SOLID MODELLING USING LINEAR OCTREE REPRESENTATION

By

SHEUNG-LAI SUNNY HO

B.Math (Hons.), University of Waterloo, 1983

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

THE FACULTY OF GRADUATE STUDIES

(DEPARTMENT OF COMPUTER SCIENCE)

We accept this thesis as conforming

to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 1985

Department of COMPUTER SCIENCE

The University of British Columbia
1956 Main Mall
Vancouver, Canada
V6T 1Y3

Date OCTOBER 2, 1985

Abstract

Object representation is the backbone of any solid modelling system. Hierarchical spatial decompositions of objects called octrees introduced very efficient algorithms for boolean set operations and some restricted classes of geometric transformations. Linear octrees, a compact encoding of the octrees, result in a significant reduction of storage requirements, and lead to simpler algorithms for most modelling operations.

This thesis investigates some properties of linear octrees with emphasis on object generation. By interpreting linear octree node digits as binary numbers, some simple conversion and node trimming algorithms are found, which when combined with a node enumeration algorithm, generate the linear octrees of cuboidal volumes efficiently. A simple and uniform approach is devised to perform arbitrary geometric transformations by means of cuboid generation. Experiments shows these algorithms maintain the efficiency of special cases while degrading linearly with the number of intermediate nodes generated.

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my supervisor, Dr. Günther F. Schrack for his guidance and suggestions for this thesis, and to thank Dr. Robert J. Woodham for reading the final draft.

The variety of computing facilities at UBC has added fun to the serious academic work. Special thanks are directed to all fellow graduate students who have been very helpful in answering questions.

Chapter 1

Introduction

Solid modelling refers to the theory and practice for computing the properties of and manipulating solid objects represented by some abstract data structures. It is becoming popular in many computer graphics applications. Evolving throughout the past decade, the technology has now been transferred from the stage of research projects to production systems [REQU82,REQU83]. Areas such as engineering design, manufacturing, architecture, and medical imagery are using solid modelling as a standard tool. In recent years, even the advertisement and movie industries are using the technology to achieve special effects.

Object representation [REQU80] is the basic foundation of any solid modelling system. Until now, most systems used boundary representation and/or primitive instancing as their basic building blocks. Such schemes provide a good approximation of real solids and allow exact geometric transformations, but make boolean set operations, a necessity in solid modelling, and geometric and topological property calculations somewhat difficult. To overcome these, a spatial enumeration scheme called *octrees* [JACK80,MEAG82] was introduced, which decomposes hierarchically the three-dimensional Euclidean space (E^3) and expresses the objects residing in it in the form of ordered 8-ary trees. However, octrees often require excessive amounts of storage for their nodes and pointers even for moderately complex objects. The linear octree [GARG82] was devised to reduce the storage requirement dramatically by storing only the leaf nodes using a special encoding scheme.

This thesis takes a different look at linear octrees to take full advantage of their mathematical properties. Efficient algorithms were derived that extensively use bitwise operations. A complete set of modelling operations is also described. Chapter 2 gives a brief overview of various solid representation schemes. Chapter 3 describes the linear octree as used in this thesis, on which new operations, such as node trimming, and object generation algorithms are derived. The standard geometric transformations—translation, (arbitrary) scaling, and rotations, which make use of the new object generation algorithms, are discussed in Chapter 4. Boolean set operations—union, intersection, and difference—and rendering techniques are also included for completeness. An experimental implementation is described in Chapter 5, along with some evaluations on the solid modelling package. Chapter 6 summarizes this work, lists unsolved problems, and suggests possible extensions and future work.

Chapter 2

Object Representations

2.1 Solid Objects

The primary interest of solid modelling is manipulation of solid "physical" objects. More precisely, these objects are subsets of E^3 that are well-behaved, representable, and occupy non-zero volume; curves and surfaces are not considered valid solids in this context. In addition, boolean set operations are *regularized* so as to maintain closure. The following sections briefly outline some of the common solid representation schemes and lists their advantages and disadvantages. A thorough survey can be found in [REQU80].

2.2 Primitive Instancing

Primitive solids such as cubes, spheres, and prisms can be represented easily and exactly by just a few parameters. Most mechanical parts can be constructed by applying set operations on instances of these primitives, such as the object in Figure 2.1. Set operations can be simplified since intersections of object surfaces can be solved analytically. However, the restricted class of solids makes primitive instancing difficult to extend to a wider range of applications. Production systems do not rely solely on primitive instancing, but rather include it as a means for input.

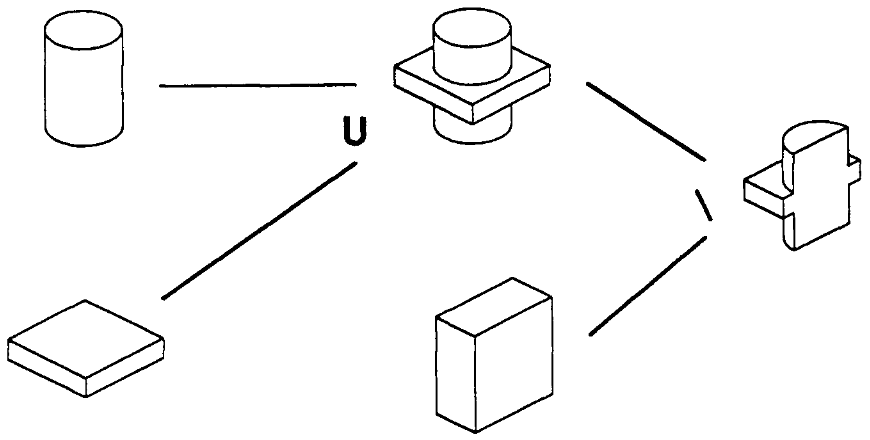


Figure 2.1: Construction of an object by set operations on primitive instances.

2.3 Boundary Representation

Boundary representation is a very versatile scheme since it can be used to represent (almost) any solid that is of interest to solid modellers. Extant systems such as *Build 2*, *Romulus* [HILL82], *PADL-2* [BROW82], and *GMSolid* [BOYS82] are all based on boundary representation. Under this scheme, an object is bounded by planar polygonal surfaces, which are defined by their bounding edges, each of which is defined by two vertices. Each distinct vertex is stored by its actual coordinate values. The surface-edge-vertex relations can be built up by "pointer" structures. Non-planar surfaces can always be approximated by a (usually) large number of planar polygons, but higher order surface patches can also be introduced to the scheme.

A major difficulty arises when performing set operations on objects in boundary representation. Determining the intersection of two objects requires solving virtually every pair of polygonal surfaces from the objects for the intersecting lines that become part of the new boundary. An improved algorithm was introduced in [MANT83] to achieve efficient boundary intersections with execution time linear in the number of faces. Nevertheless, a large amount of computation is still required compared to other geometric transformations on the same objects. Moreover, the result of set operations may not be "unique". For example, the union of the two objects in Figure 2.2 leaves superfluous edges in the middle of several planar surfaces.

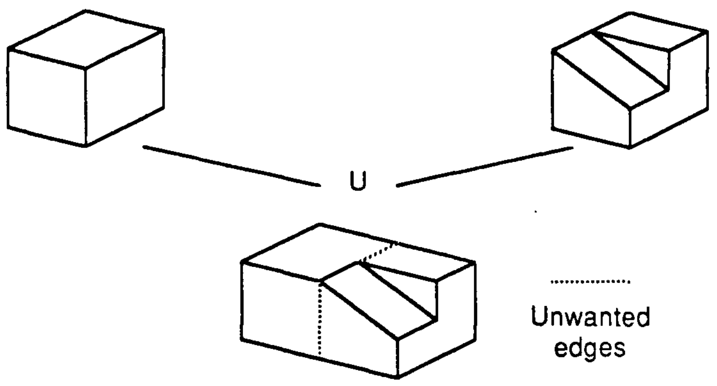


Figure 2.2: Unwanted edges produced from union of two objects.

Removing such unwanted edges may be quite time consuming.

Boundary representation may also yield "impossible" objects such as the one shown in Figure 2.3 which contains intersecting surfaces. This can be avoided if the input facility performs

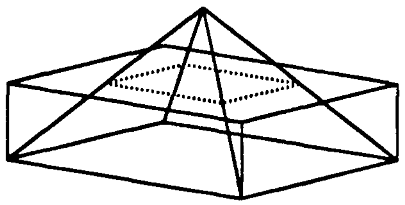


Figure 2.3: An "impossible" object from boundary representation.

some validity checks and refuses to handle such objects. Also, computation and analysis of mass properties is not trivial in boundary representation since the volume has to be deduced from bounding surfaces.

2.4 Spatial Enumeration

Spatial enumeration is a partition of a subset of E^3 into "addressable" cells, called voxels (volume elements), to form a spatial array. An object is "digitized" in this discrete space—a voxel is either part of the object or it is not. Such a scheme has many favourable mathematical properties (over boundary representation). All objects in the subspace are valid solids (no impossible objects exists; all occupy finite volume,) and their representations are unique. Computation of mass properties becomes trivial since such properties of an object as a whole can be derived from the same properties of the voxels comprising the object, which are well known because the voxels are regular cubes.

The major advantage of spatial enumeration lies in performing boolean set operations. Union, intersection, and difference of two objects can be carried out simply by comparing the individual voxels of the objects to determine their inclusion and exclusion in the resulting object. No solving of equations is required; in fact no numerical computation is needed.

There is, however, an enormous number of voxels even in a moderate decomposition of an object in the spatial array. The amount of data is directly proportional to the object's volume. The discrete nature of this scheme also makes it unfavourable for industrial designs which often require high precision. It is therefore more suitable for highly irregular objects such as tomographic scans of biological organs, which would otherwise require even more complex polyhedra using boundary representation.

2.5 Octrees

To overcome the problem of excessive storage required in spatial enumeration, a scheme call *octree* encoding was devised [JACK80,MEAG82] to take advantage of the spatial coherence of voxels in objects—adjacent voxels are grouped to form larger units to reduce the data volume.

The idea is an extension of the two-dimensional *quadtree* [HUNT79] to three dimensions. Unlike spatial enumeration, the domain (a subset of E^3) is divided into eight octants. Then each of the octants is further divided into eight smaller octants. This recursive subdivision continues until the final size of an octant reaches some desired resolution. An 8-ary tree is used to represent the subdivisions. The root is the entire domain. Then its eight *child* nodes represent the eight octants of the first division, and each of the children has eight children for further subdivisions.

An object in the octree domain is expressed by labelling the nodes of the 8-ary tree with "colours" *white, grey,* and *black.* If a node (an octant) at some level is completely filled by the object, it will be labelled black. If the node does not belong to any part of the object, it will be labelled white. Otherwise it must be partly filled by the object, and will be labelled grey. A black or white node does *not* have any *descendent* since there is no need to further subdivide the octant to determine occupancy. A grey node, on the other hand, must have (exactly) eight children for the converse reason (see Figure 2.4). Observe that all *leaves* of an octree are either

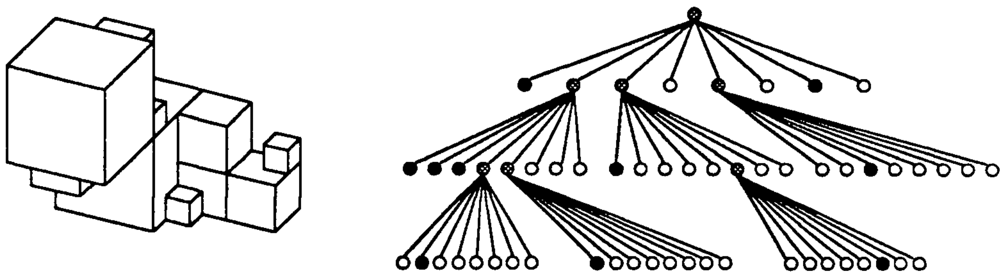


Figure 2.4: An object and its octree representation.

black or white, and a set of eight *siblings* is never all black or all white, because otherwise that would be the colour of their *parent* and they would not exist.

Octrees show promising applications in boolean set operations, interference detection (e.g. [AHUJ84]), and even ray-tracing [GLAS84]. The advantages of octrees on other applications have yet to be explored.

Chapter 3

Linear Octree Representation

3.1 The Linear Octree

In practice, the pointers and records of an octree will require a large amount of storage, since (virtually) only the leaves of the tree constitute the actual object. The internal nodes accounting for the majority of the structure do not represent "real" data. The *linear quadtree* and the *linear octree* [GARG83,GARG82] are representations which reduce a tree structure to contain only leaf nodes. Each leaf node is expressed as a string of extended base-4 and base-8 digits which identifies the *path* from which this node is reached starting at the root. Thus an octree can be represented "linearly" by a list of node numbers embedding the underlying tree structure.

The linear octree to be described here is structurally identical to Gargantini's, except for the conventions and notations used. Octants are numbered rather than using compass bearings, and their positions are reassigned. Conversion between nodes and voxels is (hopefully) clearer and easier under this numbering scheme.

