, bottom) or it can split a vertex (figure F.7).

All can be interpreted as splitting a vertex, with the operations of adding a spur vertex and splitting an edge as special cases. Here, though, the different

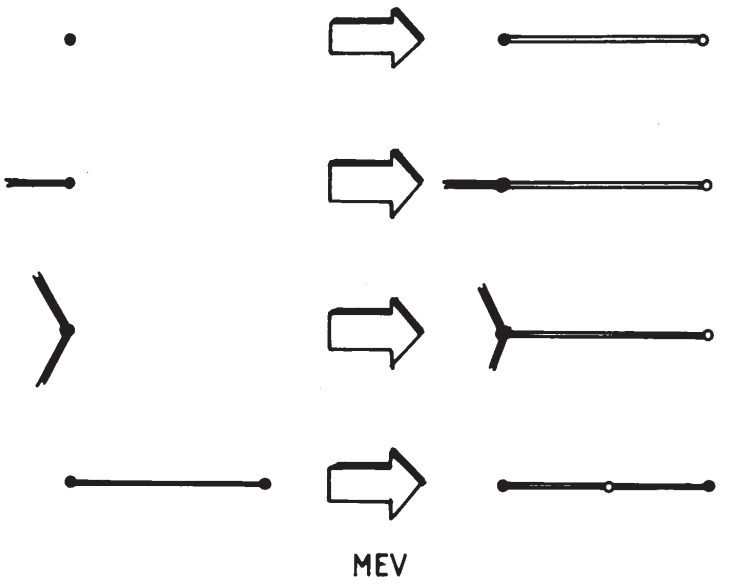


Figure F.1: Illustration of possible interpretations of the Euler operator MEV

interpretations are kept separate to make the desired operation explicit.

This diversity of function for the same basic Eulerian changes makes it difficult to specify exactly how a particular Eulerian change is to be applied in all cases. The implementations of the operators described here are intended to provide examples rather than a complete set.

Some conventions have to be decided concerning the orientation of the new entities. It is sometimes possible to use geometric tests to determine this orientation, but very often the Euler operators produce geometrically incorrect objects as part of a complex modelling operation that later changes the geometry. During these modelling operations, low-level information about the orientation of the new entities is often available. It is, therefore, unnecessary to perform complex checking, possibly on unreliable data. However, if the Euler operators are provided as a simple design tool or as part of an Application Programming Interface (API), then it may not be desirable to ask a user to supply this information. It may also be desirable to supply geometric information for the new entities, even though this is outside their topological function. This suggests that it is necessary to implement the Euler operators as a layered set. The implementation details described here concern mainly the low-level details because how the Euler operators are included in a modeller is, basically, part of the fundamental system design.

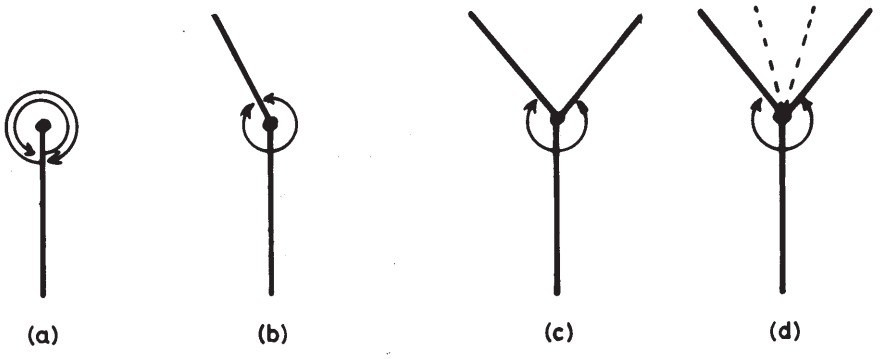


Figure F.2: Setting winged-edge pointers for a new spur edge

F.1 Make edge vertex (spur vertex and edge)

Required parameters: Base vertex
Orientation information at the base vertex (edge, loop or both)
Optional geometry: New vertex position
Curve for the new edge

Basically the process consists of creating a new edge and vertex and linking the new entities together and to a supplied vertex. Whether the new vertex is the start- or the end-vertex of the new edge is arbitrary. As the orientation of any entity should be determined with respect to other entities, a particular arrangement of the entities should not be presupposed. Here, though, it is assumed that the new edge will run from the supplied base vertex to the new vertex.

How the edge is linked around the start-vertex is determined by the orientation edge supplied as a parameter. If the base vertex has no edges or only one edge attached, then the orientation edge is superfluous and can be replaced by a NIL pointer. The operator also sets the edges clockwise and counter-clockwise around the new vertex from the new edge to be the new edge. The edge pointer of the new vertex is set to be the new edge.

With the new vertex as the end-vertex of the new edge and assuming that the representation uses direct winged-edge pointers, this means setting the Left Counter-Clockwise and Right ClockWise pointers of the new edge to point at the edge itself (figure F.2).

There are three cases to consider for the base vertex:

1. The base vertex has no edges attached.

2. The base vertex has one edge attached.

3. The base vertex has two or more edges attached.

1. If the base vertex has no edges attached, then this in effect means that the Left ClockWise pointer and Right Counter-Clockwise pointers of the new edge are set to point back to the edge itself (figure F.3, top right). The edge pointer in the base vertex is also set to be the new edge.
2. If the base vertex is a spur vertex, then the edge clockwise and edge counter-clockwise around the vertex from the new edge are set to be the spur edge. Effectively, the Left ClockWise and Right Counter Clockwise pointers are set to point at the spur edge, and the appropriate winged-edge pointers of the spur edge are set to point at the new edge (figure F.3, middle right).
3. If there are two or more edges at the vertex, then orientation information is required. This information can be supplied in the form of the edge, which will be clockwise or counter-clockwise around the vertex from the new edge, but problems occur if the edge starts and ends at the supplied vertex. A face or loop pointer could also be supplied, but is insufficient when there are more than two edges at the vertex adjacent to the loop or face. Sometimes, therefore, both an edge and a loop (or face) pointer may be required if the supplied edge both starts and ends at the same vertex. If, for example, the supplied edge has different start and end vertices, and is the edge counter-clockwise around the vertex from the new edge, say ccedge, then the other adjacent edge, say cwedge, can be found from the winged-edge pointers or winged-edge links. The edge counter-clockwise from the new edge around the supplied vertex is set to be ccedge, and the edge clockwise around the vertex to be cwedge. (Alternatively, the Left ClockWise winged-edge pointer of the new edge is set to point at ccedge, and the Right Counter-Clockwise pointer set to point to cwedge.) The clockwise edge around the vertex from ccedge is set to be the new edge, and the counter-clockwise edge around the vertex from the cwedge is also set to be the new edge (figure F.3 bottom right).

If winged-edge links or half-edges are used, then the left and right links are chained together for a spur edge. If the spur vertex is the end-vertex of an edge, then if the links are chained clockwise around a loop, then the left link is chained after the right link; otherwise, the right link is chained after the left link. If the spur vertex is the start vertex of the edge, then the right winged- edge link precedes the left winged edge link if the links are chained clockwise around a loop.

The winged-edge equivalents for the same three cases identified above, and illustrated in figure F.4, are as follows:

1. If the base vertex has no edges attached, then the winged-edge links are connected together in a closed chain.

2. If the supplied vertex is a spur vertex, then the winged-edge links of the new edge are connected between the left and right winged-edge links of the spur edge.

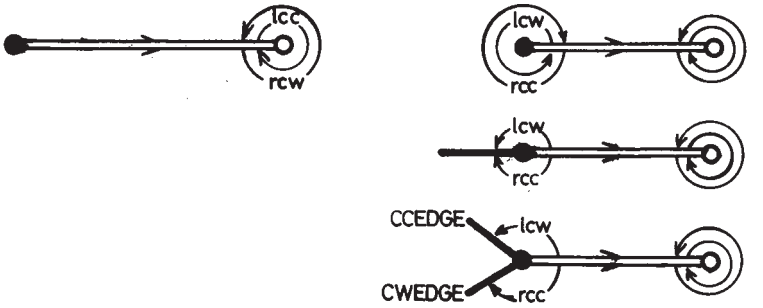


Figure F.3: Setting winged-edge pointers at the base vertex

3. When there are two or more edges at the base vertex, it means that the winged-edge links of the new edge are inserted between the winged-edge links of the ccedge and cwedge.

If vertex links are used, then the link between the new edge and the new vertex is the only one at the new vertex, so there is no problem; the link is made to point to itself. At the supplied vertex, the link between the vertex and the new edge is added between the link to the ccedge and the link to the cwedge. If the supplied vertex is an isolated vertex, the link between it and the new edge is the only link at the vertex and is made to point to itself. If the supplied vertex is a spur vertex, then the link to the spur edge and the link to the new edge are linked into a two-member circular list, so order is not important.

If geometry has been supplied, then it can be assigned to the new entities by the Euler operator. If the operator is used as a high-level tool, then the geometry is necessary; otherwise, it can be considered useful to add the geometry at the same time, but not obligatory.

As the new edge and vertex are extensions added onto an existing model, then there is no information handling problem. However, the new vertex position may lie outside the spatial extent measures associated with a face or loop, so these may be affected.

F.2 Make an edge and vertex (split an edge)

Required parameters: The edge to be split
Advised parameters: The start or end vertex of the edge to be split
The coordinates of the new vertex

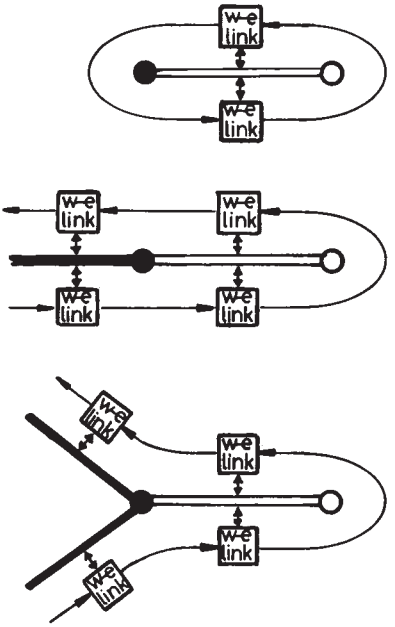


Figure F.4: Setting winged-edge links when adding an edge and a vertex

There are four possible outcomes for the operation:

1. The new edge can be added from the new vertex to the old start-vertex.
2. It can be added from the new vertex to the old end vertex.
3. It can be added from the old start vertex to the new vertex.
4. It can be added from the old end-vertex to the new vertex.

It is convenient, therefore, to make some arbitrary decisions and specify that the new edge has the same direction as the original edge, and the start or end vertex of the edge is specified as a parameter to control where the new edge is to be inserted. If the start vertex of the edge is given, then the new edge is inserted between the start vertex and the new vertex. If the end vertex is supplied, then the new edge is added between the new vertex and the end vertex. Obviously, if a vertex is supplied that is not attached to the edge, then an error should be flagged. Therefore, an alternative to supplying a vertex pointer is to specify a Boolean, TRUE for the start vertex, say, and FALSE for the end vertex. The operator itself can then find the relevant vertex, avoiding two dependent pieces of information.

It is advisable to specify the coordinates of the new vertex because the operator is often used for modifying existing, geometrically correct objects. Supplying the coordinates of the new vertex can enable the operator to adjust the geometry of the split edge to maintain consistency. This is needed if the geometry is connected directly to the edge, for example, if the edge curve is parametrised with 0 at the start vertex and 1 at the end vertex of the edge.

With winged-edge pointers, assuming that the new edge is to have the same orientation as the supplied edge, the topological changes involve inserting the new edge in place of the old edge at one vertex and connecting the new edge and the supplied edge around the new vertex. If the new edge is to be inserted between the start vertex of the supplied edge and the new vertex, then the edges clockwise and counter-clockwise from the supplied edge around the start vertex are found. The relevant winged-edge pointers of the new edge are set to be these edges, and they are adjusted to point at the new edge. The clockwise and counter-clockwise edges from the new edge around the new vertex are made to point at the supplied edge. Similarly, the clockwise and counter-clockwise edges from the supplied edge around the new vertex are made to point at the new edge, and the start vertex of the supplied edge is set to be the new vertex. See figure F.5, top.

If the new edge is to run from the new vertex to the end vertex of the supplied edge, then the Right ClockWise and Left Counter-Clockwise winged-edge pointers of the supplied edge are copied into the new edge. The edges referred to by these pointers are made to point at the new edge instead of the supplied edge. The Right Counter-Clockwise and Left ClockWise winged-edge pointers of the new edge are set to point at the supplied edge, and the

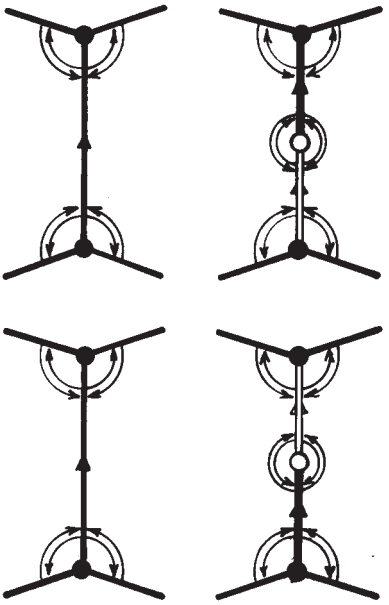


Figure F.5: Setting winged-edge pointers when splitting an edge

Right ClockWise and Left Counter-Clockwise pointers of the supplied edge are made to point to the new edge. The end vertex of the supplied edge is adjusted to point at the new vertex. See figure F.5, bottom.

If winged-edge links are used for linking edges, then one winged-edge link in the new edge is linked before and one after the winged-edge links of the supplied edge. If the new edge is inserted between the start vertex of the supplied edge and the new edge, then the right winged-edge link of the new edge is inserted before the right winged-edge link of the supplied edge, and the left winged-edge link of the new edge is inserted after the left winged-edge link of the supplied edge, assuming the winged-edge links are linked clockwise around a loop. If the new edge is inserted between the new vertex and the end vertex of the supplied edge, then the right winged-edge link of the new edge is linked after the right winged-edge link of the supplied edge, and the left winged-edge link of the new edge is linked before the left winged-edge link of the supplied edge, with clockwise linking. See figure F.6.

If vertex links are used, then the edge referred to by one of the links needs to be changed from the supplied to the new edge. The new vertex will have two links, one to the supplied edge and one to the new edge. The appropriate vertex or vertex link pointer in the supplied edge also has to be changed.

If the coordinates of the new vertex have been supplied, then the geometry of the new edge can be set from the geometry of the supplied edge, and the geometry of the supplied edge adjusted.

If the supplied edge was surrounded by a simple spatial extent measure, such as a box, then the box should be deleted or marked as invalid after the operation.

Finally, it is necessary to decide what to do about extra information, such as tags, associated with the supplied edge. The simplest thing to do is to copy them, although this may not be correct in all cases. This topic is discussed in section 8.3. In general, whenever an existing entity has to be split, it is possible that attached information has to be handled. If new entities are added as extensions, as in the MEV operation described in section F.1, then there is no information to handle. When splitting like this, it is often enough to copy the information because the new edge and the supplied edge should act as a unit until a third edge is added at the new vertex.

