Application of
Binary Space Partitioning Trees to
Geometric Modeling and Ray-Tracing

A Thesis

Presented to

The Faculty of the Division of Graduate Studies

by

William Charles Thibault

In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy

in the School of Information and Computer Science

Georgia Institute of Technology

September 1987

ACKNOWLEDGEMENTS

I would like to thank Bruce Naylor giving me enough freedom to keep things interesting while providing the guidance to make the work productive. Thanks are also due to the Computing Systems Research Lab at AT&T Bell Laboratories for supporting much of the implementation work. Chris Morgan provided many helpful comments on the draft. The people who helped to keep me sane enough to complete the degree are also to be thanked. Listing the contributions made by each person would only trivialize them. So, thanks go to my parents, Roy Lovell, Sarah Flynn, Dick and Lynn Mays, Geronimo and Diane Teer, Marsha Brown, Martin and Susan McKendry, Rachael Liss, Mike Merritt, Laurie Hodges Reuter, Carol Kilpatrick, Danny Lunsford, and the girls at the Clermont Lounge.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

SUMMARY

The Binary Space Partitioning Tree (BSP Tree) is a binary tree used to represent an organization of continuous space by recursive subdivision. The BSP tree was initially introduced to organize a set of polygons so that visible surface renderings could be produced from an arbitrary viewing position. This thesis extends the use of the BSP tree in two ways. The first, constituting the bulk of the work, uses the BSP tree to model polyhedral solids. A number of algorithms relating to this representation of a polyhedral set are introduced: determining the boundary of the set, determining volume and center of mass, evaluating set operations (union, intersection, difference), classification of points, line segments, and polygons with respect to the set, constructing the BSP tree from the boundary of the set, and constructing a boundary representation of a set represented by a BSP tree.

The BSP tree provides a unified approach to the basic problems encountered in constructing an interactive geometric modeling system. These are: representing polyhedra, rendering, geometric search, property calculation, and set operation evaluation. In previous work, these issues have been addressed with largely independent and unrelated algorithms. By solving these problems with a set of closely related algorithms based on the BSP tree, the task of constructing such a system is made easier.

The other new use for the BSP tree discussed is its use in ray-tracing. Ray-tracing can produce very realistic images, but is plagued by the requirement of large amounts of computation. The problem of finding the closest intersection a ray with a collection of objects, the basic operation in ray-tracing, is essentially a searching problem. The BSP tree is used to organize the objects so that the search can be made efficient.

CHAPTER I

Introduction

Pleased to meet you. Hope you guessed my name.

— Rolling Stones

Computer Graphics deals with techniques for automatically drawing pictures of objects that are represented inside the computer, as well as the interactive design of such objects. *Geometric Modeling* is concerned with a wider range of properties of these objects: volume, area, center of mass, interference (i.e., whether two objects intersect), finite element analysis, etc. Applications of these techniques range from the purely aesthetic, as in art, advertisement, and entertainment, to the mundane, as in the computer-aided design of appliances, buildings, and weapons. This thesis will address problems in both realms.

The work centers on the use of the Binary Space Partitioning (BSP) Tree, presented in Chapter III. The thesis shows how the BSP tree can be used for two basic purposes. One concerns its use in reducing the amount of computation required to produce realistic images containing reflections, shadows, and transparent objects using the ray-tracing technique (Chapter V). The other emphasis is on the modeling of solid objects. Algorithms for defining solids (Chapter III) and modifying them with the Boolean operations (union, intersection, and difference) (Chapter IV) are given. The viability of the techniques is demonstrated through implementation work (Chapter VI).

Solid Modeling

The recent development of geometric modeling as an identifiable field has been motivated in large part by the emergence of Computer Aided Design and Manufacturing

(CAD/CAM) as an effective tool for improving productivity in manufacturing. Other, more abstract domains have also been modeled geometrically, such as statistical distributions[Bent79] and database schedules[Yann79].

Much of the work in the modeling of geometric objects with computers has been motivated by applications in manufacturing and engineering. A number of TLAs (Three Letter Acronyms) have also been generated: CAD (Computer Aided Design), CAM (Computer Aided Manufacturing), CAE (Computer Aided Engineering), CIM (Computer Integrated Manufacturing). Some of the earliest systems, among the most widely used today, use the computer and its graphical display capabilities to manage creation and maintainance of technical drawings such as blueprints and schematics. Commercially available systems are now available that allow a designer to specify the shape of an object and view computer-generated renderings of it. This data can then be used to control automated cutting and milling machines that cut the object from a piece of stock material. For the design of electronic equipment, tools can create printed circuit boards and VLSI layouts from schematics or higher-level specifications such as algorithms. When designing a building, the architect can automatically generate views of how the building will look from arbitrary viewing positions. This can be used to generate animated sequences simulating what a person would see when walking through or around the building.

Blueprints of three dimensional objects, consisting of several "wireframe" projections, are a poor representation for most automated processes, such as determining if a given point lies within the object. Automatic conversion from blueprints to more useful representations is made difficult because ambiguities are possible in wireframes. These ambiguities are usually easy for a human to resolve, often passing unnoticed. Figure 1 shows an ambiguous object: if we interpret the object as having a hole, in which direction does the hole pass through the object? Systems for automatic wireframe (blueprint) interpretation usually rely on the operator to resolve ambiguities. Blueprints, however, remain an important

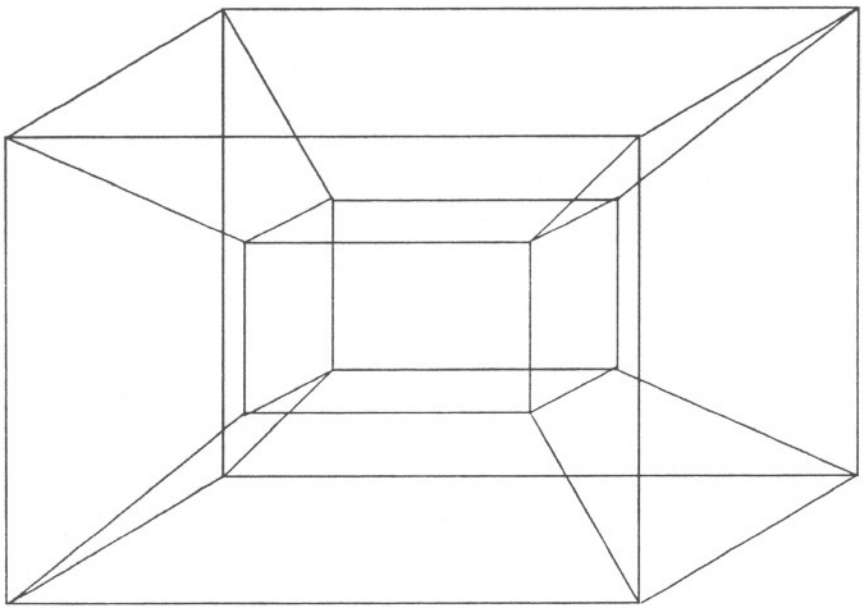


Figure 1. An Ambiguous Wireframe Representation

communication tool among humans.

The term *solid modeling* is used to refer to the modeling of "realizable" solids, i.e., objects that have a 3-dimensional extension in space (volume). The objects modeled by geometric modeling systems are often manufactured parts. 3-dimensional specification of objects can be made directly by interaction with graphical displays, with blueprint-like 2-d diagrams, or with special purpose languages. Once specified, the computer can determine the answer to various queries about the object, such a volume, moments of inertia, and surface area, as well as generating renderings of the object. With such a system, the operator can iteratively modify the object to better meet his design goals, without ever constructing physical models or prototypes.

Polyhedra

This section discusses previous work in the representation of polyhedra. For our purposes, polyhedra are solids whose boundaries are planar polygons. Boundary representations of polyhedra are often used in solid modeling, largely due to the fact that most visible surface algorithms require polygons as input. Also, polyhedra are described by linear equalities and inequalities, which allows for easily computed solutions to many problems, making them attractive for computer implementation. This has been exploited in a number of graphics workstations, such as the Silicon Graphics IRIS. These machines support, in hardware, transformation and clipping of polygons, making them effective for interactive use.

Polyhedra have been studied since the beginning of the Western Tradition. The so-called "Platonic solids" consisted of the regular 3-dimensional polyhedra: tetrahedron, cube, octahedron, dodecahedron, and icosahedron. These are the only regular polyhedra possible in 3-dimensions; a fact which so captured the imagination that they became symbolic of the five consituents of all existance: fire, earth, air, water, and ether. The astronomer (and astrologer) Johannes Kepler spent years trying to descibe heavenly motions in terms of nested, concentric spheres placed in such a manner that their interstices were defined by these five polyhedra (it didn't work).

Euler first defined the relationship between the numbers of vertices, edges, and faces of a polyhedron with the formula: $v + f - e = 2$. This was later extended to handle multiple polyhedra, polyhedra with "handles" or "through holes" and polyhedra with faces that have holes: $v + f - e = 2(b - g)$, where g is the genus (the number of passages or through holes), and b the number of disjoint bodies (objects or internal cavities). This is useful, as it provides a necessary condition for a polyhedron to be physically realizable.

When computers became available, the problem of representing a polyhedron in a

computer arose. Two basic types of representations have been developed: *boundary representations* (B-reps), and *volumetric representations*.

Boundary representations enumerate the components of the boundary of the polyhedron, its faces, edges, and vertices. It can include purely geometric information, the position and orientation of each element of the boundary, or may also include connectivity (topological) information. A natural representation for this topological information mirrors the connections between elements of the boundary of the polyhedron: each vertex is associated (say, via pointers) with a some number of edges and faces, each edge is associated with two vertices and two faces, each face is associated with some number of edges and vertices (where the numbers of faces and vertices are equal). One can also order edges about vertices, and vertices about faces. This information-rich data structure is called a "winged-edge", topological, net, or graph representation.

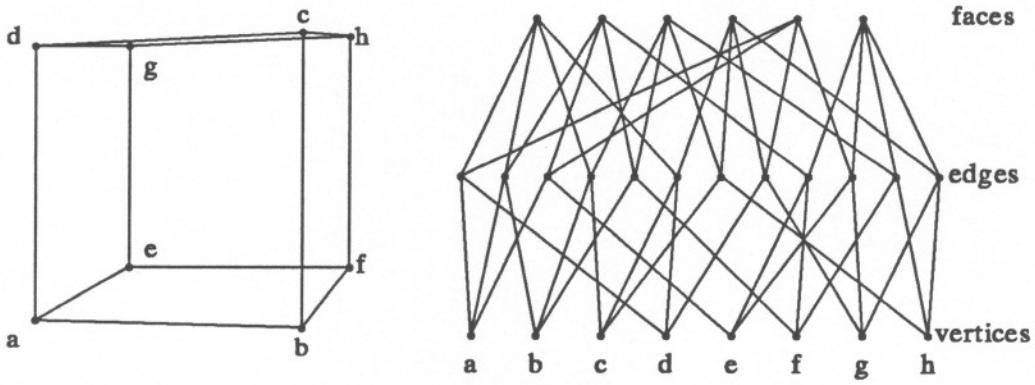


Figure 2. One Possible Graph Representation for a Cube

(Figure 2.) The amount of information stored in the data structure should be sufficient for the intended application.

Volumetric representations express the set in terms of simple primitive solids, where each primitive is described by a simple characteristic function. Commonly used primitives

include linear halfspaces and axis-aligned cubes. For example, convex polyhedra can be represented by a set of halfspaces whose intersection defines the polyhedron. Concave polyhedra can be represented in terms of set operations on half-space primitives [Brai75].

Set Operations

A convenient means of specifying complex objects is to combine a number of simple primitives with the set operations union, intersection, and difference. A Constructive Solid Geometry (CSG) representation is a binary tree in which the internal nodes represent one of the (regularized) set operations union, intersection and difference; the leaves of the CSG tree are primitive solids[Requ80, Mort85]. The primitives commonly include the quadrics: sphere, cone, cylinder, paraboloid, etc. To perform some geometric operation or query on the object described by a CSG representation, the CSG tree itself can be used. This has the advantage of using the representation actually specified by the user, and can provide the most accurate result possible. Also, a single framework, ray-casting (which is described later in this chapter), can be used for handling most operations. Ray-casting can handle a large class of primitives, requiring a single operation to be supported for each primitive, that of intersecting a line with the primitive.

The major drawback of using the CSG representation directly for answering qeometric queries is that the result will usually require large amounts of computation. One reason for this is that each such computation must determine the effects of the set operations. Consider the problem of determining where a given point lies with respect to an object represented by a CSG representation. A simple approach is to test the point to each primitive, and then combine these results according to the set operations. This "combining" step based on the set operations must be repeated for each point tested. Any query answered by sampling the CSG representation, such as with a number of rays or points, will require large amounts of computation.

