Chapter 2
How Objects Are Modelled

Various methods have been developed for representing shapes. CAD systems have included several of these techniques at different times in their development. A short historical perspective is given in the first section of this chapter. At present the main technique used is called "Boundary Representation", a solid modelling technique which is described later in this chapter.

2.1 History

The history of CAD is quite long, but you won't find it described here. The purpose of this section is not to mention particular systems, however important, but to introduce the techniques. Figure 2.1 shows the main time line of modelling development.

Early modelling systems were capable of creating wireframe, or line, drawings of shapes. However, while pictures may communicate information between people they are not enough for computer applications. Both the car industry and the aircraft industry needed to create and manufacture complex shapes. Surface modelling techniques and systems were developed in order to facilitate toolpath generation for machining. It was also realised by various people that solids could also be modelled and these techniques arrived during the 1970s. In the beginning there were several different techniques, but during the 1980s two became predominant and eventually only one technique became used for the majority of CAD/CAM applications. Although these techniques were largely ignored by the CAD developers at the outset, they became accessible and are now common. An overview of various techniques is given below.

I. Stroud and H. Nagy, *Solid Modelling and CAD Systems*,
DOI: 10.1007/978-0-85729-259-9_2, © Springer-Verlag London Limited 2011

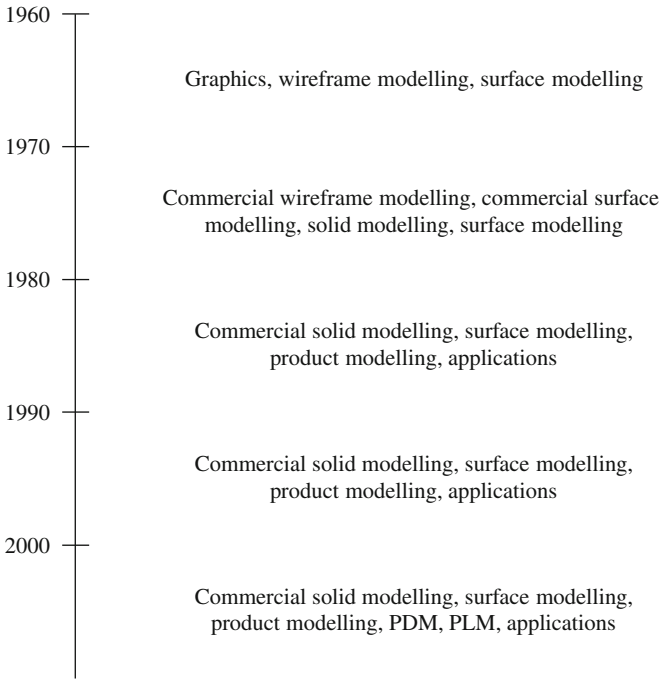
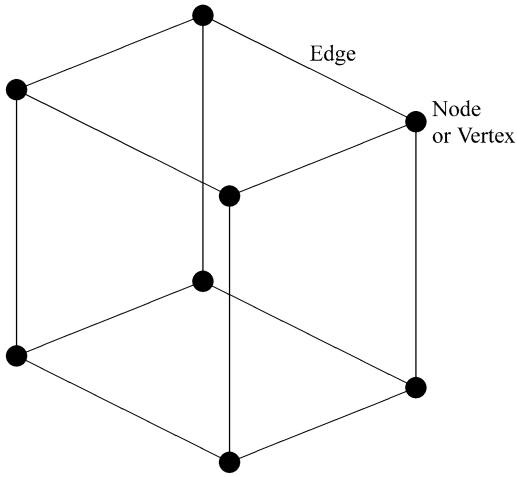


Fig. 2.1 Modelling and CAD timeline

Fig. 2.2 Simple 3D wireframe model



2.2 3D Wireframe Modelling

Wire frame models consist of nodes, or vertices, and links between them, called edges, as shown for a cube in Fig. 2.2.

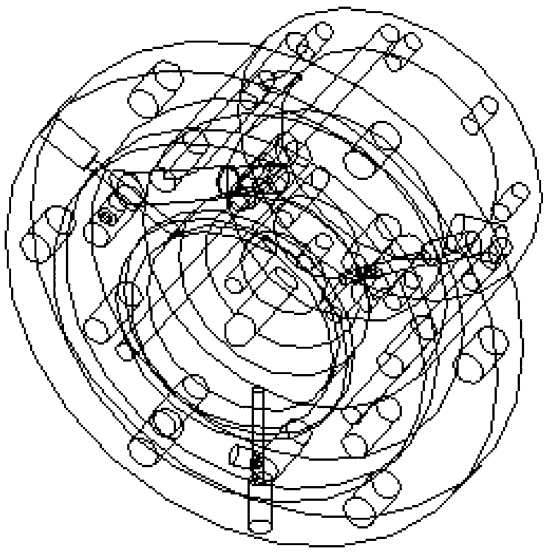


Fig. 2.3 Complex wireframe object

This is enough to produce pictures but several straightforward and useful operations cannot be performed without surface information. Hidden line removal, for example, for realistic image creation is impossible unless you have surface information. Figure 2.3 shows a complex object in wireframe mode where it is difficult to see the object represented in the figure. More importantly, automatic toolpath generation for machining cannot be done without surfaces. This particular shortcoming prompted development of surface modelling systems, described later.

Another problem is that it is possible to create objects which it is impossible to realise. See for example Fig. 2.4 which shows a well-known optical illusion which can be made with just point and lines.

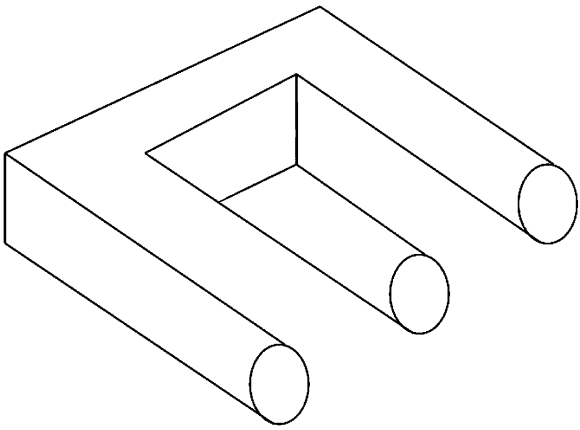


Fig. 2.4 Impossible object

Wireframe modelling has a place, though, in modelling as a support for other operations, as described in Chap. 4. It is important, though, to keep wireframe modelling as a support, or to keep it simple. It is possible, with a lot of effort, to create a complex model, but it becomes harder as the complexity increases.

2.3 Drafting Systems

Drafting systems were essentially electronic drawing boards. They could be used to create planar line drawings and made copying and amendment very easier. However, while a step forward they preserved the problems of classic design techniques, the possibility of creating incorrect drawings and that multiple views had to be created separately.

Some of the operations in solid modelling, notably linear and circular extrusion, seem to have their roots in 2D drafting. However, there are many benefits from using 3D models so drafting systems will not be dealt with here.

2.4 Surface Modelling

Surface modelling was a step forward in that these systems can represent the space between the edges in a wireframe model. Geometric modelling as a subject has a life separate from solid modelling and continues as an important subject for research.

Surface modelling systems model portions of an object with complex surfaces. Objects such as car bodies, aircraft and ships use complex surfaces of this type. With some products it is not necessary to have a solid model behind and a surface modelling system is sufficient. For other uses surface modelling systems have been used to create objects which appear closed, although the surface patches are not joined, simply placed adjacent in space. Such a lack of connection can cause problems because there is no information to check consistent orientation and connection. There are also problems which are inherent in the mathematics of surfaces because surface patches have four sides, usually, though three-sided patch mathematics has also been formulated. Real objects often disobey this requirement leading to mismatches between patch boundaries.

However, the usefulness of surface modelling means that the techniques are often incorporated into solid modelling systems. Perhaps the first such modelling system was the GPM volume module by Kjellberg et al. [1]. The use of surfaces and other complex geometry in CAD systems is something which will be described in more detail in Chap. 5.

2.5 Solid Modelling

This book is concerned with the application of solid modelling techniques in CAD systems. Solid modelling began in different places sometime at the beginning of the 1970s. In Cambridge Ian Braid, in the BUILD series of systems, developed the boundary representation technique. Work was also done on this by Kimura in the GEOMAP system. Also in Japan, in Okino, the half-space technique was being worked on which emerged as the TIPS system. Similar ideas were being worked on in Rochester by Voelcker and Requicha, which resulted in the important PADL system using the CSG technique. Generalised sweeping was also used as a solid modelling technique, representing objects as two-dimensional forms and extrusion definitions. Cellular modelling, both with uniform cells and adaptive cells, the so-called octree technique were examined. A good presentation of these is given by Jared and Dodsworth [2] and more explanation than here is given in Stroud [3].

As far as CAD systems go, it is important to know that Constructive Solid Geometry, or CSG, systems were used initially while now Boundary Representation has taken over. A simple illustration to show the way in which these two methods work is shown in Fig. 2.5. On the left you have the CSG representation. The object at the bottom is modelled as a tree structure in which a rectangular block and a cylinder are first "added" and then a cylinder is subtracted to form the hole. To the right you have the Boundary Representation version. The solid at the top is modelled using a connected set of faces, each of which is a bounded surface portion. The faces are shown in exploded form at the bottom right of the figure.

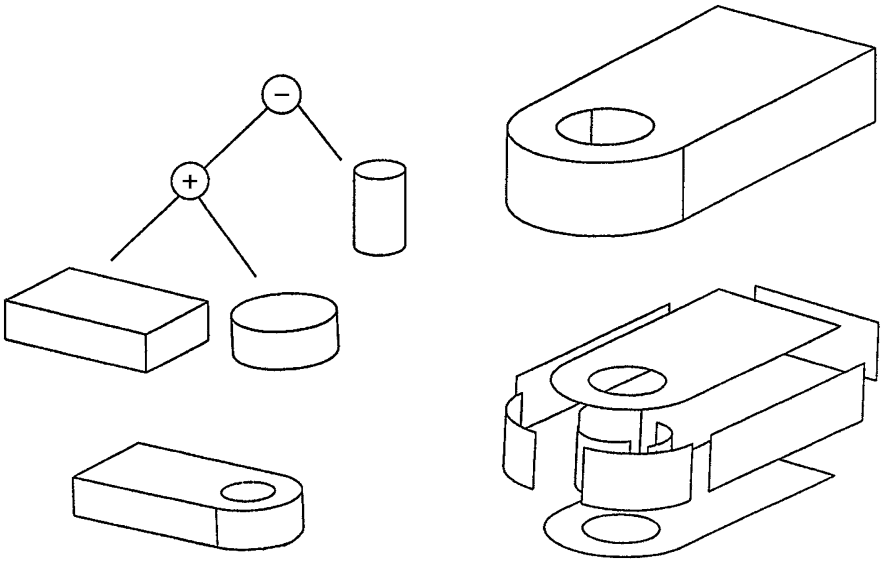


Fig. 2.5 Representing solids with CSG and Brep

To summarise, CSG models the part as a tree structure where the leaves are positioned primitive objects and the intermediate nodes are Boolean operations to combine them. Boundary Representation models the "skin", the interface between solid and non-solid.

2.6 Constructive Solid Geometry

As stated above, constructive solid geometry, or CSG, systems represent an object by combining a set of primitive objects. See Okino et al. [4], Requicha and Voelcker [5] or the interesting monadic variant, UNIBLOCK, by Katainen [6]. Each of these primitives is made of a set of half spaces defining point sets. In two dimensions, with lines instead of planes, this is illustrated in Fig. 2.6.