F.3 Make an edge and vertex (split a vertex)

Required parameters: The vertex to be split
Two edges bounding the set of edges to be moved or kept
Optional parameters: Coordinates of the new vertex
Curve of the connecting edge

Figure F.7 illustrates some alternative applications of the operator. The edge labelled as 'edge 1' is the edge clockwise around the vertex from the first edge to be moved. The edge labelled 'edge 2' is the edge counter-clockwise around the vertex from the last edge to be moved (figure F.7, top). If edge 2 is counter-clockwise around the vertex from edge 1, then applying the operator is equivalent to adding a spur edge and vertex (figure F.7, middle). If edge 1 and edge 2 are the same edge, then applying the operator is almost the same as splitting that edge, except that the other edges at the vertex are moved to the new vertex (figure F.7, bottom).

One way of implementing the operation is by setting the neighbouring edge pointers of the boundary edges (the first and last edges to be kept, and their neighbours) to point to the new edge, and then using the vertex partitioning utility described in section 3.2.3. The new edge and vertex are created and linked together so that the start vertex of the new edge is the supplied vertex, and the end vertex is the new vertex (say). The edge counter-clockwise around the supplied vertex from edge 1, edge 1a in figure F.8, and the edge clockwise around the supplied vertex from edge 2 (edge 2a) are found. The edge counter-clockwise around the supplied vertex from edge 1 and the edge clockwise around the supplied vertex from edge 2 are set to be the new edge. The edges clockwise and counter-clockwise around the supplied vertex from the new edge are set to be edge 1 and edge 2, respectively. If edge 1a

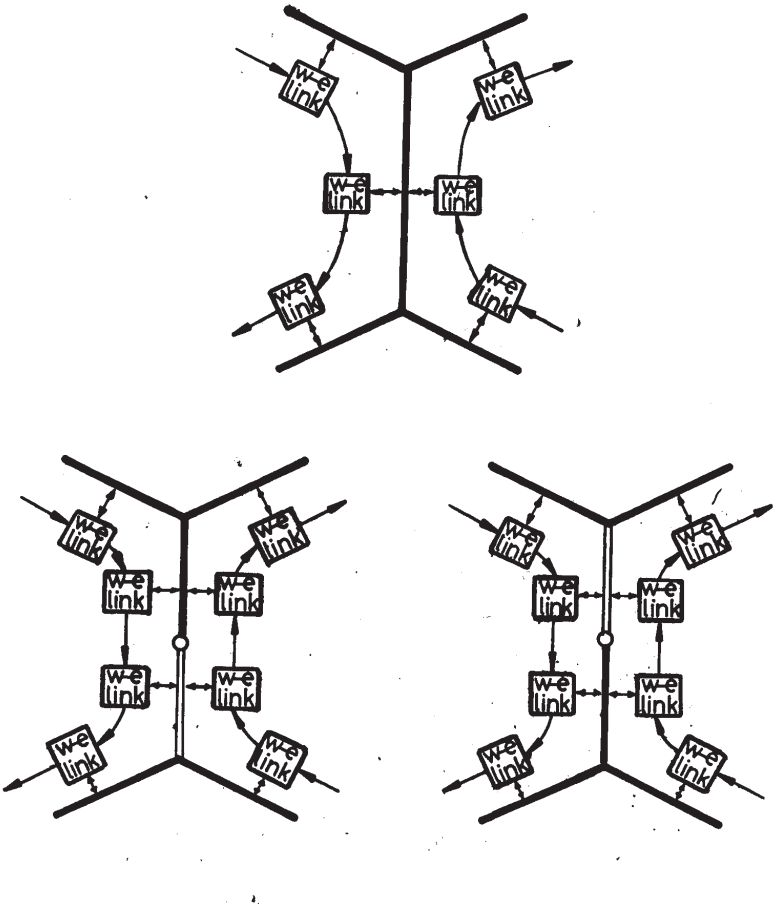


Figure F.6: Setting winged-edge links when splitting an edge

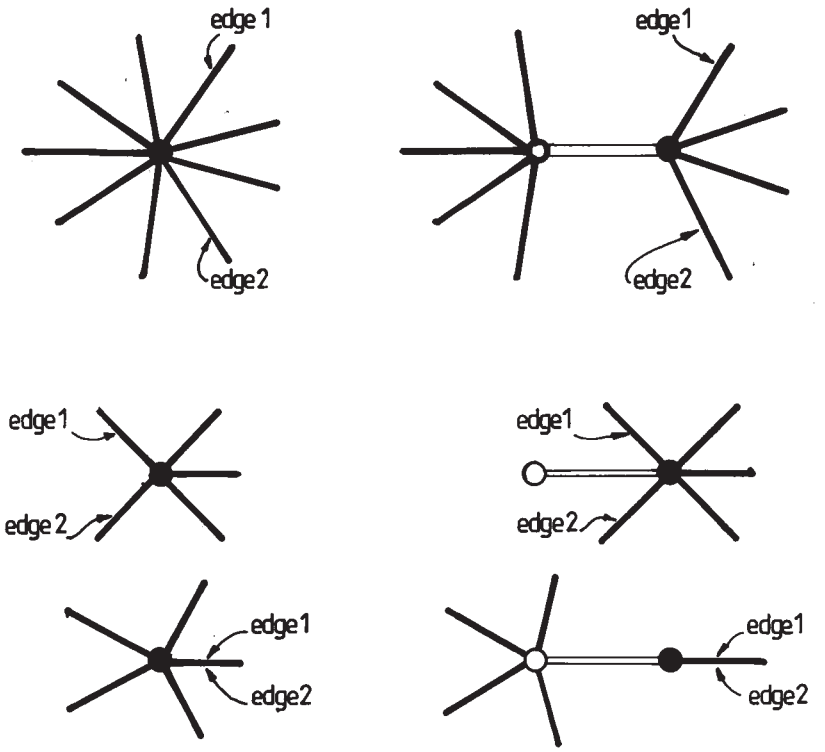


Figure F.7: Different cases when splitting a vertex

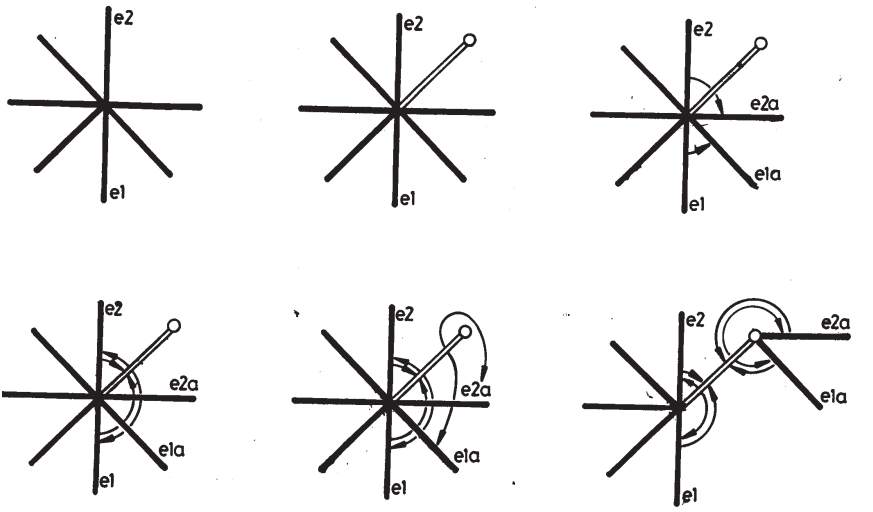


Figure F.8: Adding the new edge and vertex to split a vertex

is edge 2 (and by implication edge 2a is edge 1), then the edges clockwise and counter-clockwise around the new vertex from the new edge are set to be the new edge itself. Otherwise, the edge clockwise around the supplied vertex from edge 1a and the edge counter-clockwise around the supplied vertex from edge 2a are set to be the new edge, and the edges counter-clockwise and clockwise around the new vertex from the new edge are set to be edge 1a and edge 2a, respectively. Note that this leaves the structure in a funny state; it is repaired by the vertex partitioning utility that starts at the new edge and steps around the edges, clockwise or counter-clockwise around the supplied vertex, replacing the pointer to that vertex with a pointer to the new vertex.

With winged-edge pointers, the edges clockwise and counter-clockwise around the vertices can be set directly. With winged-edge links, the right winged-edge link of the new edge is inserted between the appropriate winged-edge links of edge 1 and edge 1a, which should be adjacent, and the left winged-edge link of the new edge between the appropriate links of edge 2a and edge 2, which should also be adjacent. With vertex links, the links clockwise from the edge 2a link around to the edge 1a link are moved to the new vertex. One link of the new edge is connected between the edge 2a and edge 1a links, the other between the edge 1 and edge 2 links.

If the coordinates of the new vertex do not match those of the supplied vertex, it may be necessary to adjust the geometry of the edges moved to the new vertex, and the surfaces of the faces meeting at the vertex. There is potentially so much change that this operator should be used with care, adding different vertex positions only with geometrically incomplete objects.

F.4 Make an edge and a face

Required parameters: The two vertices to be connected
Orientation edges at the two vertices
Optional parameters: An edge that is to be in the new face
The curve of the new edge
The surface of the new face
Logical to determine whether inner loops are to be checked

This is part of a collection of three operators that connect two given vertices with a new edge. The others are MEKH (Make Edge and Kill a Hole-loop) described in section F.6 and MEKFB (Make Edge and Kill a Face and Shell) described in section F.7. Although basically similar they are separated because they have slightly different parameter sets and different use-profiles. The operation described here, to make a face and an edge, is perhaps the most widely used. The operations can be simply differentiated from the status of the vertices. If they lie in the same loop, then the operation will be a Make Edge and Face operation. If the two vertices lie in different loops in the same face, then the operation will be a Make Edge and Kill Hole-loop operation. If the vertices belong to different objects then the operation will be the Make Edge and Kill a Face and Shell operation. If the vertices belong to different faces belonging to the same object, then the operation has been incorrectly specified. See Braid et al. [13].

As explained above, the basic information for the operation is given by the two vertices to be connected. The face to be split can be determined from the orientation edges. If only the face is given, then there will be problems if more than two edges at one or both vertices are adjacent to that face. Giving the orientation edges is also important to resolve ambiguity where the vertices have more than one face in common. The optional edge parameter can be used to determine which of the sets of edges will form the new face. If this parameter is not included, then the order of the vertices can be used to determine which will be the new face, say, making the new face always on the right side of the new edge. Obviously these extra parameters place an extra burden on the operator to check that they are consistent, that the two vertices and the edge parameter, if used, are adjacent to the face to be split.

Basically, the operation involves creating a new edge and face, connecting the new edge to the supplied end vertices, and then separating the original loop containing the two vertices into two loops, one of which surrounds the new face. Extra steps are involved, after this basic process. One of these steps involves moving any hole-loops from the original face to the new face that are surrounded by the new loop of edges. Another extra step is to sort out any information attached to the original face.

Loop partitioning is a geometric part of the operator, and it lies at a higher level than the basic topological function of the operator. If the operator is being used at a low level, for example, for sweeping when side faces are being

added, it should be unnecessary to perform this partition if the side faces are always added as new faces. Another consideration for the sweeping example is that adding side faces makes the face temporarily geometrically inconsistent, which will affect the 'loop-in-face' test. For these reasons, it seems advisable to perform this check only in certain cases, such as if the operator is applied directly as a user tool or if the geometry of the vertices and the curve of the new edge are consistent with the geometry of the face being split.

As when splitting an edge, coping with data attached to the face to be split is a difficult process that was discussed more in section 8.3. For some data, such as surface finish, it may only be necessary to copy the data. For other data, such as feature data, which may contain shape information or face identification, or names, or implicit shape modification elements, the task is more complex. It is desirable, if possible, to handle data at a low level to facilitate the work of the user. However, this task is harder to perform than the topological and geometrical functions of the operator.

If the face being split has a box surrounding it, or some other simple measure of spatial occupancy, then splitting the face affects this measure. The simplest method of handling this information is to delete the information and recalculate it when it is next needed. This avoids recalculating the measure extensively at intermediate stages of an operation, especially if the object is temporarily geometrically inconsistent. If the new face has a different surface to the original face, then it may also be necessary to note that any spatial occupancy measure associated with the facegroup (if used) should be recalculated.

The new edge is made and connected to the supplied end vertices in much the same way as described in the operators for making an edge and a vertex. If winged-edge pointers or vertex links are used, then linking the new edge will create two separate rings of edges that have then to be separated. With winged-edge links, there is slightly more of a problem because the chain of links has to be broken. The new edge can be connected as a sort of spur edge, with both winged-edge links adjacent in the loop. The winged-edge links can then be traversed, starting at any edge to be in the new loop, possibly one given as a parameter, and reconnecting the links to the new loop. When the first winged-edge link belonging to the new edge is found, the link pointers are reset and the traversal switches to a link belonging to one of the orientation edges, continuing until the starting link is again found.

Some care has to be taken if the two vertices belong to a hole-loop in the face, because it is necessary to decide whether the old or the new loop is the hole-loop, the other being the perimeter of the new face.

The start pointers of the old and new loops have to be checked and reset, if necessary. It is simplest if the pointers are reset to refer to the new edge or its links.

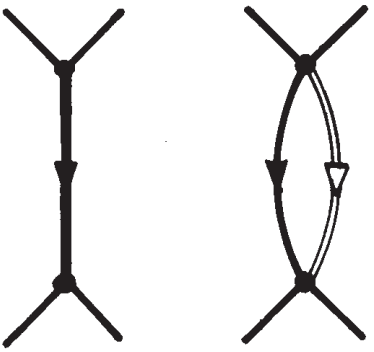


Figure F.9: Slicing an edge

F.5 Make an edge and a face (slicing an edge)

Required parameters: The edge to be sliced
Optional parameters: The surface of the new face

This is a special case of the Make Edge and Face operator to create a face bounded by two edges. It is used for such operations as sweeping graphs and expanding degenerate sheet objects to be volumetric objects. It is slightly more efficient than using the MEF operator, even though the effect is the same. The effect of the operator is illustrated in figure F.9.

A new edge is created and connected with the start vertex of the supplied edge as its start vertex and the end vertex of the supplied edge as its end vertex. Assuming that the new face is added to the left of the supplied edge, the left loop pointer of the new edge is set to be the left loop of the supplied edge, and the right loop pointer is set to be the new loop. The left loop pointer of the supplied edge is reset to point at the new loop. The new edge is connected clockwise from the supplied edge at the end vertex and counter-clockwise from the supplied edge at the start vertex.

Once again there is a potential problem with how to handle information attached to the supplied edge. There are two possible strategies: to copy it to the new edge or to reattach it to the new face. The two modelling operations that employ edge slicing, cited above, have different information handling requirements. When sweeping a graph, as described in section 6.3, edges are sliced and the two matching parts put on opposite sides of the resulting sheet object. In this case, it would seem logical to copy information attached to the supplied edge for the new edge. When expanding sheet objects to be volumetric objects, as described in section 6.7, the supplied edge represents a degenerate face, so it corresponds to the new face. It would seem logical,

therefore, to move information attached to the supplied edge to the new face. Which information strategy to apply cannot be decided at the level of the operator, and has to be resolved by the calling operation.