Another problem with CSG representations involves using primitives described by quadratic or higher-order polynomials: computations on such entities are inherently computationally expensive. Also, non-linear polynomials of are not algebraically closed under set operations. For instance, the intersection curve of two quadrics can be of degree 4.

To allow for more efficient generation of a number of different views of a CSG expression, that representation can be converted into an approximate, polyhedral boundary representation of the object. The problem of determining the boundary of the CSG representation is then one of determining which faces (or portions thereof) belong to the boundary. Once the bounding faces have been determined, the rendering algorithm need only be concerned with which faces are visible, and need not determine which faces are actually on the boundary.

Geometric modeling systems

Most real-world geometric modeling systems use at least two representations [Requ83], allowing operations to be carried out with the most appropriate representation. For example, initial specification may be made with a CSG tree, on quadric or tri-cubic parametric primitives. For display, this can be converted into a polyhedral boundary representation, and the boundary used with a conventional rendering algorithm. Alteratively, if a high-quality rendering is desired, the CSG representation itself can be used as input to a ray-tracing renderer. Figure 3 illustrates a likely system architecture.

The conversion from CSG to an approximate boundary representation consisting of polygons is supported by most systems. This allows the use of "polygon engines," hardware designed for rapid transformations and renderings of polygons. One scenario for the interactive design process begins with the specification by the user of a CSG tree defining the object. The CSG tree is then automatically converted to a polyhedral boundary representation. Next, the user specifies various viewing operations and queries the system

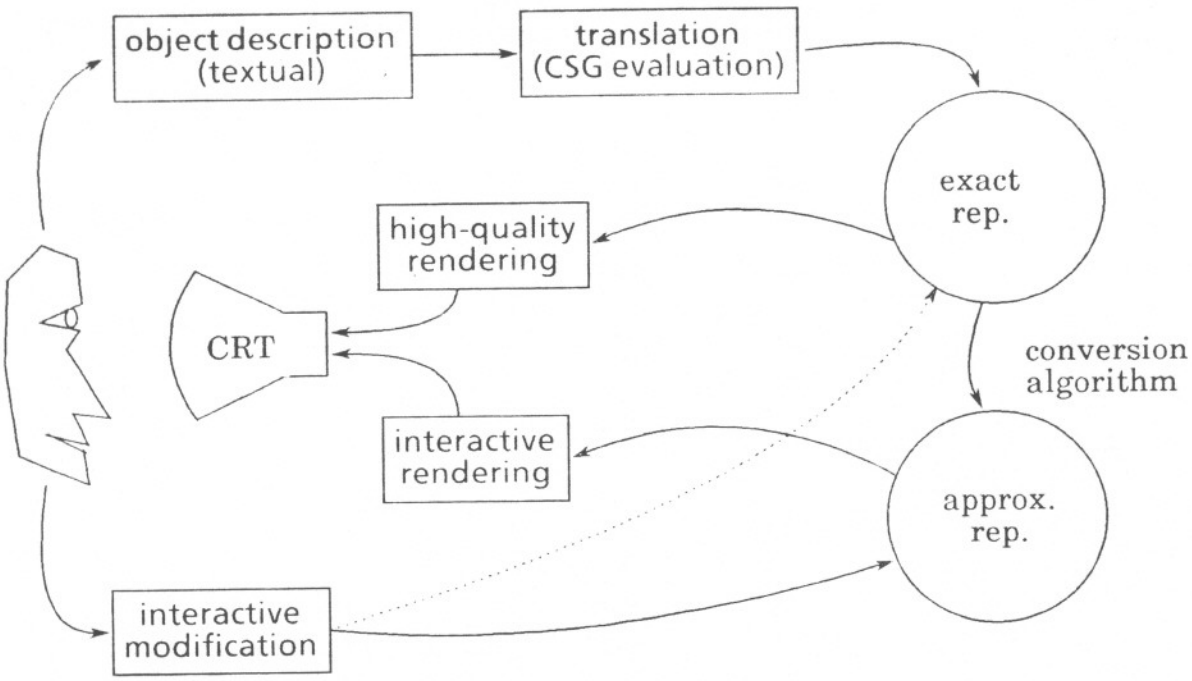


Figure 3. Architecture of a Geometric Modeling System

about various properties of the object. If the design is found lacking, the CSG tree can be modified. Modification to the CSG tree requires another conversion to be performed, and the process cycles. In an interactive setting, the CSG to boundary representation conversion could be done many times, so the conversion must be fast. Also, a large number of different views may be generated for the user to get a good sense of the spatial qualities of the object. For example, an animation of a smooth rotation of the object is often helpful.

In another scenario, given a boundary representation, the user "tweaks" the object by perturbing certain faces, edges, or vertices. This has the benefit of fast feedback, since the displayable representation is being modified and no conversion is being done. However, it is an open question as to how the result of such a modification can be mirrored in the CSG tree[Requ83] .

Yet another scenario has the user modifying the object incrementally by applying an additional set operation with a single primitive. A cylinder can be used to produce a number of mounting holes via repeated application of a set difference operation, for example. The user's actions can either modify the CSG tree, requiring conversion to be performed for visual feedback of each operation, and/or the operation can be performed on the boundary representation directly.

Geometric Search

Efficient algorithms on a collection of geometric entities, as is the case for all algorithms, require a fast way to access the data describing them. *Geometric search structures* are data structures that organize geometric data for quick access.

When considering a set operation on two polyhedra, for instance, it must be determined for each face of each polyhedron whether the two faces intersect. This is $O(n^2)$ if no geometric search structure is used. To acheive reasonable preformance, some sort of geometric search structure must be supported in a geometric modeling system.

Outline of Thesis

Geometric modeling systems have suffered from having to use separate and unrelated algorithms and data structures for various operations such as viewing, property calculation, set operations, and geometric searching. This makes the systems large and complicated.

This thesis discusses how a single data structure, the Binary Space Partitioning tree, or BSP tree, can be used to perform many of the operations required for geometric modeling in a unified framework. The first set of algorithms, constituting the bulk of the work, applies the BSP tree to geometric modeling of polyhedra. Techniques are presented for the definition and interactive modification of 3-dimensional polyhedral solids via the set operations union, intersection, and difference. The resulting representation is shown to be effective for visible surface renderings, property calculations, and for further modification

with set operations. The BSP tree provides a means to search space efficiently, a property that is useful for these operations. The anticipated uses of these techniques are in the areas of Computer Graphics, Computer Aided Design (CAD) and Robotics. Most of the techniques presented are applicable in any dimension, and may find application in areas that make use of non-convex polyhedra in higher dimensions, such as Operations Research.

Ray-tracing is the other use for BSP trees which is discussed. Ray-tracing uses the ideas of geometric optics to produce realistic images by tracing rays of light through a scene to simulate shadows, reflection, and refraction. Although capable of producing impressive images, the technique is computationally expensive. Techniques are presented that use the BSP tree to reduce a major aspect of this expense: the number of objects that must be tested for intersection with a given ray. Also, ray-tracing of BSP tree representations of polyhedra is discussed. These same algorithms can be used for *ray-casting*, a closely related technique that is used for determining properties such as volume.

CHAPTER II

Past Work in Geometric Modeling

Let attention be at a place where you are seeing some past happening, and even your form, having lost its present characteristics, is transformed.

— from the Vigyana Bhairava Tantra

Representation of Solids

Several techniques for specification, representation, and modification of solids have been used or proposed. Most are dependent on the particular representation being used. Existing representation schemes can be classified as follows (after Requicha [Requ80]):

1. Cell Decomposition -- The object is decomposed into a number of cells whose interiors do not intersect. In the simplest sort of decomposition, *spatial enumeration*, the object is represented as a list of disjoint cells of fixed size arranged in a regular grid. (In 3-d, the only possible decomposition of space into equal-sized, regular polyhedra uses cubes[Coxe63]. Decompositions using two distinct regular polyhedra, such as tetrahedron and octahedron, are also possible [Full75].) This can require a large amount of storage. A more general decomposition allows the cells to be of different shapes and sizes. For example, any polyhedron may be decomposed into a set of disjoint tetrahedra. The octree and BSP tree representations (described below) produce cell decompositions, in which the cells are of differing sizes and are organized by a hierarchical tree structure.

2. Constructive Solid Geometry (CSG) -- Objects are described by set-theoretic expressions on a collection of primitive solids (cubes, cones, etc.). In point-set

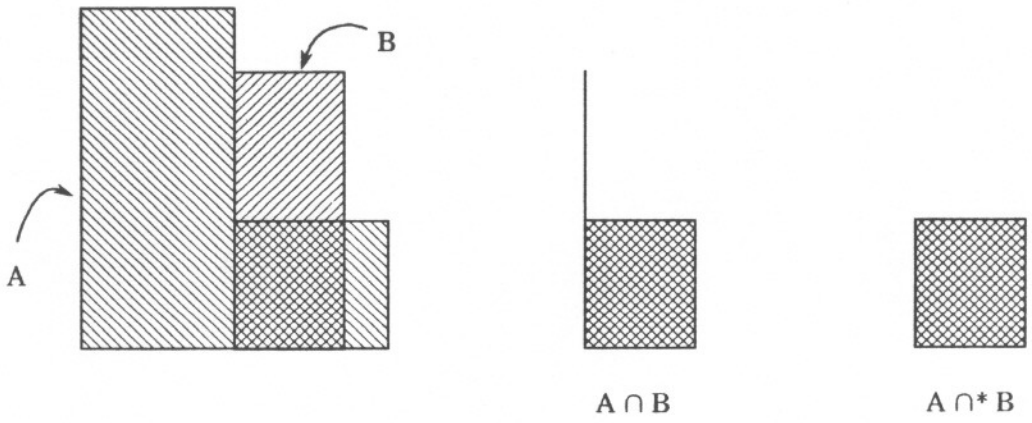


Figure 4. The Regular Set Operations Prevent Dangling Edges

topological terms, these objects have non-empty interiors. ***Regular sets*** have been proposed as a formalism for use in solid modeling [Requ78]. The approach serves to insure that the results of set operations are solids. Briefly, the *closure* of a set consists of the set together with its boundary. A set is *regular* if it is the closure of its interior[Requ78]. Note that the concept of interior, and therefore, regularization, is dependent on dimension: a line segment has a 1D interior, but it's interior in 2D is empty. This implies that the 1D-regularization of a line segment is that line segment, and the 2D-regularization of a line segment is empty. The regularized set operations behave as the usual ones, but return the closure of the interior of the result.

Consider the 2D example of Figure 5-1. Polygons A and B have some boundary points in common. The usual intersection operation $\cap$ leaves a "dangling edge" that that does not bound a 2D solid, and hence has no (2D) interior. The regularized operation $\cap^*$ does not produce such an edge. The expression is usually stored as a CSG tree, in which primitives are at the leaves and internal nodes represent set

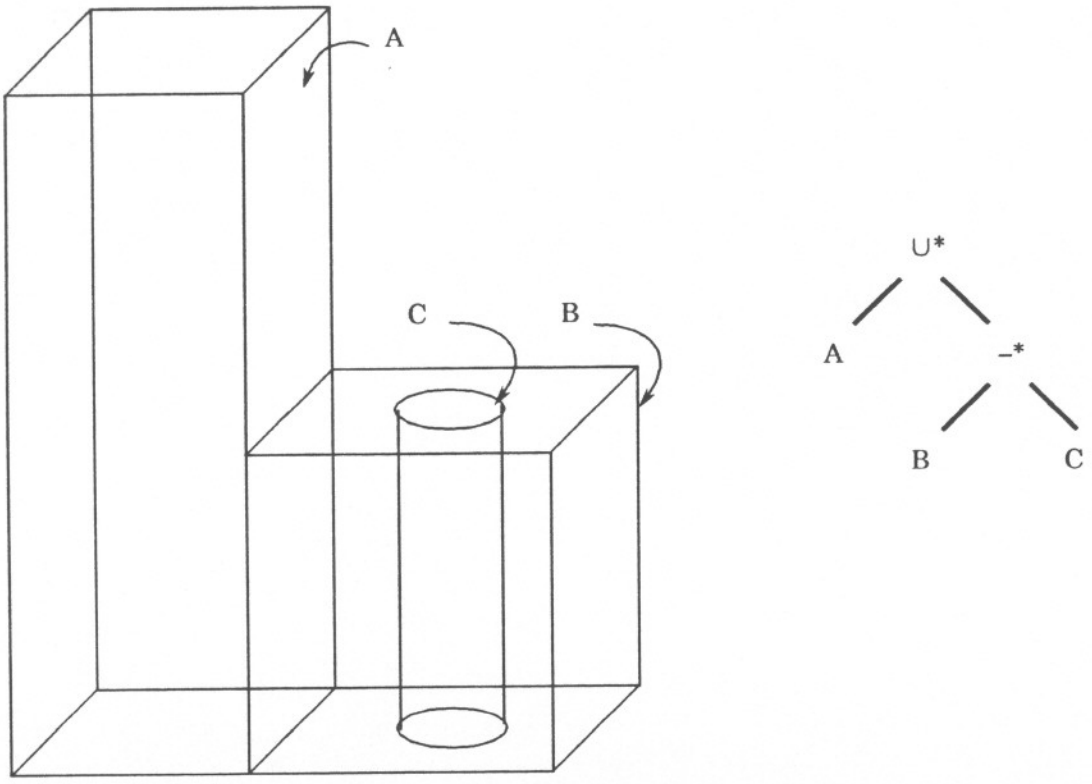


Figure 5. An Object Defined with a CSG Tree

operations (union, intersection, and difference). This is a commonly used representation. One reason for its popularity is the ease of textual specification. The syntax of CSG expressions is similar to that of arithmetic expressions, and can be described by simple grammars, making parsing easy. (Chapter VI describes an implementation of such a parser using the UNIX tools *yacc* and *lex*.) The parsed expression is often represented with a tree structure, and the term *CSG tree* is often used.