The central portion, the square, can be defined with the equation:

$$(x \geq -1) \wedge (x \leq +1) \wedge (y \geq -1) \wedge (y \leq +1)$$

Similar sets of relations can be established to define a set of three-dimensional primitive shapes. A set of normal primitives might be:

- Rectangular block
- Wedge
- Cylinder
- Cone
- Sphere
- Torus

When building a model, these primitives are created and positioned and then they are combined by applying Boolean operations. These Boolean operations are different to those described in Sect. 4.1. With CSG the Boolean operations are logical combinations of type AND, OR, INTERSECT. Consider the two objects in Fig. 2.7.

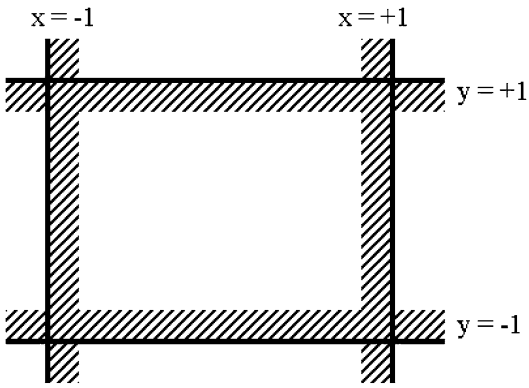


Fig. 2.6 Simple object with half space representation

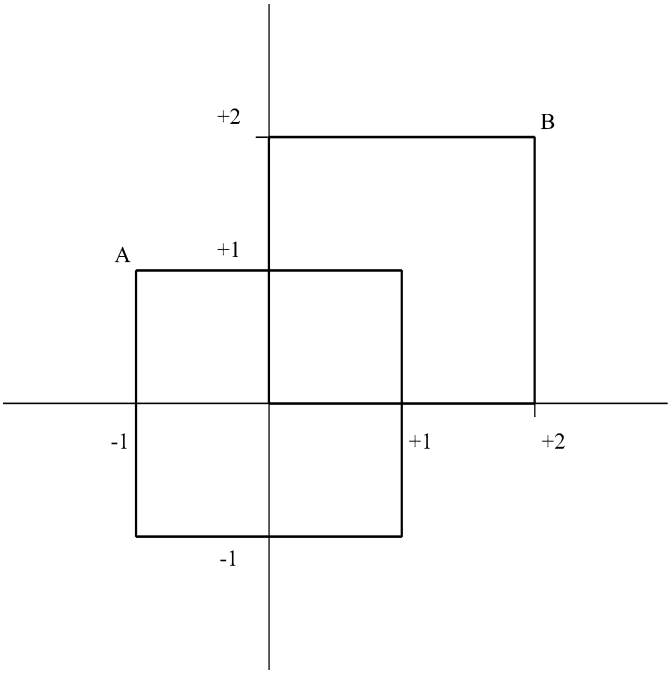


Fig. 2.7 Addition of two squares

These are two squares, one from -1 to 1 in the x and y directions, the other from 0 to 2 in the x and y directions. The combination of the two can be represented simply by the following expression:

$$A \vee B = ((x \geq -1) \wedge (x \leq +1) \wedge (y \geq -1) \wedge (y \leq +1)) \vee ((x \geq 0) \wedge (x \leq +2) \wedge (y \geq 0) \wedge (y \leq +2))$$

The resulting area is shown in Fig. 2.8.

Subtraction can be done in a similar way. The relationship is essentially everything in A which is not in B, or $A \wedge (\sim B)$ or:

$$A \wedge (\sim B) = ((x \geq -1) \wedge (x \leq +1) \wedge (y \geq -1) \wedge (y \leq +1)) \wedge ((x \leq 0) \vee (x \geq +2) \vee (y \leq 0) \vee (y \geq +2))$$

The result is shown in Fig. 2.9.

For the intersection the corresponding equation is:

$$A \wedge B = ((x \geq -1) \wedge (x \leq +1) \wedge (y \geq -1) \wedge (y \leq +1)) \wedge ((x \geq 0) \wedge (x \leq +2) \wedge (y \geq 0) \wedge (y \leq +2))$$

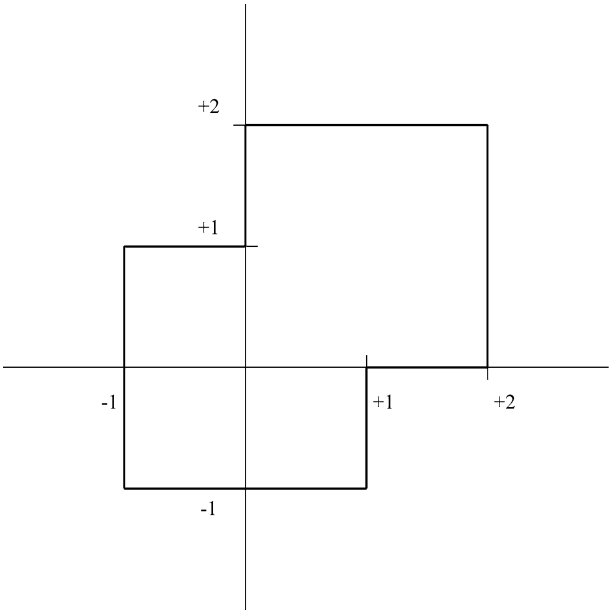


Fig. 2.8 Addition of two squares

which can be simplified to the equation:

$$A \wedge B = (x \geq 0) \wedge (x \leq +1) \wedge (y \geq 0) \wedge (y \leq +1)$$

The result is shown in Fig. 2.10.

There is, though, a slight complication in that with some operations you can get hanging geometry. If, from the result in Fig. 2.9, you subtract the block defined as:

$$(x \geq -1) \wedge (x \leq 0) \wedge (y \geq -1) \wedge (y \leq +1)$$

you get a hanging edge with the naive scheme mentioned above, as shown in Fig. 2.11.

To get round this problem the CSG researchers developed the notion of "interior" and "closure". See Tilove and Requicha [7]. Since this book is not about CSG techniques, rather than describe this in detail it is more important to see how this works in practice.

To create a 3D model, a number of 3D primitives are created and combined in the same way as outlined above. Consider the object in Fig. 2.12. This might be modelled by the set of primitives and operations shown in Fig. 2.13.

Another alternative is shown in Fig. 2.14.

Which to choose depends on the way that a user mentally decomposes the shape. As can be seen from the figures, though, while the end result is the same,

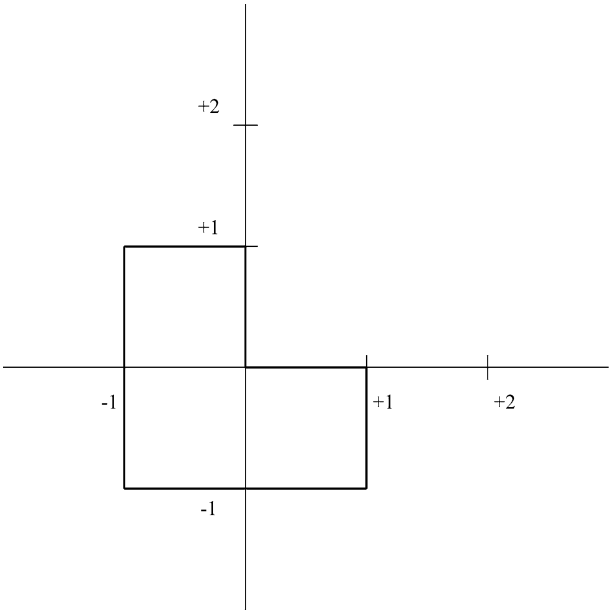


Fig. 2.9 Subtraction of one square from another

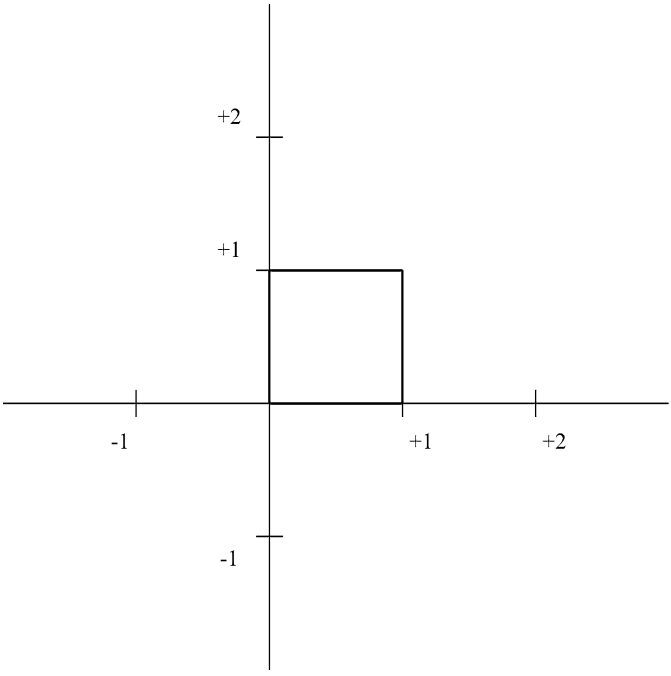


Fig. 2.10 Intersection of two squares

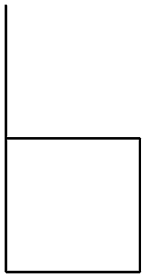


Fig. 2.11 Result with hanging edge

Fig. 2.12 Simple object

the intermediate steps and the resulting tree structure are quite different. This has been cited by several people as a problem with CSG because it means that shapes cannot be compared by simply comparing the tree structure.

Another drawback of CSG modelling is that it is not always convenient to formulate a design operation in terms of Boolean operations. Consider a chamfer operation such as that shown in Fig. 2.15. This is conveniently defined in a CAD system by picking the edge and giving an appropriate chamfer depth value.

In a CSG system how would you do that? For a start, it is necessary to note with the edge is convex or concave. For a convex edge this means subtracting a wedge of the appropriate size. For the concave edge in the figure this means adding the wedge. In the case in the figure, the faces where the edge ends are perpendicular to the edge. If they were not then it would be necessary to preshape the wedge. The basic size of the wedge is, of course, computed from the length of the edge.

What is more awkward is that the edge doesn't exist.

In the strict CSG sense the object on the left of Fig. 2.15 would probably be made by subtracting one block from another. The concave edges in the interior of