3.2 Terms and Notations

Domain resolution

The resolution specifies the number of levels of an octree or linear octree object domain. A domain of resolution r contains $(2^r)^3$ voxels defined by the set of triples

$$\{(x, y, z) \mid (x, y, z) \in \mathbf{I}^3, \quad 0 \leq x, y, z \leq 2^r - 1\}.$$

(The restriction that x, y, and z are integers is obvious in context and will be omitted unless an ambiguity arises.)

Linear octree nodes

The nodes in a linear octree correspond to the leaf nodes (full or black nodes) in the 8-ary tree of the explicit octree. Since internal nodes only appear in explicit octrees, the term "node" will not be ambiguous when referring to linear octrees.

Node level

The level of a linear octree node ranges from 0 to r. The level 0 node has a size equal to the entire octree domain. A level r node is the smallest "addressable" volume having the size of a voxel.

Linear octree encoding

A linear octree node is identified by an r-digit "octal" number using 9 distinct symbols. The digits 0 to 7 are naturally adopted for the eight octants of each level as shown in Figure 3.1. The ninth symbol, F, is a *filler* for a *full* node (hence the letter F) at a level $l < r$ where fewer than r 0-to-7's are required to identify the node—the trailing $(r-l)$ digits are F's. For example, the level 2 node occupying suboctant 7 of octant 4 in a

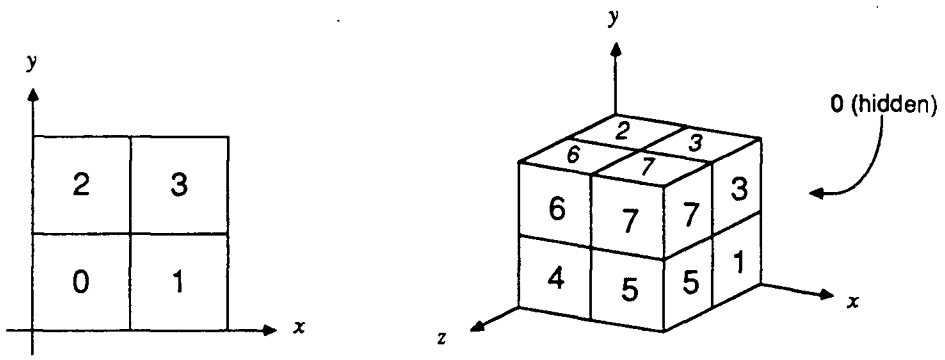
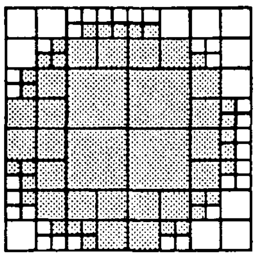


Figure 3.1: Digit assignments for linear quadtrees and linear octrees.

resolution 3 domain is encoded as **47F**. The assignments of **0** to **7** were deliberately chosen such that the three-bit binary numbers of **0** to **7** have their bit positions correspond to the x, y, and z-axis from least to most significant bit respectively. This becames an important property for conversion between nodes and voxel coordinates.

Node Ordering

All nodes in an octree-encoded object are unique and disjoint; their union comprises the object. The explicit octree implies an ordering of sibling nodes which linear octree nodes must preserve. However, while the notion of order in an octree applies to each set of sibling nodes separately, the order of nodes in a linear octree applies to all r-digit node numbers globally. The usual numerical order of **0** to **7** is used, with the addition of defining **F** greater than **7**. Figure 3.2 is a quadtree-encoded circular disk with its quadtree node numbers (digit assignment as in Figure 3.1) listed in order. Note that the most significant digits distinguishing two node numbers (from the same object) are never **F**'s, since otherwise one would be the descendent of the other. The definition **F** > **7** is critical for the node comparisons used in boolean set operations in Section 4.2.



0013	0202	103F	200F	231F	321F
0023	0203	1120	201F	2321	3220
003F	021F	1122	2021	2330	3221
0101	022F	1123	2023	2331	3300
0102	023F	12FF	203F	30FF	
0103	03FF	130F	21FF	310F	
011F	100F	1312	2210	3110	
012F	1012	132F	2211	3112	
013F	1013	1330	2213	312F	
0201	102F	1332	230F	320F	

Figure 3.2: A circular disk of diameter 15 in a domain of resolution 4 consists of 54 nodes.

Lowest and Highest Order Voxels of a Node

Each black node is a cube consisting of multiple voxels (except for a level r node which consists of a single voxel). The two corner voxels (x_0, y_0, z_0) and (x_1, y_1, z_1) completely determine the volume of space

$$\{(x,y,z) \mid x_0 \leq x \leq x_1, \quad y_0 \leq y \leq y_1, \quad z_0 \leq z \leq z_1\}$$

occupied by the node. If all the voxels in a node are expressed as level r nodes and listed in the order described above, the node representing the voxel (x_0, y_0, z_0) will be the first one in the sequence while the node for (x_1, y_1, z_1) will be the the last. Therefore these two voxels are designated the ***lowest order*** and ***highest order*** voxels respectively.

3.3 Basic Operations

3.3.1 Conversion of a node to its lowest and highest order voxel coordinates

An interesting observation is that the linear octree encoding of a voxel (its node number) is an alternative interpretation of the binary numbers of its x, y, and z-components of its spatial coordinate (x, y, z). This property allows straight forward and low cost conversion between a node and its corner (voxel) coordinates. The procedure is best illustrated by the examples in Figure 3.3.

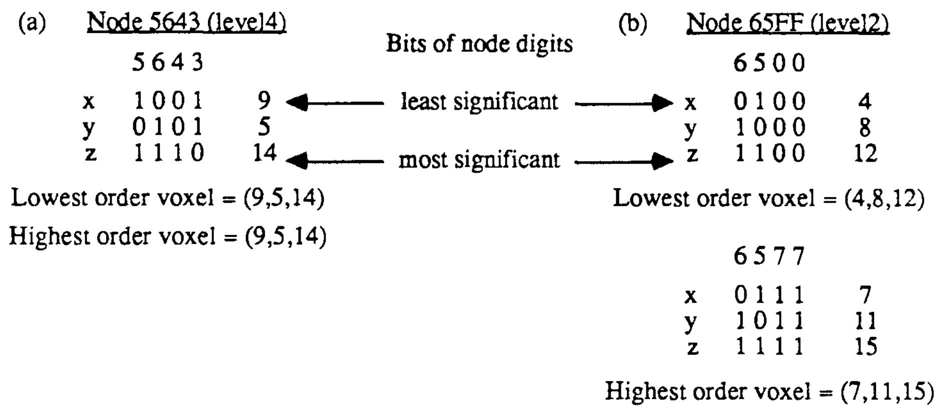


Figure 3.3: Conversion of octree nodes to their corner voxels.

3.3.2 Conversion to the largest node from its lowest order voxel coordinate

Conversion to a node from its lowest order voxel coordinate is *not* an exact inverse of the conversion operation of Section 3.3.1 because two nodes at different levels can share the same lowest order voxel. Thus, a given voxel will not generate a unique octree node unless the desired level is specified. Generating the largest (lowest level number) node will be of interest in object generation discussed in later sections. Figure 3.4 shows the inverse conversions of the nodes from Figure 3.3.

3.3.3 Sequential node generation

With the foregoing notion of node sequencing, it is desirable to enumerate all the nodes of the entire octree domain. The three sequences

(a) 000, 001, 002, ..., 007, 010, 011, ..., 775, 776, 777

(b) 0FF, 1FF, 2FF, ..., 7FF

(c) 000, 001, 002, ..., 007, 01F, 02F, ..., 07F, 1FF, 2FF, ..., 7FF

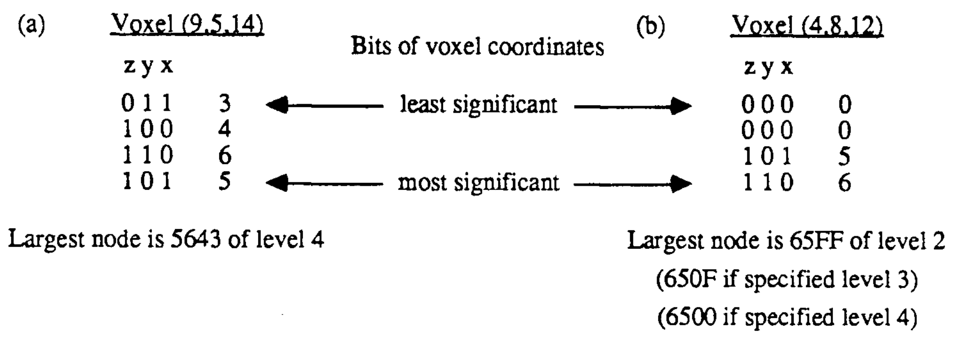

(a)
Voxel (9,5,14)
z y x
0 1 1 3
1 0 0 4
1 1 0 6
1 0 1 5
Bits of voxel coordinates
least significant
most significant
(b)
Voxel (4,8,12)
z y x
0 0 0 0
0 0 0 0
1 0 1 5
1 1 0 6
Largest node is 5643 of level 4
Largest node is 65FF of level 2
(650F if specified level 3)
(6500 if specified level 4)

Figure 3.4: Conversion to octree nodes from their lowest order voxels.

enumerate the nodes in a domain of resolution 3. But as demonstrated, there is not a unique sequence of disjoint nodes comprising the domain. The reason lies in the fact that a node such as 725 is included in 72F which is in turn included in 7FF. This non-uniqueness also introduces the ambiguity in determining what should be the *successor* of a node, say, 377—should it be 37F?, 3FF?, 400?, or 4FF?

As an essential part in object generation, a node and its successor must be disjoint, and the successor must have a maximal size, hence 4FF is *the* successor of 377. Therefore the successor of a node is defined to be the next sibling node if there is one, or the successor of its parent node if it is the last child node. The recursive property of this definition may result in a successor several levels higher than the node itself. As an extreme case, the nodes 777, 77F, 7FF, and FFF do not have a successor in the resolution 3 domain.

It is interesting to find that 'successor of' is a many-to-one relation. For example, 177, 17F, and 1FF all have the successor 2FF. Thus the notion of *predecessor* could not be easily defined since it would be a one-to-many relation. Being of no further interest, 'predecessor of' will be left undefined in this discussion.

3.3.4 Trimming against a lower bound

A node is trimmed against a given bound by *expanding* it into its children nodes and, if necessary, further expanding its children nodes until no descendent node crosses the bound. The *first* node (with respect to node ordering) in this expanded list satisfying the bound will be the result of the trimming operation.

For trimming against a lower bound, the resulting node will have its lowest order voxel coordinate no less than the given bound. The algorithm compares the **F**'s in the node with the corresponding binary digits of the lower bound and replaces some of these **F**'s by **0**'s, or **1**'s (**1** for x-bound, **2** for y-bound, **4** for z-bound) according to the following rule: From the position of the leftmost **F** to that of the rightmost 1-bit of the bound, a 0-bit converts that **F** to a **0**, and a 1-bit to a **1** (or **2** or **4** for y and z respectively). If the rightmost 1-bit locates on the left of the leftmost **F**, then the node would not be affected by the trimming. Figure 3.5 shows an example

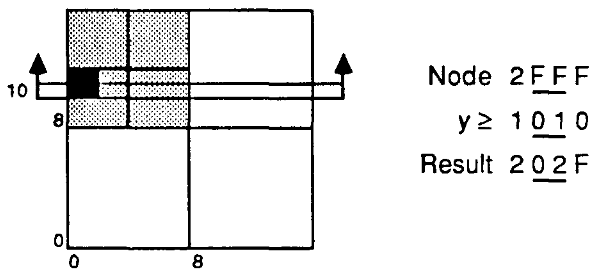
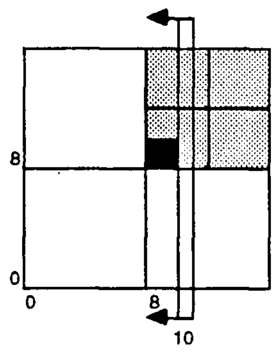


Figure 3.5: Node **2FFF** trimmed by lower bound $y = 10$.

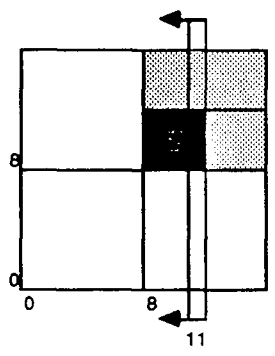
of this algorithm applied to a quadtree. For quadtrees there is, of course, no z-boundary to consider.

3.3.5 Trimming against an upper bound

Trimming against an upper bound results in a node with its highest order voxel coordinate not greater than the given bound. A similar yet significantly different rule applies: from the position of the leftmost F to that of the first 1-bit of the bound on the right of this F, change all F's to 0's. If all bits are 1 after this 1-bit, the last F should not be changed. If there is no 1-bit to the left of the leftmost F, then all F's would be changed to 0's. All other digits remain unchanged. Two examples in Figure 3.6 illustrate the rule. It should be noted that the upper



(a) Node $3\underline{FF}F$
$x \leq 1\underline{01}0$
Result $3\underline{00}F$



(b) Node $3\underline{FF}F$
$x \leq 1\underline{01}1$
Result $3\underline{0F}F$

Figure 3.6: Node 3FFF trimmed by upper bounds $x = 10$ (a), and $x = 11$ (b).

bound of the resulting node ***does not necessarily*** touch the given bound. This operation is very asymmetrical to 'trimming against a lower bound' because it only ***expands*** the node to its ***first*** descendent node that satisfies the requirement.