Figure 6 shows an example of an object defined with a CSG tree. Another

feature of the CSG representation is that it permits a unified method for answering queries: ray-casting [Roth82] (discussed below in Section "Set Operations"). This permits the 3-dimensional problem to be reduced to a set of 1-dimensional problems that are easily solved. New primitives are easily added, since only the ray intersection test must be implemented. However, a large number of such rays may be necessary.

3. Boundary Representations -- Another popular representation, a boundary representation (or B-rep) defines a solid by its enclosing surfaces (planes, quadric surfaces, parametric patches, etc.).

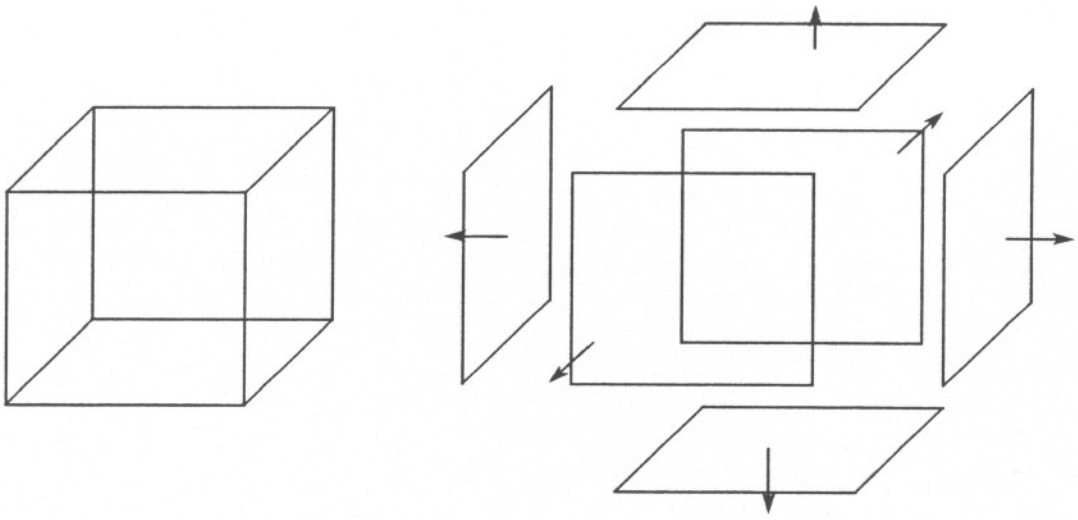


Figure 7. A Boundary Representation of a Cube

Representing a solid by a set of polygons on its boundary is the representation used as input to most algorithms to generate visible surface renderings (Figure 7).

The boundaries of a d-dimensional object are a set of (d-1)-dimensional polyhedra, called faces. These can in turn by described by their (d-2)-dimensional boundaries, continuing in this fashion until reaching 0D, the vertices of the object.

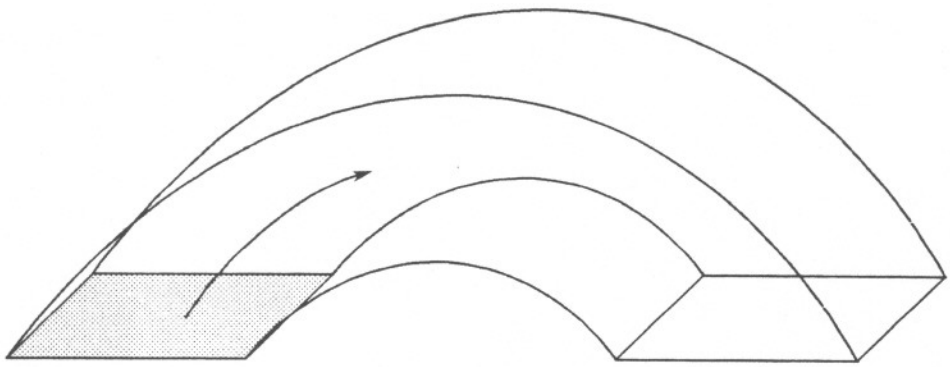


Figure 8. Sweeping a Quadrilateral Along a Curve

4. Sweep Representations -- A solid is defined by the moving a 2-d or 3-d object through space along some trajectory (Figure 8). Volumes of revolution can be defined by rotating a 2-d cross-section about some axis.

5. Procedural Modeling -- This method defines several generic objects (e.g., cones, spheres, prisms) and allows parametric specification of a particular instance. For example, regular prisms could be specified with tuples of the form ("PRISM", M, N, R), where M, N and R are the length, number of sides, and radius of the prism, respectively (Figure 9). A procedure specific to the class of object would be called with the appropriate parameters whenever the object is queried. The difficulty with this representation is that the algorithms to handle queries are difficult to write and are specific to the particular class of object, making interactions between objects of different

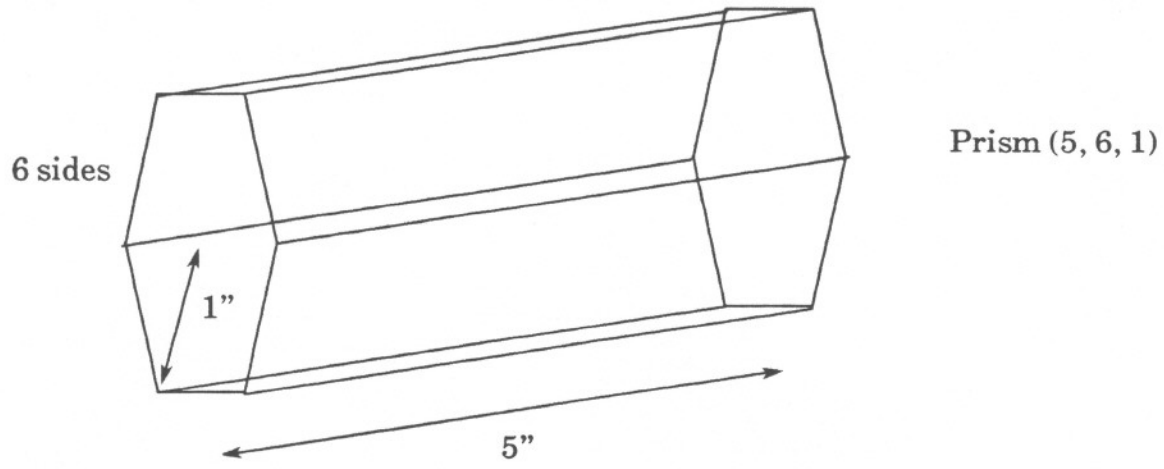


Figure 9. Parameters Define the Properties of the Prism

types problematical. Recently, a framework for handling interaction between procedural objects has been proposed [Ambu86]. Specific types of procedural objects, those characterized as "recursive subdivision" schemes, were addressed. This includes certain fractals [Four82] and parametric patches. Interactions based on spatial proximity of different objects are addressed in an object-based model, by defining certain operations that make use of the semantics of the shared subdivision scheme. This can be used to make a surface of one type conform to (lie on) a surface of another type: for example, a fractal model of a crinkled foil champagne wrapper is made to conform to the neck of a bottle modeled with parametric patches.

The BSP tree representation presented in Chapter III has the qualities of a cell decomposition, as well as those of a boundary representation. It can also be considered as a restricted form of CSG representation, in which each cell is described as the intersection of half-spaces, and the entire object as the union of these cells.

Representations for Interactive Solid Modeling

In an interactive setting, fast user feedback requires that property calculations and visible surface renderings can be done cheaply. For this reason, solids described with computationally "expensive" surfaces are approximated by solids described with simpler linear surfaces. This approximation is often used as a secondary representation. The two most prevalent approaches are the octree and polyhedral boundary representations.

Octrees

The *octree* [Jack80, Meag82] represents a hierarchical cell decomposition. It is the 3-d anologue of the quadtree [Same84]. At least one system [Requ83]) uses it as an auxiliary representation, constructing it from a CSG representation and using it for interactive operations. First, the representation is described.

The octree shares qualities with the spatial enumeration and cell decomposition representations. It is an 8-way tree representing a regular, hierarchical partitioning of space. (Figure 10) Each node of the tree corresponds to a cubical region of space, called a *voxel*. The root node describes the cubical universe. Each leaf of the tree is assigned a value describing the space in its region: "in" or "out" of the set being modeled. Each internal node of the tree has eight children, corresponding to splitting the voxel into eight equal-sized cubes. The children are ordered in a consistant fashion throughout the tree. The depth of the octree determines the size of the smallest voxel. The tree is constructed to a depth deemed sufficient to provide an adequate approximation of the solid. The representation has the advantages of simplicity, suitabilty to operations performed easily by computers (such as integer division by 2 (right shift), comparison), and regularity. Algorithms for rendering [Doct81] can easily generate multiple views of a given object. Rotation by ninety-degree angles can be performed by reordering the children of each node [Jack80]. Volume calculations are straightforward tree traversals. It seems best suited to representations of

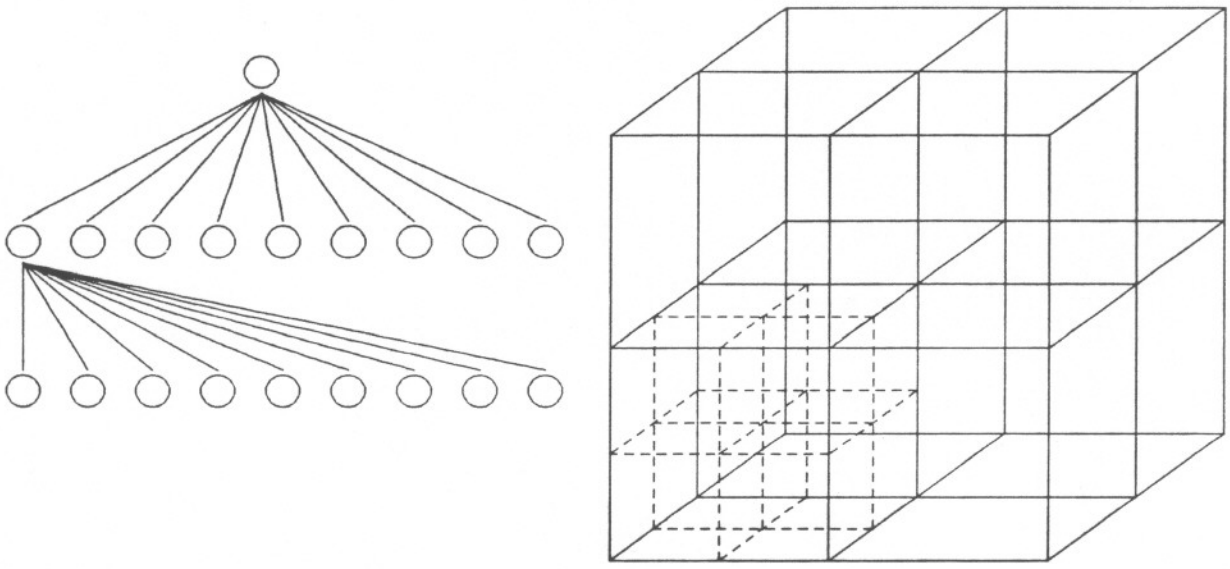


Figure 10. An Octree of Depth 2

complex, irregular objects such as human tissues reconstructed from digital data, such as CAT scans [Yau83].