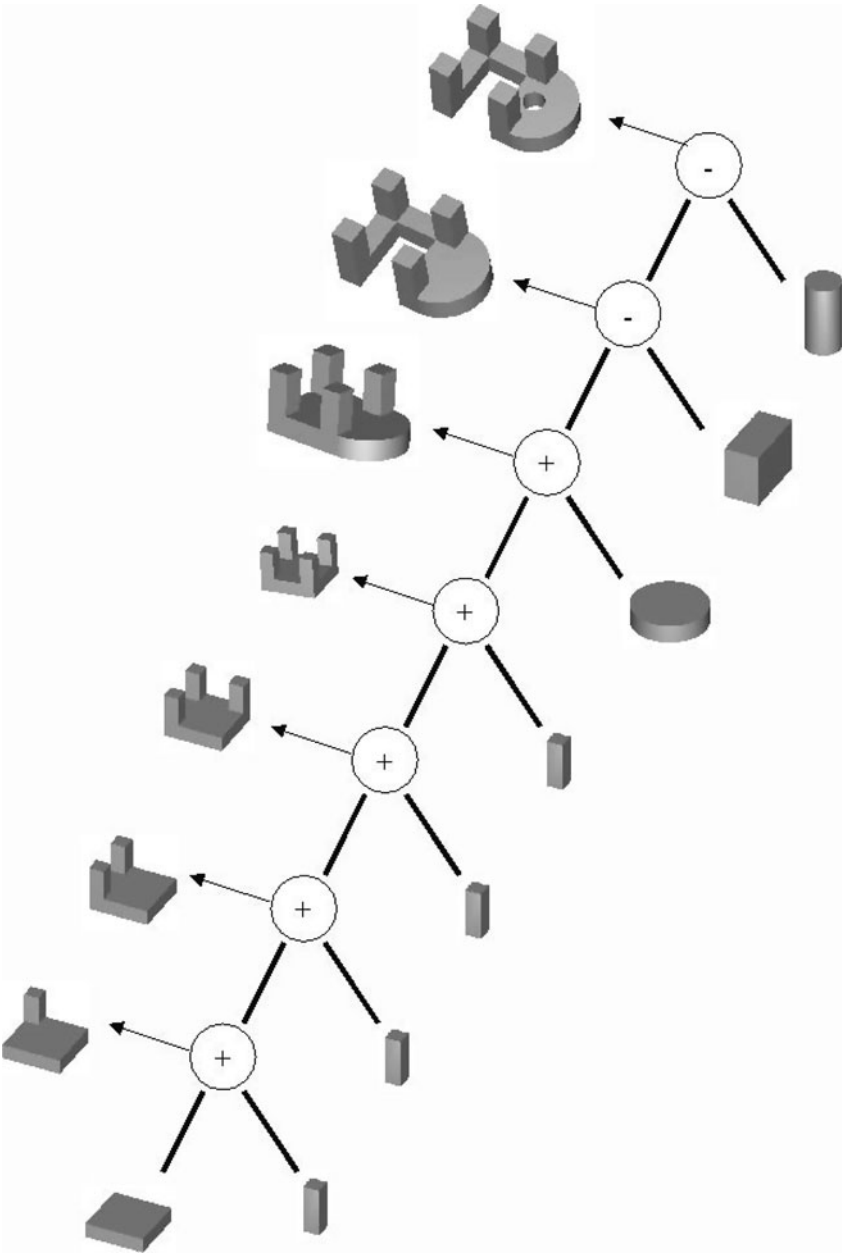


Fig. 2.13 First CSG decomposition of simple object

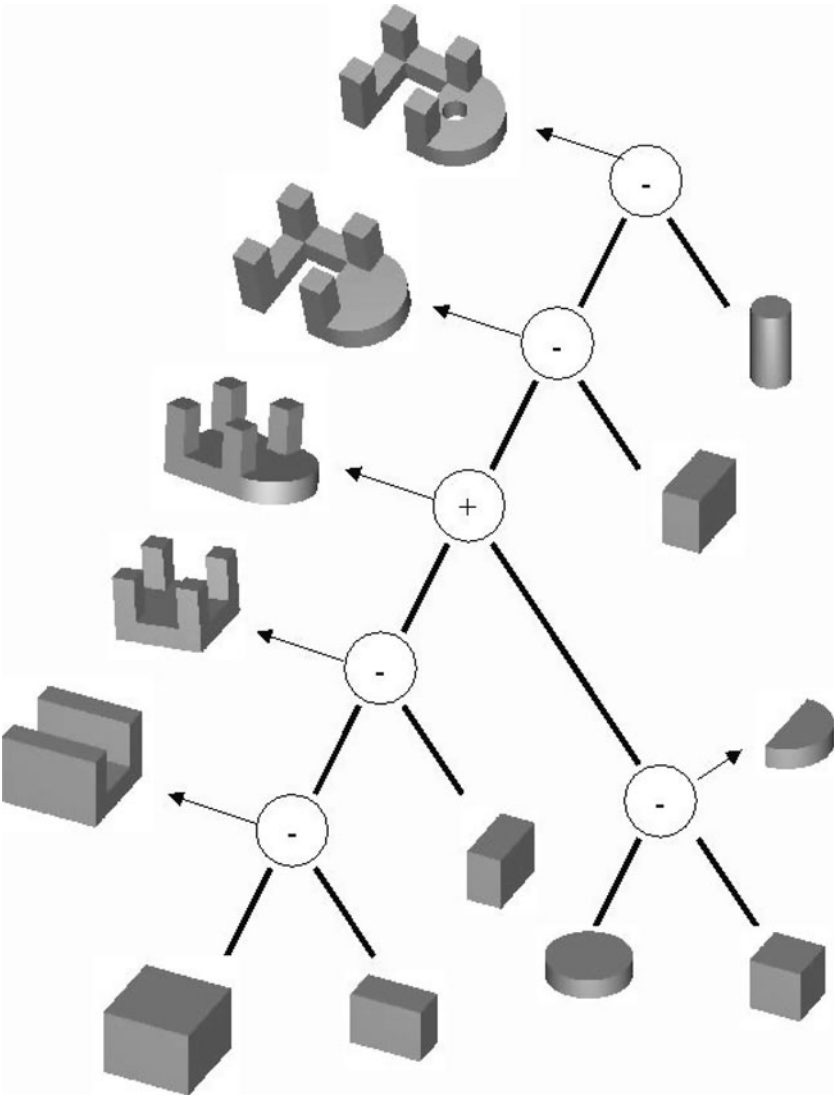


Fig. 2.14 Second CSG decomposition of simple object

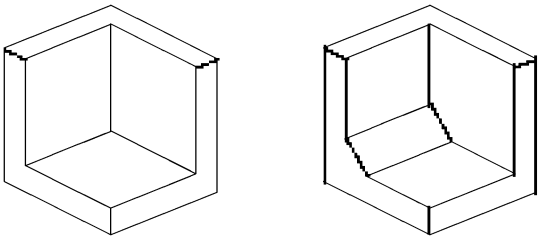


Fig. 2.15 Chamfering an edge in an object

the object appear as a bi-product of the definition. They aren't actually explicitly defined. In Okino's system, graphical images were produced by slicing the result object was sliced with planes and the resulting intersection lines drawn to show the result. The "edge" is then just a visual element in the image. In the PADL system of Requicha and Voelcker the strict CSG approach was eventually supplemented with an explicit Boundary Representation model for graphics purposes. This means that there is an explicit edge in the graphics model, but this is not strictly part of the CSG model. In the CSG philosophy you would have to define the wedge explicitly, shape its ends, if necessary, and determine whether to add it or subtract it from your model. This is possible for modelling a known part but makes design harder.

Another type of problem comes when a shape is to be extruded. This implies that the 2D shape has to be decomposed into squares, circles, triangles, etc. Each of these basic shapes corresponds to a primitive. The primitives are then added together to produce the extruded shape as a volume. Extrusion is much more straightforward in boundary representation modelling.

One of the obvious differences between CSG and Boundary Representation modelling was the richness of the Boundary Representation modelling set. A whole range of operations, linear and circular extrusions, Boolean operations, chamfers, tweaks, etc. were defined compared to the Boolean operations of CSG. It is perhaps this richness together with the flexibility of Boundary Representation for degenerate and special objects that led to the current domination of boundary representation.

2.7 Boundary Representation

The currently widely used technique for solid representation in CAD/CAM is the boundary representation technique. These techniques are described in detail in [3]. The purpose here is to put these into a context so that it is possible to understand the implementation of CAD/CAM systems, how they work and why they work that way. Some of the basic concepts are reproduced here to help explain these details.

Boundary representation models of solids have two basic parts: the topology and the geometry. These are kept separate for practical reasons. The topology defines the structure of the model, the geometry its shape, Fig. 2.16.

The main elements of the topology of a model are faces, edges and vertices. There are other elements which are there for practical or efficiency reasons, Fig. 2.17. The loop, for example, is necessary to represent multi-connected faces, but it is accessed via the face which it bounds.

The topology provides links to other elements and so makes the structure connected. For many operations you refer to a topological element of the model. Blending, for example, might take a face, edge or vertex as input, depending on the implementation. If it takes a face then the usual logic is that all edges around the face will be blended. Similarly, if a vertex is given then all edges at the vertex will be blended, usually with the vertex as well.

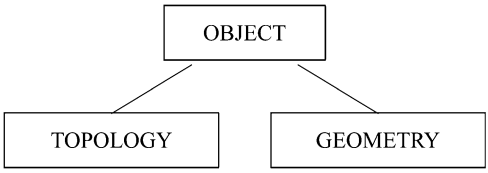


Fig. 2.16 Basic data structure subdivision

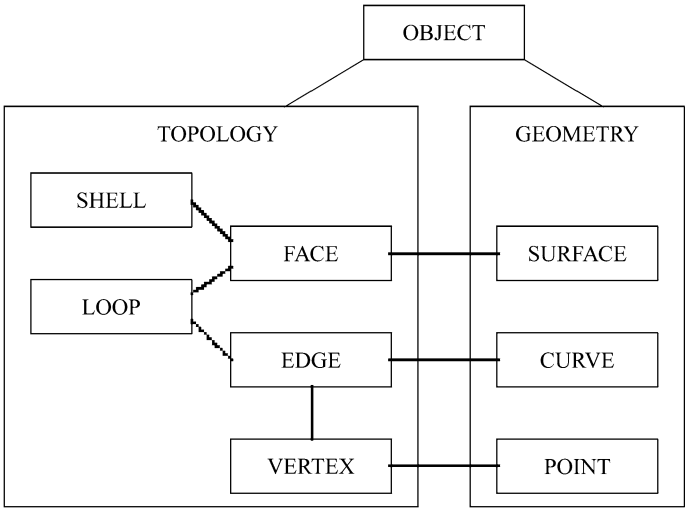


Fig. 2.17 Basic elements of the data structure

Each face is part of a surface. Each edge is a portion of a curve and a vertex lies at a point in space. This relation is illustrated in Fig. 2.18.

A full, single model data structure and definitions might be as shown in Fig. 2.19.

The data structure definitions for this structure are shown below. It is not really important to memorise these because this structure is not unique and is just a simplified example structure. About the only place where you need be concerned with these is for data exchange. They are shown here to explain how the models work. Each entity is a block of computer memory consisting of a consecutive set of memory words. The definitions are:

class body
int number;
shell *pshell;
edge *pedge;
vertex *pvert;
body *next;

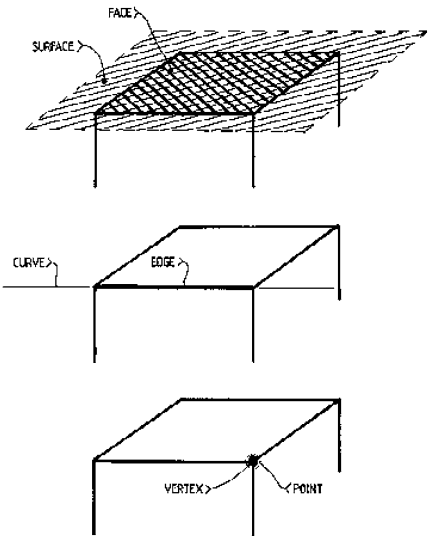


Fig. 2.18 Faces-surfaces, edges-curves, vertices-points

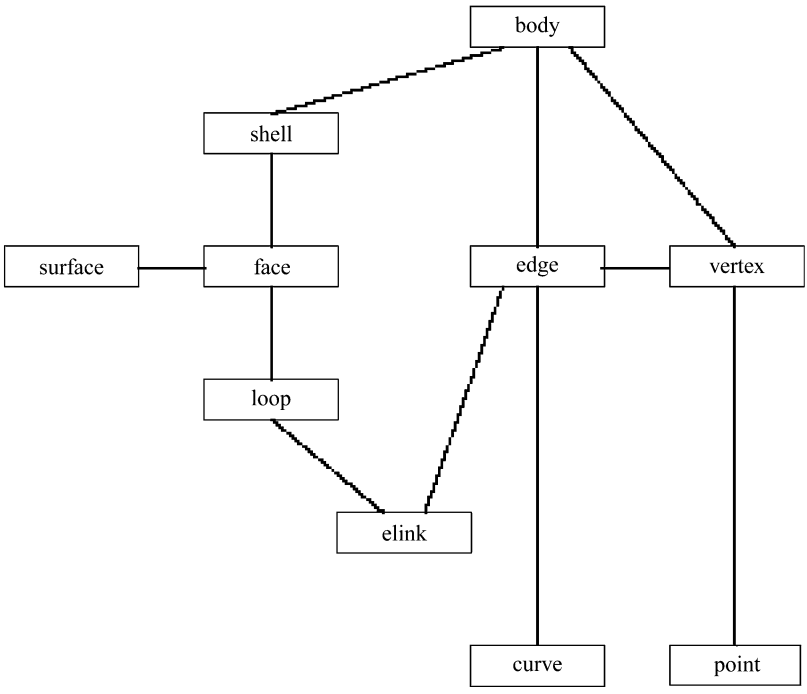


Fig. 2.19 Simple boundary representation data structure

class shell
int number;
face *pface;
shell *next;
body *pbody;

class face
int number;
shell *pshell;
loop *ploop;
surface *surf;
face *next;

class loop
int number;
elink *eref;
loop *next;
face *pface

class elink
int number;
elink *cclink;
elink *cwlink;
loop *ploop;
edge *pedge;

class edge
int number;
elink *rlink;
elink *llink;
curve *pcurve;
vertex *svert;
vertex *evert;
edge *next;

class vertex
int number;
union(loop,edge) *pref;
point *pos;
vertex *next

As an example of how an object is modelled, consider an object such as a cube (Fig. 2.20).