An optional specification, if winged-edge links are being used, is to give the winged-edge link of the edge to be sliced.

F.6 Make an edge and kill a hole-loop

Required parameters: The start and end vertices for the new edge
Orientation edges at these vertices
Optional parameters: A curve for the new edge

This is the second of the three Euler operators mentioned in section F.4, which add an edge between two vertices. The conditions for applying this operator are that the vertices lie in different loops in the same face.

This operation is easier to implement than that to Make a Face and Edge in that the edges affected are packaged in the loop to be killed, and so can easily be traversed with the traversal procedure described in section 3.2.1. If both loops are hole-loops, then, say, the loop that contains the second supplied vertex will be the one killed. If one loop is a hole and the other a perimeter loop, then the hole-loop will be deleted. If both loops are perimeter loops, both different, but in the same face, as with cylinders or cones with no fake edges, then the loop that contains the second vertex can be killed.

If winged-edge pointers or vertex links are used, then the edges in the loop to be killed are first traversed and the reference to that loop replaced by a reference to the loop that survives. A new edge is made and attached in the same way as already described in earlier sections. With winged-edge links, the process is slightly different because the process involves inserting the chain of links from the loop to be killed into the other loop. The new edge is added almost as a spur edge in the loop to be kept. The winged-edge links from the loop to be deleted are then inserted between the right and the left winged-edge links of the new edge.

F.7 Make an edge and kill a face and body (shell)

Required parameters: The two vertices to be connected
Orientation edges at each vertex
Advised parameters: The curve of the new edge

This is the third of the'connect vertices with an edge' type of operation mentioned in section F.4. This operator joins vertices in different objects. The main use for such an operator is probably to connect vertices in simple models,

allowing a user to define, say, separate elements of a shape and then connect them. It could also be used to connect vertices in more complicated objects as part of more complex operations such as joining coincident faces. These two uses are illustrated in figure F.10. This operator is equivalent to making one face a hole-loop in another face, with the Make Hole, Kill Face and Body (shell) operator (see section F.9), and then using the Make Edge and Kill a Hole loop operator, as illustrated in figure F.11. Separating the operation into these two steps may be clearer when, say, joining coincident faces (see section 6.4). However, if the two objects are wire objects, as in figure F.10b, then it seems more logical to use the MEKFB operator. It is largely a matter of taste, but it is unnecessary to limit the set of Euler operators to a spanning set and seems more sensible to implement operators that are logical for the desired purpose.

The important characteristic of this operator is the merging of the two object structures. This involves merging any "all elements in body" structures, and any information associated with the objects.

Once the object structures have been coalesced, the process of joining the vertices is very much the same as already described in section F.6. The supplied orientation edges are used to determine how the edge is attached to the two vertices; the loop of one face is inserted into the other.

As far as information handling is concerned, the operation kills a face and merges two objects. Information attached to the face being killed may, possibly, be simply deleted, although this depends in part on the geometry of the merged faces. Merging the objects requires merging any associated information sets, with all associated problems of conflicting information. One strategy is to leave the information merging process to the higher level operation using this operator so that conflicts can be sorted out where more knowledge about the goal of the application is available.

F.8 Make a vertex, a face, and a new object

Required parameters: None
Optional parameters: The vertex coordinates and the surface of the face

This operation is the basic Eulerian object creation operator. Basically it creates a new object and then a face and a vertex in this new object. If the modelling datastructure allows connections between loops and vertices, then the vertex has to be connected properly to the face as well. As the object is completely new, then there is no information handling problem.

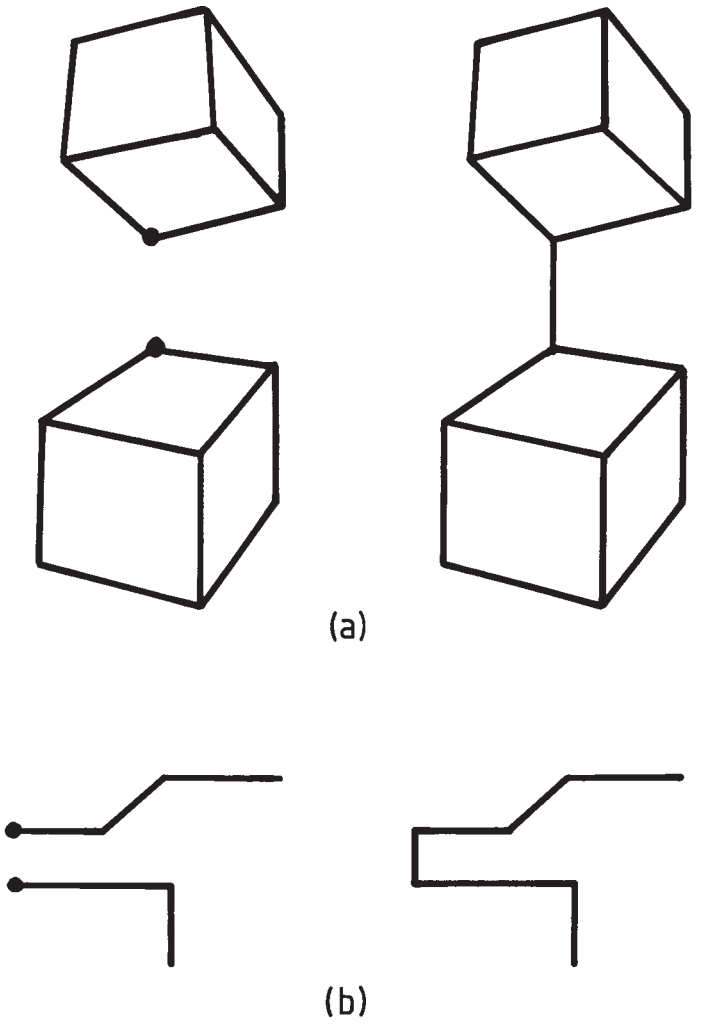


Figure F.10: Making an edge and killing a face and body (shell)

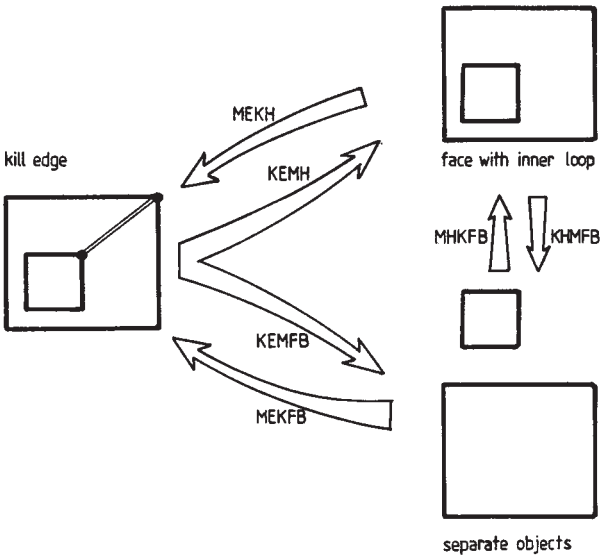


Figure F.11: MEKFB operator as a combination of MHKFB and MEKH operators

F.9 Make a hole-loop and kill a face

Required parameters: The face to be killed
The face into which the new hole-loop will be inserted

This operation will either merge two objects, increase the genus, or reduce the multiplicity of an object, depending on whether the faces are in the same object, lie in the same shell, or belong to different shells. If the faces belong to different objects, then the operation will merge two objects. It is equivalent to gluing a boss onto a face. If the faces belong to the same object, then it is necessary to use the shell traversal operation to determine whether the operation increases the genus (i.e., when the faces belong to the same shell), figure F.12 top, or decreases the multiplicity (when one or both faces lie in cavities in the object), figures F.12 middle and bottom.

The face to be killed should have no hole-loops. If it does, then they can be removed using the operator for making a face and killing a hole-loop, as described in section F.13. However, this should be done as a preliminary step, rather than calling that operator from this one, and turning this operator into a complex one, so the presence of hole-loops in the face to be killed is treated as an error condition. If the faces lie in different objects, it is necessary to merge the object structures, in the same way as described in the previous

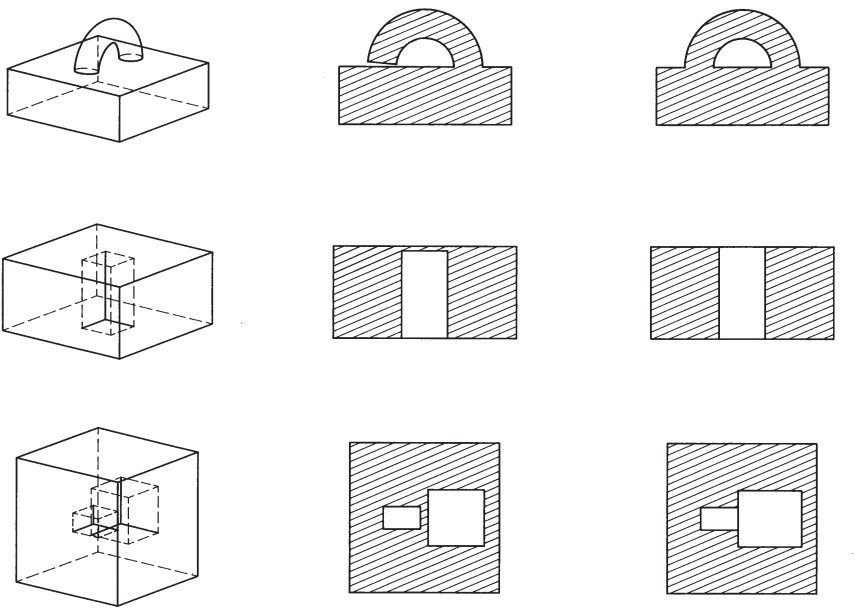


Figure F.12: Illustrations of make a hole-loop and kill a face operation

section. Once the preliminary steps have been made, the perimeter loop of the face to be killed is simply transferred to the destination face as a hole-loop, and the first face killed.

Some extra housekeeping steps need to be performed if the two faces belong to the same object. Genus is not specifically represented in the object datastructure, so if the faces are in the same shell, there is no problem, and no need to reorganise the datastructure. However, if they are in different shells in the same object, then it is necessary to merge the shells. This check is best performed before the new hole-loop is made. It can be easily performed by traversing all topology reachable from the face to be killed until the destination face for the hole-loop is found. If this destination face is not found, then the faces are in different shells, and a shell merge has to be performed. If the destination face is found, then the operation increases the genus and no work need be done.

F.10 Make a vertex and a hole-loop

Required parameters: The face in which the new hole-loop is put
Optional parameters: The coordinates of the new vertex

This operator is useful when inscribing on faces as the initialisation of a new hole-loop. It simply involves creating a new vertex in the object containing the supplied face, a new loop in the face, and setting the loop to point at the vertex, and the vertex to point at the loop. If there is no direct connection between loops and vertices in the datastructure, only, say, between edges and loops, then there is a problem in maintaining the link between the loop and the vertex. Otherwise it is possible to treat the isolated vertex loop in much the same way as other loops. Ideally there should be a link between vertices and loops in the datastructure.

F.11 Kill an edge and vertex

Required parameters: The edge to be killed
The vertex to be killed

This operator is the inverse to the Make Edge and Vertex operation described in sections F.1, F.2, and F.3. It has, therefore, three main functions: to remove spur edges and vertices, to remove vertices at which there are only two edges (i.e., to merge two edges by removing their common vertex), and to coalesce two vertices. In fact these three functions can be thought of as removing an edge and coalescing its end vertices. It differs from the edge removal operator described in the next section, F.12, in that it also removes a vertex. If, say, the edge is a spur edge, then the operation described in F.12 will remove the edge and leave the spur vertex as a hole-loop in the face.

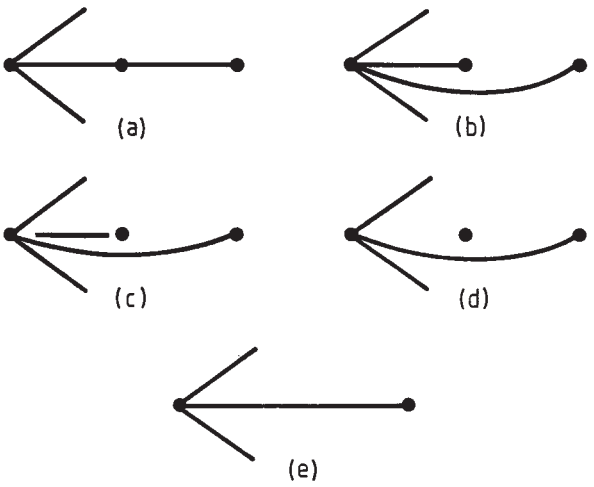


Figure F.13: Killing a vertex and merging two edges

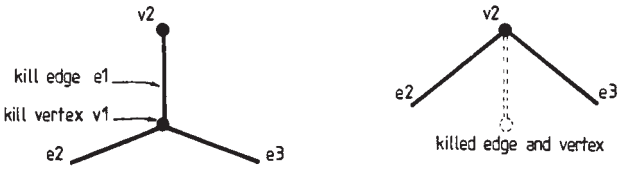


Figure F.14: Killing an edge and merging two vertices

Simply stated, the operator traverses all edges at the vertex to be killed and reattaches all but the edge to be killed to the other vertex clockwise from that edge. The spur edge and vertex are then killed. This process, applied to merge two edges, is illustrated in figure F.13. The original topology is shown in figure F.13a. The edge to be kept is reattached, as shown in figure F.13b. The edge to be killed is detached from both end vertices, as shown in figure F.13c, and deleted, as shown in figure F.13d. Finally the vertex is killed, leaving the topology shown in figure F.13e. Another example is shown in figure F.14. Edge e1 and vertex v1 are killed, and edges e2 and e3 are reattached to vertex v2. Note, also, that if the modeller has not been designed to cope with edges with the same start and end vertex, then it is necessary to check for other edges besides the one being deleted, which connect the vertices being merged.

As far as information handling is concerned, the three cases should be

treated differently. If a spur edge and vertex are deleted, any information attached to the edge or vertex can be deleted. If two edges are merged by removing the common vertex, then it may be necessary to merge the information sets attached to the edges. If two vertices are to be coalesced by removing the common edge, then it may be desirable to merge information sets connected to the vertices.

Finally, there is the question of how the geometry is handled. If a spur edge and vertex are deleted, then there is no problem. If two edges are merged, then it may be desirable to alter the curve attached to the surviving edge, unless the deleted edge had zero length, or the geometry of the edges is independent from the start and end vertex positions. If two vertices are being merged, then there will be a problem with every edge originally attached to the deleted vertex, unless the vertex positions coincide. This, like the information handling problem, requires that the three cases are differentiated.