Its major drawback is the amount of storage required to approximate surfaces that do not lie in "convenient" orientations, viz., nonplanar, not orthogonal to a coordinate axis or not positioned at distances attainable by a small number of divisions by two. Hunter and Steiglitz [Hunt79] show that the storage required is proportional to the surface area of the object. Also, since the partitioning always occurs along planes orthogonal to the axes at fixed positions, arbitrary rotations [Meag82] and translations [Jack80] require rebuilding the octree for the object at the new orientation.

Later work [Carl85, Carl87, Aya185, Nava86] extends the octree representation to address some of these shortcomings. Instead of just two leaf types, in and out, the types

"vertex," "edge," and "face" are defined. These leaves are associated with a boundary representation of the solid within the cell it represents. Although these extensions can reduce the storage required and allow polyhedra to be represented exactly, the simplicity of the representation is lost. We will see later (Chapter IV) that algorithms for set operations are also complicated considerably.

The simplicity of the octree has motivated at least one hardware implementation of viewing and set operation algorithms[John84].

Set Operations

When some operation is to be performed on a set described by a CSG tree, two approaches can be taken. One is to perform the operation using the CSG tree itself, applying the operation to the leaves directly, and for internal nodes, combining the results from its subtrees based on the set operation at that internal node [Lee82, Athe83, Roth82]. The other approach is to first convert the CSG representation to one that can be worked with more easily.

Ray-casting

By intersecting a ray with each primitive, the intersection of the ray and the object can be determined by combining the resulting 1D intervals according to the set operations[Roth82]. (Figure 11.) By generating a number of parallel rays arranged in a grid, a number of disjoint samples result. This reduces the 3D problem to a set of simpler 1D problems. However, this is only an approximation, and a large number of rays must be tested for an accurate result.

This method has the advantage of requiring only a single geometric operation to be supported for each primitive, that of intersecting the primitive with a ray. This allows the system to be easily extended with new primitives.

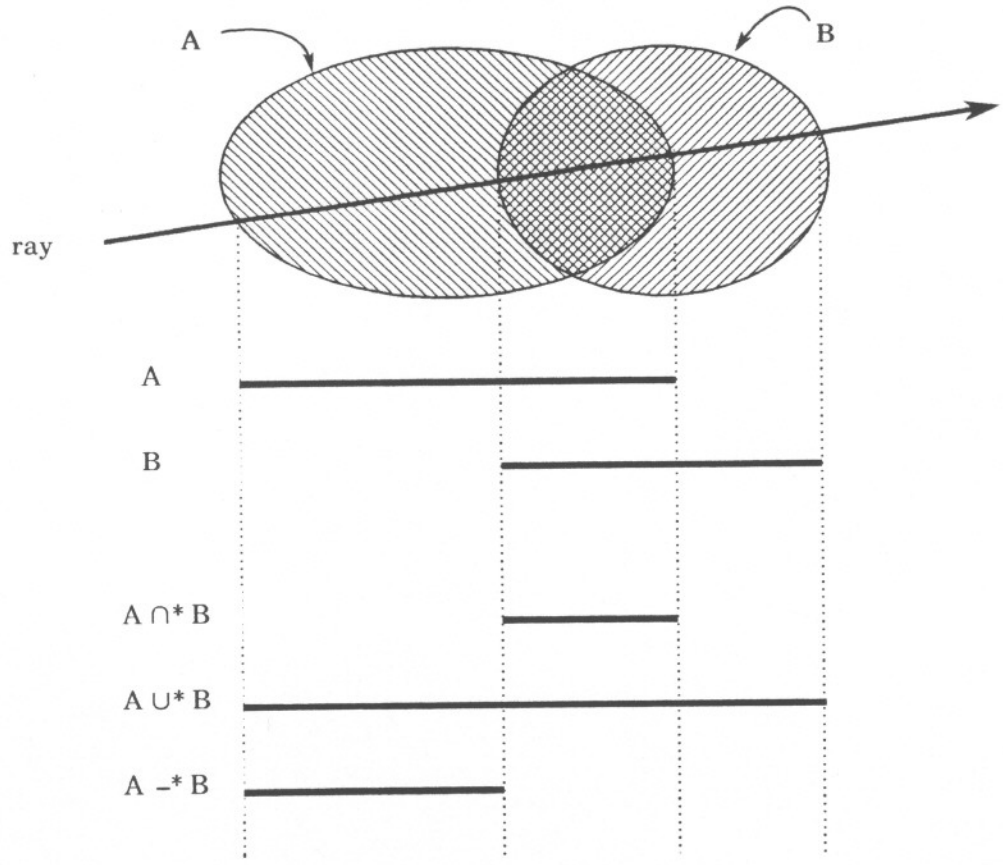


Figure 11. Ray-Casting to Solve a Set Operation.

When this is being used to generate visible surface renderings, the entire process must by repeated for each new viewing position. Atherton [Athe83] has presented a hybrid visible surface algorithm for polyhedral objects that mitigates this problem somewhat. It combines the ray-casting technique with the classical scan-line algorithm [Fole83]. The rendering is generated by considering each horizontal scan-line of the CRT raster. Ray-casting is used to determine which surface is visible at places where the visibility may change, such as at the intersection of two faces from different primitives. In regions where this does not change,

the linear nature of the polygons on the boundary allow a series of adjacent pixels to be written all at once. Although this approach is more efficient than casting a ray for each pixel, it shares that approach's drawback of requiring the entire process to be repeated for each view generated.

Evaluating Set Operations using Boundary Representations

The generation of several different visible surface renderings of a given object can be made more efficient if only the boundaries of the actual object are considered. That is, if the set operations are evaluated first, that work need not be repeated for each successive view. For example, the boundary of a 3D object defined with set operations on quadric primitives would consist of the 2D quadrics on the boundary, along with the 1D curves descibing the boundaries of these 2D surfaces. This is the approach taken in the GMSolid system developed at General Motors [Sarr83], and the work of Levin [Levi79, Levi76]. The resulting boundary is then used to generate hidden line renderings. Algebraic techniques can be used to determine the intersection curves of quadratic surfaces (as in the aforementioned systems), but for most higher-order surfaces, numerical techniques must be used.

The above approach has the drawback of requiring computation on non-linear entities. Techniques that attempt to avoid this first construct piecewise-linear, approximate (polyhedral) boundary representations of the primitives. The collection of faces of the primitives contains the boundary of the result as a subset. Finding intersections of linear faces (polygons) can be done with relatively inexpensive computations. The published algorithms for this CSG-to-boundary conversion[Mant82, Requ85, Laid86] share the same basic structure. Pairs of primitives combined with some set operation are considered, producing a new boundary representation that can then be combined with some other primitive, and so on, continuing until all primitives have been considered. First, intersections of faces are found. These are used to split the affected faces. Then, the faces are classified

TABLE 1. Dispositions of Faces Based on Set Operation

operation	*bd* A ∩* *int* B	*bd* A ∩* *ext* B	*bd* B ∩* *int* A	*bd* B ∩* *ext* A	*bd* A ∩* *bd* B
A ∪ B	discard	keep	discard	keep	same→keep, else discard
A ∩ B	keep	discard	keep	discard	same→keep, else discard
A - B	discard	keep	keep	discard	same→discard, else keep

as lying in the interior or exterior of the other primitive. Faces of the appropriate classification (depending on the set operation) are kept, the others discarded. (Table 11.) When faces of the two objects lie in the same plane and overlap, the orientation of the faces ("same" or "opposite") is used to determine which portions to keep.

An Example of an Algorithm for Set Operations

One such algorithm is now presented. The purpose of this section is to illustrate, using one of the relatively less complex of the published algorithms, how complex such an algorithm can be. In [Laid86], an algorithm is presented that performs a regularized set operation on two boundary representations of polyhedra (consisting of convex polygons), resulting in a boundary representation of the resulting polyhedra. The algorithm first splits any faces of either polyhedron that intersect the other polyhedron. This essentially requires testing each face of one polyhedron to each face of the other. Bounding boxes are used to quickly determine cases in which faces cannot possibly intersect. A bounding box gives the minimum and maximum extent of the object (here, a face) in all dimensions, thereby defining a box with axis-aligned sides. When the bounding boxes of two faces do not intersect, the enclosed faces do not intersect. Each face that intersects a face of the other polyhedron is split into subfaces that do not intersect the interior of the face of the other polyhedron. This is done by an exhaustive case analysis (17 cases) of the ways in which two faces may intersect. Vertices that lie on the boundary of both polyhedra are marked as such (BOUNDARY), and all others as UNKNOWN. Faces that are coplanar with a face(s) of the other polyhedron are marked SAME or OPPOSITE, according as the normal of the face is parallel or anti-parallel to the face of the other polyhedron.

Once this has been done, the polygons that have at least one UNKNOWN vertex are classified as lying in the interior or exterior of the other polyhedron. This is done by casting a ray from the center of such a face through the B-rep of the other polyhedron, and counting the number of times the ray intersects the boundary of the other polyhedron. If this number is odd, the polygon lies in the interior of the polyhedron. An even number of intersections indicates that the polygon lies in its exterior. If the ray does not intersect the other polyhedron, the ray is "jiggled" by some random perturbation. The new ray is again tested for intersection with the other polyhedron. This "jiggle-and-test" process is not guaranteed to terminate, but the authors report that one jiggle is usually all that is required in practice.

This classification, IN or OUT, is given to each UNKNOWN vertex of the face. Vertex connectivity information maintained in the boundary representation is then used to propogate this classification to UNKNOWN vertices that are connected to the vertices of this face. The propogation stops at BOUNDARY vertices, and therefore does not cross into regions of the boundary that lie on the other side of the boundary of the other polyhedron. The process of classification by ray casting and subsequent propogation of classification continues until all UNKNOWN vertices are marked as IN or OUT. This propogation step eliminates the need for many repititions of the expensive ray cast.

Once all vertices of a face have been classified, the polygon itself is classified. The polygon's classification is that of any non-BOUNDARY vertex; these are ensured to all be of the same classification by the initial splitting step. Polygons of the appropriate classification are then maintained in the result; OUT and SAME polygons are retained for union, IN and SAME for intersection, IN from one and OUT and OPPOSITE from the other for difference.

This algorithm has several aspects that complicate its implementation. Handling the splitting of polygons in 17 different ways is one such aspect. Others are the implementation

of the ray-test, and maintenance of connectivity relationships in a shared-vertex B-rep. Also, the use of bounding boxes can cheaply eliminate from consideration certain non-intersecting faces, but the lack of structure on the collection of boxes keeps intersection testing a $O(n^2)$ operation: each face (or bounding box) of one object is tested against each face (or bounding box) of the other object. Bounding boxes, although relatively easy to test for intersection, often fit "loosely" around the bounded object, leaving a large amount of void space (space inside the box that is not part of the bounded object). The larger the amount of void space, the higher the probability that a positive test for box intersection does not indicate intersection of the bounded objects.

The above algorithm only works for 3D polyhedra. Putnam and Subrahmanyam[Putn86] present a technique that is independent of the dimension of the two polyhedra. Their approach is to describe each nD object as an embedding hyperplane and a set of (n-1)D objects that lie in this hyperplane and bound the object. This recursive definition terminates at 0D, the vertices. The algorithm is also recursive in dimension. Although elegant and general, no consideration for reducing the n^2 comparisons required in each dimension is mentioned.

Set Operations on Octrees

Given two octrees in the same coordinate system, the set operation involves traversing both trees in parallel. The root of each tree represents the cubical universe. All children of the root correspond to the same regions for both trees. This is due to the strong connection between the coordinate system and the placement of the partitioning planes of the octree. When the set operation algorithm finds that one of the operand trees is a leaf, the result tree for that region is either the tree rooted at the corresponding position of the other tree, or a leaf. (In either case, it is also possible that the result has all of its leaves complemented, replacing "empty" with "full," and vice-versa.) The simplicity of the

algorithm suggests a straightforward hardware implementation.

In the "extended octree" schemes[Nava86, Carl87], a similar parallel traversal is done, but the handling of leaves is more complex. Leaves of these extended octrees can contain a B-rep of the object within the region. Techniques similar to those used for set operations on B-reps must then be used. This complicates the algorithm significantly.

Geometric Search

The time and space complexity of several geometric searching problems have been studied (e.g., [Prep85, Bent79, Kirk83, Edah84]), but the results are not directly applicable to geometric modeling. This is due to the nature of the problems studied: the 'database' being searched in these methods is usually a set of discrete points in n-dimensions, and the queries are either 0-d points or n-d orthogonal parallelipipeds. In a geometric modeling system, the sets considered are continuous (solids, lines), and the queries can have arbitrary geometry. For example, a polyhedral representation of a robot arm might be used in the query, "Does the robot arm intersect any other object in its environment?"