In memory, if you create such an object, you might have a structure like that shown in Fig. 2.21, with the circular nodes representing entities and the lines representing pointers. The **elink**s are shown as small circles without numbers, the right elinks on the top row, the left ones on the lower row.

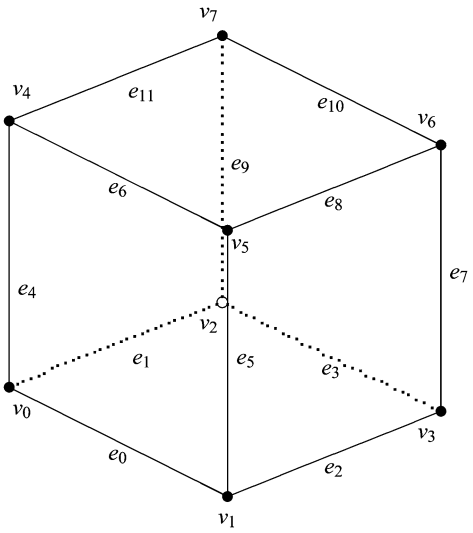


Fig. 2.20 A simple cube

If you were forced to build the datastructure manually in a CAD tool you would probably not use a CAD system. You, or anyone, would probably also make many mistakes because it is just too complicated. Frankly, the datastructure diagram in Fig. 2.21 is a mess, but it illustrates graphically what is in memory. Building this up is hidden by operations and suboperations so that you can work logically. [One thing you should note, though, is that the boundary representation is a very localised representation, which has advantages and disadvantages for CAD]. The real data structure of a solid modeller is fundamental to a CAD system, but it is not directly of interest to a user. To explain this, you might consider that the structure of a modelling system is a little like an onion. This is illustrated in Fig. 2.22.

At the heart of the modelling system are the data structures, of course, but these are hidden behind a set of interrogation and manipulation routines. This is done so that the data structure can be modified without disturbing the whole modeller. The geometry, also, is hidden. When working with algorithms the method is to work with the categories curve and surface rather than with geometric types such as straight line, circle, sphere, cone, etc. This makes it possible to change the geometric set, if necessary. For the user the representation should be completely transparent. It should not be necessary to know whether geometry is represented by a general type, such as NURBS, or by explicit types such as planes, straight lines, cylinders, etc.

On the next level up are simple traversal routines, for finding neighbouring elements in the datastructure or finding sets of connected entities. Also, there is an important class of operation, called the Euler operators, which are used to create complex operations. There are also more complex utilities, such as copying and transforming objects.

Complex operations, like Boolean operations or extrusions, are built on top of the simple utility routines to make them more stable if changes are made to the

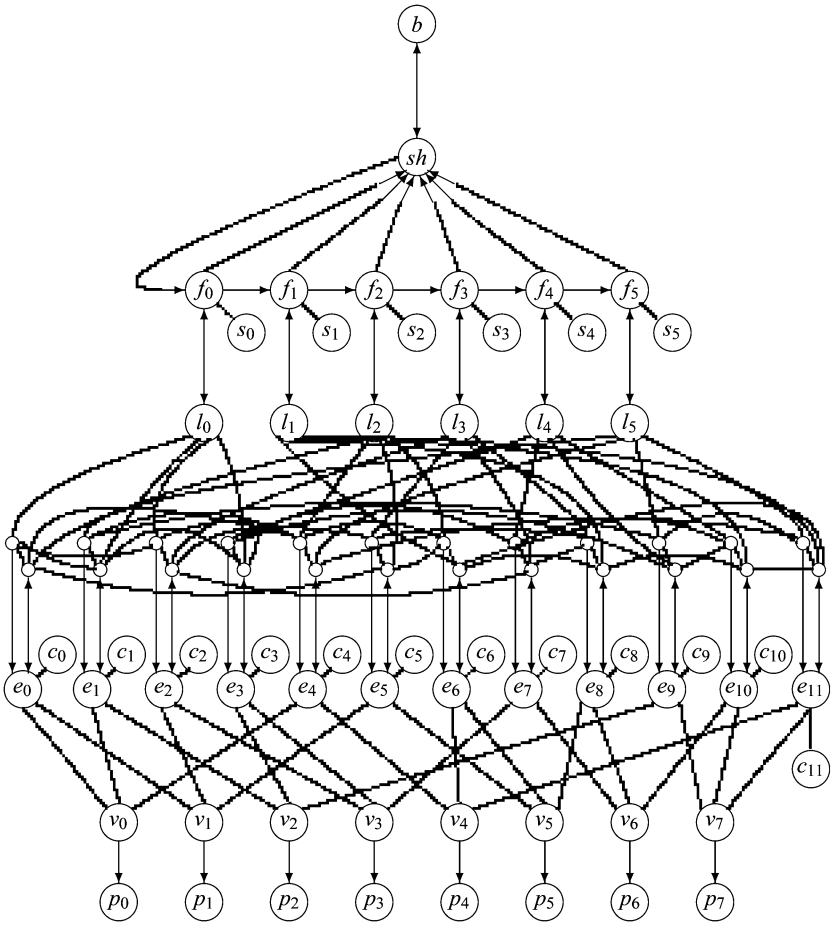


Fig. 2.21 Simple cube topological datastructure

basic datastructures. Applications can be built on top of the high level routines. The boundaries of these may be a bit fuzzy but this has been a general strategy since it was established in the BUILD system.

2.7.1 Finding Entities from Each Other

The datastructure handling routines are described more fully in [3]. Among the important topological routines are the traversal routines. There are multi-element traversals, such as finding all edges in a body, or all connected faces edges and vertices in a body part. These traverse the exact data structure and return a set of

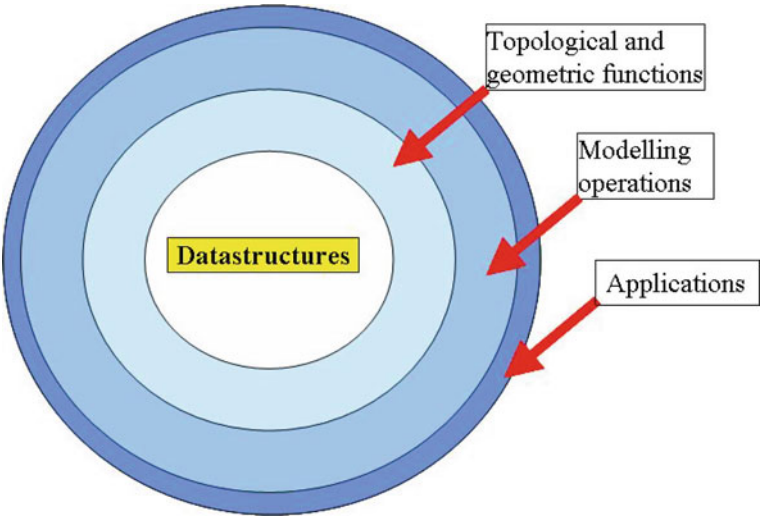


Fig. 2.22 Modelling "onion" or layered development

elements in the form of a simple list. The exact connections between the elements are handled by the routine, the user just sees the list, which is a well understood form. There are also single entity traversals to return neighbouring elements, such as the vertex at the opposite end of a given edge from a given vertex. The geometrical routines include such things as intersections, curve tangent at a point, surface normal at a point, creating surfaces by extruding, or sweeping, curves.

The single entity traversal routines, from [3] are:

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).

The next level of operations above these are what are termed Euler operators. These are very useful for developing different algorithms so are dealt with in more detail.

2.7.2 Finding Sets of Connected Entities

As stated in Stroud [3], 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. These take a structure known by the implementer and converts it to a simple list which can be handled by an application routine.

Using the classification from Stroud [3], the traversal procedures can be divided into four groups based on the usual type of datastructure.

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 under body

Topological set traversal:

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

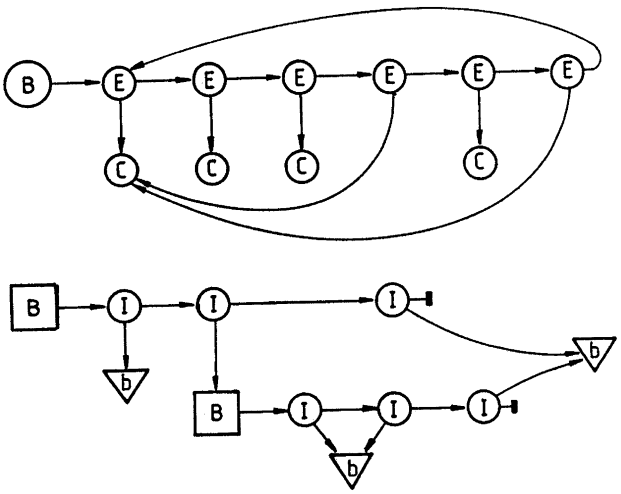


Fig. 2.23 Topological structures

Some entities can be shared in structures, so the shared entity traversals are needed to make sure that you do not process the same entity twice. For example, curves may be shared by several edges if, say, a single curve is broken into several pieces. A possible structure is illustrated in Fig. 2.23, top (from [3]). This can be a problem when transforming a body because it would be wrong to transform a curve once for each edge referring to it. Another possibility is that there are shared instances in assemblies, as illustrated in Fig. 2.23, bottom. 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 used to process all faces in an object, for example, or all single objects in an assembly.

The last group of traversal functions is important and 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.

2.7.3 Euler Operators

This section is based on a chapter from [3]. Some of this text is copied directly from there for completeness.

In mathematical terms the standard 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. This chapter describes the basic Euler operators for the simpler datastructure, non-manifold datastructures will be discussed in Chap. 6. 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, Hillyard and Stroud [8] as:

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, $\underline{b}$ 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 [9], by Braid et al. [8], by Eastman and Weiler [10], and by Mäntylä [11, 12], and an extension to handle non-manifold models is described by Luo [13]. The description given by Braid et al. is most appropriate here, so what follows is based on that work.

There are 99 possible Euler operators which change the number of any element by at most one, i.e. perform transitions between adjacent nodes in Euler space. Of these, 60 are obvious combinations leaving 39 unique operators. Any change, any simple or complex operation can be described in terms of combinations of these Euler operators. Since 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 full list of Euler operators as well as the shorter list are given in Appendix A.

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 e.g. Giblin [14] and Braid

et al. [8]) 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 which 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 which 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 be implemented. Indeed, the effect of some of them is rather obscure.

2.7.3.1 Spanning Sets and Decompositions

As described by Braid et al. [8], it is possible to choose a set of five Euler operators which 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:

(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 which 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 MEV (Make and Edge and Vertex) operations, five MFE (Make a Face and Edge) operations, and one MBFV (Make Body, Face and Vertex) operation. Since 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.

Note, though, that it is possible to create a series of cube creation sequences from different combinations of these thirteen elements. Some of these are shown in Figs. 2.24, 2.25, 2.26, 2.27 and 2.28.