F.12 Kill an edge

Required parameters: The edge to be killed
Optional parameters: A Boolean or a face pointer to indicate which face is to be killed in the Kill Face and Edge operation

This is (almost) the inverse operation to those creating an edge between two vertices. The effect of the operation can be determined from the attributes of the supplied edge. If it is a spur edge, then the operator makes the spur vertex a hole-loop, or both end vertices into hole-loops if both are spur vertices. If it is a wire edge, then the edge is killed and a hole-loop is made. Otherwise, the operator removes a loop. Note that this operator is not inverse to the MEKFB, Make Edge Kill Face and Body (shell), operator. Both the MEKFB and the MEKH, Make Edge Kill Hole, operator produce wire edges, but the kill-edge operator described here results in a KEMH for wire edges by default. The KHMFB operator described in section F.13 can then be used, if desired. However, for the special case where the object contains only wires, then it is reasonable that the effect of this operator is to kill an edge and make a face and a new object (KEMFB).

If both start and end vertices of the edge are spur vertices, then the edge is disconnected from both. One of the vertices is made into a separate hole-loop. If the edge was the only edge in the object, then one vertex should be made into a new object, and the hole-loop into the perimeter of a new face (figure F.15).

If only one vertex is a spur vertex, then that is made into a hole-loop. The edges clockwise and counter-clockwise from the supplied edge at the other vertex are made to refer to each other, and the supplied edge is deleted (figure F.16).

If the edge is a wire edge, then first the edge is disconnected from the

Figure F.15: Killing the last edge in a body

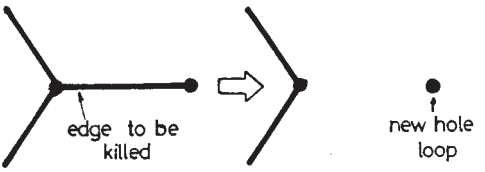


Figure F.16: Killing a spur edge

start and end vertices. The edges counter-clockwise and clockwise from the edge around each vertex are set to point to each other. This separates the original loop into two portions, and it is then necessary to traverse the edges in one portion, moving them into a new loop. This is done, starting from the supplied edge, and following edges successively using the winged-edge pointers, winged-edge links, or vertex links until the supplied edge is again found. Once done, it has to be decided whether the new loop is a hole-loop or a perimeter loop. If the supplied edge belonged to a hole-loop, then both portions will be hole-loops. If not, then a simple check is to calculate the approximate area of both loop portions, making the loop with the greater area into the perimeter, and the other into a hole-loop. However, as mentioned above, if the object containing the supplied edge contains only wire edges, then instead of making one portion a hole-loop and one a perimeter, one loop should be moved into a new face in a new object (figure F.17).

If the edge is neither a spur nor a wire edge, then the operation kills a loop. If both loops of the edge are hole-loops, it performs a KEHG, Kill an Edge, a Hole-loop, and reduce the Genus. If only one loop of the edge is a hole-loop, then the operator should kill the face on the other side. If both loops of the edge are perimeter loops, then the supplied Boolean or face pointer can be used to determine which is to be killed and which survives.

The KEHG variant of the operation can occur if the edge is part of a through hole in a lamina, as illustrated in figure F.18. The operation replaces the references to one hole-loop in the set of edges in the loop with references to the other hole-loop, thus removing it from one side of the object and making the surviving edges into wire edges. It can be thought of as performing a MFKHG (Make a Face, Kill a Hole-loop and decrease the Genus) followed by

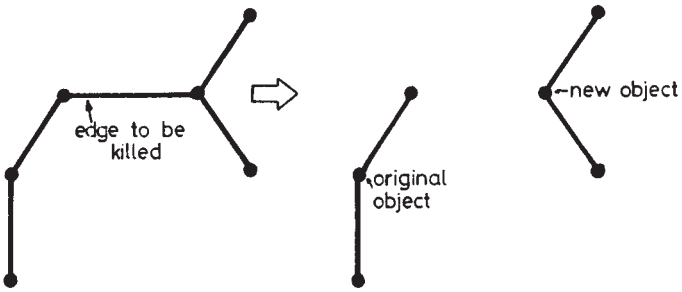


Figure F.17: Killing a wire edge

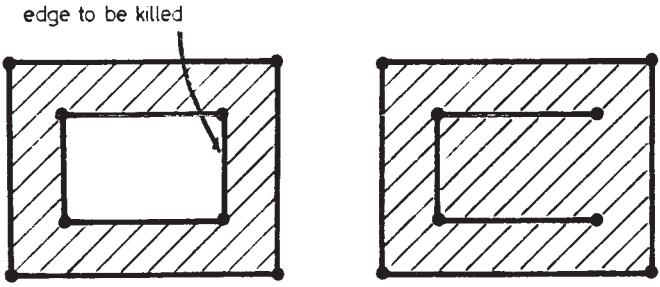


Figure F.18: Killing an edge separating two hole-loops

a Kill Face and Edge operation.

The KFE, Kill a Face and Edge, operator merges two faces by removing the edge between them. Starting from the supplied edge, the loop to be killed is traversed, replacing references to that loop with references to the loop on the other side of the edge. The edges clockwise and counter-clockwise from the edge at both start and end vertices are made to point to each other. Any hole-loops in the face to be killed are moved to the surviving face.

With the Kill Face and Edge operation, it is necessary to merge the information sets attached to the two faces being merged. The Kill Edge, Hole-loop, and Genus operation may affect any high-level descriptions of the object. The other operations probably do not affect information sets, because wire edges and spur edges are both more common as intermediate forms during modelling operations rather than as accessible parts of the model for users.

One final point, concerning the geometric side of the operation, it should be noted that there may be practical restrictions on whether geometry can be removed. One possible representation for cylinders, say, uses a single wire edge in the curved surface. A possible representation for cones might have

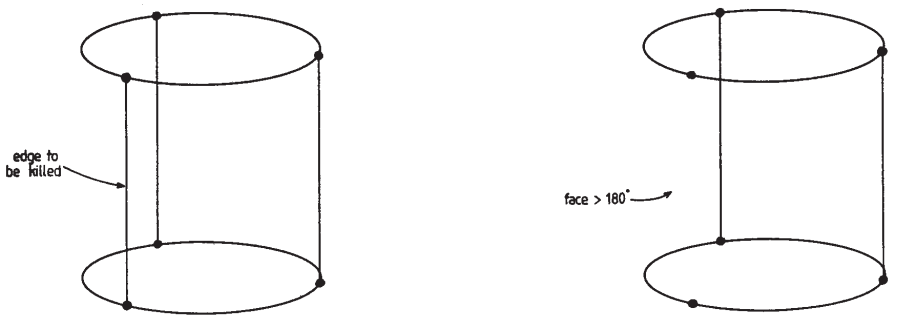


Figure F.19: Killing an edge between two curved faces

a spur edge from the cone apex to the conical face boundary. These edges may be necessary for the correct functioning of other parts of the modeller, so it may not be possible to simply remove them. Similarly, if there are two adjacent curved faces, as shown in figure F.19, then removing a common edge will create a single face extending through more than 180 degrees, which may also cause problems. It is necessary to check for these conditions even though the checks may be put into the calling modelling operations rather than these low-level ones.

F.13 Kill a hole-loop and make a face

Required parameters: The hole-loop to be killed

This is the inverse operation to that described in section F.9 and may decrease the genus, increase the multiplicity of an object, or create a new separate object.

The topological changes involved are relatively simple. A new face entity is created, and the supplied hole-loop is removed from the face that it bounds, and set to be the perimeter of the new face. However, thereafter there are possible organisational changes according to whether the operation changes the genus or the multiplicity. The surface of the new face is the same as that of the original face of the hole-loop, but it is negated. This should not be a problem for most general surface representations, but it may cause problems if the surface is a complex one with separate topological representation.

Once the hole-loop has been 'blocked-off' with the new face, the topological traversal utility can be used to find out if the new face belongs to the same shell as the original face of the hole-loop. If it does, then the operation reduced the genus of the object (figure F.20). If the faces are not connected, then either a separate object has been made (figure F.21, top) or a new cavity has been

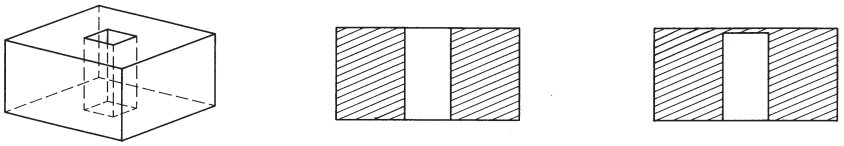


Figure F.20: Making a face, killing a hole-loop, and decreasing the genus

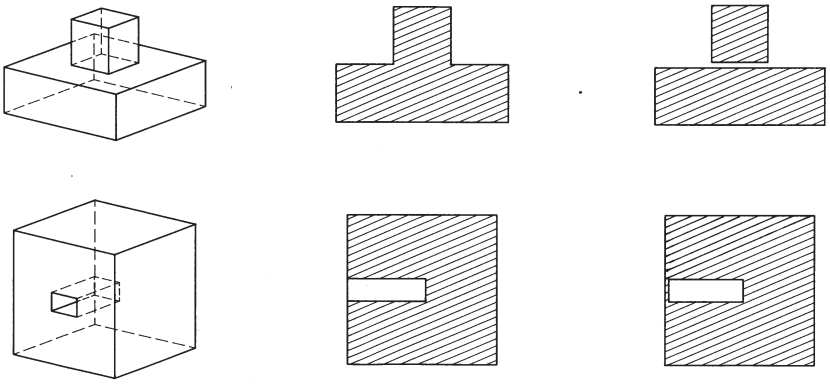


Figure F.21: Making a face, killing a hole-loop, and increasing the multiplicity

created (figure F.21, bottom). The same sort of classification as that used by Kyprianou [74] for feature representation can be used to sort out the cases. If the original loop contained only convex edges, and so the new face is bounded by only concave edges, then the new shell is an internal cavity in the object. If the original hole-loop contained only concave edges, and thus the new face is bounded by only convex edges, then a new object has been created and should be put into a separate object structure. If the hole-loop contains a mixture of convex and concave edges, as with the object in figure F.22, then an error should be signalled, because the result will be geometrically incorrect. The object should probably be modified before applying the operator.

F.14 Kill a vertex and a hole-loop

Required parameters: The vertex to be killed

This is a very simple operator to remove the special case of a hole-loop consisting of a single vertex from the datastructure. It simply removes the hole-loop from the face which includes it, and the vertex from the object which contains it. It is useful, for example, for finishing off stepwise deletions of contours in a face.

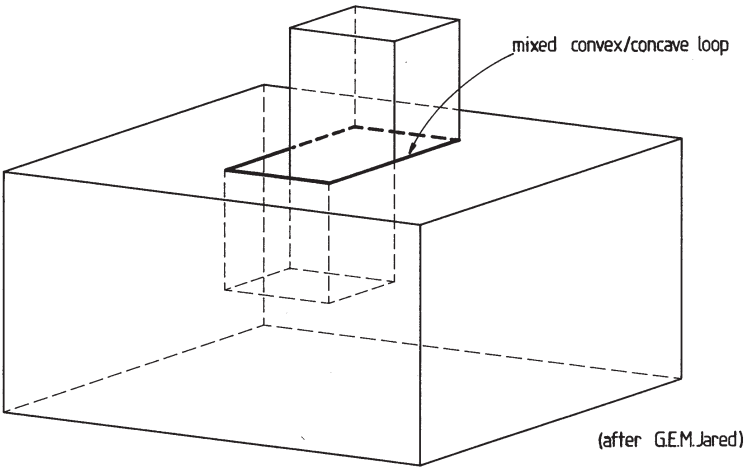


Figure F.22: Mixed concave/convex loop

F.15 Merge coincident vertices

Required parameters: The vertices to be merged
Optional parameters: The face to be collapsed

The operator merges two coincident vertices, creating a new face or removing a hole-loop (see figure F.23). The operator combines two Euler operators: the MFKV (Make Face and Kill Vertex) operator and the KVH (Kill Vertex and Hole- loop) operator. It is thus very closely related to the Make Edge and Face operator and the Make Edge Kill Hole-loop operator. The operator can be thought of as adding an edge between the coincident vertices and then deleting this zero-length edge and one of the vertices.

If one or both vertices are spur vertices, then the merging process is easier because it is only necessary to determine how one extra edge should be inserted at the other vertex. If two or fewer edges are adjacent to the face being collapsed at the first vertex, then the edge adjacencies can be simply determined. If there are more than two, a geometric check is necessary to find out between which pair of edges at the vertex the spur edge should be inserted. If neither vertex is a spur vertex, then all edges at the second vertex should lie between one pair of edges at the first vertex.

F.16 The null operator

There are several useful changes to the topological structure of a model that do not create or delete topology, but simply rearrange it. They form repeated

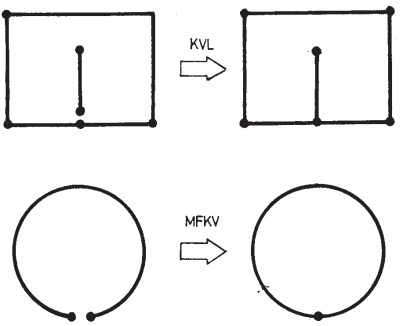


Figure F.23: Merging two vertices, creating a new face, or killing a hole-loop

steps or well-defined steps in operations, so they can be classed as functions in their own right. They correspond to the null operator in the list of Euler operators in table 1.

Examples of this type of null operator are contained in the BUILD/FFSolid chamfer operation, implemented by Graham Jared (see [12]), and another one in the 'join coincident faces' operator (see section 6.4).

There are two examples of null operators in the chamfer operator. One 'slides' the end of an edge along a second edge (figure F.24a). The other 'slides' an edge along two others (figure F.24b). Both are equivalent to a Make Edge and Face and Kill Face and Edge, but, as with the 'slice edge' operator described in section F.5, they are more efficient because they are very specific.

In the face joining process, the step to join two edges with one common vertex, as shown in figure F.25, corresponds to a null operator. It, too, can be simulated using other Euler operators, a Make Edge and Face, a Kill Face and Edge, a merge vertex variant of the Kill Edge and Vertex operator, and finally a Make Edge and Vertex operator.

These examples can be considered as Euler operators because they rearrange the topology. They differ from, say, operations that change just the geometry of a topological element because these can be considered as basically pointer changes rather than geometrical. They correspond to circular 'tours' over the Euler network and illustrate the fact that objects with different topology map on to the same node in the network. Their use can be justified for practical reasons, because they are more efficient than performing the same changes using other Euler operators.