The problem of searching a set of 1D values is well understood, having been the topic of much work over the years. In the common case of searching a list of numbers ('find the smallest entry greater than or equal to some value'), we can use the fact that there exists a total ordering on numbers to construct, for example, a binary search tree. By comparing the key (the value being searched for) to the value stored at an internal node of the tree, we can eliminate from consideration all elements stored in one subtree. Higher-dimensional techniques must use geometric properties to define ordering relations and searching strategies.

Applying the *divide-and-conquer* paradigm used in binary search trees to the problem of search in an n-dimensional space requires the introduction of the notion of *space partitioning*. The space in which a model lies can be partitioned into a set of subspaces by

geometric entities, called *partitioning sets*. These partitioning sets can be any surface that partitions a space into two sets such that given any continuous curve connecting two points in each of the partitions, that curve must intersect the surface. The objective is to reduce our problem to asking questions about the objects in individual subspaces of the partition.

All previously published set operation algorithms that operate on boundary representations are essentially similar, in that sets of faces corresponding to all classifications of one object with respect to the other are generated, and then the appropriate ones kept and the others discarded, based on the particular set operation being performed. The difficulty with the algorithms of [Laid86] and [Requ85], for example, is that each face of each object must be tested for intersection with all faces of the other object.

Tilove[Tilo84] describes several methods of reducing the amount of this work. The basic idea behind these and all such methods is to decompose the problem into subproblems of restricted spatial extent. The restricted problems are simpler than the original. Also, it seems that boundary-based algorithms are inherently complex, both conceptually and in implementation.

One method proposed by Tilove for CSG evaluation is to restrict each subproblem to the region occupied by some primitive in the CSG tree. In an initial pass, all primitives are tested for pairwise intersection. This information is then used to "prune" the CSG tree describing the region occupied by each primitive. If a primitive A lies entirely in the exterior of a primitive B, then the position occupied by A in the CSG tree for the region occupied by primitive B is replaced by a representation of the empty set. If A is indeed disjoint from B, and is combined with some CSG subtree via the intersection operator (at A's parent), then A's parent can be pruned, discarding A and its sibling subtree and replacing A's parent with a representation of the empty set. (Although not mentioned by Tilove, a similar simplification can be performed when a primitive lies wholly in the interior of another.

Figure SIMPLIFY in chapter IV summarizes these simplification rules.)

Another method described by Tilove is the use of a regular grid to define the subproblems. This has the advantage of partitioning the problem into disjoint subproblems. Also, testing for intersection is made simpler by virtue of the regularity and orthogonal orientation of the grid cells.

Mantyla and Tamminen [Mant83] describe a hierarchical structure to define subproblems. They address the problem of evaluating a single binary set operation. In this setting, the problem can be framed in terms of geometric searching. The faces of one object are stored in a hierarchical data structure mirroring a hierarchical decomposition of space by axis-aligned planes. Non-trivial subproblems, that is, those in which neither operand is entirely in the exterior or interior, are identified by a geometric search of this data structure. The "search key" consists of the faces of the other object in the set operation. Geometric comparisons of these faces with the partitioning planes of the spatial decomposition are used to direct the search through the tree. Instead of comparing each face of each object to each face of the other, the comparison need only be made for the faces found to be in nearby regions of space. Requicha states [Requ80] that any realistic modeling system must provide some sort of geometric search structure.

Once a boundary representation, consisting of a set of polygons, has been determined, these can be used with any one of a number of rendering algorithms. (See [Roge85] for a survey of such algorithms.)

CHAPTER III

Representing Polyhedra with BSP trees

I represent that remark.

— Curly Joe

The idea upon which the BSP tree is based was originally proposed by Schumacker [Schu69]. The BSP tree was developed as a means of preprocessing polygonal models for efficient solution to the visible surface problem by scan-conversion[Fuch80, Nayl81, Fuch83], and was later extended to ray-tracing [Nayl86]. After formally introducing the BSP tree, we discuss how it is used to model polyhedra. Several algorithms are then discussed. First, the existing work on using the BSP tree for rendering is reviewed. Next, it is shown how the BSP tree can be used for one type of geometric searching problem, that of classification of an object with respect to the represented polyhedron, where the object is a point, line segment, or polygon. The concepts introduced by these algorithms are then applied to the problem of finding the boundary of a polyhedron represented by a BSP tree. Having shown how the BSP tree can be used, techniques to build the BSP tree from a boundary representation of a polyhedron are discussed. Finally, algorithms for computing the volume and center of mass of the polyhedron represented by a BSP tree are given.

Notation and Definitions

A *Binary Space Partitioning tree* (BSP tree) represents a recursive subdivision by hyperplanes of d-dimensional space. (Hyperplanes are planes in 3-d and lines in 2-d.) An example of a BSP tree in 2-d is given in Figure 12. In Figure 12(a), arrows indicate the orientation of (or, the normal vector to) each partitioning line. Each such line is associated with an internal node of the BSP tree in Figure 12(b). The right subtree of each internal

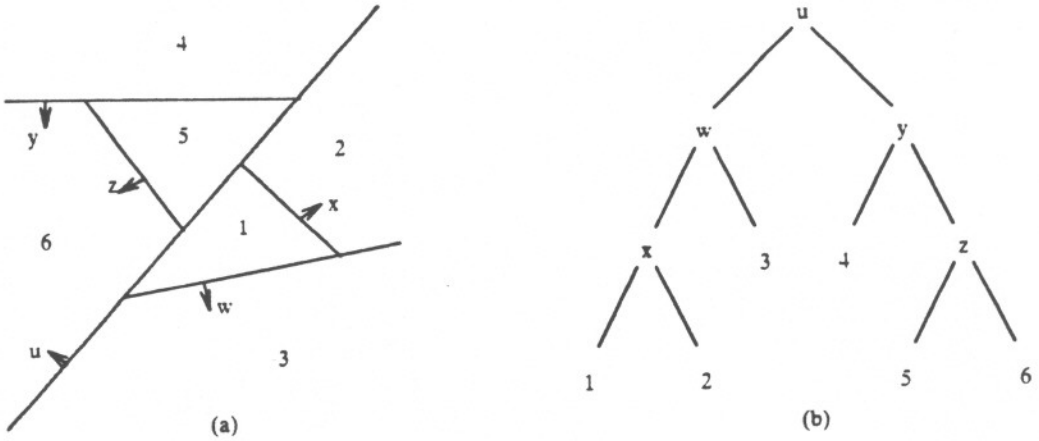


Figure 12. Geometry of a 2-d Partitioning (a) and its BSP Tree (b)

node represents the region of the plane lying to the side of the line pointed to by the arrow. The left subtree represents the other side. For clarity, leaves are numbered to correspond to the region each represents.

For a hyperplane $H = \{(x_1, \ldots, x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} = 0\}$, the halfspace lying "in front of" H is $H^+ = \{(x_1, \ldots, x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} > 0\}$, and the halfspace lying "behind" H is $H^- = \{(x_1, \ldots, x_d) | a_1x_1 + \cdots + a_dx_d + a_{d+1} < 0\}$). The "front" side of H lies to the side of H in the direction of the hyperplane's normal, $(a_1,\ldots,a_d)$.

Each node v of the BSP tree represents a convex region of d-space, $R(v)$ (defined below). Each internal node v of the tree is associated with a *partitioning hyperplane*, H_v, which intersects the interior of $R(v)$. The hyperplane partitions $R(v)$ into three sets: $R(v) \cap H_v^+$, $R(v) \cap H_v^-$, and $R(v) \cap H_v$. The d-dimensional region "in front of" or "to the positive side of" H_v is represented by the right child of v, *v.right*. The region "behind" or "to the negative side of" H_v is represented by the left child, *v.left*. The intersection of H_v and $R(v)$ is called the *sub-hyperplane* of H_v, $SHp(H_v)$, and is convex, of dimension d-1, and may

or may not be bounded.

Each $R(v)$ is the intersection of halfspaces defined by the path from the root to v. More formally, for each edge (v,v_2) in the tree, associate a halfspace $HS(v,v_2)$. If $v_2 = v.left$, then $HS(v,v_2) = \bar{H_v}$. Otherwise, $v_2 = v.right$, and $HS(v,v_2) = \dot{H_v}$. Let $E(v)$ denote the set of edges on the path from the root to v. Then, $R(v) = \bigcap_{e \in E(v)} HS(e)$. The root node represents all of d-space. Each $R(v)$ is *open* in the topological sense [Lay82]. The leaves of the BSP tree correspond to the un-subdivided polyhedral regions, called *cells*. We will use the terms "leaf" and "cell" interchangably, and our meaning should be made clear by the context. Cells are convex, and may or may not be bounded. A *trivial* BSP tree has no internal nodes, consisting only of a single leaf.

The above data structure, a binary tree with internal nodes labeled with plane equations, is called a *generic BSP tree*.

Often it is useful to *augment* the generic tree with additional information. To model polyhedra, the *labeled-leaf* BSP tree is used. Each leaf (cell) of a generic BSP tree is classified by labeling it as either *in* or *out*, according as the cell is entirely within or outside of the polyhedron. Each *in*-cell has a non-empty interior, by construction. The union of the closure of all *in*-cells therefore defines a regular set. A labeled-leaf BSP tree T is said to *represent* a regular set S when S is identical to the union of the closure of the *in*-cells of T. (Equivalently, T represents the set defined by the closure of the union of its *in*-cells.)

The labeled-leaf BSP tree can be further augmented with the boundary of the set it represents. Since each cell of a labeled-leaf BSP tree lies entirely in the interior or exterior of the polyhedron, any point on the set's boundary must lie in a sub-hyperplane. A *boundary-augmented* BSP tree associates a boundary representation of the (d-1)-dimensional boundary lying in a given sub-hyperplane with the internal node of the tree representing that

sub-hyperplane. In 3-d, the boundary in a given sub-hyperplane is a polygon. If a portion of the boundary lying in some sub-hyperplane is of dimension (d-2) or less, it is not used to augment the sub-hyperplane node. In other words, the portion of the boundary associated with a node is regularized in the dimension of the sub-hyperplane. Any such part of the boundary will be in the closure of one or more (d-1)-dimensional faces associated with some node whose sub-hyperplane is adjacent to the given sub-hyperplane. In the next section it is shown how such a boundary-augmented tree can be used.

Rendering

The BSP tree was originally used for the preprocessing of a set of polygons that described the boundary of some set of objects. This took the form of a boundary-augmented tree without labeled leaves. A boundary-augmented BSP tree can be traversed to define a *visibility priority ordering* on these polygons [Fuch80, Nayl81]. Two polygons, A and B, are related by the visibility priority ordering $A < B$, if, for a given viewing position, it is possible that polygon A obscures B, and that B cannot obscure A. Suppose we are given a boundary-augmented BSP tree. Consider an internal node, v, augmented with polygon P. Assume that the viewing position is located in the front halfspace of H_v. Then, for any polygon Q lying in the front halfspace, $Q < P$. Similarly, for any polygon R lying in the back halfspace, $P < R$. In other words, any polygon lying to the same side of a hyperplane as the viewing position could occlude some polygons lying in the hyperplane, which could, in turn, obscure some polygon on the other side of the hyperplane.

The tree can be traversed to yield the polygons in visibility priority order as follows. At each internal node of the tree, test the viewpoint against the partitioning hyperplane represented by that node. First, recurse on the "far" subtree (which is the left or right subtree according as the viewpoint is in front of or behind the plane), then output the polygons lying in the hyperplane, and then recurse on the "near" subtree.