The ratio of edges to vertices in the final object is 3:2 and the ratio of edges to faces is 2:1. This means that every vertex has three edges and every edge runs between two vertices. Similarly, every edge has two adjacent faces and every face has four edges. Figure 2.24 shows what happens when trying to maintain the ratio of edges to faces. Figure 2.25 shows what happens when trying to maintain the ratio of edges to vertices.

Figure 2.26 shows what happens when all the MEV operations are applied first followed by all the MFE operations.

Figure 2.27 shows the other extreme case, when all the MFE operations are applied first followed by all the MEV operations.

Finally, Fig. 2.26 shows a real sequence when creating a cube.

Figure 2.29 illustrates some of what happens. The Euler space is six-dimensional, as stated earlier, but taking just three of those, the edge, face and vertex number dimensions you can just about represent this on paper. The small dots represent all combinations of faces, edges and vertices in the cube range, that is: $0 \leq v \leq 8,\ 0 \leq e \leq 12,\ 0 \leq f \leq 6$. Not all these combinations are valid. The valid combinations are governed by the simplified equation $v - e + f = 2m, m$ is usually

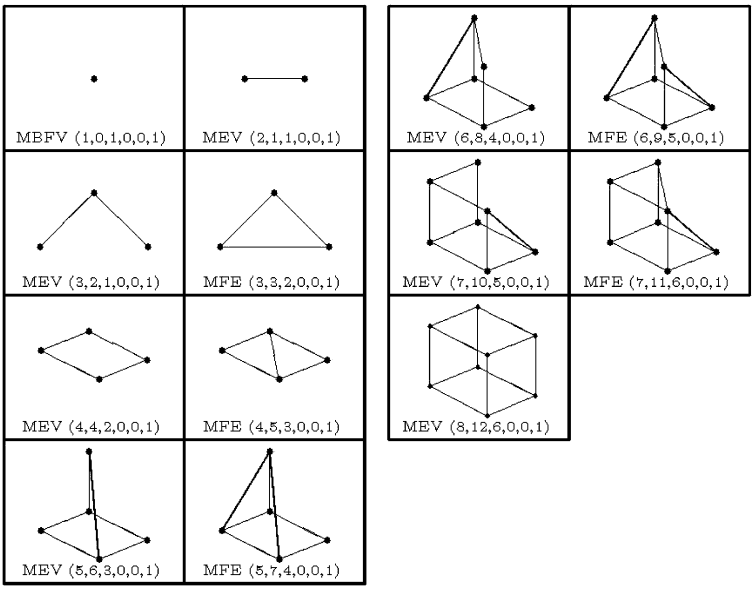


Fig. 2.24 Creating a cube with Euler operators (1)

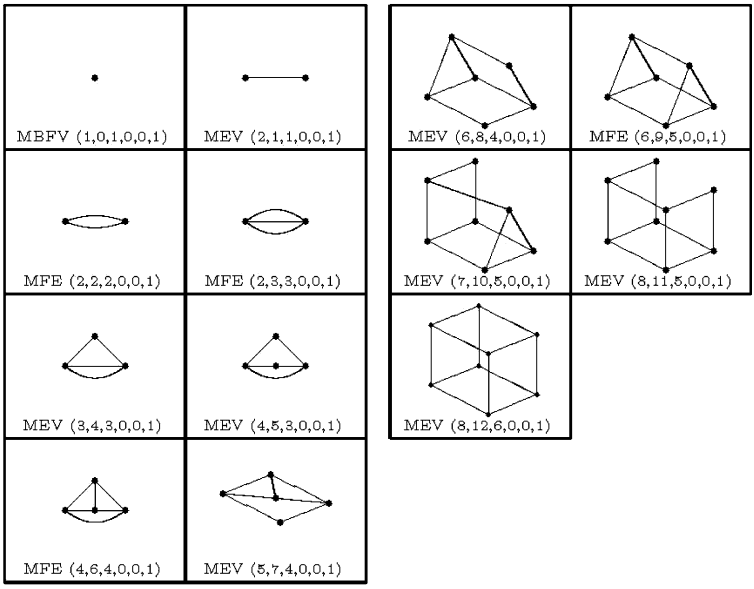


Fig. 2.25 Creating a cube with Euler operators (2)

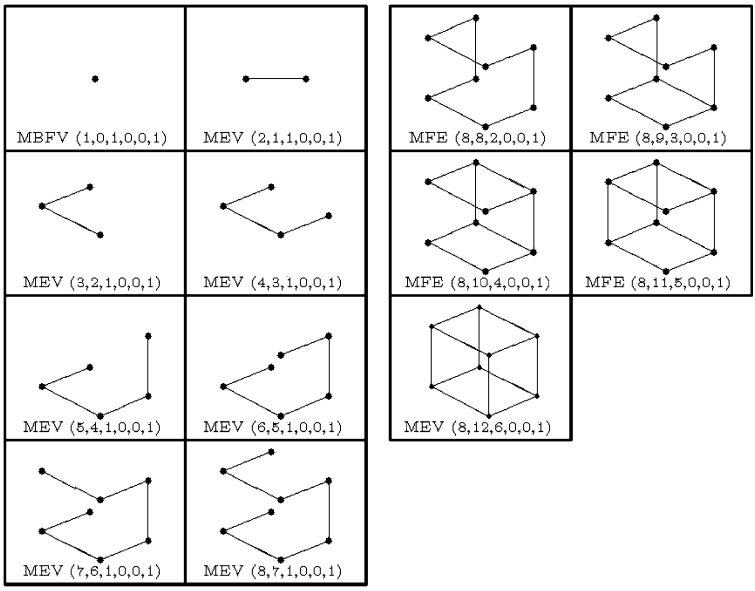


Fig. 2.26 Creating a cube with Euler operators (3)

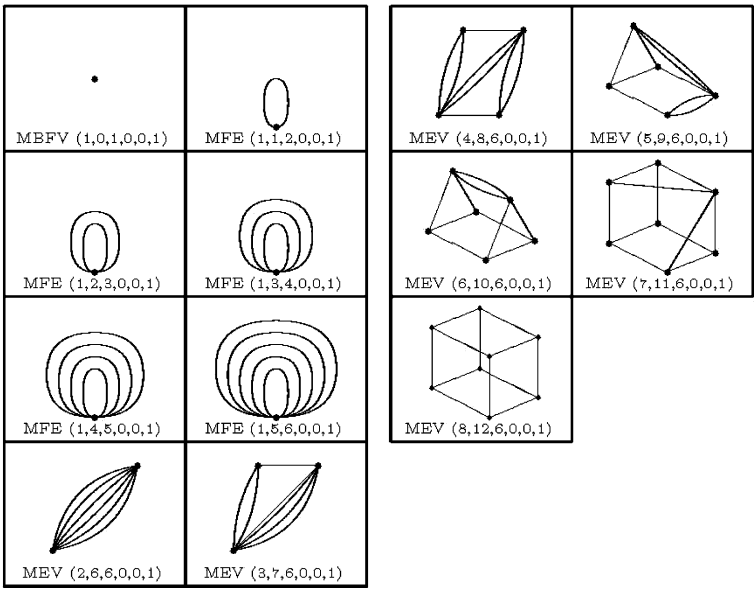


Fig. 2.27 Creating a cube with Euler operators (4)

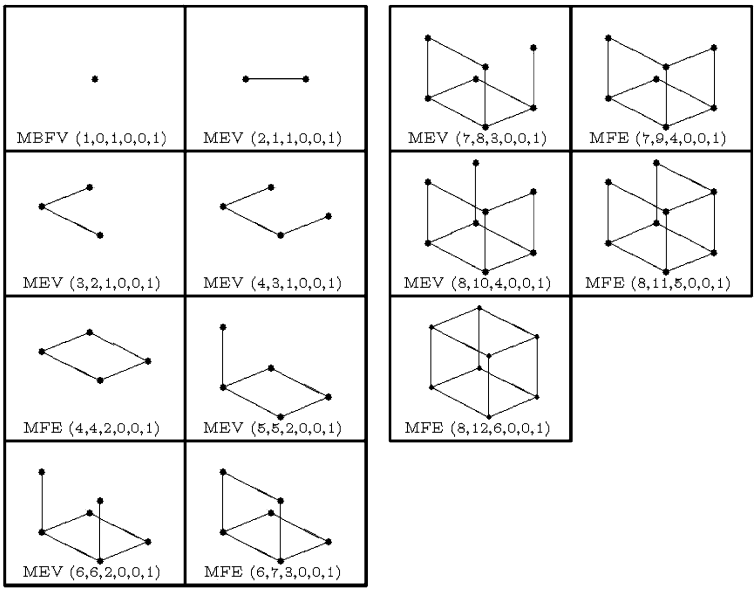


Fig. 2.28 Creating a cube with Euler operators (5)

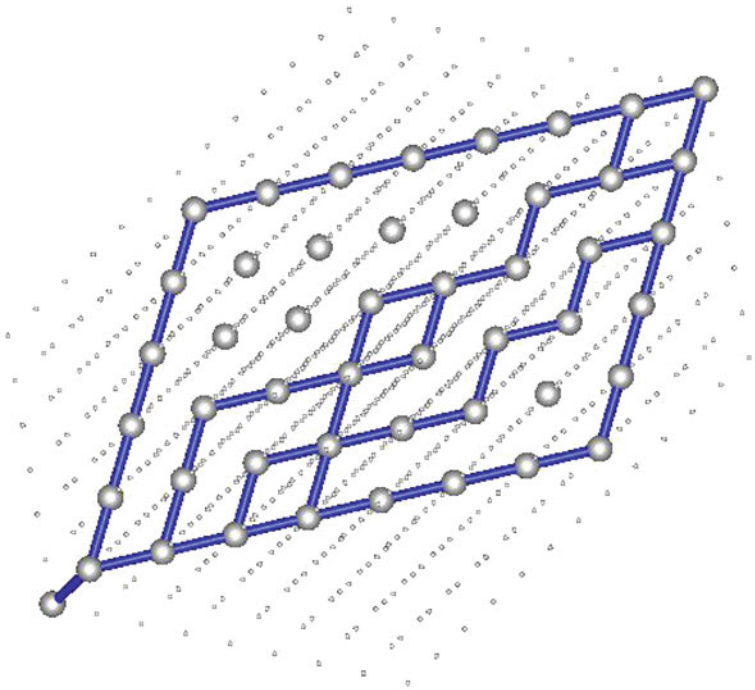


Fig. 2.29 Creating a cube with Euler operators (1)

1, but the origin is also a valid Euler point. The points which obey this equation are shown larger in the figure. The points with $v = e = 0, f = 2$ and $f = e = 0, v = 2$ are excluded by the rules defined previously. The process of building a cube means walking, more like staggering, through this point field in a particular sequence. The previous five figures show some of these sequences.

The next section describes how the Euler operators are used.

2.7.4 Stepwise Algorithms

Normally you don't see the Euler operators and the other lower level operators when you use a CAD system. The onion-layer that you see concerns high-level operations which make specific changes to a model. These, in their turn, are usually built on sequences of small changes, using Euler operators and other simple operations to make a set of small changes. Such a way of building an operation is termed a "stepwise algorithm".

More history. In the BUILD system, which established many basic principles of boundary representation modelling, there were two types of modelling operation: (1) Boolean operations; and (2) everything else.

Boolean operators are a powerful general tool performing a global comparison of two objects to produce a result. Many other operations were developed which performed specialised changes which were sometimes difficult to achieve except by using special code. This second category was important because, at that time, the Boolean operations were relatively slow because they performed a global check on objects. The other operations, sometimes called "local operations" performed localised changes, sometimes producing a global effect, but did not check the consequences. This meant that it was sometimes possible to create invalid objects termed "self-intersecting objects". Nowadays things have changed. Boolean operations have become faster, based principally on results by Smith in BUILD, and so several operations now create volumes and then apply a Boolean operation to add or subtract this from the part being modified. This is explained in Chap. 4, but since this is not the only way to do things, and since not every operation can be done that way, here is an explanation of the stepwise methodology.