The two trimming algorithms described above work ***only*** if the node is known to cross the trimming bounds. Therefore a preliminary test is necessary by comparing the corner voxel coordinates of the node with the given bound. Incorrect result would occur if this rule is not obeyed.

3.3.6 Normalizing an object

The octree representation of an object is not unique if all sibling nodes at some part of the tree are black. For example, instead of having one node 52F, there are eight nodes 520, 521, ..., 527 that occupy the same space. This may result from modelling operations on objects described later. It is possible to *normalize* these nodes by sorting them in order and condensing them to a minimum number such as the circle in Figure 3.2. (Note that not all groups of four nodes forming a square make a larger node, the circle *is in fact* normalized.) Although being in unnormalized form does not affect an object's geometrical properties, it is necessary to have objects normalized in order to carry out modelling operations efficiently.

3.4 Object Generation

3.4.1 Parametrically defined objects

Many simple geometric solids such as cuboids (or parallelepipeds), spheres, prisms, and cones are defined by just a few parameters such as length, height, width, or radius. Most manufactured parts are built by combinations of these solids; and in fact many existing solid modelling systems support only this family of solids since they require little amount of storage, and modelling operations can be performed quite easily. Generating such objects in an octree domain is, however, not as easy as it seems. Boundary following algorithms such as [SAME80] generate quadtree representations of regions from chain-coded boundaries. For a parametrically defined solid, this means that one must digitize the surfaces of the solid and apply a similar surface-following algorithm to generate the octree of the enclosed space. This technique can be applied to general solids, concave or convex.[1] But in most situations, only convex primitive solids need to be considered. (At least the four geometric solids mentioned above are all convex.) Complex objects can be built by modelling operations on convex primitives. In addition, while generating nodes of an explicit tree structure in any order does not affect its properties, it is critical to generate nodes of a linear octree in some sorted order to avoid the normalization process which is dominated by the time-consuming sorting step. Facing these considerations, a simple trial and error algorithm has been devised:

1. The level 0 node is tested for inclusion inside the parametrically defined surface.
2. If the test returns "inside", then the node is part of the object and will be accepted.
3. If the node is outside, then it will be discarded.
4. Otherwise the node must be partially inside and outside. In this case, the node is expanded into its eight children and all are tested, beginning with the first one.
5. When one node is done, try its successor as defined in Section 3.3.3.

The restriction to convex objects is imposed to make the inclusion tests easier. Inclusion of all vertices of a node implies the inclusion of the entire node. (Exclusion of all vertices, however, does not imply exclusion of the node.)

3.4.2 Cuboid

A cuboid with its sides parallel to the principal axes is the simplest object to generate. It can be filled exactly by octree nodes of different levels. The two vertices (x_0, y_0, z_0) and (x_1, y_1, z_1) where $x_0 \leq x_1$, $y_0 \leq y_1$, and $z_0 \leq z_1$ completely define the cuboid.

[1]All lines joining two different points inside a convex solid lie completely inside that solid.

Assuming a domain of resolution 4, the above algorithm can be applied by initiating the node sequence with FFFF to be tested for acceptance. A better choice is to start with the largest node which has its lowest order voxel at (x_0, y_0, z_0). Immediate acceptance and rejection can be determined by a few comparisons with x_0, y_0, z_0, x_1, y_1, and z_1.

When a node crosses a boundary, it could be expanded into its eight descendents simply by changing the leading F to 0. The following nodes will be generated sequentially by finding the successor of each one, which again requires the changing of (usually) one octree digit. This is virtually an in-order traversal of the tree without using any recursion or stack—the digits in the current node are actually the information that the stack would have saved.

Incidentally, expanding and retrying a node one level at a time is not very attractive. "Trimming" a node against a given bound as described in Sections 3.3.4 and 3.3.5 will expand the node to the appropriate level in one step. As are window clipping algorithms, the node trimming here should be applied to each of the six bounds of the cuboid in a pipelined fashion.

As one can see, the sequential node generation algorithm, combined with the trimming operations, provide an efficient method to generate the normalized linear octree of any volume in the octree domain. It also becomes apparent why the resulting linear octree is normalized, since all nodes are introduced in order, and a node is always considered before its descendent.

3.4.3 Ellipsoid

The ellipsoid is of interest because it is the generalized form of the sphere, and it is also a typical solid with non-planar surfaces. The same basic algorithm can be applied to the ellipsoid, except that the trimming algorithms are no longer useful since they only apply to planes parallel to the principal axes. There remains the one-level-at-a-time expansion algorithm. A more general trimming algorithm could be applied to parametrically defined surfaces, but is likely to involve complicated equations solving for the desired level of expansion. The benefit gained may not be worth the effort.

Nevertheless, some provision can be made to improve the ellipsoid generation algorithm. Since generating the linear octree of a cuboid is relatively cheap, the smallest enclosing cuboidal volume of the ellipsoid is found and generated first. Next, the nodes in this volume are enumerated and the algorithm is applied once again to the ellipsoid, using the level-by-level expansion scheme. This suggests a two stage trimming algorithm—trim against the smallest bounding volume, then against the object's boundaries.

In practice, the two stages are combined into a trimming-pipeline. A node accepted by the first stage is immediately fed into the second stage for the inclusion test. As will be seen below, this trimming algorithm is also applicable for geometric transformations, in particular, rotation, since they can be viewed as object generation as well.

3.5 Other solids

With the established object generation algorithms, it should be straight-forward to generate convex solids such as cones and cylinders. An algorithm to convert a boundary representation to a linear octree was given in [TAMM84] using a connected components labelling technique. Thus it is possible to generate linear octrees of general solids expressed in boundary representations for most applications. In the meantime, we have enough capability to generate simple objects to show some other important features of the linear octree representation.

Chapter 4

Modelling Operations

4.1 Geometric Transformations

The basic capability of any solid modeller is to perform the common geometric transformations translation, scaling, and rotation. Because of their discrete nature, octree encodings often lead to non-numerical algorithms that require only manipulation of data structures. With linear octrees, these manipulations can be further simplified to operations on octal digits and bits. As one will expect, there are some tradeoffs. Due to the restriction to discreteness, it is difficult to perform the geometric transformations arbitrarily, particular rotations which are used often. The algorithms to be described here will apply to arbitrary translation, scaling, and rotations up to the limited resolution of the octree domain, yet at the same time will maintain the efficiency of some specific transformations characteristic to linear octrees.

4.1.1 Translation

Translation of an object within the octree domain does not affect its physical properties, but can result in a drastic change in its octree representation. This phenomenon can be observed by considering the translation of a single (large) node. A node at level l has sides of length 2^n where $n = r - l$ and r is the resolution of the domain. A displacement by multiples of 2^m where

$m \geq n$ leaves the node in one piece—simply relocated. If $m < n$, however, the translation will fracture the node into lower levels, hence smaller nodes to fill the displaced volume. For the quadtree example shown in Figure 4.1, the node **12FF** ($l = 2$, $n = r - l = 2$), when translated

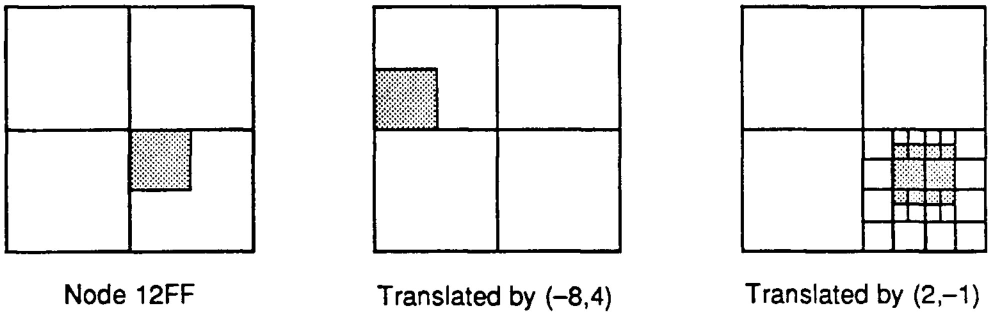


Figure 4.1: Translation of a node by different multiples of 2.

by $(-8, 4)$ results in a single node **20FF** since -8 and 4 are multiples of 2^3 and 2^2 respectively. In contrast, translation by $(2, -1)$ breaks the node into 10 smaller ones since 2 and -1 are multiples of 2^1 and 2^0 respectively (but not 2^m for larger m's).

Fujimura et al. [FUJI83] described a translation algorithm that treated these two cases separately. Here, a far simpler algorithm is used. Each node will be translated individually (which is what most other algorithms do). A node is "moved" to its *target* position simply by adding the translation vector to its lowest and highest order voxel coordinates. The two translated voxels then define a cuboid in the octree domain, whose octree will be generated by the cuboid generation algorithm described in Section 3.4.2. After all nodes are translated, the resulting nodes are sorted and condensed into a normalized linear octree. This additional step, of course, is not required for algorithms operating on explicit octrees; it is a tradeoff between the two schemes.

The question on how the algorithm distinguishes between $m \geq n$ or $m < n$ may arise.

The interesting fact is that it is not necessary to make such a distinction. Since the cuboid generation algorithm enumerates nodes in the octree domain sequentially, the target will be filled by a normalized linear octree regardless of how many nodes it is composed of. If the translation is *good* ($m \geq n$), then only one node will be generated. If it is *bad* ($m < n$), then the trimming algorithms will expand large nodes into smaller ones efficiently.

The worst situation occurs when an octree of low complexity (with a few nodes) undergoes a bad translation that breaks it into numerous small nodes. Each node will give rise to a linear octree which contains a number of nodes proportional to the surface area of the original node [MEAG82], causing an "explosion" of nodes. (Some improvement will occur after normalizing the linear octree.) For some physical objects consisting of a single component, it is preferable to determine the boundary of the entire object, translate it, and then regenerate a new octree. Surface detection algorithms would be required, however, which are beyond the scope of this thesis.

4.1.2 Scaling

Scaling of an object by 2^m on all principal directions is particularly simple for octrees. For $m > 0$, the top m levels of the octree will be deleted making one of the level m subtrees the new root. (Which subtree to choose depends on the enlarged octant which becomes the new domain.) For $m < 0$, $|m|$ new levels will be introduced to the top of the tree, pushing the old root downward to level $|m|$; branches below level $|m|$ in this new octree will be pruned. In linear octrees, these operations reduce to shifting of octree digits. Each node number will be shifted left m places with F's replacing the least significant m digits for $m > 0$, and shifted right $|m|$ digits for $m < 0$.

Arbitrary scaling, however, is the real problem. The same if not a worse fragmentation problem as for translation exists. Scaling a node by a factor other than a power of 2 will result in generating a large number of nodes as exemplified in Figure 4.2. A simple and uniform

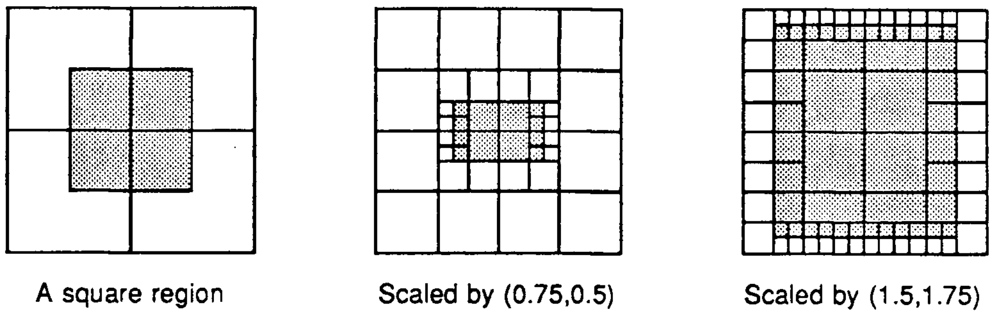


Figure 4.2: Scaling a node by different factors.

approach will be to scale each node of the object exactly (probably using real arithmetic) and regenerate the octree in the resulting cuboid, similar as translation is handled. Different scaling factors will be treated identically, but the advantage of the 2^m-case will remain in effect.

It is worth noting that a scaling factor of less than 1 will cause a loss of information, since re-sampling occurs over a smaller grid at the same resolution. A node could be scaled to nil, resulting in a degenerated cuboid. In this case, the cuboid generation algorithm will simply return a null linear octree—the empty node set.

A point of reference must be defined for the scaling operation. In $\mathbf{R}^3$ this is the origin. In an octree domain, the zero point would not be a practical choice since there are no negative axes, and a translation of objects to this point will be truncated. As no other point presents itself as a natural choice, it was decided, *ad hoc*, to choose the center of an octree domain, i.e. the common vertex shared by the eight level-1 nodes, to be the invariant point.