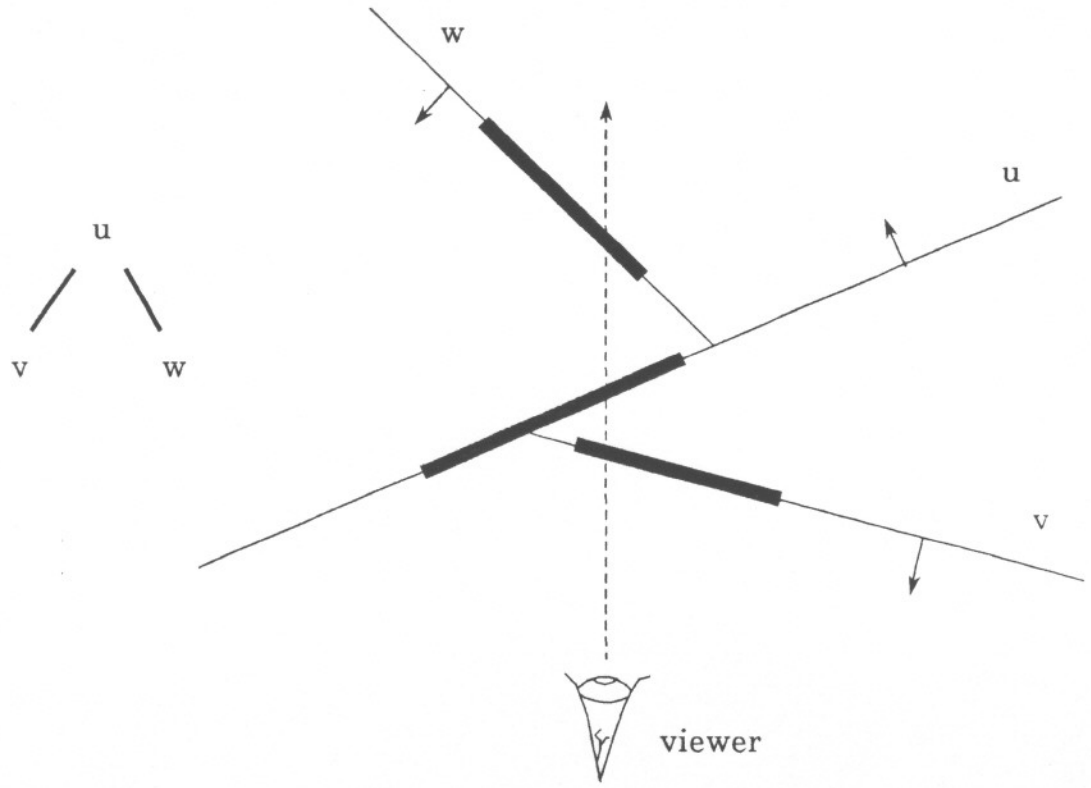


Figure 13. Illustration of Visibility Priority

(Figure 14.) The order in which the polygons are output is the visibility priority ordering, from lowest to highest. This ordering and can be used to drive the painter's algorithm. The painter's algorithm is based on the characteristics of the frame buffer (bit-map) display device. A frame buffer consists of a quantity of memory, enough to associate at least one bit with each displayable point on a television screen, and circuitry to read the memory during screen refresh, using memory contents to control the intensity of the electron beam(s). A new object will be displayed on the first screen refresh cycle to occur after the appropriate memory locations have been written. The usual memory write operation is destructive. Thus, if we write objects to memory in a back-to-front order (lowest-to-highest priority),

```
procedure Render_BSP ( v : BSPTreeNode; eye : point_3D );

        d := evaluate H_v at eye;
        If d > 0 then
                BSP_render ( v.left, eye );
                output v's embedded polygons;
                BSP_render ( v.right, eye );
        else
                BSP_render ( v.right, eye );
                output v's embedded polygons;
                BSP_render ( v.left, eye );

end; (* Render_BSP *)
```

Figure 14. Algorithm to generate a Visible Surface Rendering from a Boundary-Augmented BSP Tree

"close" objects will be drawn later, effectively obscuring previously written ("distant") objects. This is similar to the way a painter can obscure anything on the canvas by painting over it. (Reversing the order (i.e., first the near subspace then the far subspace) results in a front-to-back ordering.)

This algorithm has been used[Thib87, Fuch83, Gigu85] to generate visible surface renderings fast enough for interactive use. The BSP tree can also be rendered with ray-tracing, as discussed in Chapter V.

Classification algorithms

Three algorithms to determine the classification of an object with respect to a polyhedron represented by a labeled-leaf BSP tree are now presented. *Inserting* an object into a BSP tree involves determining which cells and sub-hyperplanes of the BSP tree contain the object. If the object lies in more than one cell, it is split into subsets such that each lies entirely within a cell. It may also be the case that the object or portions of it lie in a sub-hyperplane of the BSP tree.

The *classification* of a set X with respect to a set S consists of three subsets of X: $X \cap^{*\prime} int\ S$, $X \cap^{*\prime} ext\ S$, and $X \cap^{*\prime} bd\ S$, where the operation $\cap^{*\prime}$ is defined as the usual regularized intersection, except that the closure of the result is performed in the subspace X. In other words, the sets are regularized in the dimension of X. Classifying a set X with respect to another represented by a BSP tree T is done by inserting X into T. Portions of X lying in cells of T are classified according to the corresponding leaf label: *in* or *out*. Since no explicit information on the classification of sub-hyperplanes is maintained in a labeled-leaf BSP tree, portions of X lying in a sub-hyperplane of T are classified with respect to the subtrees v.left and v.right. The results of these classifications are combined to determine the classification of the portions in the sub-hyperplane. For example, a point in a sub-hyperplane that has an *in*-cell on one side and an *out*-cell of the other is on the boundary; a point with two *in*-cells on either side is in the interior.

Point classification

The *point classification* problem can be stated: "Given a set S and a point x, determine if x lies in the interior of S, the exterior of S, or the boundary of S." We assume S is regular and we have a BSP tree T representing S.

Figure 15 gives pseudo-code of a recursive algorithm. Note that since classifications are only associated with cells, the algorithm recurses on both subtrees when the point lies in a hyperplane H_T.

The running time of the algorithm is proportional to the size of the tree. The worst case tree would have all sub-hyperplanes having a common intersection. The worst case running time would occur when the point to be classified lies in this common intersection. In this case, both subtrees of each node encountered would be traversed. However, most points do not lie in any hyperplane. Classification of these points only requires traversing a path from the root to a leaf. The running time in these cases is then on the order of the depth of

```
procedure Classify_Point (x : Point;  v : BSPTreeNode) returns {in, out, on}

        if v is a leaf
                return the leaf's value (in or out)
        else
                d := dot_product(x, H_v).
                if d < 0 then
                        return point_classify (x, v.left)
                else if d > 0 then
                        return point_classify (x, v.right)
                else (* d lies in the partitioning plane *)
                        l := point_classify (x, v.left)
                        r := point_classify (x, v.right)
                        if l = r then
                                return r
                        else
                                return "on"
end;  (* Classify_Point *)
```

Figure 15. Algorithm for Point Classification

the tree.

Classifying a Line Segment

The line segment is represented by its endpoints x and y. To insert a line segment, the endpoints are tested against the hyperplane of the root node. If both endpoints lie to the same side of the hyperplane, the line segment cannot intersect the hyperplane, and the insertion recurses on the appropriate subtree. If the endpoints lie on opposite sides of the hyperplane, then the line segment intersects the hyperplane. The point of intersection p is calculated using the method presented in the Appendix. This new point p is used as the endpoint of two new line segments: (x,p) and (p,y). Each of these is then recursively classified with respect to the appropriate subtree. When a leaf is encountered, the current

```
procedure ClassifyLineSegment ( Segments: set of LineSegment;
                                v : BSPTreeNode )
                returns < LinS, LoutS, LonS : set of LineSegment >

  If v is a leaf then
      case v.label of
          in: return < P, ∅, ∅ >
          out: return < ∅, P, ∅ >
  else
      for each L in Segments do
          < L_left, L_right, L_coincident > ∪= partition L by H_v
      if L_left ≠ ∅ then
        < L_left_inS, L_left_outS, L_left_onS > := ClassifyLineSegment (L_left, v.left)
      if L_right ≠ ∅ then
        < L_right_inS, L_right_outS, L_right_onS > := ClassifyLineSegment (L_right, v.right)
      if L_coincident ≠ ∅ then
        < co_inL, co_outL, co_onL > := ClassifyLineSegment (L_coincident, v.left)
        < co_inLinR, co_inLoutR, co_inLonR > := ClassifyLineSegment (co_inL, v.right)
        < co_outLinR, co_outLoutR, co_outLonR > := ClassifyLineSegment (co_outL, v.right)

      LinS  := L_left_inS ∪ L_right_inS ∪ co_inLinR
      LoutS := L_left_outS ∪ L_right_outS ∪ co_outLoutR
      LonS  := L_left_onS ∪ L_right_onS
          ∪ co_inLoutR ∪ co_inLonR ∪ co_outLinR ∪ co_outLonR
      return < LinS, LoutS, LonS >
end (* ClassifyLineSegment *)
```

Figure 16. Algorithm for Line Segment Classification

line segment is classified according to the leaf's label.

If both endpoints lie in the hyperplane, the segment must be classified with respect to both subtrees. This is done by first classifying the segment with respect to the left subtree. This returns three sets of line segments, corresponding to "in", "out", or "on" classifications. "On"-segments are on the boundary of the set, and need not be classified with the right subtree. The "in"-segments are classified with respect to the right subtree. Those segments from this second classification that are "in" lie in the interior of the set, those that are "out" or "on" are on the boundary of the set. Similarly, the "out" segments resulting from the classification with respect to the left subtree are classified with respect the the right subtree.

Those segments from this second classification that are "out" are in the exterior of the set, and those that are "in" or "on" are on the boundary. (Figure 16.)

The result of the insertion process is 3 sets of line segments, those *in*, *on*, or *out* of the set. Each set in turn consists of subsets reflecting the geometric relationship between the original line segment and the BSP tree, as manifested by the insertion process.

Classifying a Polygon

Given a labeled-leaf BSP tree T representing a set S, and a representation of a polygon P, P is to be classified with respect to S. Insertion of P into T is again the basic operation. P is represented by an ordered list of vertices $(p_1, p_2, \ldots, p_n)$, such that there is an edge between each consecutive pair of vertices modulo n. At a node v of T, the polygon is partitioned by H_v. First, each vertex of P is compared to H_v. If the polygon has vertices on both sides of H_v, the partitioning operation splits the polygon into two parts. This essentially involves partitioning the line segments describing the edges of the polygon. If the polygon is not split, the partitioning operation determines if the polygon lies in front of, behind, or in hyperplane H_v. The portion lying in H_v^+ are recursively inserted into v.right, and the portion lying in H_v^- into v.left. Figure 17 shows the algorithm to partition a polygon. Note that the partitioning operation finds three regularized intersections: $H^+ \cap^* P$, $H^- \cap^* P$, and $H \cap^* P$, where regularization is in the dimension of P. The algorithm to partition a polygon is basically the same as those used to "clip" a polygon to the sides of a viewing window (see[Fole83]). The difference from such a clipping algorithm is that *both* parts of the result of a split are retained. The algorithm shown handles convex polygons. Extension to concave polygons is straightforward.

In the pseudo-code in Figure 17, the function TestVertex (v,H) returns 0 when $v \in H$, 1 when $v \in H^+$, and -1 when $v \in H^+$.

```
procedure SplitPolygon ( H : hyperplane;  P : polygon ) returns <LeftPoly, RightPoly, CoPoly : polygon>
        in H⁻ := in H⁺ := false
        prev_vertex := last vertex in P's list
        last_state := TestVertex ( prev_vertex, H )
        current_vertex := first vertex in P's list

        for i := 1 to number of vertices in P do

           next_vertex := the vertex after current_vertex in P's list
           state := TestVertex ( current_vertex, H )

           if state≠last_state and state≠0 and last_state≠0 then

                 in H⁻ := in H⁺ := true
                 p := point of intersection of H and the edge joining
                          prev_vertex and current_vertex
                 Add p to list for LeftPoly
                 Add p to list for RightPoly

           if state=-1 then

                 in H⁻ := true
                 Add current_vertex to list for LeftPoly
           else
                 if state=1 then

                    in H⁺ := true
                    Add current_vertex to list for RightPoly
              else
                    Add current_vertex to list for LeftPoly
                    Add current_vertex to list for RightPoly

           last_state := state
           prev_vertex := current_vertex
           current_vertex := next_vertex
        end for

        if not in H⁻ and not in H⁺ then
           LeftPoly := RightPoly := ∅
           CoPoly := Poly
        else
           CoPoly := ∅

        return <LeftPoly, RightPoly, CoPoly>

end  (* SplitPolygon *)
```

Figure 17. Algorithm to Partition a Polygon

As the insertion progresses, the polygon eventually makes its way into some set of leaves and sub-hyperplanes of T. The portions of the polygon lying in a cell of T can be classified with respect to S based on the cell's label. The portions lying in sub-hyperplanes of T are classified in a manner similar to that used to classify a point or line segment lying in a

subhyperplane. The algorithm is identical to that for line segment classification, with one modification: code for partitioning the polygon with the hyperplane replaces that for partitioning a line segment. Also note that the classification of a coincident polygon (i.e., one lying in the sub-hyperplane of node v) with respect to v.left will never return an "on" result.

Generating the boundary of a labeled-leaf BSP tree

Given a labeled-leaf BSP tree, it is possible to augment it with polygons constituting its boundary. This is done for each internal node of the tree by first producing a polygon representing the subhyperplane H_v, $SHp(H_v)$. This polygon is then classified with respect to the tree rooted at v, using the method of the previous section.