2.7.4.1 Linear Extrusion

An example of this type of operation is the extrusion operation. The sequence for creating a cube is shown in Fig. 2.28. Another example, for a six-sided shape is shown in Fig. 2.30. Figure 2.30a shows the original figure. The first step in the extruding the base shape is to add an edge and a vertex in the extrusion direction using an MEV operation, Fig. 2.30b. A second edge is added in the same manner, Fig. 2.30c and then a cross-edge added, Fig. 2.30d. The sequence then continues,

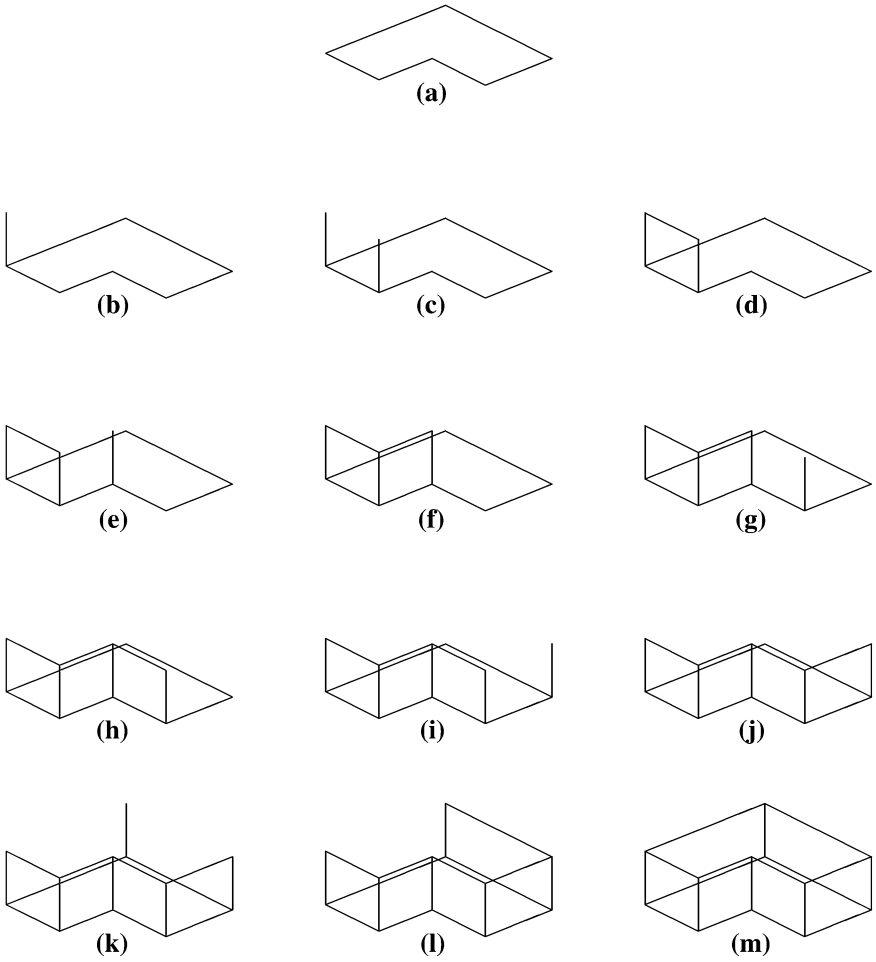


Fig. 2.30 Linear extrusion in a stepwise manner

add edge-in-extrusion-direction (Fig. 2.30e), add cross-edge (Fig. 2.30f), add edge-in-extrusion-direction (Fig. 2.30g), add cross-edge (Fig. 2.30h), add edge-in-extrusion-direction (Fig. 2.30i), add cross-edge (Fig. 2.30j), add edge-in-extrusion-direction (Fig. 2.30k), add cross-edge (Fig. 2.30l). Finally a single cross-edge is added to join the last two extensions, (Fig. 2.30m).

A linear extrusion of a face (a flat shape has two faces, one on top and one on the bottom) can be defined as a sequence of steps, one for each edge surrounding the face. The very first step is to create a single extrusion edge. Then, for each edge until the last, one extrusion edge and one cross edge are created. For the final edge a single cross edge is created between the last extrusion vertex and the first extrusion vertex created in the special first step. Each "step" is governed by an edge. The edges are found from the loops, or contours, around the face.

In pseudo-code form this might look like:

```
for all loops in face do
for all edges counter-clockwise round loop do
BEGIN
if (edge IS start OF loop)
v0 = oldv = MEV(vcwel(edge, loop), extrusion direction);
if (edge IS last OF loop) v = v0;
else v = MEV(vccel(edge, loop), extrusion direction);
e = MFE(oldv, v); oldv = v
END;
```

The function "vcwel" finds the vertex clockwise from a vertex from a given edge round a given loop. Similarly, the function "vccel" finds the vertex counter-clockwise from a vertex from a given edge round a given loop. These are two of a set of simple relational functions for traversing the data structure. The complete set is described in [3].

This is the very simplest form of extrusion. Nowadays this is the extrusion that is done. When you extrude a shape drawn on an existing face then the shape to be extruded is made into a 2D shape, extruded and then combined using a Boolean operation. More about this later.

2.7.4.2 Splitting an Object

Splitting an object with a plane can also be defined in a stepwise manner. In the extrusion example the edges were found and extruded in an ordered sequence. For the splitting algorithm here this is not done so there are some special cases which need to be recognised and handled.

The pseudo code is:

```
for all faces in object do split_face_with_plane(face, plane);
pull_results_apart;
```

This is illustrated in Fig. 2.31.

Most of the work is in the split_face_with_plane function. As can be seen from the outline of the split operation in Fig. 2.31 this inserts edges where the section plane cuts the face (e.g. Fig. 2.31b). These edges are then sliced (e.g. Fig. 2.31c). If one or both end vertices of the slice edges are also end vertices of other slice edges, the vertices are split (e.g. Fig. 2.31f). Eventually, splitting the vertices creates two disjoint loops which become the outer loops of two new faces.

A complete operation is a little more complicated. The basis is still a stepwise construction centred around the central function, split_face_with_plane, but this needs to identify different special cases. For example, what happens if they are several new section edges in the face? What happens if an edge of a face lies in the

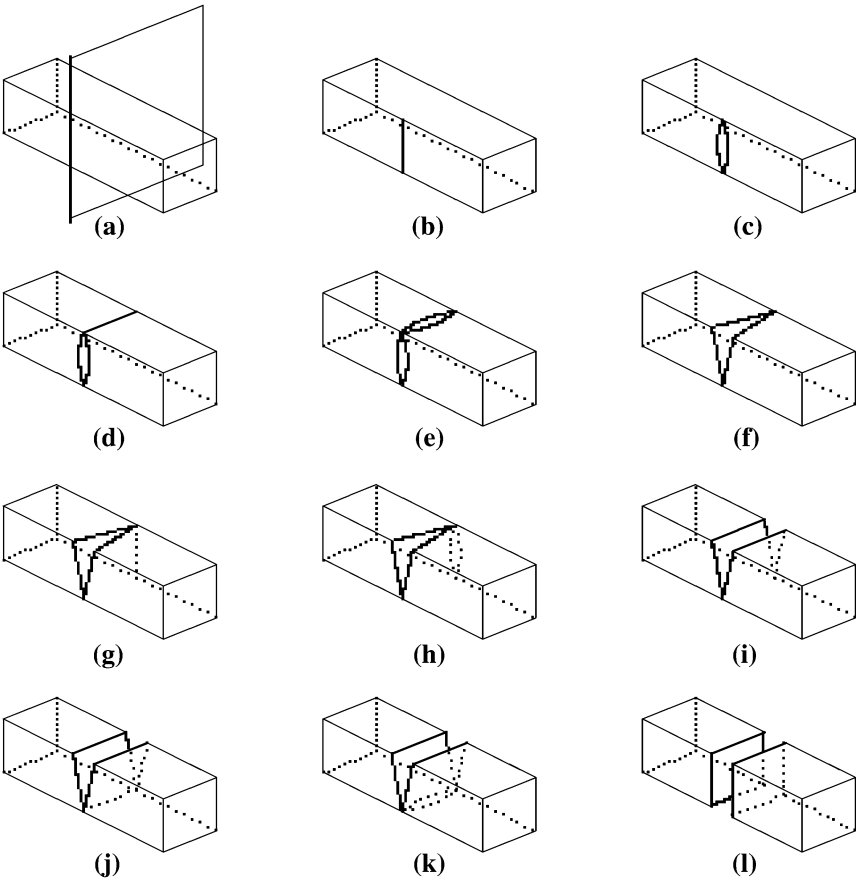


Fig. 2.31 Splitting an object with a plane

section plane? Extending the function to identify special cases keeps the stepwise character, though.

For the first question, if there are multiple section edges then these are created and sliced separately, as shown on the top line of Fig. 2.32.

Multiple section edges may also just touch a boundary, as in the case shown on the bottom line of Fig. 2.32. In this case, as well, the section edges are created and sliced, but there is an extra step to slice the common vertex, leaving the result in Fig. 2.32f.

For an edge lying in the section plane, such as that shown in Fig. 2.33, there are two cases. If the edge is convex then the object does not cut the section plane at that edge, so the edge is ignored. If the edge is concave, as in the figure, then the edge is sliced. Note, though, that the sliced edge has to be ignored when the other adjacent face is processed.

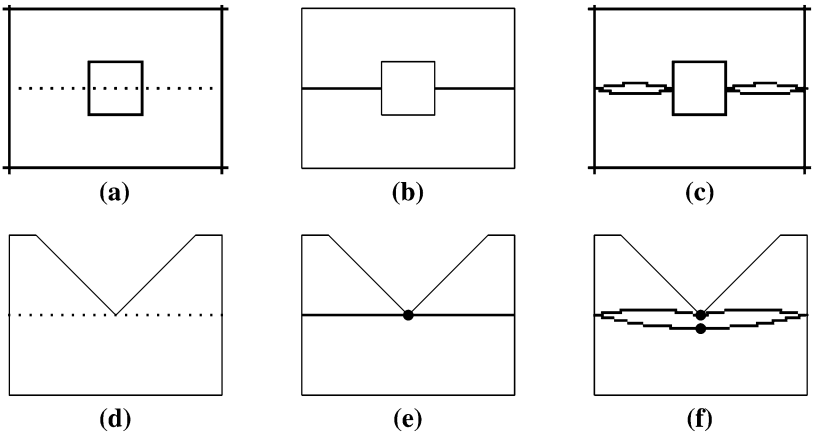


Fig. 2.32 Handling multiple section edges in a face

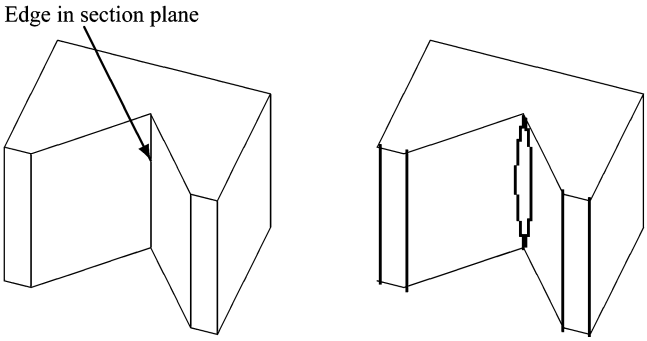


Fig. 2.33 Edge lying in section plane

The same set of simple steps are used to create the section seam. Once the section seam, or seams, are complete the connected sets of faces are moved into separate bodies.

These are just simple illustrations of how stepwise algorithms are built up from repetitions of basic steps.

2.7.4.3 Etcetera

The aim, her,e is not to run through a lot of modelling algorithms. These are described in Stroud [3] and short descriptions of several operations are given in Chap. 4. The aim is to illustrate the stepwise notion of building up operations as sequences of simple steps.