4.1.3 Rotations

The center of rotation is taken to be the center of an octree domain, as for scaling, and for the same reasons. Rotations about the x, y, and z-axes[1] are identical in all respects. Without loss of generality, it is sufficient to consider only rotation about the z-axis, and hence the following discussion will be restricted to two-dimensional rotation on quadtrees.

Rotations by multiples of 90° will be considered *good*. All other angles will cause severe fragmentation, as shown in Figure 4.3. There is no intermediate "badness"—10° is just as bad

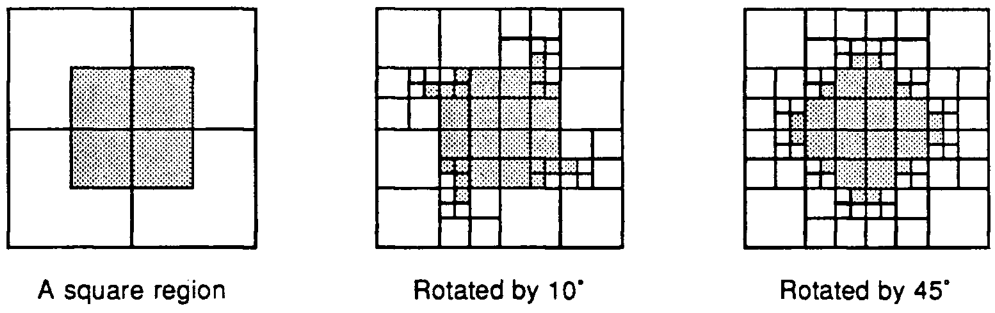


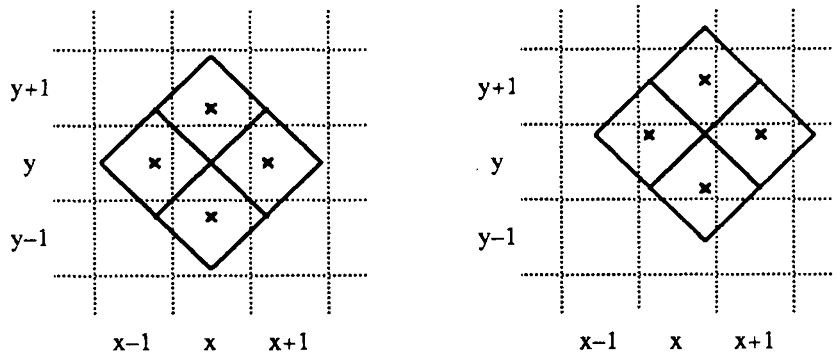
Figure 4.3: Severe fragmentation results for almost any angle of rotation.

as 45°. For rotation by 90°, it is only necessary to permute the eight children of each node in the explicit octree [MEAG82], or apply a permutation function on all octal digits in each node of the linear octree [GARG82].

A rotation by an arbitrary angle θ can always be normalized such that $0° \leq \theta < 360°$, and then decomposed into a multiple of 90° plus ϕ where $0° \leq \phi < 90°$. Each (linear octree) node can be pre-rotated through 90°'s by permutation as given in [GARG82], then the rotation of the remaining angle ϕ can be carried out.

[1]Actually axes parallel to the three principal axes origined at the center of the octree domain.

Each target node is bound by four rotated planes from the original node and by two unrotated ones which are perpendicular to the axis of rotation (z-axis). Projecting the node onto the xy-plane will result in a 2-D rotation of four straight lines forming a square. To regenerate a quadtree for this rotated node (square), each pixel can be mapped onto a new one inside the target. By replacing each pixel by the point at its center, we could rotate this point and declare the pixel on which it lies the result of the mapping. This *forward mapping*, however, is not 1-1. It could map two pixels onto one or leave some pixels in the target unmapped as shown in Figure 4.4. Alternatively, each scanline can be rotated as a group rather than as individual



(x,y) not mapped by any pixel. (x,y) being mapped twice.

Figure 4.4: Holes and duplicates resulting from forward mapping.

pixels. Braccini et al. [BRAC80] studied an optimal line drawing algorithm and encountered a similar problem of holes between adjacent scanlines. They proposed a suboptimal algorithm to maintain continuity but some points would be "plotted" twice. Rotating the entire node would not result in much improvement since the same problems would arise between adjacent nodes. The implication of these problems on octrees is rather serious. Unmapped holes change the topology of an object and greatly complicate its octree structure. The possibility of multiple-mapped voxels requires that a test for existence must be carried out for each new voxel before it is added to the result—a very expensive operation.

A *backward mapping* can be used to overcome the above problems. The target is represented by the four rotated boundaries of the node. Pixels are re-sampled by determining whether the center of each lies inside the target. Each point (pixel) can belong to at most one target because nodes are disjoint, and no holes will be created since adjacent nodes transform into continuous target boundaries (Figure 4.5). Similar arguments follow if this mapping is applied to octree

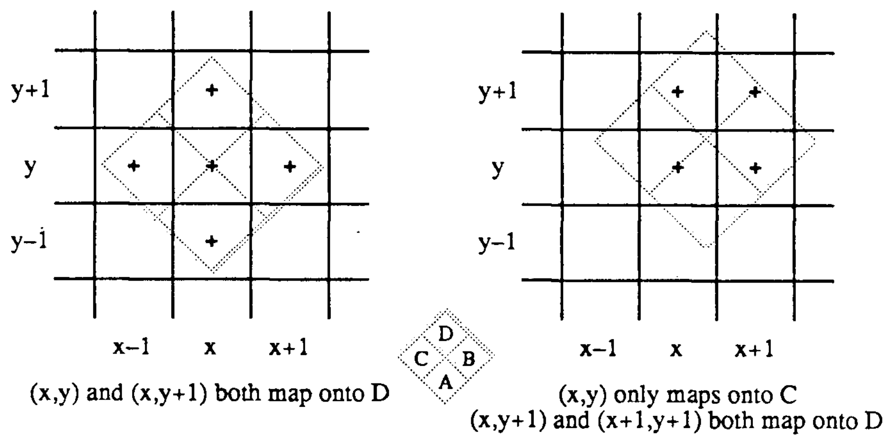

y+1
y
y-1
x-1
x
x+1
(x,y) and (x,y+1) both map onto D
D
C
B
A
y+1
y
y-1
x-1
x
x+1
(x,y) only maps onto C
(x,y+1) and (x+1,y+1) both map onto D

Figure 4.5: Re-sampling of pixels in target under backward mapping.

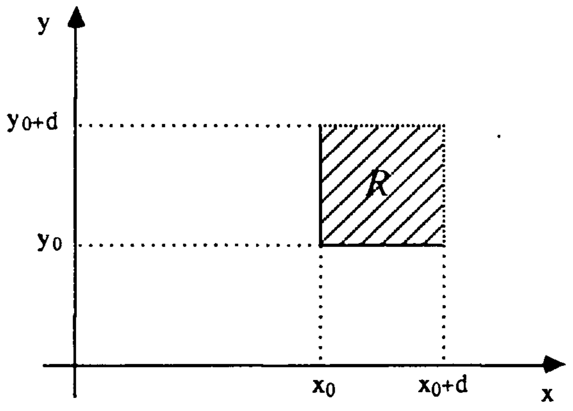
nodes by considering their corner voxels. Hence the same object generation algorithm as for ellipsoids (Section 3.4.3) can be employed.

The lowest and highest order pixels (x_0, y_0) and (x_1, y_1) of a node determine the boundary equations

$$\begin{aligned} x &= x_0 \\ y &= y_0 \\ x &= x_1 + 1 = x_0 + d \\ y &= y_1 + 1 = y_0 + d \end{aligned}$$

where $d = 2^{r-l}$ with $r =$ domain resolution and $l =$ node level enclosing the region R occupied by the node as shown in Figure 4.6. Rotating a point (u, v) through ϕ gives (u', v') where

$$\begin{aligned} u' &= u \cos\phi - v \sin\phi, \\ v' &= u \sin\phi + v \cos\phi. \end{aligned}$$



$$R = \{(x, y) \mid x_0 \le x < x_0 + d, \quad y_0 \le y < y_0 + d\}$$

Figure 4.6: Region occupied by a node of size $d \times d$.

Two lines with slopes $\tan\phi$ and $\dfrac{-1}{\tan\phi}$ passing through (u', v') have equations

$$\begin{array}{lrclrcl} & (y-v')-(x-u')\tan\phi & = & 0 \quad \text{or} & (y-v')\cos\phi-(x-u')\sin\phi & = & 0 \\ \text{and} & (y-v')\tan\phi+(x-u') & = & 0 \quad \text{or} & (y-v')\sin\phi+(x-u')\cos\phi & = & 0 \end{array}$$

respectively, which, when expressed in terms of u and v, simplify to

$$\begin{array}{lrclcl} & y\,\cos\phi - x\,\sin\phi & = & u\,\sin\phi\,\cos\phi + v\,\cos^2\phi - u\,\cos\phi\,\sin\phi + v\,\sin^2\phi & = & v \\ \text{and} & y\,\sin\phi + x\,\cos\phi & = & u\,\sin^2\phi + v\,\cos\phi\,\sin\phi + u\,\cos^2\phi - v\,\sin\phi\,\cos\phi & = & u. \end{array}$$

Substituting (u, v) by (x_0, y_0), $(x_o + d, y_0)$, and $(x_0, y_0 + d)$, the rotated region R' will be

$$R' = \{(x, y) \mid x_0 \le y\,\sin\phi + x\,\cos\phi < x_0 + d, \quad y_0 \le y\,\cos\phi - x\,\sin\phi < y_0 + d\}.$$

Inclusion of a point (x, y) in R' can be tested by substituting it into the four inequalities

$$y\,\sin\phi + x\,\cos\phi \ge x_0 \tag{4.1}$$

$$y\,\sin\phi + x\,\cos\phi < x_0 + d \tag{4.2}$$

$$y\,\cos\phi - x\,\sin\phi \ge x_0 \tag{4.3}$$

$$y\,\cos\phi - x\,\sin\phi < x_0 + d. \tag{4.4}$$

For each target, d will be fixed; so the right hand sides can be kept constant, leaving four multiplications per inclusion test. Dividing Inequalities 4.1 to 4.4 by $\cos\phi$ when $0 \le \phi < 45°$ and $\sin\phi$ when $45° \le \phi < 90°$ results in two different groups of tests which can reduce the number of multiplications by half.

A node may be expanded to one level lower during the inclusion test. Since octrees have fixed size nodes, the highest order voxel coordinates of the expanded node can be computed from those of the lowest order voxel by simple addition, which means only two multiplications are required to test each enumerated node from the smallest bounding rectangle of the target. As stated earlier, the center of rotation is actually the center of the octree domain, which means all voxel coordinates must be translated by $(-D, -D, -D)$ where $D = 2^{r-1}$ before applying the inclusion test. This pre-translation, when incorporated into the equations above, however, will not affect the complexity of the test; it still needs only two multiplications to test an enumerated node.

4.1.4 Compound Transformations

A reasonable extension of simple transformations is to allow combinations of any number of them to form a single transformation. Compound transformations on points in E^3 can be easily handled using homogeneous coordinates and 4×4 transformation matrices. Such manipulations on octrees, however, are not as straight forward as matrix multiplications. Doing each transformation separately will not give satisfactory result since "rounding errors" will build up at each step due to re-sampling. Being relatively "exact", combinations of translations and scaling operations can be carried out by transforming each node using the ordinary homogeneous coordinate method, and then regenerate the linear octree for the target volume. Rotations are exceptionally difficult. A compound rotation about different axes will rotate all six faces of a node, which means a new set of inclusion tests must be designed involving equations of planes rather than lines in the xy-plane.

4.2 Boolean Set Operations

Perhaps the greatest advantage of octrees occurs when performing boolean set operations on octree-encoded objects. The operations union, intersection, and difference must be regularized according to Tilove [TILO80] such that no "invalid" object will result. This is a particularly difficult problem for boundary representations because it requires extra attention to maintain regularity. Algorithms such as Franklin's [FRAN82] classify all edge segments (original and generated) to be included as results of different operations. While such an algorithm computes all set operations at once, it is rare in practice that more than one operation is required for the same objects. Octrees provide regularity naturally without additional effort, since "invalid" objects such as faces and line segments are simply not representable. (For this reason, the regularized union ($\cup^*$), intersection ($\cap^*$), and difference ($\setminus^*$) will be written as '$\cup$', '$\cap$', and '$\setminus$' in the following sections.[2])

The octree approach to set operations is basically a pairwise comparison of two black nodes, one from each object. As nodes are spatially sorted, it is not necessary to compare every pair of black nodes. An in-order traversal of the two trees in parallel will be sufficient to carry out a merge sort-like algorithm to determine which black node to include in the result of the set operation. For linear octrees, the merging algorithm is further facilitated by the fact that the objects are already in the form of two sorted lists of nodes, making the tree traversal step trivial. The major task lies in a node comparison operation which depends on the particular set operation. Each of the union, intersection, and difference operation is sufficiently different to be implemented separately as a "stand alone" algorithm.