To construct a polygon representing the sub-hyperplane of H_v, a "maximally finite" representation of the hyperplane is first constructed, i.e., a polygon large enough to be considered unbounded for all practical purposes, but still within the limits of the resolution of computer arithmetic. This polygon is found by orthogonally projecting, along the x_i axis, a maximally finite representation of the $x_i=0$ plane into H_v. The axis of projection, x_i, is chosen to be the largest component of H_v's normal. (Figure 18.) For simplicity of presentation, assume that the axis of projection is the z-axis. The vertices of the polygon representing the $z=0$ plane are (M,M,0), (-M,M,0), (-M,-M,0), and (M,-M,0), where M is chosen to be the fourth root of the largest representable floating point value, in order to ensure arithmetic over/underflow does not occur in subsequent operations. If the plane equation of H_v is $ax+by+cz+d=0$, then the coordinates of the projection of vertex $(x',y',0)$

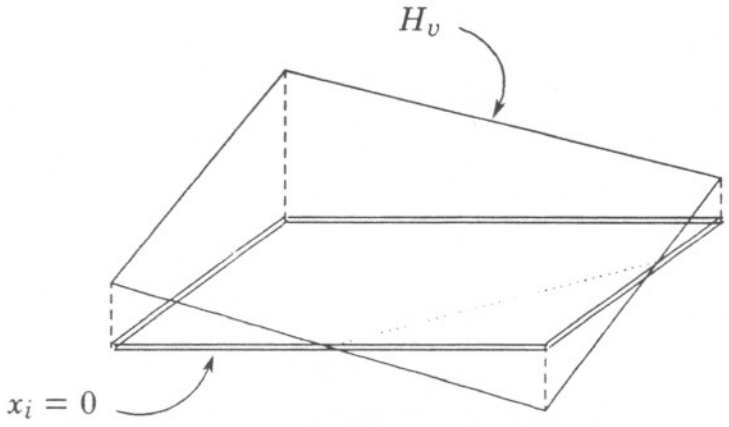


Figure 18. Projecting a Representation of $x_i = 0$ onto H_v.

```
procedure Generate_subhp ( v : BSPTreeNode ) returns Polygon

        I := index of largest component of Hv's normal
        S := projection along xI axis into HV
                of "maximally finite" rep. of the xI = 0 plane

        for each e ∈ E(v) do
                S := S ⋂ HS(e)

        return S

end  (* Generate_subhp *)
```

Figure 19. Algorithm to Generate a Polygon Representing the Sub-Hyperplane of H_v is $(x', y', (-ax-by-d)/c)$.

Having found a representation of the hyperplane, it is then transformed into a representation of the sub-hyperplane. This is done by inserting the polygon into the tree along the path from the root to node *v*. Recall that the sub-hyperplane is the intersection of H_v and R(v), and that R(v) is bounded by partitioning hyperplanes on a path from v to the root of the BSP tree. The hyperplanes on this path to the root successively clip the polygon. The intersection of the polygon with the appropriate halfspace is retained at each stage; specifically, if v is in the left (right) subtree of an ancestor w, we clip the polygon to the back (front) halfspace of H_w. (Figure 19.)

Once this clipping is complete, a polygon representing the sub-hyperplane of H_v remains. The polygon is now classified with respect the subtrees of v, as in the previous section. To maintain the convention of outward-pointing normals, boundary polygons "in" v.left and "out" of v.right are oriented so their outward-pointing normals point in the same direction as H_v's normal, and the boundary polygons "out" of v.left and "in" v.right are oriented in the opposite direction.

The boundary polygons could also be generated with two classifications and a "glue." The subhyperplane representation is inserted into v.left, generating two lists of polygons, corresponding to *in*- and *out*-cells of v.left. A copy of the subhyperplane representation, with its orientation reversed, is then inserted into v.right, generating two more lists of polygons. The two lists of polygons lying in in-cells on either side of the hyperplane are then "glued" together (as discussed in Chapter IV), which "cancels out" polygons that overlap and are of opposite orientation. This has the effect of eliminating any polygons in regions bounding in-cells on both sides. We are then left with polygons bounding in-cells on one side, out-cells on the other, and hence on the boundary.

Constructing BSP trees from boundary representations

The algorithms presented below for constructing BSP tree representations of regular

sets use boundary representations of the input polyhedra, but are independent of the particular B-rep used. For purposes of discussion, 2-d polygons are represented as lists of vertices, and 3-d polyhedra as sets of 2-d polygons. Polygons are consistantly oriented, so that normals point to the exterior of the solid. The techniques handle arbitrary polyhedra (including concave polyhedra with holes). The boundary representation is assumed to be *valid*, defining a closed surface. As much as is possible, the algorithms are discussed independently of the dimension, d, of the objects being modeled. The term *face* refers to the (d-1)-dimensional boundaries of a d-polyhedron. The (d-1) dimensional hyperplane which embeds a face f and has f's orientation is denoted by H_f. The interior, exterior, and boundary of a set S are denoted by *int* S, *ext* S, and *bd* S, respectively.

Given a boundary representation of a convex polyhedron S, a BSP tree representation of S is constructed as follows. Recall that any convex set can be represented by the intersection of halfspaces. A simple BSP tree for a convex set is then a list of hyperplanes. Each internal node has one "out" child. The last node in the list has one "out" and one "in" child, with R(the "in" child) = *int* S. (Figure 20.) An algorithm to construct this tree from the boundary representation of S would remove faces from S one at a time, essentially replacing each with a hyperplane that embeds the face.

For concave polyhedra, some faces may have to be split. An algorithm to construct a BSP tree from an arbitrary boundary representation of S is given in Figure 21. To "partition faces of S with H," each face in the set S is tested against the hyperplane H. If H splits a face, representations of the two parts are returned in left_S and right_S. Faces returned in left_S (right_S) have no vertices in the front (back) halfspace of H. Faces returned in coplanar_S lie entirely in H. Figure 22 shows a concave polygon and the BSP tree output by the algorithm.

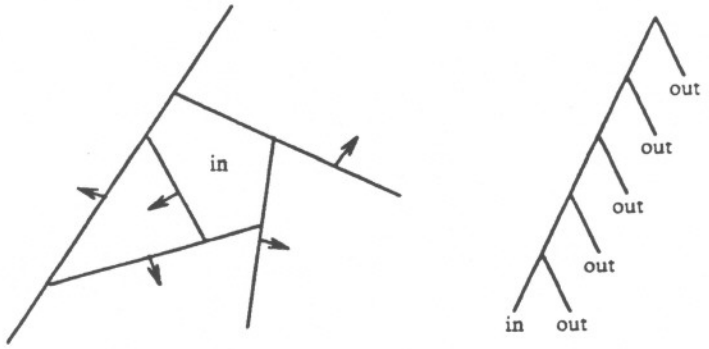


Figure 20. A Convex Set and its BSP Tree

Procedure Build_BSPT implements essentially the same algorithm as is used to preprocess polygons for visibility priority ordering [Fuch80, Nayl81], with one significant addition. When a partitioning operation finds no polygons to one side of the partitioning plane, that region lies either entirely within the interior or the exterior of the object, and the resulting leaf is classified as "in" or "out". Procedure Build_BSP can easily differentiate between the two cases because hyperplanes are chosen that embed faces, and faces have outward-pointing normals.

When a partitioning of the polyhedron's faces results in a region containing none of the polyhedron's faces, it must be the case that all faces lying in the splitting sub-hyperplane have the same orientation. The homogeneous region must be either *in* or *out*. A face lying in the hyperplane with one orientation would indicate that the region was *in*, while a face with the opposite orientation would indicate it was *out*. If faces of both orientations lie in the hyperplane, the homogeneous region would be both *in* and *out*, a contradiction.

All boundary points of a polyhedron represented by a BSP tree lie in sub-

```
procedure Build_BSPT ( S : set of faces ) returns BSPTreeNode

        Choose a hyperplane H that embeds a face of S;
        new_BSP := a new BSP tree node with H as its partitioning plane;
        <left_S, right_S, coplanar_S> := partition faces of S with H;
        append each face of coplanar_S to the appropriate face list of new_BSP;

        if (left_S is empty) then
                if (coplanar_S has the same orientation as H) then
                        (* faces point "outward" *)
                        new_BSP.left := "in";
                else      new_BSP.left := "out";
        else
                new_BSP.left := Build_BSPT ( left_S );

        if (right_S is empty) then
                if (coplanar_S has the same orientation as H) then
                        new_BSP.right := "out";
                else      new_BSP.right := "in";
        else
                new_BSP.right := Build_BSPT ( right_S );

        return new_BSP;

end; (* Build_BSPT *)
```

Figure 21. Algorithm to Build a BSP Tree from a Boundary Representation

hyperplanes, but not all sub-hyperplanes must contain faces. More balanced trees can be constructed by allowing use of hyperplanes that do not embed faces, but, for example, split the polyhedron into two equal-sized sets. When partitioning the boundary with a hyperplane that does no contain a face, it may happen that one side of the hyperplane winds up with no part ot the boundary. This is the point at which an *in* or *out* value is to be assigned to the resulting leaf of the BSP tree. However, the technique of comparing the embedded face's normal to the hyperplane (as in Procedure Build_BSPT) cannot be used. The portion of the boundary lying to the other side of the hyperplane can be examined to determine the leaf's value. This is discussed in Chapter IV under "In/Out Testing."

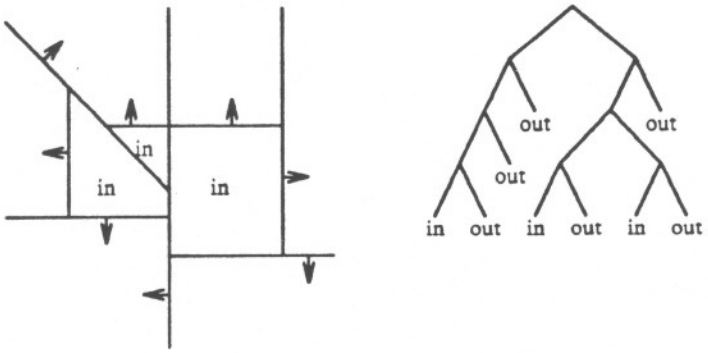


Figure 22. A concave set and its BSP tree

Metric properties

In order for a model of a geometric entitity to be useful, it should be possible to use it to answer almost any geometric query automatically. In previous sections, algorithms for classification and boundary generation were presented. In this section we discuss algorithms for determining volume and center of mass.

Volume

The volume of a polyhedron represented by a BSP tree is the sum of the volumes of the in-cells. Each in-cell is a convex polyhedron. Two algorithms for computing the volume of a convex polyhedron are given in [Cohe79]. These algorithms require a boundary representation of the convex polyhedron as input. This is not directly available from the BSP tree representation, in which the cell is described by the intersection of halfspaces on the path from the root of the BSP tree to the leaf representing the cell. To use these algorithms, then, a method of determining the boundary representation of a cell is needed.

In particular, the algorithms require the locations of the vertices of the polyhedron

and the topological information as to how these are grouped into edges and faces. Geometric information about the edges and faces (e.g., line or plane equations) are not required.

A modification of the technique for boundary generation from a labeled-leaf BSP tree can be used to generate a boundary representation for the cell corresponding to leaf of a BSP tree. First, a representation of the sub-hyperplane of each node on the path to the leaf is generated, using the technique described in that section. This involves clipping a representation of the entire hyperplane successively to halfspaces defined by ancestor nodes. Next, each sub-hyperplane is clipped to the halfspaces defined by descendant nodes on the path to the leaf of interest. This results in a set of boundary representations of faces of the cell. It remains to coalesce these faces, merging vertices and edges shared by more than face, producing a graph representation of the polyhedron. This can be done by comparing the position of each vertex of a given face to each vertex of the other faces.

Lasserre [Lass83] gives an algorithm for the volume of a convex polyhedron that uses an input representation that is more convenient for use with BSP trees. The polyhedron is represented as the intersection of half-spaces, specifically, as a system of linear inequalities. The algorithm handles "redundant constraints," i.e., half-spaces that do not share part of their boundary with the boundary of the polyhedron. This makes it ideally suited for the purpose of finding the volume of a cell corresponding to a leaf of a BSP tree. The half-spaces defined by the path from the root of the tree to this leaf constitute the input to the algorithm.

The algorithm is based on an algebraic formulation of the volume of a d-dimensional convex polyhedron in terms of the (d-1)-dimensional volumes of its (d-1)-dimensional faces. The technique essentially finds the volume of the i-th (d-1)-face by first eliminating the i-th variable from the system of inequalities (by back-substitution). The resulting system defines a (d-1)-dimensional polyhedron, whose volume is found recursively. Recursion is terminated

when back-subtitution is no longer possible. The author's state that a symbolic algebra program can be used to construct an analytic expression for the volume of a given polyhedron in this manner.