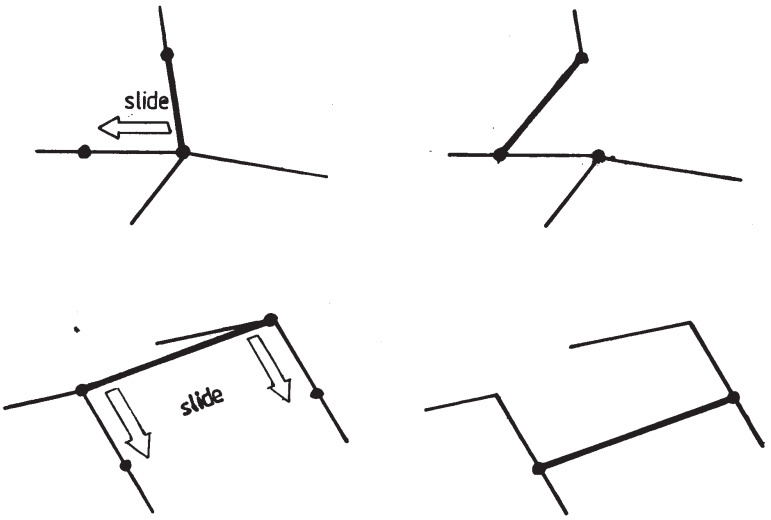


Figure F.24: Null operators for chamfering

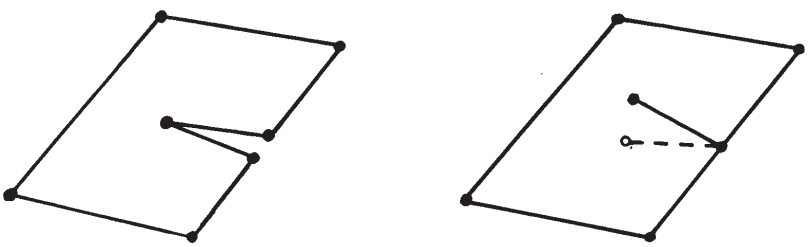


Figure F.25: Null operator for joining two edges

Appendix G

Modelling implementation notes

The following are some notes on implementations of operations in chapter 6.

G.1 Boolean operations

The implementation of the Boolean algorithm was not very efficient because the code was developed in FORTRAN IV. It is useful to use subsidiary structures, lists of edges and vertices, to assist matching elements in the intersection boundary. Another problem, that of re-implementing the joining procedure, required more work than was possible in the time available, so it was not completed. The proposed joining scheme is described later in this section.

A more serious problem for the GPM implementation of the algorithm described in section 6.1 concerned geometric tolerances and the way the geometry changes during modelling. Such problems remain as serious inherent handicaps for the implementation. An illustration of one particular problem, originally stated by Solomon (private communication), is shown in figure G.1. In the figure, an infinite curve is found to enter the face, but the intersection where it leaves the face is not found. In the BUILD implementation developed by Braid and Smith, the face–face comparison creates the intersection curve as described in section 6.1, but when comparing this curve with one of the faces, the surface of the other is also used. The edge/surface comparison is more stable than the edge/curve comparison, so this makes the BUILD implementation slightly better than the GPM implementation. However, the problem of geometric inexactitude has not been adequately solved because it is inherent to the nature of computers.

There are some advantages in using parametrised curves in the model because parameter values of points are commonly used, and so it is unnecessary to keep reparametrising for every interrogation, as has to be done with

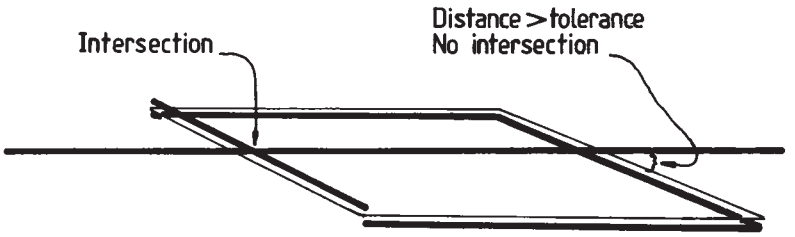


Figure G.1: Effect of geometric inaccuracies on intersections

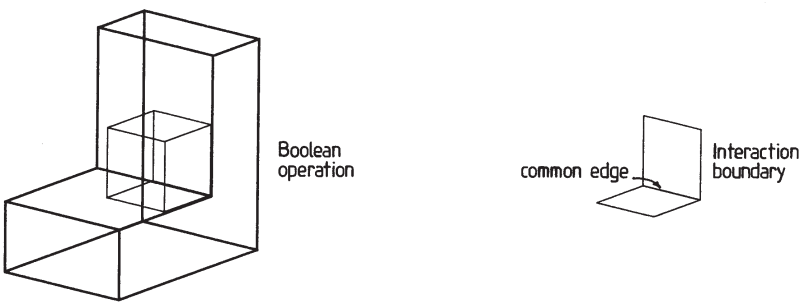


Figure G.2: Overlapping intersection rings

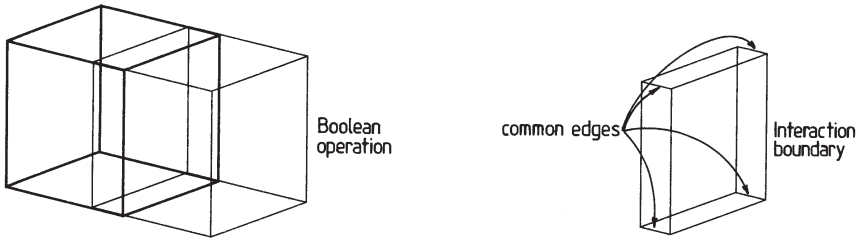


Figure G.3: Multiple overlapping intersection rings

non-parametrised representations. The disadvantage is that they need to be reparametrised if associated with edges that are altered.

The original GPM implementation used the uniform Boolean method of Braid. This method has several drawbacks. The extra negation steps are time-consuming and can be avoided; they are not necessary for obtaining the original intersection edges, just for the final step. Another, more serious, drawback with the original implementation is that the intersection rings sometimes overlap; that is, there are common edges, as illustrated in figure G.2. It was necessary, therefore, to introduce a test, when joining the objects around the rings, to make sure that only matching edges were joined. Even with this test, the joiner was only partly successful. Examples such as that in figure G.3 proved too serious a challenge to the original version to continue trying to keep patching it up. Instead a second version was started, although only partly completed.

The second version of the joining process takes the edges from the two groups and joins them as pairs, rather than as complete rings. The case analysis for the edges is shown in figures G.4 to G.7. They represent sections through the edges; each of the lines represents an adjacent surface. The implementation was only partly completed, and so there are no results by means of which the two joining methods can be compared. The lines in the diagram represent cross sections through the edge and represent the comparative positions and orientations of the faces meeting at the edge.

G.2 Sweeping/Swinging

The implementation of the sweep algorithm in both BUILD and the GPM volume module has several problems connected with curved geometry. The current imperfect evaluation of movable edges for circular sweeping does not allow 'undoing' of circular sweeps; otherwise, the sweep operation is self-inverse; i.e., sweep can be used to undo a previous sweep operation. What is required is a better analysis of face and edge extent. This is also connected to the use of 'fake' edges for artificially dividing curved faces. If the modeller