4.2.1 Union

Two linear octree nodes N_1 and N_2 are either disjoint, or equal, or one is the descendent of the other. That is, either $N_1 \cap N_2 = \emptyset$, or $N_1 \subseteq N_2$, or $N_1 \supset N_2$ if a node is considered as a set

[2]Some authors use '$-^*$' and '$-$' for the difference operation.

of points. It follows that

$$N_1 \cup N_2 = \begin{cases} N_2 & \text{if } N_1 \subseteq N_2 \\ N_1 & \text{if } N_1 \supset N_2 \\ N_1 + N_2 & \text{otherwise} \end{cases}$$

where $N_1 + N_2$ denotes $N_1 \cup N_2$ given that $N_1 \cap N_2 = \emptyset$. Thus the union of two objects S_1 and S_2 can be determined locally by comparing the nodes $N_1 \in S_1$ and $N_2 \in S_2$ as follows:

1. If $N_1 \subseteq N_2$, discard N_1.
2. Else if $N_1 \supset N_2$, discard N_2.
3. Else if $N_1 \prec N_2$, add N_1 to the result.
4. Else if $N_1 \succ N_2$, add N_2 to the result.

The symbols '$\prec$' and '$\succ$' apply to the ordering of node numbers as defined in Section 3.2. N_1 and N_2 will be enumerated sequentially from S_1 and S_2 respectively. Whenever a node is discarded or accepted, a new one will be retrieved from the corresponding object. When one object is exhausted, all the nodes from the other object will be added to the result. Since it is possible to group nodes from the two objects to form larger nodes, normalization should be carried out by condensing the collected nodes to produce the final result. Sorting is not required since the merging algorithm ensures resultant nodes to be sorted. This "clean up" step can be performed in parallel while nodes are being accepted.

The four cases can be determined by a digit-by-digit comparison between two node numbers: scanning from the most significant digits, locate the first different pair. If one digit is F, the node containing this digit has the other node as its descendent. Otherwise, the numerical order of the digits reflects the order of the nodes themselves. Two node numbers are equal if and only if all their digits are equal.

4.2.2 Intersection

Closely resembling the union operation, the intersection of N_1 and N_2 is defined as

$$N_1 \cap N_2 = \begin{cases} N_1 & \text{if } N_1 \subseteq N_2 \\ N_2 & \text{if } N_1 \supset N_2 \\ \emptyset & \text{otherwise} \end{cases}$$

and the pairwise node comparison becomes:

1. If $N_1 \subseteq N_2$, add N_1 to the result.
2. Else if $N_1 \supset N_2$, add N_2 to the result.
3. Else if $N_1 \prec N_2$, discard N_1.
4. Else if $N_1 \succ N_2$, discard N_2.

Nodes are retrieved as in union operation. However, when one object is exhausted, the remaining nodes of the other object will be discarded since they will not be part of the exhausted one. The major difference in intersection is that no normalization is needed because the resulting object cannot contain more nodes then either of the original objects, assuming that they are normalized in the first place.

4.2.3 Difference

The difference $N_1 \setminus N_2$ is defined as the set of all points belonging to N_1 but not N_2. More precisely,

$$N_1 \setminus N_2 = \begin{cases} \emptyset & \text{if } N_1 \subseteq N_2 \\ \text{expand}(N_1, N_2) - N_2 & \text{if } N_1 \supset N_2 \\ N_1 & \text{otherwise} \end{cases}$$

where expand(N_1, N_2) gives a minimal list of nodes expanded from N_1 which contains N_2. As the example in Figure 4.7 shows, this expansion is always possible since $N_1 \supset N_2$ implies

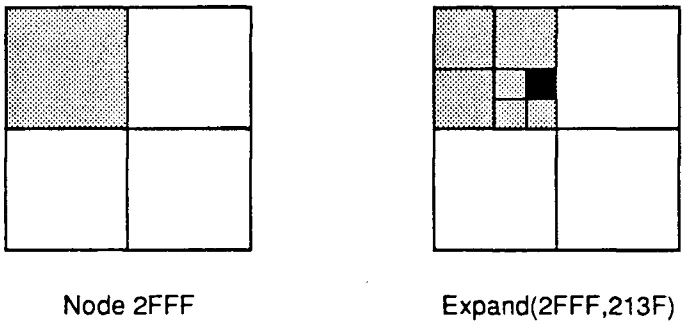


Figure 4.7: Partition of **2FFF** under **expand(2FFF, 213F)**.

that N_2 is a descendent of N_1. The '$-$' operation "removes" N_2 from the expanded list. The algorithm for the difference operation is significantly different from union and intersection as shown below:

1. If $N_1 \subseteq N_2$, discard N_1.
2. Else if $N_1 \supset N_2$, expand N_1 one level down, make N_1 the first node of the expanded list, and return to step 1.
3. Else if $N_1 \prec N_2$, add N_1 to the result.
4. Else if $N_1 \succ N_2$, discard N_2.

The expansion algorithm is identical to the one used in ellipsoid generation described in Section 3.4. Bauer [BAUE85] suggested a queue to store all the nodes expanded from a large one. Here, the expanded nodes are enumerated by the successor function defined in Section 3.3.3. A flag will be set while expansion is underway to indicate that no actual retrieval should take place for the object S_1. Instead, nodes will be "retrieved" by finding the successor of the discarded or accepted node. Successor generation will terminate when all the expanded nodes are listed, i.e.

when the successor of the original N_1 is reached. The flag will then be reset and actual retrieval will resume. No additional information is required to keep track of the expanded nodes except a flag and the terminating node.

When S_1 is exhausted, all nodes in S_2 will be discarded since they will not contain points belonging to S_1. If S_2 is exhausted first, all nodes in S_1 will be added to the result since they will not interfere with S_2. Again, no normalization is needed as the difference cannot have more nodes than S_1.

Chapter 5

Implementation and Results

5.1 A Solid Modelling Package

A solid modelling package based on linear octrees has been implemented in C under UNIX[1] on a VAX 11/750.[2] It consists of a function library and a set of drivers available as UNIX commands. The function library contains modules that manage the internal data structure, conversions between different formats of data, basic operations on linear octree nodes, input/output routines, etc., and the six independent modelling operations (see Figure 5.1). The set of driver programs provides high-level object generation and modelling operations. All data are transferred using the standard UNIX input/output conventions. Interface with the internal octree structure is completely transparent to the user. The flexibility of the function library allows building of special purpose programs with little effort.

[1]UNIX is a trademark of AT&T Bell Laboratories.

[2]VAX is a trademark of Digital Equipment Corporation.

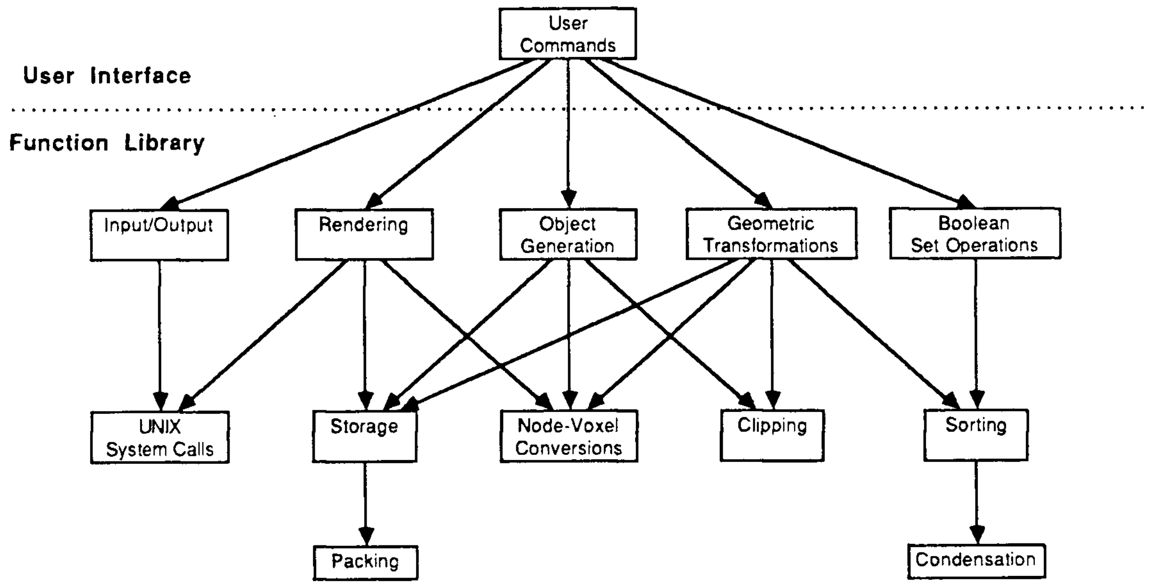


Figure 5.1: An overview of the solid modelling package.

5.2 Representation of Linear Octrees

5.2.1 Internal Representation

One of the original motivations to use linear octrees is to significantly reduce the storage requirement of the pointer-structured explicit octrees. Recent studies by Lauzon [LAUZ85] show that it is possible to condense quadtrees even further by using a run-encoding scheme; a similar method should apply to octrees. However, the more compact octrees are stored, the more difficult it is to unveil the encoded object, decreasing the efficiency of accessing the nodes of the tree. The spectrum ranges from high speed access on one end to compact storage on the other.

In the current implementation, an active linear octree node will have its octal digits stored individually in addressable units (**char**'s or **int**'s in C) as an array since the conversion and trimming algorithms require octal digits to be decomposed further into bit fields. For an actual object with thousands of nodes, however, such underutilization of memory will not be sensible—only 4 bits (for 9 symbols) in one byte or word (32 bits) are used. A compression (or packing) scheme is employed to store nodes as base-9 numbers, treating the digit F as 8. The unsigned integer available in C with 32 bits on a VAX is capable of storing $\lfloor \log_9 2^{32} \rfloor = 10$ base-9 digits in one word, which gives an octree domain with a maximum of $(2^{10})^3 = 1024^3$ voxels, adequate for high-resolution objects. More importantly, this packed format allows nodes to be compared as unsigned integers rather than numerical strings and greatly improves the time-consuming sorting process. For simplicity, it was decided to store all nodes this way in **unsigned**'s for any domain resolution up to 10. Higher resolutions are not currently supported.

Lookup tables are used to avoid multiplication in base-9 conversions. A resolution r node number can be packed into one word with r additions, one for each digit. Unpacking a node takes $r\lceil \log_2 9 \rceil = 4r$ additions since each digit can be determined by testing the 8, 4, 2, and 1 multiples of a power of 9. The packed nodes are stored in dynamically allocated array blocks each containing a maximum of 125 nodes (this number was chosen arbitrarily; some particular value may optimize memory allocation). A linear octree encoded object is represented by the data structure shown in Figure 5.2. It is possible to add more attributes such as colour and/or composition as additional fields to an object. The linear octree only describes the topology of the object.

5.2.2 External Representation

An object can be stored externally in files or can be *piped* to another command for further processing. UNIX supports stream-oriented input/output; thus an object has to be transformed into a string of bytes for external storage. The following format is designed for this purpose: the

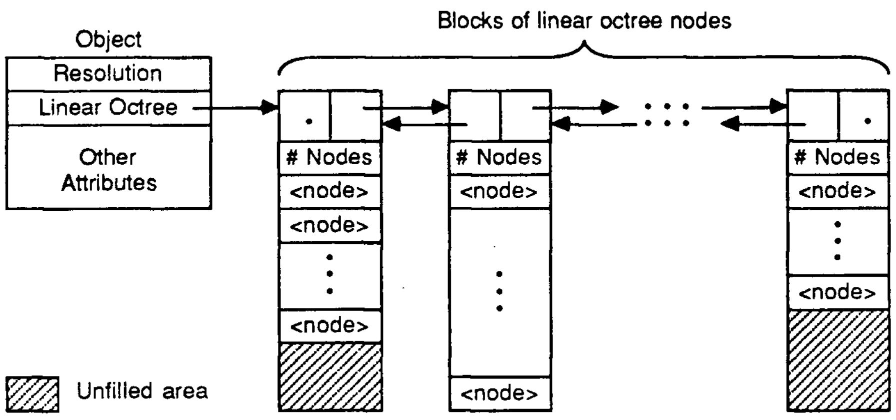


Figure 5.2: Data structure of an object.

first byte will contain the resolution of the object; to be followed by sets of four bytes (32-bits), each representing an unsigned base-9 octree node number. Within a 4-byte field, the format of the unsigned number is not defined, as it depends on the specific input/output operation used to produce it.

The packed format is not intended to be used for high level access. If the user wants to input octree nodes manually, he can prepare a file with the first line stating the resolution, followed by one linear octree node number on each line, and convert this file to the packed format for further processing. Details are covered in Appendix A.

5.2.3 Input/Output

All object manipulation functions operate only on the internal linear octree format; therefore any object in the packed external format must be "read" in and transformed to the internal data structure described above. Results of these operations can be output in packed external format. The input/output module is the only means for this transformation.