2.7.5 Complex Utilities

Finally in this section come the complex utilities. These are functions which perform well-defined common operations which are useful to higher level functions. Examples are copying objects, deleting objects, transforming objects, the dual function.

2.7.5.1 Copying

Copying is illustrated in Fig. 2.34. The entities in the object to be copied are renumbered temporarily so that they have consecutive numbers starting at zero. Lists of new entities matching the original entities are created. The original entities are then traversed and the corresponding new entities are linked into structures using the numbers as logical pointers to list elements.

Take a cube as an example. The cube might have one shell, six faces, six loops, twenty-four loop-edge links, twelve edges, eight vertices, six surfaces, twelve curves and eight points. The shell is numbered 0, the faces, loops and surfaces are numbered 0–5, the loop-edge links are numbered 0–23, the edges 0–11 and the

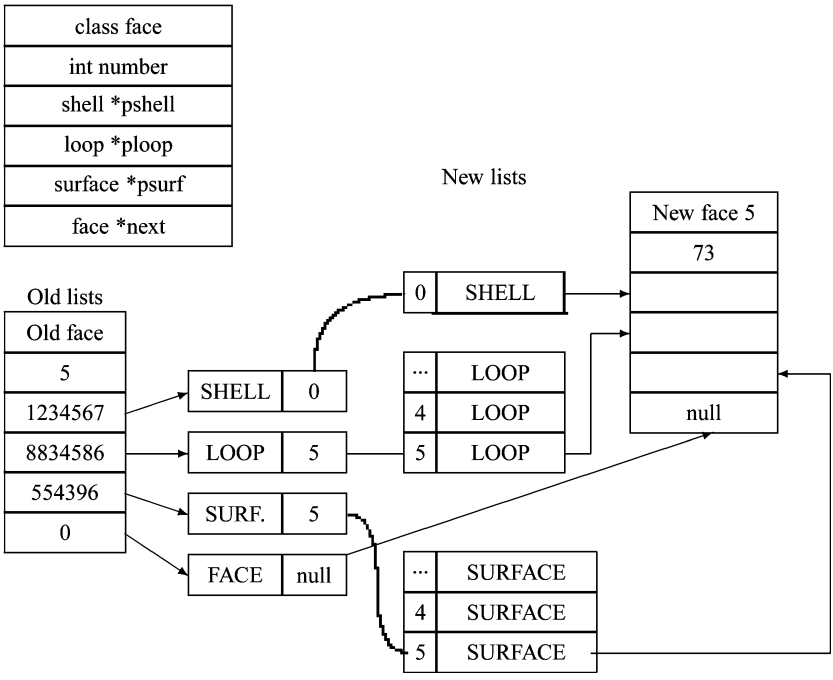


Fig. 2.34 Writing a face to disk

vertices and points numbered 0–7. A list of one new shell is created, a list of six faces is created and so on, for each type of entity.

Figure 2.34 shows what happens for face number 5 when linking the data. The datastructure definition is used to interpret the data. If there are pure numerical fields (except for the entity number), which there aren't in this case, they are simply copied. For the face, the first relevant field is the shell pointer field, *pshell*. In the original face this shell is numbered zero, so a pointer to the shell in position 0 of the list of new shells is copied into the *pshell* field of the new face. Note that the list indices start from zero, here. The next field is the loop pointer field, *ploop*. The loop referred to in this field from the original face is numbered 5, so a pointer to the loop in element 5 of the list of new loops is copied into the *ploop* field of the new face. Similarly for the surface, the surface referred to in the *psurf* field of the original face is numbered 5. A pointer to the surface in element 5 of the new surfaces is copied into the *psurf* field of the new face. The final field, the *next* field is a face pointer, but this is NULL, so the *next* field of the new face is set to NULL.

This process is repeated for all the entities of the original object.

2.7.5.2 Deleting

Deleting can be applied to single elements or to whole structures. It is useful to use Euler operations to delete connected edges, faces, vertices or hole-loops in an object. This means that the resulting structure left is correct. Once the element has been deleted from the topological structure it can be removed completely by disconnecting it from any lists in the object to which it belongs. This, again, is done with specialised low-level routines which handle the datastructure directly.

However, although it is possible to delete complete objects, or shells also, using Euler operators it may be more efficient to delete all entities more directly. At any rate, it is necessary to understand the difference between deleting using Euler operators, which disconnect elements, and pure deletion which probably doesn't tidy up completely surrounding elements.

2.7.5.3 Transforming

Transforming objects is done on three levels. What appears to be an object might be an instance in an assembly, which means that the transformation is multiplied into the instance transformation, and doesn't change the original geometry. Transformation matrices will be discussed in Sect. 5.2.2. If the object has a transformation attached then, again, the transformation may simply be accumulated with the object transformation instead of changing the real geometry. The third level really changes the geometry.

Accumulating transformations tends to be preferred to really changing the geometry for several reasons. First of all it is faster, second, some systems allow non-uniform scaling, which changes the nature of the geometry, which is usually

transformed to general numeric forms. However, it is harder to go back from the numeric to the simpler forms, so you get what is called "geometric migration", to be discussed later, in Sect. 5.3.

Really transforming single objects changes the geometry of an object but not its topological structure. This is one example of where you need to identify elements which are used several times and transform them only once. To transform the geometry it is best to compile a list of geometric elements using the set traversal tool described earlier and then process each one, changing its nature if necessary, and applying the transformation.

2.7.5.4 Dualling

Dualling is a strange operation that normally you would not want to use, nor even know about, but it is useful as a background structure for some operations. An example is shown in Fig. 2.35.

In effect, the dual operation creates a new object with a vertex for each face and a face for each vertex, which means that the dual of a cube (with eight vertices, twelve edges and six faces) is an octohedron, with six vertices, twelve edges and eight faces. It is a graph theoretic operation which is used, maybe directly for some smoothing operations, or indirectly as a navigation aid for unfolding objects, for example.

The operation also uses a technique such as that described for copying. The first step, renumbering the entities, is the same as for copying. However, instead of creating lists of new entities of the same number of elements as in the original object, there is a switch. For a cube, a list of eight new faces and a list of six new vertices are created. Edges are reconnected appropriately. For example, take an

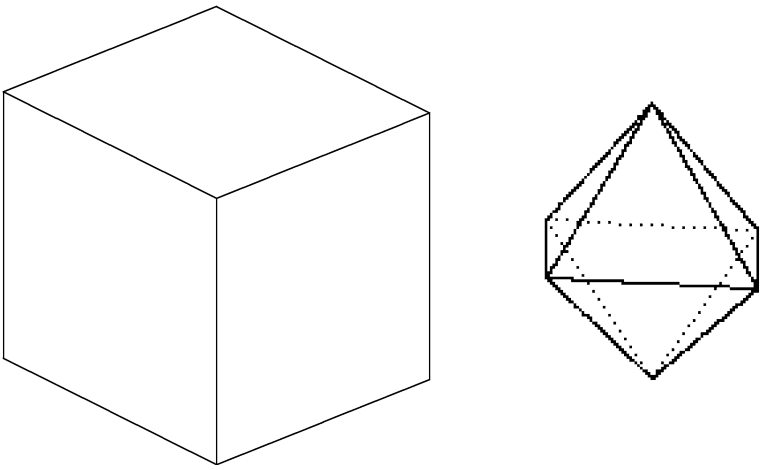


Fig. 2.35 Cube and dual (from Stroud [3])

edge lying between faces A and B, running between vertices C and D in the original object. In the new object, the corresponding edge will run between vertices A and B and lie between faces C and D.

2.8 Complex Geometry and Integration

The next topic to mention briefly concerns the use of geometry in modelling. Geometry in modelling systems is usually of two types: analytic or numeric. Analytic geometry concerns specific forms, such as plane surfaces, cylinder surfaces, cone surfaces. Numeric geometry is a general form the shape of which is determined by a set of points called control points. Normally these are mixed in a manner which is transparent for the user. You should not need to know which is used because the system should use the appropriate one and modify it according to set rules.

As an example, try the example in Fig. 2.36.

I don't know who created this example, possibly Fjällström in his dissertation [15], but it is a useful example to show the use of complex surfaces. Create the object shown in the figure and then blend the six edges meeting at the central vertex, marked "*v*" in the figure. Even though the original geometry is simple, when the edges are blended it is usual that there is a complex blend surface, or surfaces, instead of the vertex. Similarly, intersections between curved surfaces may produce complex curves which are calculated automatically.

This is what happens in normal modelling and is taken care of automatically. Complex geometry is used to model all surfaces not modelled explicitly. There are other occasions, though, when numerical geometry is used explicitly. For example, for car bodies or other products with aesthetic shapes, or for functional forms like turbine blades. In this case the shape is usually created as a surface form, although

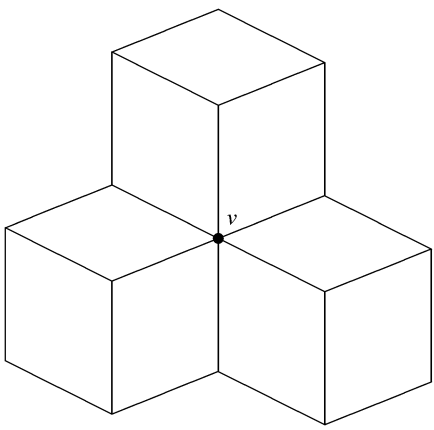


Fig. 2.36 Complex blend example

if these are used in isolation the problem is then to integrate them with solid models.

The integration of solid and surface modelling has long been the subject of research. One method was the sol-called "SETSURF" method, which sets a surface into a face (Braid [16]). Another interesting method was to smooth polyhedral models by inserting sort-of blend surfaces (Chiyokura and Kimura [17]). Hybrid modelling, with elements suitable for representing surfaces, was introduced by Kjellberg et al. [1]. This hybrid methodology is now common in CAD systems and is described further in Chap. 6.

2.9 The Cylinder Test

The "cylinder test" is a small diversion from the main theme of this chapter. It involves creating a cylinder and circular shape to identify how the system handles closed geometry. This should be transparent for users, but this is not always so. Identifying this gives clues to the way the system works.

In the original research system, BUILD, cylinders were represented with three side edges, because the "point-in-face" test counted angles. The first commercial kernel system, Romulus—a forerunner of today's Parasolid and ACIS systems, advanced this, but maintained a single edge down the cylinder side, presumably also for the "point-in-face" test. The ACIS kernel has no side edges but has two vertices on the top and bottom circular edges.

So, use your CAD system to create a cylinder and count the number of edges around the cylindrical surface.

- Zero – The modeller in the CAD system is modern and transparent for the user.
- One – The modeller is a little old-fashioned, you will get artefact edges of rotational objects.
- Two – CATIA has this, and this avoids another problem, that a single edge is a so-called "wire" edge with the same face on the left and right. These are also artefact edges and may not need to be at 180 degrees.
- Three or more – old-fashioned, maybe a BUILD derivative would have three, but there should be no need to have these in modern systems.

A similar problem can be found at the level of 2D. Both cylinders and circles are examples of "continuous" geometry, that is, geometry which curves back on itself. A true circle may have neither start nor end, but this is not true for CAD geometry. For practical purposes it is necessary to have a start and end, albeit at the same place. This is connected with parametrisation of geometry, which is a necessary mechanism in many algorithms. For a circle, the parametrisation may run from 0 to 2π.