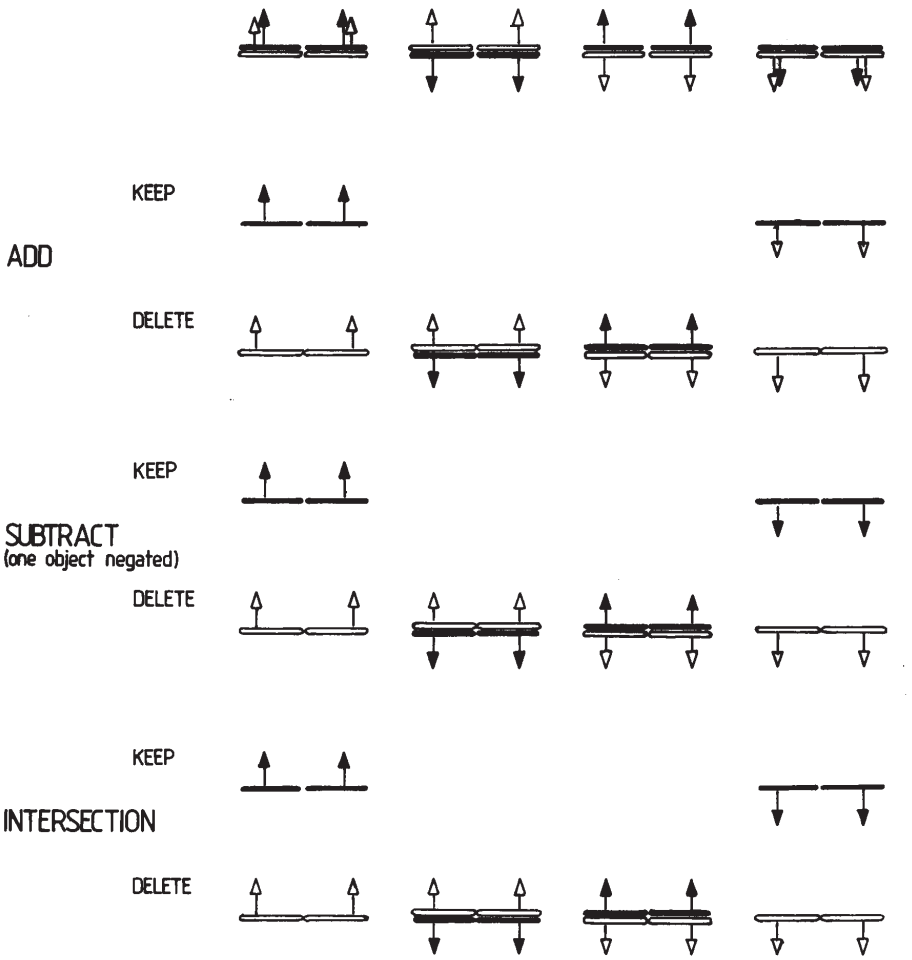


Figure G.4: Case analysis for edge joining 1

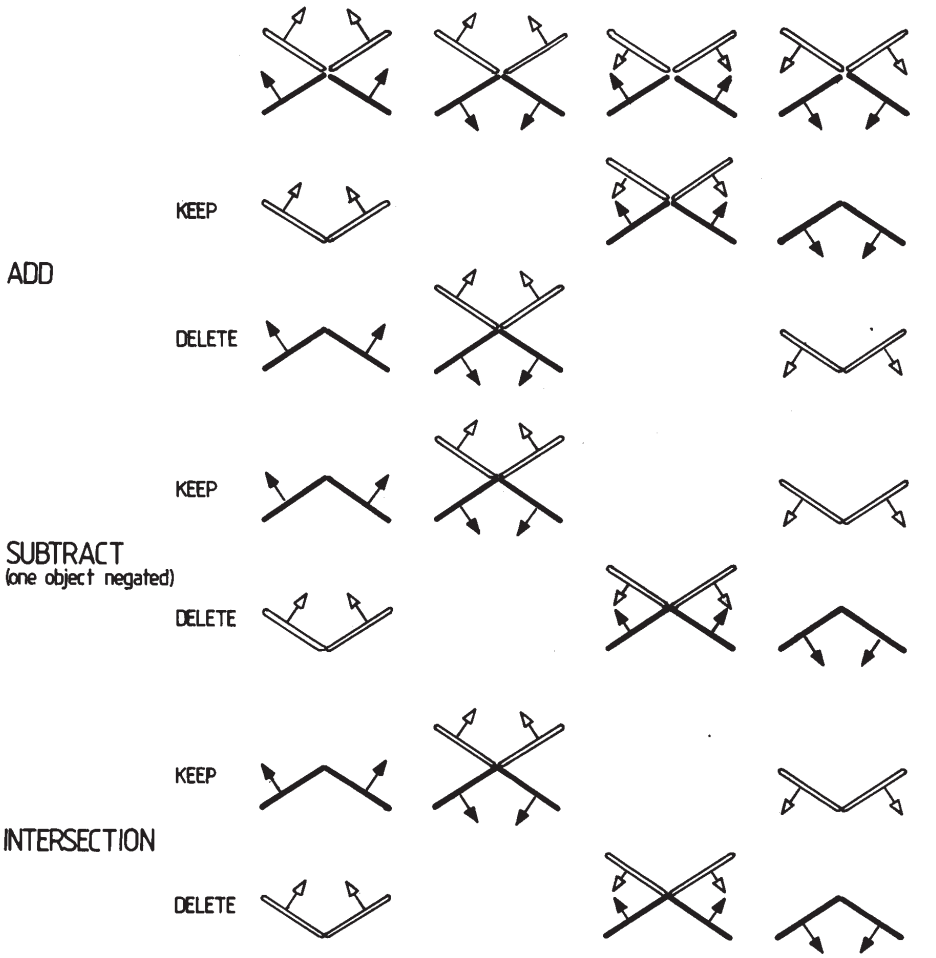


Figure G.5: Case analysis for edge joining 2

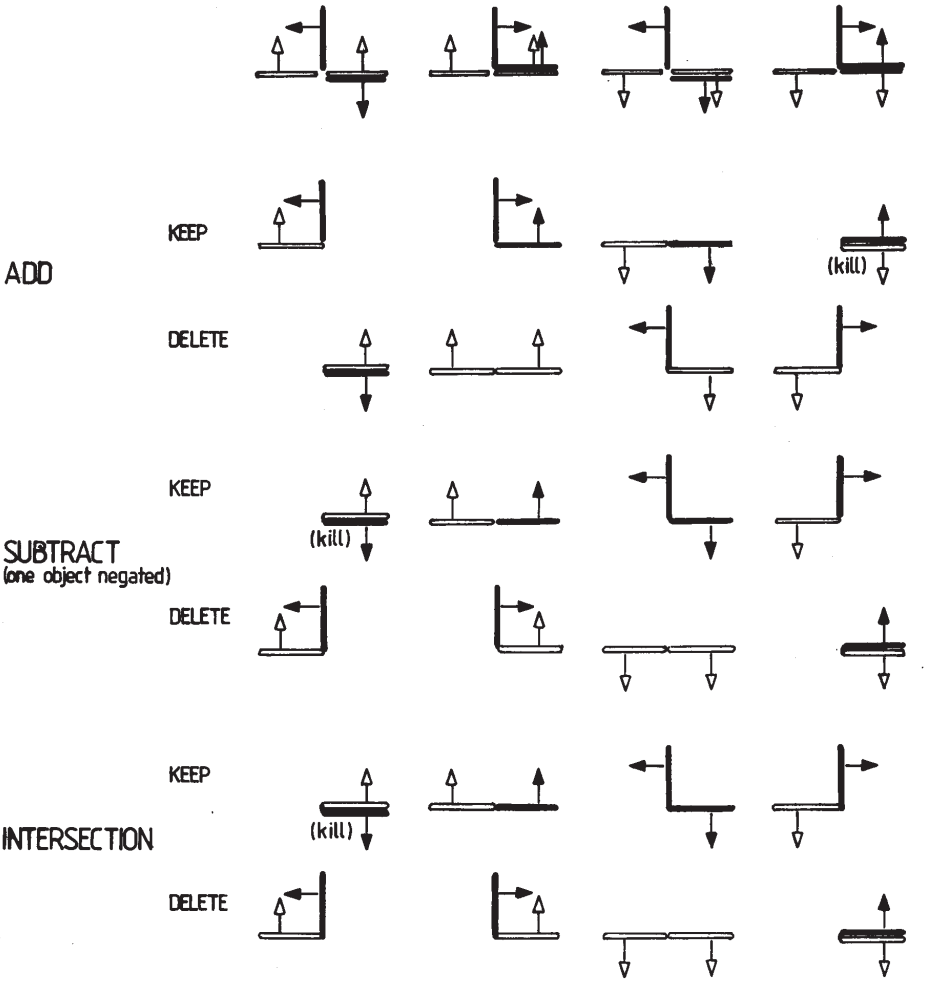


Figure G.6: Case analysis for edge joining 3

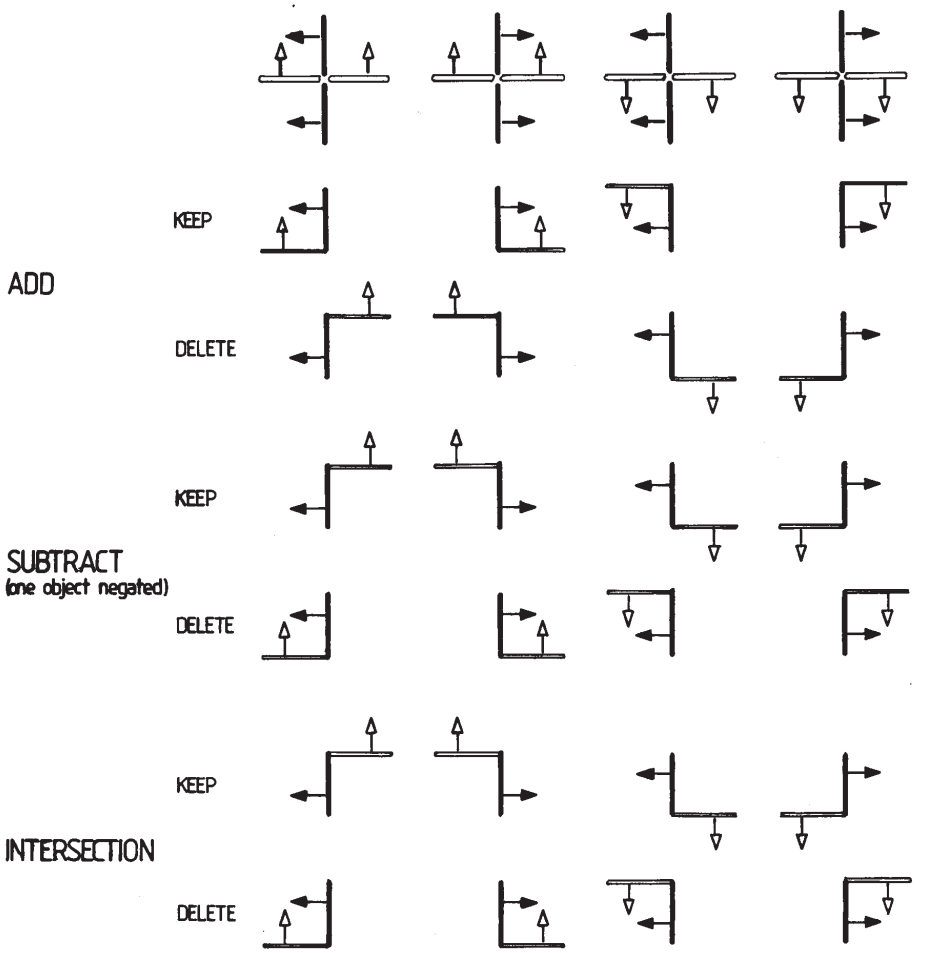


Figure G.7: Case analysis for edge joining 4

does not need these, then the classification problem disappears.

There is some inefficiency due to sweeping edges separately. Where two adjacent edges in the loop are movable, the first sweep will create extra entities that are removed by the second sweep. This is a direct cost of having stepwise modelling. If the conditions for a movable vertex are widened, so that a vertex is allowed to be moved when all adjacent edges in the face being swept are movable and all other attached edges are either extended or shortened, then it can upset the calculation of the swept side surfaces.

The current code for obtaining a swept vertex can only negotiate past one adjacent face, not up multi-storey sides. This means that sweeping can create objects with coincident faces, i.e., non-manifold objects. It is unclear how much of a problem this is. If there is an object with several adjacent co-planar faces then it is not in a minimal form, and the Booleans (both in BUILD and the GPM volume module) reduce the object to a minimal form. Other operations, too, tend to remove redundant edges, so for planar sweeping it would seem to be an academic problem rather than a real one. Circular sweeping is somewhat different. Here, there can be several adjacent faces in the same surface, separated by the so-called 'fake' edges, to ensure faces do not exceed 180 degrees in extent. There seems no obvious reason why the method used to obtain a swept vertex could not be called recursively to solve this problem.

Sweeping faces composed entirely of wire edges has to be handled as a special case, because wire edges are, by the definition given in section 6.2, movable. However, it is reasonable to treat these objects as special cases, since wires contained in faces are somewhat different from a face containing only wires. Section 6.3 deals with sweeping the special wireframe models in the GPM volume module. The sweeping method requires that each edge, when swept, generates a pair of faces, one on each side of the new sheet object. This differs from sheet objects generated by the original BUILD where there was a separate face for each swept edge on one side of the sheet object and a single 'rubber' face on the other side.

If an edge being swept is adjacent to a face smaller than that created by the swept edge (see figure G.8), i.e., the edges of the side face lie inside the edges created by sweeping the end vertices of the edge, this is not detected and handled properly. This is a weakness in the algorithm; it is not coped with by the algorithm as it is now, and would require more work to be done to analyse the new side edges. The solution might require the sort of imprinting process described in section 6.9 and a face–face glue to make the object correct.

There is a problem with using transformation matrices to modify the geometry of the face for swinging. If the swing is in a complete circle, especially if it is done in many steps, numerical errors can creep into the geometry causing problems for the exact face-face glue.

In the GPM volume module, degenerate or sheet objects (laminae and shell objects) and volumetric objects were different classes of object, and there was a convention that the sides of a sheet object are contained in separate main

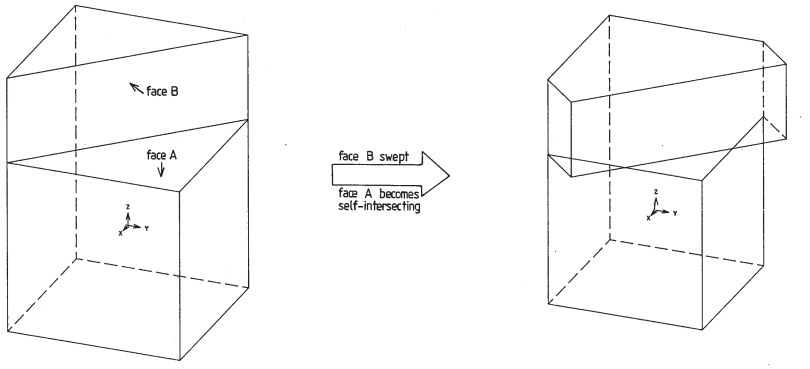


Figure G.8: Generating swept faces adjacent to smaller faces

facegroups, and that volumetric models had one main facegroup for each separate shell of faces, edges, and vertices [48]. This means that when converting a sheet object into a volumetric model, there was a pre-processing step to transfer the topology and geometry to a different class of owner object, and to merge the two main facegroups into a single one, although they were preserved as 'children' of the new main facegroup. In BUILD, there is a different set of criteria when sweeping sheet objects. There is only one basic class of object model, so there is no need to convert between model types. Also, there is only one party (or facegroup) level, with no special interpretations for the facegroups. What is required depends on the basic decisions made when the modeller is planned, so that they are uninteresting when describing the basic algorithm.

G.3 Sweeping wireframe models

Several problems are associated with this operation. Trying to convert general wireframe models into Eulerian objects is not an easy problem. The best idea would seem to be to place severe restrictions on the use of wireframe models and avoid the problems rather than solving them.

The first algorithm described, implemented in the GPM volume module, could be adapted to produce non-manifold structures of the type defined by Weiler [143] instead of the GPM sheet object structure. Instead of extension edges, a suitable non-manifold edge structure would be created. This is more difficult with the second, untried algorithm because the associativity between the matching edges is lost.

G.4 Gluing coincident identical topology

G.4.1 Face joining

The face joining process described above has been implemented in the FFSolid modeller. An additional case, not covered by the algorithm as described, is where there is a single vertex common to both faces. This can occur when a triangular face with one vertex on the rotation axis is swung in a complete circle. Although this could be flagged as an error case by the swing process, the join procedure was extended to cope with this by splitting the vertex and creating a zero length edge common to the faces.

A pre-condition for the algorithm to work is that both faces have the same topology and geometry. This can be checked either as a first step or for each pair of edges being joined. Checking the geometry and topology before joining the faces is preferable because it avoids stopping the joining process prematurely with the faces partly joined.

G.4.2 Edge joining

The operation is primarily useful for joining two coincident edges after an edge has been swung in a complete circle, or for joining sheet objects together at boundary edges. However, the operation can be applied under different conditions that do not prevent the operation working, but may cause odd effects. If, for example, the faces adjacent to one edge lie on one side of an object, and the faces adjacent to the other edge lie on the opposite side of the object then the net effect may be to create a slit in the object bounded by the two edges. If one or both edges is an internal edge, i.e., lies between faces in the same side of an object, then problems may arise in the partitioning of the faces into facegroups. As described in section 3.4 it is useful to have a convention that faces on each side of a degenerate sheet object are contained in separate structures. The safest course of action is to limit the scope of the operation to joining boundary edges, that is, edges that lie between different sides of an object.

There is not the same range of problem cases where the edges share a vertex as where there is no common vertex. This is because both edges must, necessarily, belong to the same object. This limits the cases to two basic ones: joining boundary edges and joining non-boundary edges. Again, it seems best to limit the scope of the operation to only joining boundary edges. There is an inverse operation to the glue operation, for sheet objects, that of splitting a sheet object at an edge. As implemented in the GPM volume module, the splitting operation is supplied with one edge and searches for a matching edge on the opposite side of the object. If the edges that are joined are non-boundary edges, then, after the operation, the edge left has no corresponding edge on the opposite side of the object. Thus, a glue followed by a split will either produce an error or a object with different topology from the origi-

nal object (depending on exact implementation and original topology of the object). However, when joining boundary edges, a glue followed by a split will result in an object with the same topology as the original object. If it is necessary to join non-boundary edges, then it is better to supply a separate function, with its own restrictions, so that the meaning of the operation becomes clear to a user.

G.5 Reflect

Although it is a local operator, it is desirable that the reflect operation perform a check so that it does not create self-intersecting objects. The check can be performed in two ways: quick, cheap but not entirely accurate; or slow and exact.

The quick check involves testing the positions of all vertices and the centres of all curved edges to see whether they straddle the reflection plane. This test is sufficient for checking planar objects and objects with cylindrical or conical faces not extending through more than 180 degrees, but it can miss intersections in other cases. It is, however, relatively simple to implement and fairly quick.

The accurate check is very similar to the procedure for sectioning an object with a plane surface (section 4.8). It involves intersecting the surface of each face with the reflect plane and then testing the results back with the face to find possible interactions. The procedure is obviously slower than the quick test, above, and so it may detract from the speed of the reflect operation as a local operator.

For volumetric models, edges and vertices in the reflect plane that are not adjacent to faces lying in the plane may cause self-intersection problems. Edges can be joined with the edge joining procedure to produce degenerate type models, but there is no way of representing joined vertices with the standard, idealised datastructure described in chapter 3.

Reflecting in curved surfaces is not considered here. There seems little justification in going to the considerable trouble of implementing such a tool, because there seems to be little call for it. Reflecting in planar surfaces has clear use in design, but not reflecting in curved surfaces. It could probably be performed in a similar manner to that outlined in the algorithm description in section 4.5, but the geometric calculations would be difficult at best, probably requiring approximating geometry.

G.6 Bend

One practical problem is that geometry shared between entities in the bent part and the static part has to be coped with, if allowed. Surfaces can be shared either by referring to them several times or by referring to them from a facegroup containing several faces. If a face to be modified does not have

its own surface, then the surface must be copied and the copy transformed to avoid modifying faces in the static part of the object. To copy the geometry, the face can either be isolated in its own (new) facegroup, if the facegroup referred to a surface, and the new facegroup is made to refer to a copy of the surface, or the surface can be copied and the face can be made to refer to the copy. Similarly for curves. If an edge to be modified refers to a curve that is used elsewhere, then the curve is copied and the edge is changed to refer to the new curve, which can then be modified.

There are obvious problems that have to be tackled by this method of creating the bend seam. One problem is when the bend plane cuts through a vertex, not one created as part of the bend seam. Here, there will be several candidate faces as the next face, and it is necessary to use a geometric check to discover which one is the next face to be checked.

Another problem is where the bend plane produces several unconnected portions of the bend seam. Here it is necessary to follow several possibly distinct portions of the bend seam. It is necessary to have some sort of 'stacking mechanism' to preserve unprocessed results temporarily while a current bend seam is processed. The BUILD implementation does not take these into account as it was developed to bend part of an object with only a simple structure at the bend seam. A general bend algorithm would have to cope with these problems, but the problem seems to require only a simple extension of the seam insertion algorithm rather than a major change.

If the bend seam does not completely separate the object into two pieces, then problems will occur when trying to transform one part of it. When traversing the object from the face indicating which part is to be bent, the whole object will be transformed, rather than just a part of it. Also, in the current implementation, the bend wedge would then be reclosed and so have degenerate geometry, i.e., zero-length edges and zero-area faces.

If the bend axis passes through the object, then the changes made by the current algorithm are incorrect and the object will become self-intersecting. If it is necessary to bend objects about an axis through the object, then some faces may disappear and the geometry of the resulting object will be different. This more general bending operation is a more complex problem than that originally solved by the algorithm outlined above.

Finally, the bend algorithm as described inserts material in the object, which may not be what happens when a physical object is bent. To produce a more accurate simulation of the physical process would probably only require a modification of the bend seam geometry calculation rather than an extensive modification.

G.7 Giving sheet objects thickness

The algorithm outlined in section 6.7 does not handle the cases where faces disappear during the offset process. It is difficult to know whether this is a

serious problem. The original aim of the algorithm was to convert degenerate representations of thin plate models into volume models. It is probable that, for most cases, no faces will disappear, but it should be noted that the algorithm is not a general offsetting process.

All internal edges are simply offset by the operation; i.e., there is no rounding of the internal edges. This is an arbitrary decision that has both advantages and drawbacks. If the internal edges are automatically rounded, it may suggest that a bending process will be used to create the part. This may well be true, but it will be wrong if the two faces meeting at the edge are to be welded, say. Also, it may be possible to round off the relevant edges as a separate process, thus simplifying the original offset process. All in all, the question seems to be an open one that depends more on how the modeller is to be used than on modelling techniques. This is unsatisfactory, but it is possible to add the extra geometry and topology instead of offsetting the curve geometry if this is required, treating all sharp edges as degenerate faces and slicing them in the manner described in the algorithm.

The choice of side face geometry is arbitrary. A more complete analysis of how the degenerate side faces should be expanded was done in the GPM assembled plate construction module [49], which allowed a variety of shapings to be imposed for interpretation when drawing objects. This was not emulated in the implementation of sheet objects in the GPM volume module. In some cases, this might require a more sophisticated geometric calculation when generating surfaces for the new side faces. However, if specially shaped side faces are required for, say, welding, then new topology may have to be added. One way of doing this is as a post-processing step. To do it as part of the offsetting process would require more work, but it should be possible, at least in a limited way.

The algorithm, as implemented, could only be applied to objects containing 'easy' geometry. The GPM volume module contains separate representations for many curves and surfaces, and general representations to handle all other cases. For the special representations, it was possible to implement offset geometry calculation, but general quadrics, say, represented as 4×4 matrices, were not handled, nor were spline curves. For this reason, the process was limited in scope.