5.2.4 Node Retrieval

Nodes in the internal format can be randomly accessed. Sequential node retrieval and insertion is provided by using a "pointer" (named *indicator* in the code) independently of the linear octree structure to indicate the next position, similar to sequential file read/write. Random node deletion is not supported since there is no immediate need. Each function accessing this structure requires passing the indicator as an extra parameter. A retrieved node is considered active and is always unpacked into an array of digits, with element 0 the most significant digit of the node. The packing scheme is completely hidden from the conversion and trimming algorithms.

5.2.5 Normalization

The data structure only acts as internal storage; it does not impose any ordering of the nodes—it simply stores nodes according to the given indicator. Sorting and condensation must be executed explicitly by separate functions in order to keep a linear octree normalized. A heapsort is implemented to work on the data structure.

The condensation algorithm operates directly on the packed format (i.e., without unpacking a node into octal digits), using the fact that two consecutive nodes at level l differ by 9^{r-l}. An array of r integers is used for stacking consecutive nodes before releasing the condensed node. The condensation module can be used for condensing only one linear octree at a time since external variables are used to improve efficiency.

5.3 Operations on linear octree nodes

Two modules, node-pixel conversion and trimming, form the basis of the linear octree package. Both modules implement the algorithms exactly as described in Section 3.3. Bitwise operations of C are used extensively, hence ideally such low level algorithms should be implemented in assembler or even hardware.

5.4 Modelling Operations

5.4.1 Translation

Translation, scaling, and rotations accept **float** (real) arguments to maintain consistency since arbitrary scaling and rotations are allowed. The octree domain can be viewed as

$$\{(x,y,z) \mid (x,y,z) \in \mathbf{R}^3, \quad 0 \leq x,y,z \leq 2^r - 1\}.$$

Results of transformations are trimmed against this volume.

For translation, the displacement vector components are rounded to the nearest integers before adding to each node of the object. As noted previously, fragmentation caused by a bad translation will seriously affect performance. Each component of the displacement vector contributes to the problem. Thus it is reasonable to determine the badness of each component and handle them in some order so as to avoid fragmentation. Performance does not depend solely on the components. Table 5.1 indicates the relations of good and bad objects (low complexity),

	Component	
Object	Good	Bad
Good	Good	Bad
Bad	Bad	Good or Bad

Table 5.1: Badness of objects vs. badness of translation components.

translation components, and results (complexity, hence performance). A bad object might be the result of a bad translation in the first place and could therefore be "recovered" to a good one. In general, however, it is not possible to know the complexity of an object; after all, the modelling operations do not depend on that information.

Based on the translation components, there is a better chance to get a good object by working with a good component first. Experiments show that the cutoff point of good and bad is at multiples of 4—translation components of multiples of 1 and 2 are far worse than those of 4, 8, etc., the latter ones tend to perform equally well in the resolution 10 (and less) domain. In fact, the performance of the 1 and 2-cases degrade so sharply from the good ones that they dominate the overall translation. It seems reasonable to handle these bad components as separate translations according to the following rule: translate each bad component separately, but good components can be grouped together and "piggy-backed" on one of the bad ones. This rule is implemented in the **translate** command at the user level while the library version is kept general.

5.4.2 Scaling

Arbitrary scaling requires real multiplications. Rather than multiplying each node by the scaling factors, a lookup table is pre-computed to map each voxel coordinate onto its scaled (integer) value. For a resolution 10 domain, 512 **short int**'s are needed for each of the x, y, and z scaling factors. The other 512 units are the negative side with respect to the center of scaling and can be computed from the same tables. Initializing these tables takes approximately 0.1 CPU seconds, negligible compared to octree regeneration, although extra memory is needed. Negative scaling factors can not be handled with this table, since they would reverse the roles of lowest and highest order voxels of each node.

5.4.3 Rotations

Rotations about the x, y, and z-axes are identical except for some xyz-permutations on line equations, but they are kept separate for simplicity and efficiency. Again, lookup tables are set up for $2^l t$ where $0 \leq l \leq r$ and t being the six trigonometric functions $\sin\phi$, $\cos\phi$, $\tan\phi$, $\csc\phi$, $\sec\phi$, and $\cot\phi$, since they are fixed for each rotation. The permutation functions for 90° rotations are defined in static lookup tables. Many computations are exactly identical for all three rotations, and are therefore grouped into a "main" function, which calls a specific rotation depending on the given x, y, or z code.

Rotations are the only operations that require real comparisons since the target is bound by line equations. Particular attention must be paid to ensure the accuracy of these line equations; a slight rounding error can cause mis-mapping of voxels that leads to duplicated nodes which the condensation module cannot handle. To maintain high accuracy, all lookup tables are declared `double` and all computations are carried out in `double` as a characteristic of C, but all values are rounded to `float` (single precision) before comparisons. Slight errors will then be "smoothed out" when rounded. Tiny values will be rounded to 0 by adding and then subtracting a larger quantity from it.

5.4.4 Normalization

All geometric transformations require normalization of their result. It is not sensible to sort all the resulting nodes in one step, particularly after a bad transformation where up to ten times the number of nodes in the original object would be generated and most of them can be grouped into larger nodes. Thus partial sorting is carried out on every 1000 nodes generated (some manageable amount), then merged and condensed into the node list of the final result. Since nodes in a linear octree are processed sequentially, they tend to operate regionally within an octant at some level, which means that a partial list is more likely to be merged into a localized segment in the final list. This observation prompts for a fast merging that skips blocks of nodes in the final list and locates the block where actual merging occurs. Complete condensation is thus not guaranteed, therefore a slower node-by-node merging must be carried out when all nodes are collected. A flag is used to select the mode of the merging function. Because condensation occurs continuously, the memory allocation can be kept close to minimal.

5.4.5 Boolean Set Operations

The union, intersection, and difference operations are structurally very similar. Nodes are retrieved simultaneously from two input objects for comparisons. Two symmetrical loops control the retrieval of one object while anchoring a node of the other. When a retrieved node number is larger than that of the anchored node, processing will be switched to the other loop where the roles of the two objects are exchanged. An infinite loop encloses these two control loops to achieve the switching. Condensation is performed in one step on the result of a union since the worst case does not have more nodes than the total of the original objects.

5.5 Auxiliary Library Functions

5.5.1 Object Generation

Object generating functions should be considered as an input facility of the solid modelling package and not part of the standard library. New object generators can be added if necessary to enhance input using the basic operations such as trimming and condensation. Only the cuboid and ellipsoid generators are currently implemented. They accept integer parameters specifying the lowest order voxel coordinates and the x, y, and z-dimensions of the object. Real comparisons are used for ellipsoid generation to determine node inclusion.

5.5.2 Rendering

Similarly, rendering is an output facility of the package. Hidden surface removal is straight forward with octrees since nodes are spatially sorted. Two approaches exist [FRIE85]: the back-to-front method displays nodes from farthest to nearest with respect to the view point. This can be easily achieved by displaying the far octants before the near ones recursively for each level of the octree [DOCT81], which means a pre-order traversal of the tree where the order is defined according to the octant of the view point. The front-to-back method displays the near part of the octree first and proceeds towards the far side. Nodes obscured by previous displays will be ignored and not output. Although complex, this method will be more efficient since traversal of some octants can be bypassed if known to be hidden.

For this thesis, rendering was not studied in depth, although a primitive back-to-front algorithm is implemented for linear octrees. It re-sorts the nodes in a linear octree according to the viewing octant by applying a permutation on each octal digit, sorts them, and then applies a reverse permutation to restore them back to the original nodes. Quicksort is used since the nodes appear quite randomly after the first permutation. (But no sorting should be carried out if the traversal leaves the order unchanged or in exact reverse order, which are the worst cases for quicksort.) The visible vertices of all nodes are then output to be displayed on a particular device. An object can currently be "drawn" on an image file to be viewed on the Jupiter 7 graphics terminal under MTS[3] following a fairly tedious procedure described in Appendix A—it is barely usable.

[3]Michigan Terminal System of UBC Computing Center.

5.6 User Commands

All input/output, modelling operations, and rendering functions are available as UNIX commands. Parameters of these functions are supplied as command-line arguments on UNIX and objects are read and written via **stdin** and **stdout**, thus output from one operation can pipe into another as input. For example, to generate a $30 \times 20 \times 50$ cuboid at the center of a resolution 10 domain rotated $30°$ about the y-axis, to be stored in file **simple_object**, issue the command

```
cuboid 497 502 487 30 20 50 10 | rotate y 30 > simple_object
```

from the shell. Only a few primitive error checks are implemented at this time to prevent the package from crashing by invalid inputs. All error messages (even though not too informative) are produced by the user program on **stderr**; the library functions never print error messages but return error codes instead.

5.7 Results and Evaluation

Formal proofs of algorithm efficiencies are not provided in this thesis, although some evaluations were performed regarding execution times and storage requirements. Execution times are the 'user' times obtained by the UNIX command **/bin/time** which shows the (VAX 11/750) CPU time (in seconds) used by the user programs, excluding system calls. Storage requirements (in bytes) for both internal and external formats are about 4 times the number of nodes in the linear octree since each node is packed into one word.

Table 5.2 shows that the time required to generate the linear octree of a cuboidal volume is basically linear to the number of nodes generated for the good cases, and is proportional to the intermediate nodes created in the bad cases due to fragmentation; these effects are not shown in the tables since their linear octrees are normalized. Slight increases are observed when the same number of nodes are generated at higher order regions of an octree domain or at higher resolutions, but do not seem to dominate the overall times. Ellipsoids involve real arithmetic and are thus slower.

Times for translation, scaling, and rotations are given in Tables 5.3 to 5.5. Two resolution 10 objects, a cube with lowest-order corner at $(480, 480, 480)$ and extending 64 units towards all three positive axes, and a sphere enclosed in the same volume, are used for the tests. Note that the identity transformations are not implemented as no-ops. In all cases, the times directly reflect the fragmentation of nodes due to bad transformations as explained in Section 4.1.

Solid	x	y	z	Δx	Δy	Δz	Resolution	Seconds	Nodes
Cuboid	0	0	0	16	16	16	4	0.0	1
Cuboid	0	0	0	16	16	15	4	0.2	340
Cuboid	0	0	0	16	15	15	4	0.5	590
Cuboid	0	0	0	15	15	15	4	0.7	778
Cuboid	0	0	0	15	15	15	6	0.8	778
Cuboid	0	0	0	15	15	15	9	0.9	778
Cuboid	0	0	0	15	15	15	10	0.9	778
Cuboid	1	1	1	15	15	15	4	0.5	778
Cuboid	0	0	0	64	64	64	10	0.0	1
Cuboid	0	0	0	64	64	63	10	6.5	5460
Cuboid	0	0	0	64	63	63	10	13.0	10542
Cuboid	0	0	0	63	63	63	10	18.4	15288
Cuboid	1	1	1	63	63	63	10	15.8	15288
Cuboid	479	479	479	63	63	63	10	17.8	15288
Cuboid	960	960	960	63	63	63	10	18.3	15288
Ellipsoid	0	0	0	64	64	64	10	33.1	12496
Ellipsoid	0	0	0	63	63	63	10	50.1	11987
Ellipsoid	480	480	480	64	64	64	10	33.0	12496
Ellipsoid	480	480	480	64	63	64	10	38.4	12048
Ellipsoid	480	480	480	63	64	63	10	45.1	12244
Ellipsoid	480	480	480	63	63	63	10	49.9	11987

Table 5.2: Execution times for object generations.

			Cube-64		Sphere-64	
t_x	t_y	t_z	Seconds	Nodes	Seconds	Nodes
0	0	0	0.0	8	43.4	12496
128	128	128	0.0	8	43.5	12496
64	64	64	0.0	8	44.6	12496
32	32	32	0.0	1	42.8	12496
16	16	16	0.4	57	49.7	12489
8	8	8	1.5	316	48.7	12503
4	4	4	7.7	1499	50.9	12237
2	2	2	34.8	6546	96.5	12300
1	1	1	164.7	27385	387.6	12356
−480	−480	−480	0.0	1	44.6	12496
−512	−512	−512 *	0.0	1	42.2	1562
544	0	0 *	0.0	0	12.1	0

* Nodes moved outside of the octree domain are truncated.

Table 5.3: Execution times for translation by (t_x, t_y, t_z).

			Cube-64		Sphere-64	
s_x	s_y	s_z	Seconds	Nodes	Seconds	Nodes
1	1	1	0.1	8	50.1	12496
2	2	2	0.1	8	53.8	12496
4	4	4	0.1	8	54.3	12496
0.5	0.5	0.5	0.1	8	27.4	3064
0.25	0.25	0.25	0.1	8	16.5	808
1.1	1.1	1.1	96.6	35112	253.8	14464
0.9	0.9	0.9	58.5	20672	148.8	9872
0	1	1 *	0.0	0	1.5	0

* Scaling table of zeroes signals an object to be nullified.

Table 5.4: Execution times for scalings by (s_x, s_y, s_z).

		Cube-64		Sphere-64	
Axis	θ	Seconds	Nodes	Seconds	Nodes
x	0	0.0	8	34.2	12496
x	−90	0.0	8	43.1	12496
y	10	105.8	12384	510.7	12944
z	45	110.2	11280	569.8	12928

Table 5.5: Execution times for rotations by θ degrees.