A test you can do in CATIA v5 is to create a circular curve and then cut it with a vertical line, as shown on the left of Fig. 2.37. Instead of being divided into two pieces the circle is in three pieces, as can be seen by selecting the parts. The parts

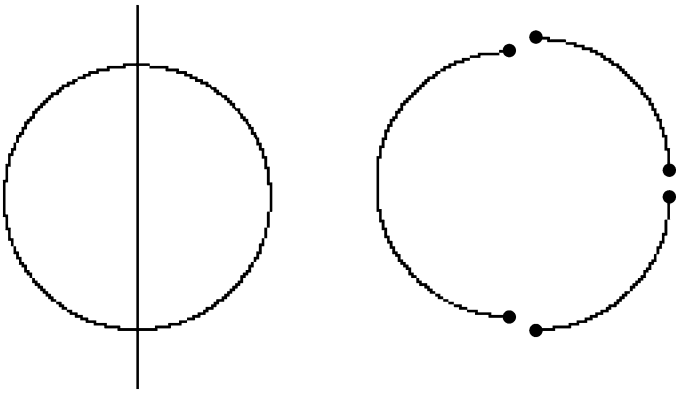


Fig. 2.37 Cutting a circle

are shown a little displaced on the right of Fig. 2.37. The reason is, apparently, that the point dividing the right-hand "half" of the circle is the start- and end-point of the circle. Instead of adjusting the circle and using one of the new intersection points when it is cut, CATIA maintains the original point. There may be a good reason for this, but I can't think of one. For the user it provides an oddity which does not seem logical at first glance. A personal opinion is that it would have been better to make this sort of artefact invisible to the user.

2.10 Assemblies

Assemblies are a frequent problem because the implementation method is not well understood. Chapter 13 explains more about assemblies. A modern assembly structure is shown in Fig. 2.38.

A major conceptual difficulty concerns how objects which occur more than once in an assembly are represented. There are two ways of doing this, by physically copying the object or by referring to it more than once. The latter

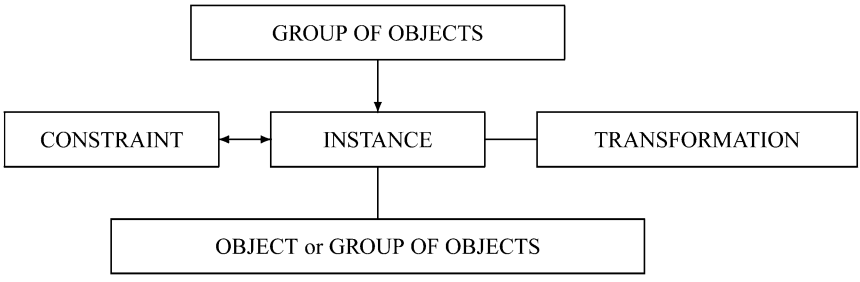


Fig. 2.38 General assembly datastructure

method is the "computer science" solution, which has several advantages, but many people seem to want to copy the object and then do not understand why the CAD system does not "know" that the copies are the same object. Figure 1.57. shows the expected structure of an assembly with multiple elements. A simplified assembly is shown in Fig. 2.39.

A simple rule is that, in an assembly, if multiple parts with the same shape are used, then there should be one object model of the part shape and multiple instances. Each instance has a transformation associated with it containing the information about repositioning of the part from its defined position to the position at which it is needed. Transformations are described in Sect. 5.2.2.

If one of the instances is to be modified then it is normal to copy the common object and to change the instance to point to the copy, which is then modified, as in Fig. 2.40. If the referenced object is modified without copying then all instances will show the change. It is, therefore, necessary to think about the effect you want before modification, if you want the change propagated to all instances or to change just one instance. Some systems let you perform the re-instancing operation explicitly, to "make-unique" one instance. This leaves you in control of the changes.

There is a special type of part used in assemblies which is "standard". These can be parts which are to be purchased from suppliers, such as nuts, bolts, motors, etc. Normally these should not be modified and so may be "blocked" from being changed. Such standard parts may come in 3D catalogues from suppliers. It is necessary, then, to consider the format in which they are communicated. This could be in a standard format, such as STEP (see Chap. 9) or in native format.

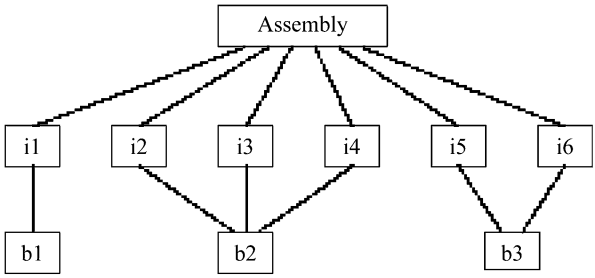


Fig. 2.39 Simple instance structure

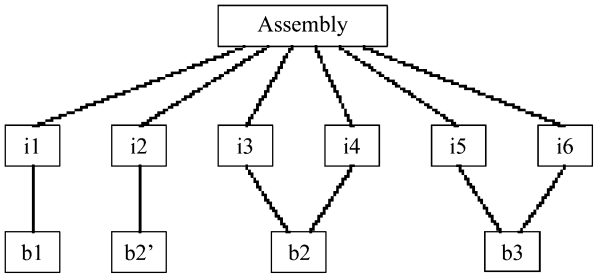


Fig. 2.40 Simple instance structure

If they are in a neutral format, like STEP, then you will probably only get the final shape anyway, which makes modification difficult.

If parts, standard or otherwise, are to be modified physically, say by drilling a hole in a baseplate, or cutting off a bolt, say, then the method mentioned above for copying instances loses the connectivity between the parts. This is a pity, since this is likely to increase the work in the manufacturing stage. A method for doing this would be to assign local modification "trees" to the instances themselves and to allow only material removal operations. I have not seen this in any CAD system yet, so part association would have to be done on paper or by some other means, such as an external database, or PLM (Product Lifecycle Management) system.

2.11 Chapter Summary

This chapter describes, briefly, solid modelling methods. The details of solid modelling are outside the scope of this book. However, the different modelling methods have different implications for the user, and hence a cursory knowledge of the methods is useful for understanding what is going on. The CSG method was described, which uses Boolean operators on primitive objects to build models. The method currently used for the majority of CAD systems, Boundary representation, was also described. Boundary representation was described in more detail because this is the current method used in most CAD systems. Specifically, some simplified datastructures, the layered manner of implementing systems as well as some of the more common operations were described.

2.12 Representation Exercises

These exercises are intended to test your understanding of representations.

2.12.1 CSG Decompositions

Figures 2.41, 2.42, 2.43 and 2.44 show simple objects to try decomposing into CSG trees of primitive objects and Boolean operations.

2.12.1.1 The Cylinder Test

Perform the "cylinder test", described in Sect. 2.9, using your CAD system.

Fig. 2.41 Object 1 to decompose in CSG trees

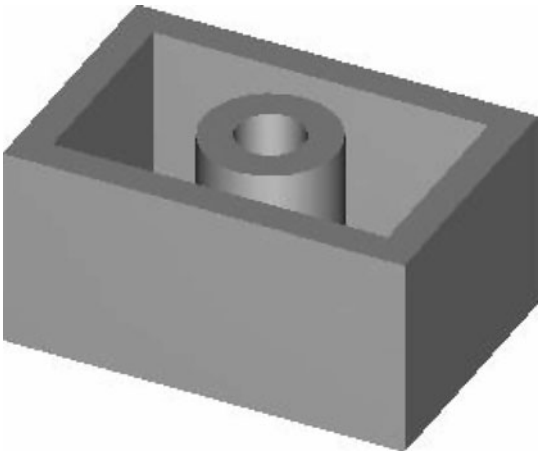


Fig. 2.42 Object 2 to decompose in CSG trees

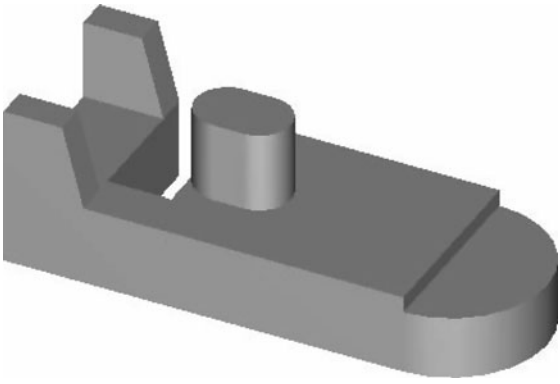


Fig. 2.43 Object 3 to decompose in CSG trees

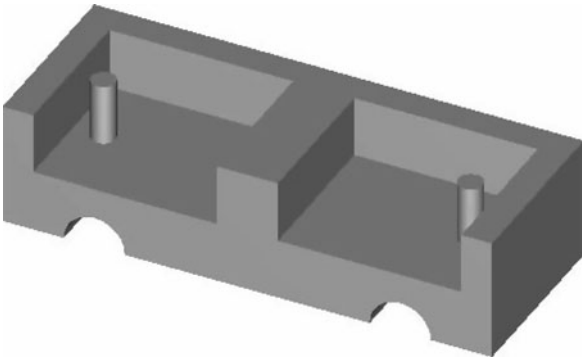


Fig. 2.44 Object 4 to decompose in CSG trees

References

1. Kjellberg, T.A., Lindholm, G., Sorgen, A., Haglund, G.: GPM Report 10: GPM—Specifikation Volymgeometri. Department of Manufacturing Systems. KTH, Stockholm, Sweden. Confidential document, ISBN 91-85212-54-7 (1980)
2. Jared, G.E.M., Dodsworth, J.: Solid modelling. In: Rooney, J., Steadman, P. (eds.) Principles of Computer-aided Design. Pitman Publishing in association with the Open University, ISBN 0 273 02672 0 (1987)
3. Stroud, I.A.: Boundary Representation Modelling Techniques. Springer, Heidelberg (2006)
4. Okino N., Kakazu Y., Kubo H.: TIPS-1: technical information processing system for computer-aided design, drawing and manufacturing. In: Proceedings of the Second PROLAMAT, vol. 73, pp. 141–150. (1973)
5. Requicha, A.A.G., Voelcker, H.B.: Constructive solid geometry. Technical Memo. No. 25, Production Automation Project, University of Rochester, NY (1977)
6. Katainen, A.: Monadic set operations. CAD J **14**(6) (1982)
7. Tilove, R.B., Requicha, A.A.G.: Closure of Boolean operations on geometric entities. Computer-Aided Des. **12**(5), 219–220 (1980)
8. Braid, I.C., Hillyard, R.C., Stroud, I.A.: Stepwise construction of polyhedra in geometric modelling, (1978). In: Brodlie, K.W. (ed.) Mathematical Methods in Computer Graphics and Design. Academic Press, London (1980)
9. Baumgart, B.G.: Geometric modelling for computer vision. AD/A-002 261, Stanford University (1974)
10. Eastman, C.M., Weiler, K.: Geometric modeling using the Euler operators. Research Report 78, Institute of Physical Planning, Carnegie-Mellon University (1979)
11. Mäntylä, M.: A note on the modeling space of euler operators. Comput. Vis. Graph. Imag. Process. **26**, 45–60 (1984)
12. Mäntylä, M.: An introduction to solid modeling. Computer Science Press, Maryland, ISBN 0-88175-108-1 (1988)
13. Luo, Y.: Solid modelling for regular objects renewed theory, data structure and Euler operators. Ph.D. Dissertation, Computer and Automation Institute (1991)
14. Giblin, P.J.: Graphs, Surfaces and Homology. Chapman and Hall, London, ISBN 0 412 21440 7 (1977)
15. Fjällström, P.-O.: Integration of Free-Form Surfaces and Solid Modelling. Ph.D. Dissertation, Department of Manufacturing Systems, PS-Lab, IVF/KTH, Stockholm, Sweden (1985)
16. Braid, I.C.: Notes on a geometric modeller. CAD Group Document 101, Cambridge University Computer Laboratory (1979)
17. Chiyokura, H., Kimura, F.: Design of solids with free-form surfaces. Comput. Graph. (SIGGRAPH 83 Proc.) **17**, 289–298 (1983)

http://www.springer.com/978-0-85729-258-2

Solid Modelling and CAD Systems
How to Survive a CAD System
Stroud, I.; Nagy, H.
2011, XXII, 689 p., Hardcover
ISBN: 978-0-85729-258-2