G.8 Planar sectioning

Voids are a problem for sheet objects. Although it is theoretically possible to create voids inside sheet objects, their meaning is somewhat unclear, and the geometric tests needed to handle them are impossible to apply.

The sectioning code works only with planes, not with general surfaces. Several changes would have to be made to the current code if it is necessary to handle sectioning by general surfaces. With closed surfaces, such as cylinders (which are closed in one direction) and spherical surfaces, it may be necessary

to introduce arbitrary delimiting (or 'fake') edges to break up a face into pieces smaller than 180 degrees.

Otherwise, the same potential for numerical instability exists with the sectioning code as with the Boolean operations.

G.9 Imprinting

The process, as described above, only imprints on a single face. It could be extended to imprint objects onto facesets as well as onto single faces. As the imprint process progresses, it is possible that the original face becomes split into smaller sub-faces. To handle this condition, all new faces created are chained together as they are created. If it is necessary to imprint a sheet object onto a set of faces, then all that is necessary is to chain these together before imprinting begins. However, if it is necessary to imprint on the curved faces of a cylinder, say, which are separated by 'fake' edges, it is unclear how to present the operation to a user. If the faces are chained together, the imprint may result in several faces, separated by 'fake' edges even though the imprinted faces may be small.

The topology of the imprinted face may not match exactly to the topology of the original sheet object shape being imprinted. When the surface generated by sweeping the edge being imprinted is created, it is not oriented exactly with the edge. Sometimes extra vertices are created where closed curves wrap around.

The complexity of the face increases as the imprint progresses because the new edges are integrated with the face structure. This means that the process slows down when imprinting complex shapes.

G.10 Creating the dual of an object

Creating the dual of an object is not a commonly useful tool for design, and as such, their usefulness is limited. The operation was described as an illustration of the technique of using the original object as a 'map' for the modelling operation.

Creating the geometry of the dual is one obvious problem, as already described. Creating duals of general objects is not advisable unless only the connectivity, i.e., the topology, is of interest.

If the object being dualled has hole-loops in any face, then the dual will create multiple vertex duals of the face that are not connected, and that may not even lie at the same position. To overcome this problem, a non-manifold vertex representation is needed.

Bibliography

[1] Ansaldi, S., De Floriani, L., Falcidieno, B. "Form feature representation in a structured boundary model", In "Image Analysis and Processing", ed. V. Cantoni, S. Levialdi, and G. Musso, Plenum Press, New York, pp. 111-120. 1985.

[2] APT Long range program staff IIT Research Institute, "APT Part Programming", McGraw-Hill Book Company, 1967.

[3] Bain, G. "Celtic Art The Methods of Construction", pub. Constable 1951.

[4] Baumgart, B. G. "Geometric modelling for computer vision", AD/A-002 261, Stanford University, 1974.

[5] Baer, A., Eastman, C., Henrion, M. "Geometric modelling: a survey", CAD Journal, vol. 11, no. 5, September 1979.

[6] Bartels, R. H., Beatty, J. C., Barsky, B. A. "An introduction for splines for use in computer graphics and geometric modelling", ISBN 0-934613-27-3, pub. Morgan Kaufman, 1987.

[7] Benouamer, M., Michelucci, D., Peroche, B. "Error-Free Boundary Evaluation Using Lazy Rational Arithmetic: A Detailed Implementation", Proceedings of the Second Symposium on Solid Modeling and Applications, Eds. J.Rossignac, J. Turner and G. Allen, pp. 115-126, ACM Press, 1993.

[8] Besl, P. J. "Surfaces in Range Image Understanding", ISBN 0-387-96773-7, Springer-Verlag, 1986.

[9] Boor, C. de "A practical guide to splines", ISBN 0-387-95366-3, Applied Mathematical Sciences, vol. 27, Springer, 2001.

[10] Braid, I. C. "Designing with volumes", CANTAB Press, Cambridge Computer Laboratory, University of Cambridge (Ph.D. Dissertation), 1974.

[11] Braid, I. C. "Notes on a geometric modeller", CAD Group Document 101, Cambridge University Computer Laboratory (1979).

[12] Braid, I. C. "An Interface between Geometric Modellers and Application Programs", CAM-I Report R-80-GM-04 (1980).

[13] Braid, I. C., Hillyard, R. C., Stroud, I. A., "Stepwise construction of polyhedra in geometric modelling", in "Mathematical Methods in Computer Graphics and Design", ed. K. W. Brodlie, Academic Press, 1980.

[14] Brun, J. M. "EUCLID: A manual", Equipe Graphique du Laboratoire d'Informatique pour la Mechanique et les Sciences de l'Ingenieur LIMSI, Orsay, France, 1976.

[15] Bosser, L., Stroud, I. A., Wingard, L. "Modelling extensions to the GPM Volume Module", Dept. of Manufacturing Systems, KTH, Stockholm, Sweden (1985).

[16] Carleberg, P. "STU report", KTH, Industriell Produktion, 1985.

[17] Carleberg, P. "Product Model Driven Direct Manufacturing", Solid Freeform Fabrication proceedings, pp. 270-276, September 1994

[18] Chiyokura, H., Kimura, F., "Design of solids with free-form surfaces", Computer Graphics (SIGGRAPH '83 Proc.) vol. 17, pp. 289-298, 1983.

[19] Chiyokura, H., Kimura, F., "A method of representing the solid design process", IEEE Computer Graphics and Applications, vol. 5, pp. 32-41, 1985.

[20] Chiyokura, H. "Solid Modelling with DESIGNBASE", Addison-Wesley Publishing Company, ISBN 0-201-19245-4, 1988.

[21] Choi, B. K., Barash, M. M., Anderson, D. C. "Automatic Recognition of Machined Surfaces from a 3D Solid Model", Computer Aided Design, vol. 16, no. 2, pp. 81-86, March 1984.

[22] Choi, S. W., Seidel, H.-P. "Linear one-sided stability of MAT for weakly injective 3D domain", Proceedings Seventh ACM Symposium on Solid Modeling and Applications, Saarbrucken, pp. 344-355, June 17-21, 2002.

[23] Corney, J., Clark, D. E. R. "A feature recognition algorithm for multiply connected depressions and protrusions", pp.171-183, Proceedings Second Symposium on Solid Modeling and Applications, Montreal, May 1993

[24] Coxeter, H. S. M. "Regular Polytopes", Dover Publications Inc., ISBN0-486-61480-8, 1973.

[25] Csabai, A. "Layout Modelling and Evaluation Methods and Tools", EPFL, STI-IPR-LICP, Switzerland (Ph.D. dissertation), 2003.

[26] Csabai, A., Xirouchakis, P., "On Medial surface approximations of extrusions", Journal of Engineering with Computers, vol. 20, no. 1, pp. 65-74, March 2004.

[27] Culver, T., Keyser, J., Manocha, D. "Accurate computation of the medial axis of a polyhedron", Proceedings Fifth ACM Symposium on Solid Modeling and Applications, Ann Arbor, pp. 179-190, June 9-11, 1999.

[28] The "Geometric Modelling Society" home page is currently at *http://www.bath.ac.uk/ẽnsab/GMS/gms.html* or can be found by a search engine. A link to DJINN documentation is included.

[29] Dutta, D., Hoffmann, C. M., "On the skeleteon of simple CSG objects", ASME Journal of Mecanical Design, vol. 115, pp. 87-94, March 1993.

[30] Eastman, C. M., Weiler, K. "Geometric Modeling Using the Euler Operators", Research Report 78, Institute of Physical Planning, Carnegie-Mellon University (1979).

[31] Etzion, M., Rappoport A. "A boundary sampling algorithm for computing the Voronoi graph of a 2-D polygon", Technical report, Institute of Computer Science, The Hebrew University of Jerusalem, (1996).

[32] Etzion, M., Rappoport A. "Computing the Voronoi Diagram of a 3-D polyhedron by separate computation of its Symbolic and Geometric parts", proceedings "5th Symposium on Solid Modeling and Applications", pp. 167-178, (1999)

[33] Farouki, R., Ramamurthy, R., "Degenerate point/curve and curve/curve bisectors arising in medial axis computations for planar domains with curved boundaries", Computer-Aided Geometric Design, vol. 15, pp. 615-635, 1998.

[34] Faux, I., Pratt, M.J., "Computational Geometry for Design and Manufacture", Ellis-Horwood, 1978.

[35] Farin, G. "Curves and Surfaces for Computer-Aided Geometric Design", Academic Press, ISBN 0-12-249054-1, 1988.

[36] Fjällström, P.-O. "Integration of Free-Form Surfaces and Solid Modelling", Dept. of Manufacturing Systems, PS-Lab, IVF/KTH, Stockholm, Sweden (Ph.D. Dissertation), 1985.

[37] Fjällström, P.-O. "Smoothing of Polyhedral Models", IBM Svenska Aktiebolag, S-16392 Stockholm, Sweden, 1986.

[38] Fjällström, P.-O., Wingård, L., Johansson, J., Kjellberg, T. A. "Integrering av skulpterade ytor i GPMs volymmodul: En förstudie", Dept. of Manufacturing Systems, KTH, Stockholm, Sweden (In Swedish), (1983).

[39] Fournier, A., and Wesley, M. A. "Bending polyhedral objects", CAD Journal, vol. 15 no. 2, March 1983.

[40] Foley, J. D., van Damm, A., Feiner, S. K., Hughes, J. F. "Computer Graphics principles and practice", Addison-Wesley, 1984.

[41] Foskey, M., Lin, M. C., Manocha, D. (2003) "Efficient computation of a simplified Medial Axis", Proceedings Eighth ACM Symposium on Solid Modeling and Applications, Seattle, pp. 96-107, June 16-20, 2003.

[42] Gàal, B. "Automatic Generation of NC Data for 5 Axis Milling from a Solid Modeller", Cranfield, 1987.

[43] Giblin, P. J. "Graphs, Surfaces and Homology", Chapman and Hall, ISBN 0 412 21440 7, (1977).

[44] Glasner, A. "Glasner on Celtic Knots, Part 1", IEEE Computer Graphics and Applications, vol. 19, no. 5, September/October 1999.

[45] Glasner, A. "Glasner on Celtic Knots, Part 2", IEEE Computer Graphics and Applications, vol. 19, no. 5, November/December 1999.

[46] Glasner, A. "Glasner on Celtic Knots, Part 3", IEEE Computer Graphics and Applications, vol. 19, no. 5, January/February 2000.

[47] Gossard, D. C., Lin, V. "Representation of Part Families through Variational Geometry", from "Advances in CAD/CAM", Proceedings of the 5th International IFIP/IFAC Conference on Programming Research and Operations Logistics in Advanced Manufacturing Technology PROLAMAT 82, pp. 47-53, North-Holland publishing company, 1982.

[48] GPM Report 18a,b,c Volume module, system specifications, Department of Manufacturing Systems, Royal Institute of Technology, Sweden and VTT, Finland, (1981).

[49] GPM report 22: Reference Manual: Assembled Plate Constructions, NTH/SINTEF, Trondheim, Norway, (1981).

[50] GPM Information Brochure "GEOMETRIC PRODUCT MODELS - CAD/CAM SOFTWARE", Information Report, ISBN 87-981360-0-3, December 1982.

[51] Grayer, A., "A Computer Link between Design and Manufacture", University of Cambridge, (Ph.D. Dissertation), September 1976.

[52] Gursoy, H. N. "Shape Interrogation by Medial Axis Transform for Automated Analysis", M.I.T., (Ph.D. dissertation), 1989.

[53] Held, M. "On the Computational Geometry of Pocket machining", Lecture Notes in Computer Science 500, ISBN 3-540-54103-9, ISBN 0-387-54103-9, Springer-Verlag, 1991.

[54] Held, M., Lukacs, G., Andor, L. "Pocket machining based on contour parallel tool paths generated by means of proximity maps", Computer-Aided Design Journal, vol. 26, no. 3, March 1994.

[55] Helldén, H. "Design Based on Product Modelling", Final Project Report, Dept. of Machine Design, KTH, Stockholm, Sweden (1985).

[56] Henderson, M. R. "Extraction of feature information from three dimensional CAD data", Purdue University (Ph.D. dissertation), May 1984.

[57] Hillyard, R. C. "Dimensioning and Tolerancing in Shape Design", University of Cambridge Computer Laboratory (Ph.D. dissertation), Technical Report No. 8, June 1978.

[58] Hoffman, C., "Geometric and Solid Modeling – An Introduction", Morgan Kaufmann Inc., ISBN 1-55860-067-1, 1989.

[59] Hoffmann, C. M., "How to construct the skeleton of CSG objects", Proceedings IMA Conference "The Mathematics of Surfaces IV", ed. A. Bowyer, pp. 421-437, Oxford University Press, 1994.

[60] Hosaka, M., Matsushita, T., Kimura, F., Kakishita, N. "A computer system for computer aided activities", Proceedings IFIP Conference on CAD Systems, Austin, TX, 1976.

[61] Hoschek, J., Lasser, D. "Fundamentals of Computer Aided Geometric Design", translated and published by A. K. Peters, ISBN 1-56881-007-5, 1993.

[62] Jared, G. E. M., Dodsworth, J. "Solid Modelling", in "Principles of Computer-aided Design", eds. J. Rooney and P. Steadman, Pitman Publishing in association with the Open University, ISBN 0 273 02672 0, 1987.

[63] Jared, G. E. M. "Generative Process Planning based on feature extraction from solid models", SERC project proposal (private communication), June 1985.

[64] Jared, G. E. M. "Recognizing and using geometric features", in "Geometric Reasoning", ed. J. Woodwark, Oxford Scientific Publications, ISBN 0-19-853738-7, 1989.

[65] Jared, G. E. M., Stroud, I. A. "Local operators in the BUILD system", from "Advances in CAD/CAM", Proceedings of the 5th International IFIP/IFAC Conference on Programming Research and Operations Logistics in Advanced Manufacturing Technology PROLAMAT 82, pp. 55-65, North-Holland Publishing Company (1982).

[66] Jared, G. E. M., Stroud, I. A. "A knowledge based expert system with geometric reasoning capabilities for process planning", 21st CIRP International Seminar on Manufacturing Systems, Stockholm, June 5-6, 1989.

[67] Johnson, R. H. "Dimensioning and tolerancing final report", Report R-84-GM-02.2, Computer Aided Manufacturing International (1985).

[68] Katainen, A. "Monadic set operations", CAD Journal volume 14, number 6, November 1982

[69] Kjellberg, T.A. "Integrerad Datorstöd for Mänsklig Problemlösning och Mänsklig Kommunikation inom Verkstadsteknisk Produktion begränsat till Produkt-utveckling, Produktions-beredning, Kon-struktion och Tillverknings-beredning. En systemansats baserad påProdukt-modeller", Dept. of Manufacturing Systems, KTH, Stockholm, Sweden (In Swedish), (Ph.D. dissertation), 1982.

[70] Kjellberg, T. A., Lindholm, G., Sorgen, A., Haglund, G. GPM Report 10: GPM - Specifikation Volymgeometri, ISBN 91-85212-54-7 Dept. of Manufacturing Systems, KTH, Stockholm, Sweden, Confidential document (1980).