Finally, the union, intersection, and difference operations are timed and listed in Table 5.6. The efficiency of boolean set operations using linear octrees becomes prominent when compared to that of the geometric transformations. The rendering procedures described in Appendix A use an existing graphics package not designed for viewing linear octrees and are too slow to give meaningful timings.

Operation	Seconds	Nodes
$S \cup C$	8.3	8
$S \cap C$	14.6	12496
$S \setminus C$	8.8	0
$C \setminus S$	20.2	12936
$S \cup S$	25.8	12496
$S \cap S$	23.9	12496
$S \setminus S$	17.9	0
$F \cup S$	8.3	1
$F \cap S$	14.8	12496
$F \setminus S$	20.8	13160
$S \setminus F$	8.3	0
$H \cup S$	12.5	6252
$H \cap S$	10.7	6248
$H \setminus S$	13.5	6580
$S \setminus H$	11.2	6248

$C =$ Cube-64
$S =$ Sphere-64
$F =$ Cuboid$(0,0,0,1024,1024,1024,10)$
(entire domain)
$H =$ Cuboid$(0,0,0,512,1024,1024,10)$

Table 5.6: Execution times for boolean set operations.

Chapter 6

Conclusion

6.1 Summary

The linear octree structure as an object representation for solid modelling is studied. The linear representation interprets leaf nodes as octal numbers that are closely related to the 3-D coordinates of voxels, leading to a reduction in storage requirements as compared to explicit octrees. By using a consistent numbering scheme for the eight octants, and interpreting the node numbers in binary form, several new algorithms have been developed.

Conversion between nodes and voxel coordinates reduce to a regrouping of bits of their binary representations. The notion of successor of a node and the node trimming operations are defined for generating the normalized linear octree of a cuboidal volume; both require only manipulations on octal digits and bits.

The cuboid generation algorithm leads to unified translation and scaling operations. Nodes are translated and scaled into cuboidal targets to be generated as partial objects and merge into the final results. All cases are handled identically, while special ones such as translation by powers of 2 and scaling by factors of 2 are kept efficient naturally as a result of their simple octree structures. Arbitrary rotations are designed as an extension of the 90°-rotations that only involve permutation of octal digits. A backward mapping algorithm is used to ensure spatial continuity and node uniqueness of the rotated object.

The union and intersection operations are just variations of the familiar merging algorithm for two sorted lists of numbers. The difference operation, however, results in expansion of large nodes into its smaller descendents. With the successor generator, no extra memory is needed to keep expanded node lists.

All basic operations are implemented as a complete solid modelling library package accompanied by a set of interface programs. Though not formally proved, the algorithms exhibit execution times which are linear to the number of nodes in the resulting linear octree for most cases.

6.2 Future Work

The sorting component of the normalization process is the bottleneck of the geometric transformations. Generating normalized targets during these operations have not been fully taken advantage of. Better sorting algorithms for partially sorted lists should outperform the heapsort currently used.

For objects at higher resolutions, a large number of nodes inside the same octant will have identical leading digits, which waste space. A hybrid octree structure would be preferable. The top few levels can be kept as a pointer-structured tree, while the lower level subtrees can be "linearized" into a list of nodes. Such a structure would also lead to more efficient searching algorithms used by rendering and interference detection.

Furthermore, there are useful operations such as neighbourhood determination and surface detection algorithms, yet to be designed, that require only bit manipulations. Applying these operations to image processing and robotic vision may explore other advantages of linear octrees.

The conversion and trimming algorithms used extensively throughout the package induce a lager amount of overhead and hence are very suitable to be executed by hardware. It is possible to build such a linear octree processor with the advent of VLSI technology. Since most octree algorithms are highly parallel in nature, a linear octree multi-processor could support real-time solid modelling for computer-aided design and manufacturing.

Bibliography

[AHUJ84] Ahuja, N. and Nash, C., "Octree Representations of Moving Objects," *Computer Vision, Graphics, and Image Processing*, 26 (2), May 1984, pp. 207–216.

[BAUE85] Bauer, M. A., "Set Operations on Linear Quadtrees," *Computer Vision, Graphics, and Image Processing*, 29 (2), February 1985, pp. 248–258.

[BOYS82] Boyse, J. W. and Gilchrist, J. E., "GMSolid: Interactive Modeling for Design and Analysis of Solids," *IEEE Computer Graphics and Applications*, 2 (2), March 1982, pp. 27–40.

[BRAC80] Braccini, C. and Marino, G., "Fast Geometrical Manipulations of Digital Images," *Computer Graphics and Image Processing*, 13 (2), June 1980, pp. 127–141.

[BROW82] Brown, C. M., "PADL-2: A Technical Summary," *IEEE Computer Graphics and Applications*, 2 (2), March 1982, pp. 69–84.

[DOCT81] Doctor, L. J. and Torborg, J. G., "Display Techniques for Octree-Encoded Objects," *IEEE Computer Graphics and Applications*, 1 (3), July 1981, pp. 29–38.

[FRAN82] Franklin, W. R., "Efficient Polyhedron Intersection and Union," *Graphics Interface '82*, 1982, pp. 73–80.

[FRIE85] Frieder, G., Gordon, D., and Reynolds R. A., "Back-to-Front Display of Voxel-Based Objects," *IEEE Computer Graphics and Applications*, 5 (1), January 1985, pp. 52–60.

[FUJI83] Fujimura, K., Toriya, H., Yamaguchi, K., and Kunii, T. L., "Oct-tree Algorithms for Solid Modeling," *Proceedings of InterGraphics '83*, 1983, pp. 96–110.

[GARG82] Gargantini, I., "Linear Octtrees for Fast Processing of Three-Dimensional Objects," *Computer Graphics and Image Processing*, 20 (4), December 1982, pp. 365–374.

[GARG83] Gargantini, I., "Translation, Rotation and Superposition of Linear Quadtrees," *International Journal of Man-Machine Studies*, 18 (3), March 1983, pp. 253–263.

[GLAS84] Glassner, A. S., "Space Subdivision for Fast Ray Tracing," *IEEE Computer Graphics and Applications*, 4 (10), October 1984, pp. 15–22.

[HILL82] Hillyard, R. C., "The Build Group of Solid Modelers," *IEEE Computer Graphics and Applications*, 2 (2), March 1982, pp. 43–52.

[HUNT79] Hunter, G. M. and Steiglitz, K., "Operations on Images Using Quad Trees," *IEEE Transaction on Pattern Analysis and Machine Intelligence*, PAMI-1 (2), April 1979, pp. 145–153.

[JACK80] Jackins, C. L. and Tanimoto, S. L., "Oct-Trees and Their Use in Representing Three-Dimensional Objects," *Computer Graphics and Image Processing*, 14 (3), November 1980, pp. 249–270.

[LAUZ85] Lauzon, J. P., Mark, D. M., Kikuchi, L., and Guevara, J. A., "Two-Dimensional Run-Encoding for Quadtree Representation," *Computer Vision, Graphics, and Image Processing*, 30 (1), April 1985, pp. 56–69.

[MANT83] Mäntylä, M. and Tamminen, M., "Localized Set Operations for Solid Modeling," *ACM Computer Graphics*, 17 (3), July 1983, pp. 279–288.

[MEAG82] Meagher, D., "Geometric Modeling Using Octree Encoding," *Computer Graphics and Image Processing*, 19 (2), June 1982, pp. 129–147.

[REQU80] Requicha, A. A. G., "Representations for Rigid Solids: Theory, Methods, and Systems," *ACM Computing Surveys*, 12 (4), December 1980, pp. 437–464.

[REQU82] Requicha, A. A. G. and Voelcker, H. B., "Solid Modeling: A Historical Summary and Contemporary Assessment," *IEEE Computer Graphics and Applications*, 2 (2), March 1982, pp. 9–24.

[REQU83] Requicha, A. A. G. and Voelcker, H. B., "Solid Modeling: Current Status and Research Directions," *IEEE Computer Graphics and Applications*, 3 (7), October 1983, pp. 25–37.

[SAME80] Samet, H., "Region Representation: Quadtrees from Boundary Codes," *Communications of the ACM*, 23 (3), March 1980, pp. 163–170.

[TAMM84] Tamminen, M., "Efficient Octree Conversion by Connectivity Labeling," *ACM Computer Graphics*, 18 (3), July 1884, pp. 43–51.

[TILO80] Tilove, R. B., "Set Membership Classification: A Unified Approach to Geometric Intersection Problems," *IEEE Transactions on Computers*, C-29 (10), October 1980, pp. 874–883.

Appendix A

The Solid Modelling Package

A.1 The Linear Octree Library

The solid modelling package is implemented in C under **ubc-cs**, a VAX 11/750 running 4.2 BSD UNIX. All sources, library, and executable codes are residing under the directory `~ho/lot/lib` (userid **ho**). Library functions are archived (`ar(1)`) in `liblot.a` and should be loaded by including it in `cc(1)` or `ld(1)`, while their source codes are stored in files beginning with lower case letters (`[a-z]*.c`). The files `README` and `Makefile` contain more information on how to use the library.

The major functions will be described here. `#include <lot.h>` must appear at the beginning of the source file before referencing these functions. Functions that are not explicitly typed are of type `int` in C, but here untyped functions do not return useful values and should be considered only as procedures. Most abbreviations used for composing names follow the conventions listed below:

`St...`:	`struct` name
`Ty...`:	`typedef`'ed name
`p...`:	pointer, pointer-to
`i...`:	index, index-of
`Blc`:	block of storage
`LUT`:	lookup table
`Obj`:	solid object
`LOT`:	linear octree
`Bn`:	linear octree node (black node of the "tree")
`Vox`:	voxel
`Ind`:	indicator
`Pkd`:	packed

Basic Operations and Conversions:

int BnLevel(Bnode, Resoln)
TyBnode Bnode; int Resoln;

Returns the node level of **Bnode** at resolution **Resoln**.

BnToVo(Bnode, Resoln, Level, pLoVox, pHiVox)
TyBnode Bnode; int Resoln, Level; TyVox *pLoVox, *pHiVox;

Returns the lowest and highest order voxels ***pLoVox** and ***pHiVox** of level **Level** node **Bnode** of resolution **Resoln**.

LVoToHVo(LoVox, Resoln, Level, HiVox)
TyVox LoVox, HiVox; int Resoln, Level;

A macro expansion to compute the highest order voxel **HiVox** from the lowest order voxel **LoVox** given its node level **Level** and resolution **Resoln**.

HVoToLVo(HiVox, Resoln, Level, LoVox)
TyVox LoVox, HiVox; int Resoln, Level;

A macro expansion to compute the lowest order voxel **LoVox** from the highest order voxel **HiVox** given its node level **Level** and resolution **Resoln**.

LVoToBn(pLoVox, Resoln, pLevel, Bnode)
TyVox *pLoVox; int Resoln, *pLevel; TyBnode Bnode;

Returns the largest node **Bnode** and its level ***pLevel** given its lowest order voxel ***pLoVox** at resolution **Resoln**.

VoToBn1(pVox, Resoln, Bnode)
TyVox *pVox; int Resoln; TyBnode Bnode;

Returns a level **Resoln** node **Bnode** (smallest node) representing the voxel ***pVox**, i.e., a direct conversion from voxel to node number is carried out.

Boolean SuccBn(Bnode, pLevel)
TyBnode Bnode; int *pLevel;

Replaces the level ***pLevel** node **Bnode** by its successor and updates ***pLevel**. Returns **TRUE** unless **Bnode** does not have a successor. Note that the resolution parameter is irrelevant since the **Level** will only "rise" towards 0.

Trimming:

TrimBnLo(Bnode, Resoln, pLevel, Coord, OcPos)
TyBnode Bnode; int Resoln, *pLevel, Coord, OcPos;

Trims a level ***pLevel** node **Bnode** at resolution **Resoln** against a lower bound **Coord**. **OcPos** specifies the bit position of an x, y, or z coordinate within the octree digit, thus specifying the x, y, or z-bound. ***pLevel** will be updated.

```
TrimBnHi( Bnode, Resoln, pLevel, Coord )
TyBnode Bnode; int Resoln, *pLevel, Coord;
```

Trims against a higher bound similar to TrimBnLo, but identically for all x, y, and z-bound.

Storage and Node Retrieval:

```
Boolean GetObj( pObjFile, pObj )
FILE *pObjFile; TyObj pObj;
```

Reads an object *pObj from file *pObjFile. Returns FALSE on bad external object file format or insufficient memory.

```
PutObj( pObjFile, pObj )
FILE *pObjFile; TyObj pObj;
```

Writes an object *pObj onto file *pObjFile.

```
PackBn( Bnode, Resoln, pPkdBn )
TyBnode Bnode; int Resoln; unsigned *pPkdBn;
```

Packs the node Bnode at resolution Resoln into *pPkdBn.

```
UnpackBn( PkdBn, Resoln, Bnode )
unsigned PkdBn; int Resoln; TyBnode Bnode;
```

Unpacks the packed node PkdBn of resolution Resoln into Bnode.

```
Boolean GetNextBn( pObj, pInd, Bnode )
TyObj *pObj; TyInd *pInd; TyBnode Bnode;
```

Returns the next node Bnode of object *pObj using indicator *pInd and advances it to "point to" the next node. Returns FALSE when last node was retrieved. To start up a sequential retrieval, pInd->pLOTBlc must be set to pObj->p.LOT and pInd->iLOTBlc to 0.

```
Boolean DelNextBn( pObj, pInd, Bnode )
TyObj *pObj; TyInd *pInd; TyBnode Bnode;
```

Identical to GetNextBn() except that a block will be freed as soon as *pInd moves away from it.

```
Boolean InsertBn( Bnode, pObj, pInd )
TyBnode Bnode; TyObj *pObj; TyInd *pInd;
```

Inserts the node Bnode into object *pObj after the position "pointed to" by *pInd. *pInd will advance to the current position. Returns FALSE on insufficient memory. To initialize the appending mode, *pInd->pLOTBlc must be set to NULL.

```
Boolean DupObj( pObj1, pObj2 )
TyObj *pObj1, *pObj2;
```

Duplicates object *pObj1 as *pObj2. Returns FALSE on insufficient memory.

```
NullObj( pObj )
TyObj *pObj;
```
Nullifies the object ***pObj** by freeing all its node blocks.

Normalization:

```
Boolean SortLOT( ppLOTBlc )
TyLOTBlc **ppLOTBlc;
```
Sorts the nodes in the linear octree blocks heading with ****ppLOTBlc** into increasing order and condenses them. Returns **FALSE** on insufficient memory. (This may happen since blocks are reallocated, but unlikely.)

```
Boolean MergeLOT( pqLOTBlc, ppLOTBlc, Fast )
TyLOTBlc *pqLOTBlc, *ppLOTBlc; Boolean Fast
```
Merges the linear octree ***pqLOTBlc** into ***ppLOTBlc** and condenses the result. If **Fast** is **TRUE**, not every node will be compared for merging and condensation, instead, blocks unchanged will simply be re-linked. ***pqLOTBlc** will be **NULL** upon return. Returns **FALSE** on insufficient memory.

```
Boolean CollectBn( PkdBn )
unsigned PkdBn;
```
Collects a packed node **PkdBn** onto an internal stack for condensation. Returns **FALSE** on insufficient memory returned from the internal function **ReleaseBn()**. **CollectBn()** must be called sequentially with a sorted list of **PkdBn**'s after an initial call to **CdInit()**, and must signal the termination of the sequence by a call to **CdDone()**.

```
Boolean CdInit( ppLOTBlc )
TyLOTBlc **ppLOTBlc;
```
Initializes the internal memory for condensing nodes into linear octree ***ppLOTBlc**. Returns **FALSE** on insufficient memory.

```
Boolean CdDone()
```
Inserts all pending nodes collected for condensation into the current linear octree. Returns **FALSE** on insufficient memory.

Object Generation and Modelling Operations:

```
Boolean Cuboid( x0, y0, z0, dx, dy, dz, Resoln, pObj )
int x0, y0, z0, dx, dy, dz, Resoln; TyObj *pObj;
```
Returns the cuboid ***pObj** of resolution **Resoln** with lowest order corner at (**x0**, **y0**, **z0**) and dimensions **dx** × **dy** × **dz**. Returns **FALSE** on insufficient memory.

`Boolean Ellipsoid( x0, y0, z0, dx, dy, dz, Resoln, pObj )`
`int x0, y0, z0, dx, dy, dz, Resoln; TyObj *pObj;`

Returns the ellipsoid ***pObj** of resolution **Resoln** enclosed in the cuboid with lowest order corner at (**x0**,**y0**,**z0**) and dimensions **dx** × **dy** × **dz**. Returns **FALSE** on insufficient memory.

`Boolean Translate( pObj, tx, ty, tz )`
`TyObj *pObj; float tx, ty, tz;`

Translates the object ***pObj** by (**tx**,**ty**,**tz**). Returns **FALSE** on insufficient memory.

`Boolean Scale( pObj, sx, sy, sz )`
`TyObj *pObj; float sx, sy, sz;`

Scales the object ***pObj** by **sx**, **sy**, **sz** times along the x, y, and z-axis respectively, centering at the center of the object's octree domain. Returns **FALSE** on insufficient memory.

`Boolean Rotate( pObj, Axis, thetaD )`
`TyObj *pObj; char Axis; float thetaD;`

Rotates the object ***pObj** by **thetaD** degrees about the axis parallel the **Axis**-axis and passing through the center of the object's octree domain. **Axis** can have one of the values **'x'**, **'y'**, or **'z'**. Returns **FALSE** on insufficient memory.

`Boolean Union( pObj1, pObj2 )`
`TyObj *pObj1, *pObj2;`

Returns the union of objects ***pObj1** and ***pObj2** and keep the result in ***pObj1**. ***pObj2** will be nullified. Returns **FALSE** on insufficient memory.

`Boolean Intersect( pObj1, pObj2 )`
`TyObj *pObj1, *pObj2;`

Returns the intersection of objects ***pObj1** and ***pObj2** and keep the result in ***pObj1**. ***pObj2** will be nullified. Returns **FALSE** on insufficient memory.

`Boolean Subtract( pObj1, pObj2 )`
`TyObj *pObj1, *pObj2;`

Returns the difference of object ***pObj2** from ***pObj1** and keeps the result in ***pObj1**. (i.e., subtract ***pObj2** from ***pObj1**.) ***pObj2** will be nullified. Returns **FALSE** on insufficient memory.

Rendering:

`View( Dev, pParFile )`
`TyDev Dev; FILE *pParFile;`

Sets up viewing parameters from data in file ***pParFile** for device **Dev**.

`Display( pObj )`
`TyObj *pObj;`

Displays the object ***pObj** on the selected device under the current viewing parameter settings.

```
Boolean DepthSort( pObj, Octant )
TyObj *pObj; int Octant;
```

Rearranges the nodes in object `*pObj` into a depth-sorted order when viewed from octant `Octant`. Returns `FALSE` on insufficient memory.

```
PutVertex( pObj, Octant )
TyObj *pObj; int Octant;
```

Outputs the 7 vertices of the 3 visible surfaces of each node in object `*pObj` as viewed from octant `Octant` to `stdout`.

A.2 User Programs

A set of driver programs are available as ordinary UNIX commands in the directory `~ho/lot`. The syntax of these commands are described below. Most of them reads and writes objects via `stdin` and `stdout` using the external object file format as described in Section 5.2.2. All errors generated are written onto `stderr`.

The object generation commands

cuboid x_0 y_0 z_0 Δx Δy Δz *resolution*

ellipsoid x_0 y_0 z_0 Δx Δy Δz *resolution*

are identical to their library counterparts with objects written out onto `stdout`.

The command **unpack** reads an object from `stdin` and dumps all the nodes in printable octal digits to `stdout`, suitable for debugging. **pack** does the exact inverse of **unpack**, which can be used for building a linear octree-encoded object manually. There is also an **analyze** command that reads an object from `stdin` and prints a tally of its nodes onto `stdout`.

The three geometric transformations

translate t_x t_y t_z

scale s_x s_y s_z

rotate *axis* *theta*

read an object from `stdin` and writes the result onto `stdout`; they all accept real arguments.

The boolean operations

union [*file1*] *file2*

intersect [*file1*] *file2*

subtract [*file1*] *file2*

have identical syntax, accepting two objects. If only one file name is given, it will be taken as the second object, and the first object will be read from `stdin`. Results will be written onto `stdout` as usual.

Rendering is not completely implemented. An object in file *file* can be displayed on the Jupiter 7 terminal under MTS using viewing parameters from file *viewpar* by following the tedious procedure below:

1. Issue the command

   see virtual *viewpar file* **>lot**

   where **virtual** is the only device supported currently, and the file **lot** must be specified exactly.

2. Issue

   seelot >*image*

   which reads the faces from file **lot** and generates a 256 × 192 grey level image onto file *image*. For compatibility with MTS, *image* contains 257 × 192 8-bit pixels with 8 grey levels, decimal values 65 to 72. Each 256-byte scanline is terminated by the newline character, decimal 10. All other values are unused.

3. Transfer the file *image* to MTS. Currently only **tip(1)** seems to work well, except for its low speed—be patient. Use 2400 baud and the **~f** command of **tip**, and make sure to use the **binary** format. The receiving MTS file must be created before the transfer.

4. Signon to MTS from the Jupiter 7 terminal. Then issue

   run j7.seelot+*juplib 0=*j7image*

   to see the result assuming *image* is transferred to *j7image*.

The following is a sample session to create the object shown in figure A.1. A different procedure was taken to put all six images onto the Raster Technologies One/25 in the UBC Laboratory for Computational Vision.

```
cuboid 464 512 480 96 24 64 10 >brick
```
(a parallelepiped at the center of a resolution 10 domain)

```
rotate y 60 <brick | intersect brick >rhombus
```

```
rotate y -60 <brick | intersect rhombus >hexegon
```
(intersecting the three cuboids results in a hexagonal prism)

```
ellipsoid 472 500 472 80 36 80 10 | intersect hexegon >head
```
(the bolt's head)

```
cuboid 506 520 480 12 16 64 10 >slice
```

```
union head cylinder | subtract slice >bolt
```
(**cylinder** is actually a scale-up of a 1 voxel thick ellipsoid—a disc)

```
see virtual vp10 bolt >lot
```
(file `vp10` already set up to view center part of a resolution 10 domain)

```
seelot >image.jupi
```

Send to Jupiter 7 or Raster Technologies One/25 for viewing.

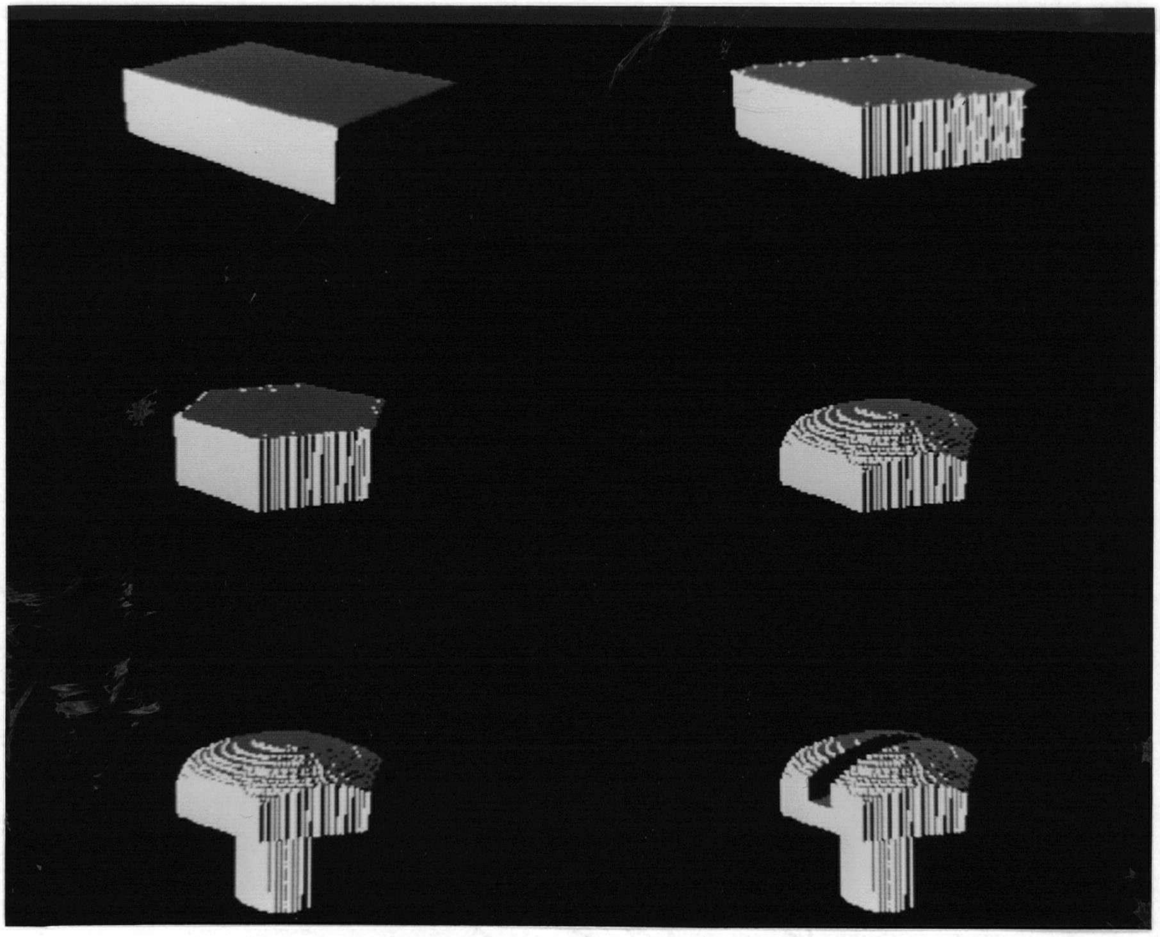


Figure A.1: Various stages in creating an object.