[71] Kjellberg, T. A., Stroud, I. A., Wingard, L. O. "GPM Volume module in CAM-I perspective", CAM-I Solid Modelling Conference, Cambridge, Dept. of Manufacturing Systems, KTH, 1983.

[72] Kjellberg, T.A. "LISP Interface to GPM", Dept. of Manufacturing Systems, KTH, (private communication), 1986.

[73] Kumar, V., Dutta, D. "An assessment of data formats for layered manufacturing", Advances in Engineering Software, vol. 28, no. 3, pp. 151-164, 1997.

[74] Kyprianou, L. "Shape Classification in Computer-aided Design", University of Cambridge (Ph.D. dissertation), July 1980.

[75] Laakko, T., Mäntylä, M. "A feature definition language for bridging solids and features", Proceedings Second Symposium on Solid Modelling and Applications, pp. 333-342, Montreal, 19-21 May, 1993.

[76] Laakko, T., Mäntylä, M. "Introducing blending operations in feature models", EUROGRAPHICS 93, Barcelona, Spain, 6-10 September 1993.

[77] Lukacs, G., Rozgonyi, Z., Andor, L., "LARK technical reference manual", CADMUS Consulting and Development Ltd., Budapest, 1992.

[78] Luo, Y., Lukacs, G. "A Boundary Representation for Form Features and Non-Manifold Solid Objects", The first ACM/SIGGRAPH Symposium on Solid Modeling Foundations and CAD/CAM Applications, Austin, TX, June 1991.

[79] Luo, Y. "Solid modelling for regular objects: Renewed theory, data structure and Euler operators", Computer and Automation Institute, (Ph.D. dissertation), 1991.

[80] Mäkelä, I., Dolenc, A. "Some Efficient Procedures for Correcting Triangulated Models", Solid Freeform Fabrication Proceedings, pp. 126-134, September 1993.

[81] Mäntylä, M. "A note on the modeling space of Euler operators", Computer vision, graphics and image processing, volume 26, pp. 45-60, 1984.

[82] Mäntylä, M. "An Introduction to Solid Modeling", Computer Science Press, ISBN 0-88175-108-1, 1988.

[83] Mäntylä, M., Sulonen, R. "GWB: A solid modeler with Euler operators", IEEE Computer graphics and applications, vol. 2, no. 7, pp. 17-31, September 1982.

[84] Marshall, A. D., Martin, R. R. "Computer vision, models and inspection", ISBN 9810207727, World Scientific Publishing, 1992.

[85] Martin, R. R., Stroud, I. A., Marshall, A. D. "Data reduction for reverse engineering", "Mathematics of Surfaces VII", Ed. T. N. T. Goodman and R. R. Martin, Information Geometers Ltd., ISBN 1-874728-12-7, pp. 85-100, 1997.

[86] Minsky, M. "A Framework for the Representation of Knowledge", in "The Psychology of Computer Vision, ed. P. Winston, McGraw Hill, 1975.

[87] Montanari, U. "Continuous skeletons from digitized images", Journal of the Association of Computing Machinery, vol. 16, no. 4, pp. 543-549, 1969.

[88] Mortenson, M.E. "Geometric Modeling", Wiley and Sons Inc., ISBN 0-471-88279-8, 1985.

[89] Müller, M., Stroud, I., Xirouchakis, P., "Preprocessing of degenerate slices in layered manufacturing", in "Developments in engineering computational technology", ed. B. H. V. Topping, Civil-Comp Press, ISBN 0-948749-70-9, pp. 63-68, 2000.

[90] Näf, M. "3D Voronoi skeletons A semi-continuous implementation of the 'Symmetric Axis Transform' in 3D Euclidean space", ETHZ, (Ph.D. dissertation), 1996.

[91] Newman, W. M. and Sproull, R. F. "Principles of Interactive Computer Graphics", McGraw-Hill, 1979.

[92] Noton, D. and Stark, L., "Eye movements and visual perception", Scientific American, June 1971.

[93] Okino, N., Kubo, H. "Technical information system for computer- aided design, drawing and manufacturing", Proceedings of the Second PROLAMAT 73, 1973.

[94] Owen, J., "STEP An Introduction", Information Geometers, Winchester, U.K., 1993

[95] Parkinson, A. "Feature Recognition and Parts Classification in BUILD", University of Cambridge CAD Group Document 112, March 1983.

[96] Parkinson, A. "The use of solid models in BUILD as a database for N.C. machining", Proceedings PROLAMAT 1985, Paris, France, pp. 293-299, June 1985.

[97] Parry-Barwick, S., Bowyer, A. "Is the features interface ready?", in "Directions in Geometric Computing", ed. Ralph Martin, Information Geometers Ltd., ISBN 1-874728-02-X, pp. 129-160, 1993.

[98] Peters, J. "Smoothing Polyhedra Made Easy", ACM Transactions on Graphics, vol. 14, no. 2, pp. 162-170, April 1995.

[99] Pfeifer, H.-U. "Methods used for intersection geometrical entities in the GPM module for volume geometry", CAD Journal, vol. 17, no. 7, September 1985.

[100] Piegl, L., Tiller, W. "The NURBS Book", Springer Verlag, ISBN 3-540-61545-8, 1997.

[101] Pratt, M.J. "Recent research in form features", ACM SIGGRAPH '87, 1987.

[102] Preparata, F. P. "The medial axis of a simple polygon", in Lecture Notes in Computer Science: Mathematical Foundations of Computer Science, eds. Goos and Hartmanis, Springer-Verlag, pp. 443-450, 1977.

[103] Ramanathan, M., Gurumoorthy, B., "Constructing medial axis transform of planar domains with curved boundaries", Computer-Aided Design, vol. 35, no. 7, pp. 619-632, June 2003.

[104] Ranta, M., Inui, M., Kimura, F., Mäntylä, M. "Cut and paste based modeling with boundary features" Proceedings Second Symposium on Solid Modeling and Applications, Montreal, pp. 303-312, May 1993.

[105] Reddy, J., Turkiyyah, G. "Computation of 3d skeletons by a generalised Delaunay triangulation technique", Computer-Aided Design, vol. 27, no. 9, pp. 677-694, 1995.

[106] Renner, G., Stroud, I. "Medial surface generation and refinement", in "Product Modelling for Computer Integrated Design and Manufacture", ed. M. J. Pratt, R. D. Sriram, M. Wozny, pub. Chapman and Hall, ISBN/ISSN: 0-412-80980-, 1997.

[107] Renner, G., Stroud, I.A., "MAT surface skeletonisation and feature extraction", CIRP World Congress on Intelligent Manufacturing Systems, Budapest, June 10-13 1997.

[108] Renner, G., Stroud, I. "Volumetric Tools for Machining Planning" Proceedings PROLAMAT 2001, "Digital Enterprise Challenges: Life-Cycle Approach to Management and Production", eds. G. Kovács, P. Bartók, and G. Haidegger, Kluwer, ISBN 0-7923-7556-4, 2001.

[109] Renner, G., Stroud, I. A., "Recombination, offsetting and decomposition of solids by medial surface transformation", proceedings First Hungarian Conference on Computer Graphics and Geometry, eds. L. Szirmay-Kalos and G. Renner, pp. 83-89, Budapest 2002.

[110] Renner, G., Stroud, I. "Techniques for the calculation of medial surfaces of solids", International Journal of Computer Applications in Technology, vol. 22, no. 2/3/4, pp. 138-156, 2005.

[111] Requicha, A. A. G., Voelcker, H. B. "Constructive solid geometry", Technical Memo. No. 25, Production Automation Project, University of Rochester, NY, (1977).

[112] Rockwood, A.P. "Blending Surfaces in Solid Geometrical Modelling", Department of Applied Mathematics and Theoretical Physics, University of Cambridge (Ph.D. dissertation), 1987.

[113] Rockwood, A.P., Chambers, P. "Interactive Curves and Surfaces - A Multimedia Tutorial on CAGD", Morgan Kauffman Publishers Inc., 1996.

[114] Sabin, M. A. "The use of piecewise forms for the numerical representation of shape", dissertation for the degree of Candidate of Technical Sciences, Magyar Tudomanyos Akademia, Budapest, 1976.

[115] Schlechtendahl, E.G. (ed.) "Specification of a CAD*I Neutral File for CAD Geometry Wireframes, Surfaces, Solids", Version 3.3, Springer Verlag, ISBN 3-540-50392-7, 1988.

[116] Sheehy, D. J., Armstrong, C. G., Robinson, D. J. "Computing the medial surface of a solid from a domain Delaunay triangulation", in Proceedings of the Third Symposium on Solid Modeling and Applications, eds. C. M. Hoffmann and J. R. Rossignac, pub. ACM, pp. 201-212, 1995.

[117] Sheehy, D. J., Armstrong, C. G., Robinson, D. J. "Shape Description by Medial Surface Construction", "IEEE Trans. on Visualization and Computer Graphics", vol. 2, no. 1, pp. 62-72, March 1996.

[118] Sherbrooke, E. C., Patrikalakis, N. M., Brisson, E. "Computation of the Medial Axis Transform of 3-D Polyhedra", Proceedings of the Third Symposium on Solid Modeling and Applications, ed. C. M. Hoffmann and J. R. Rossignac, ACM, pp. 187-199, 1995.

[119] Sherbrooke, E. C., Patrikalakis, N. M., Brisson, E. "An algorithm for the medial axis transform of 3-D polyhedral solids", "IEEE Trans. on Visualization and Computer Graphics", vol 2, no. 1, pp. 44-61, March 1996.

[120] Solomon, B. J. "Surface Intersections for Solid Modelling", Cambridge University Computer Laboratory (Ph.D. Dissertation), 1985.

[121] Shah, J., Sreevalsan, P., Rogers, M., Billo, R., Mathew, A., "Current status of features technology", CAM-I Report R-88-GM-04.4 for Task 0, 1988.

[122] Srinivasan, V., Nackman, L. R. "Voronoi diagram for multiply connected polygonal domains", I: Algorithm, IBM Journal of Research and Development, vol. 31, no. 3, pp. 361-372, 1987.

[123] Stroud, I. A. "Modelling with degenerate objects", CAD Journal, vol. 22, no. 6, pp. 344-351, July/August 1990.

[124] Stroud, I. A. "Definition of solid modelling operations using a uniform set of elementary procedures", Hungarian Academy of Sciences (Ph.D. dissertation), 1992.

[125] Stroud, I. A. "Boundary modelling with special representations", CAD Journal, vol. 26 no. 7, pp. 543-550, July 1994.

[126] Stroud, I., Xirouchakis, P. C. "STL and extensions", Advances in Engineering Software, vol. 31, no. 2, pp. 83-95, February 2000.

[127] Stroud, I., Xirouchakis, P. C. "CAGD – Computer-Aided Gravestone Design", Advances in Engineering Software, vol. 37, no. 5, pp. 277-286, May 2006.

[128] Stroud, I., Renner, G., Xirouchakis, P. "A divide and conquer algorithm for medial surface computation", Submitted to CAD Journal, 2005.

[129] Sudhalkar, A., Gursoz, L., Prinz, F. "Continuous skeletons of discrete objects", in Proceedings of the Second Symposium on Solid Modeling and Applications, eds. J. R. Rossignac, J. Turner, and G. Allen, pub. ACM, May 19-21 1993.

[130] Sugihara, K., "Resolvable representation of polyhedra", Proceedings of the Second Symposium on Solid Modeling and Applications (Eds. J. Rossignac, J. Turner, and G. Allen), ACM Press, pp 127-136, 1993.

[131] Suh, N. P. "Design axioms and quality control", 21st CIRP International Seminar on Manufacturing Systems, Stockholm, June 5-6, 1989.

[132] Szèkely, G., "Shape characterization by local symmetries", Institut für kommunikationstechnik, Fachgruppe Bildwissenschaft, ETH Zürich, Habilitationsschrift, 1996.

[133] Szirmay-Kalos L., Màrton, G., Dobos, B., Horvàth, T., Risztics, P., Kovàcs, E. "Theory of Three Dimensional Computer Graphics", Akadèmia Publishers, 1994.

[134] Tate, S. J., Jared, G. E. M., Swift, K. G. "Detection of symmetry and primary axes in support of proactive design for assembly", Proceedings Fifth Symposium on Solid Modeling and Applications, eds. W. F. Bronsvoort and D. C. Anderson, June 9-11, 1999.

[135] Tigyi, A. "Solid body medial surface calculation", Diploma work (in Hungarian), Eötvös LoràND University of Sciences, Budapest, 1995.

[136] Tilove, R. B. "Extending solid modeling systems for mechanism design and kinematic simulation", IEEE Computer Graphics and Applications, pp. 9-19, May/June 1983.

[137] Turkiyyah, G.M., Storti, D.W., Ganter, M., Chen, H., Vimawala, M., (1997) "An accelerated triangulation method for computing the skeletons of free form solid models", CAD, vol. 29, no.1, pp. 5-19, 1997.

[138] Várady, T. "Integration of free-form surfaces into a solid modeller", Computer and Automation Institute, Hungarian Academy of Sciences, 171/1985 (Ph.D. dissertation), ISBN 963 311 1927, ISSN 0324-2951, 1985.

[139] Várady, T. "Fat sectors: A new bounding region for B-rep solid modelling", Presentation at IFIP Working Group 5.2, Geometric Modelling Workshop, RPI, 1990.

[140] Várady, T., Gàal, B., Jared, G. E. M. "Identifying features in solid modelling", Computers in Industry, vol. 14, no. 1-3, pp. 43-50, 1989.

[141] Várady, T., Stroud, I. A. "Polyhedral smoothing using the X-construction method", in preparation.

[142] Vida, J. "Integrating parametric blends into a volumetric modeller", Hungarian Academy of Sciences (Ph.D. dissertation), 1993.

[143] Weiler, K. "Edge-based data structures for solid modeling in curved–surface environments", IEEE Computer Graphics and Applications, vol. 5, pp. 21-40, 1985.

[144] Weiler, K. "Topological Structures for Geometric Modeling", Rensselaer Polytechnic Institute (Ph.D. Dissertation), August 1986

[145] Wingård, L. Presentation of work on modelling with features to STU, private communication, 1991.

[146] Wingård, L., Palm, G. "The GPM ASEA robot simulation", private communication, 1983.

[147] Woo, T. C. "Feature extraction by volume decomposition", Proceedings of the Conference on CAD/CAM in Mechanical Engineering, MIT, pp. 76-94, March 24-26 1982.

[148] Wördenweber, B. "Automatic mesh generation of 2 and 3-dimensional curvilinear manifolds", Technical report 18, Cambridge University Computer Laboratory (Ph.D. dissertation), Cambridge 1981.

[149] Wozny, M.J. "System issues in solid free-form fab-rication", Solid Freeform Fabrication proceedings, pp. 1-15, September 1992.

[150] Zhu, X., Shiaofen, F., Bruderlin, B.D., "Obtaining robust Boolean set operations for manifold solids by avoiding and eliminating redundancy", Proceedings of the Second Symposium on Solid Modeling and Applications, (Eds. J. Rossignac, J. Turner, G. Allen), ACM Press, pp 147-154, 1993.

Index