A major advantage of this algorithm is that it does not require constructing the graph representation of the polyhedron, an intricate process that can be difficult to implement. It also determines redundant halfspaces, those that do not constitute a face of the polyhedron.

A technique for *approximating* the volume is the *Monte Carlo* technique. This involves testing a randomly selected set of points for inclusion in the object, and basic the volume estimate on the percentage found to lie in the object. An algorithm for point classification was given in a previous section. This approach may be attractive in environments where point classification is inexpensive, as may be the case for a vector (pipelined) processor. In such a setting, performing the same operation on long vectors (arrays) of data is very efficient.

Another approximation technique, *ray-casting*, intersects a set of regularly spaced parallel lines with the object. By considering each line segment in which a line intersects the object as a parallelepiped of fixed cross-section (determined by the spacing of the lines), the aggregate volume of these parallelepipeds approximates the volume of the object. An algorithm to compute the intersection of a line (ray) with an object represented by a BSP tree is given in a subsequent chapter.

Center of Mass

The structure of the BSP tree allows for a simple algorithm for center of mass. Assume the center of mass of a convex polyhedron (cell) can be computed. (This could be done with a modification to Cohen and Hickey's algorithm, making use of the fact that the center of mass of a tetrahedron is the barycenter of it's vertices.) Assume also that the polyhedron is of constant density.

The algorithm for computing the center of mass of a polyhedron represented by a BSP tree is a recursive, post-order traversal. To find the center of mass of the portion of the object represented by a node v of the tree, first find the center of mass and volume of each subtree of v. We can consider the sets represented by the subtrees of v as point masses, located at their respective center of mass, with mass proportional to their respective volumes. The volume of the set represented by the tree rooted at v is the sum of these voluems, and the center of mass is located along the line segment connecting the two point masses. The position of the center of mass along this line is determined by the relative masses of the two points. If the points are P_1 and P_2, with masses M_1 and M_2, respectively, then the center of

$$\text{COM} = \alpha P_1 + (1-\alpha) P_2, \quad \textit{where } \alpha = \frac{M_1}{M_1 + M_2}.$$

While the above gives an elegant solution, a more efficient method would simply find the weighted average of the barycenters P_i of the cells, where the weight of each cell is its

$$COM = \frac{\sum_{\textit{cells } i} P_i M_i}{\sum_{\textit{cells } i} M_i}$$

Discussion

BSP tree representations of a set are not unique: a given set may be represented by more than one BSP tree. This is suggested by the fact that any hyperplane can be used to partition an as-yet un-subdivided region, as long as it intersects the interior of the region. This fact leads one to ask which representations are "better" than others. The answer depends on the use which is to be made of the tree. For geometric searching applications in which a path from the root to a leaf is followed, such as ray-tracing, a minimum height tree may be desired. For applications that traverse the entire tree, such as visibility priority ordering, a tree with a minimum number of nodes may be more appropriate. Generating all possible trees and choosing the best one is too expensive. Naylor[Nayl81] has shown the

number of trees to be $\Omega(n!)$, where n is the number of faces of the polyhedron. Heuristics are used that try to choose partitioning planes at each stage of tree construction that will result in a final tree with the desired characteristics. Another result in Naylor's thesis concerns the time complexity of the process of constructing a BSP tree from a B-rep. The result assumes that $O(m^p)$ time is taken to choose each hyperplane, where m is the number of polygons considered at a given stage. (Since each polygon must be considered in the course of partitioning, p can be no smaller than 1.) It is also assumed that half (or fewer) of the polygons are split at each stage. This increases the total number of polygons by 50 percent at each stage. Letting n be the number of faces in the input, it is shown that under these assumptions the process is $O(n^p \log n)$. The question remains as to whether or not the assumption that half or fewer of the polygons are split at each stage is reasonable. The answer would require arguments based on the geometry of polyhedra, and is left for future research.

There are trade-offs to be made between the amount of work to be spent in building the tree and the expected return on this investment as evidenced by improved performance, accrued over the life of the tree. We discuss the behavior of several heuristics in Chapter VI.

This BSP tree representation is in some repects similar to the octree. Both are tree structures with leaves marked to represent membership. The octree partitions with planes orthogonal to the coordinate axes, which makes testing of a point with respect to a plane a simple comparison, but also makes for verbose and approximate representations of faces of polyhedra that are not axis-aligned. The BSP tree permits the use of partitioning planes of any orientation. This requires a dot-product computation to test a point with respect to a plane, but allows us to succinctly and exactly represent arbitrary polyhedral regions. Of course, axis-aligned planes can be used in the BSP tree when desired. We can apply a transformation to a BSP tree by transforming each hyperplane equation. Transforming an octree representation requires rebuilding the tree, since the partitioning planes must remain

axis-aligned, placed at regular subdivisions of the size of the universe.

CHAPTER IV

Set Operations

Leave off fine learning! End the nuisance of saying yes to this and perhaps to that, distinctions with how little difference!

— Lao Tzu

This chapter presents algorithms to produce a labeled-leaf BSP tree representing a set theoretic expression on polyhedra. Specifically, two cases are addressed. The first involves a single binary set operation, in which one operand is represented by a labeled-leaf BSP tree and the other by a boundary representation. The result is obtained by modifiying the input BSP tree, and so is called an *incremental* set operation. The second takes as input an arbitrary set theoretic expression (CSG tree) with boundary representations as primitives, and outputs a labeled-leaf BSP tree representing the set. This algorithm is called "CSG evaluation." The algorithms are then extended to operate on operands represented by BSP trees, in which each internal node is augmented by a boundary representation of its sub-hyperplane.

Any system for interactive modeling of polyhedra with set operations must solve three basic problems: efficient spatial search, modification of the model, and visible surface rendering. Most existing algorithms, some of which are described in a previous chapter, all suffer the drawback that the data structures and algorithms for these crucial aspects are distinct and unrelated. For example, spatial search may be handled with a data structure that is not an integral part of either the model modification or visible surface sub-problems. The one case that does unify these aspects to some extent are the octree-based schemes proposed independently by Carlbom[Carl87] and Navazo[Nava86]. However, these methods have the

drawback of requiring complex boundary-based algorithms at the boundary nodes (i.e., modification of the model is not fully unified with the other aspects). The techniques presented in this chapter address these three issues within a unified framework, provided by the BSP tree. This unification serves to reduce the complexity of the task of building such a system. This simplicity manifests itself in reducing the amount of programming that must be done. Although more efficient solutions to each of these problems may possibly exist, the unity and simplicity of the BSP tree certainly helps to offset this.

Concepts Relating to the Algorithms

Set operation algorithms may be considered to be *representation conversion* algorithms. An expression like A $\cap^*$ B, where A and B are represented as, say, boundary representations, is itself a representation of the set it denotes, the regularized intersection of object A and B. This representation may be inconvenient, however: it may take too much space, may involve large amounts of computation and complex algorithms to answer geometric queries, etc. Converting such an *implicit* representation to an *explicit* one, such as a single boundary representation or a BSP tree, may therefore be desired. The conversion from a set theoretic expression to a single data structure (the BSP tree in this case) is called *set operation evaluation*, and is a special case of the class of representation conversion algorithms.

Given a representation X, call the set it describes *Denotes*(X). A *set operation evaluation problem* is a pair (X,R), where X is a representation of a set theoretic expression, and R is a region of space. X is defined only within R. Overload the *Denotes* function to define *Denotes*(X,R) = *Denotes*(X) $\cap$ R. A *solution* to a set operation evaluation problem is a representation Y such that *Denotes*(Y, R) = *Denotes*(X, R). In a *complete solution*, Y is not a set theoretic expression but is a (component of a) data structure. In this context, a complete solution is a BSP tree.

A *partitioning* of a set operation evaluation problem (X,R) is a set of subproblems $\{(X,R_1), (X, R_2), \ldots, (X, R_n)\}$, where $\bigcup_i R_i = R$, and $\bigcap_{i \neq j} R_i = \varnothing$. If these subproblems have solutions Y_i, then the solution Y to the original problem is the union of these solutions: $Denotes(Y, R) = \bigcup_i Denotes(Y_i, R_i)$.

Partitioning is one technique that can be used to reduce a problem into simpler subproblems. In the context of set operations, simplification of a set theoretic expression is possible whenever one or more operands are found to be *homogenous* with respect to the region of interest R. A region R is homogeneous with respect to an operand S when no part of the boundary of S intersects R. Thus, homogeneity with respect to S implies that either $R \subseteq int\ S$, or $R \subseteq ext\ S$. In terms of the *Denotes* function, these cases correspond to $Denotes(S, R) = Denotes(U, R)$, where U is the universal set, and $Denotes(S, R) = Denotes(\varnothing, R)$, respectively. Once a region is found to be homogeneous with respect to all operands of a set theoretic expression, the region is homogeneous with respect to the entire expression. Determining if the region is wholly within or outside of the set represented by the expression is then a simple matter of evaluating a Boolean expression. The result is a complete solution to the expression within R.

To see how this simplification can be used to obtain a complete solution, consider the expression $A \cap^* B$ in the context of Figure 23. The region R_1 lies entirely in the exterior of B. Since the regularized intersection of two sets must contain points in the interior of both, we know that R_1 contains no part of $A \cap^* B$: $Denotes(A \cap^* B, R_1) = Denotes(\varnothing, R_1)$. Region R_2 lies entirely in the interior of B, so $Denotes(A \cap^* B, R_2) = Denotes(A, R_2)$. Results of this sort are summarized in Figure 24. In that figure, "in" means the region lies wholly in the interior of the operand, and "out" means the region lies wholly in the exterior of the operand. Note that simplifications of the sort applied to region R_2 above can be done

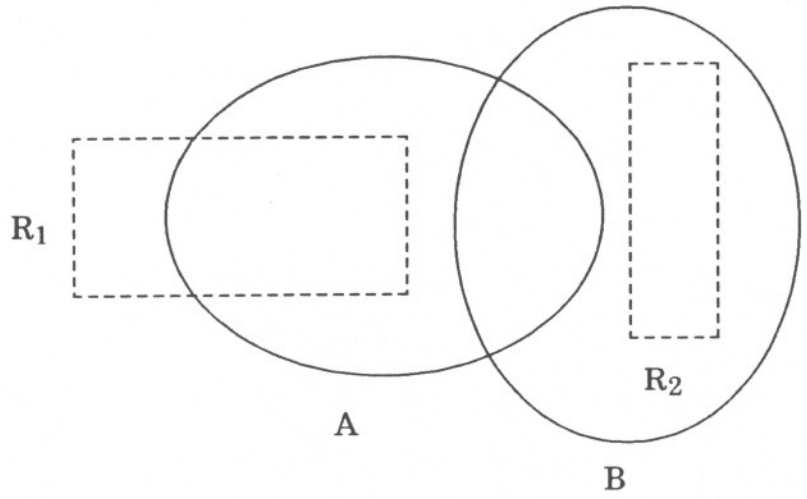


Figure 23. Example of Expression Simplification

without any knowledge about operand A; it may be represented by an arbitrarily complex sub-expression.

The algorithms presented below use a BSP tree to define each region of interest, specifically, the region R(v) associated with a node v of the tree. The BSP tree is also used to partition a set operation evaluation problem into subproblems.

Algorithms Using the BSP Tree

Complementation

First, consider the unary complementation operator. Given a set A, represented by a labeled-leaf BSP tree, a labeled-leaf BSP tree representing its complement, $\sim^*$ A, can be formed as follows. All "in" cells are changed to "out" cells, and all "out" cells to "in" cells. If the BSP tree is also boundary-augmented, the orientation of each boundary polygon is reversed. A boundary representation is complemented by reversing the orientation of every

face.

Incremental Evaluation

Incremental evaluation modifies a BSP tree to reflect the result of a set operation on the set the tree represents and one additional set, represented by a B-rep. The result of the set operation is recorded by changing the BSP tree T. These changes take two forms: (1) some subtrees of T are replaced by leaves, and (2) some leaves of T are replaced by new subtrees. Given a labeled-leaf BSP tree T representing a regular set A, and a boundary representation of a polyhedron B, an algorithm to determine A *op* B is presented, where *op* is a regularized set operation.

Figure 24 presents the algorithm as pseudo-code. At a given node v of the tree, the subproblem under consideration is (A op B, R(v)), where A is represented by the subtree of T rooted at v, and B is represented by the subset of B's boundary lying in R(v). In 3D, the boundary of B is represented as a set of polygons.

When evaluating a difference operation A $-^*$ B, the right operand is complemented, and the equivalent intersection operation A $\cap^*$ ($\sim^*$ B) is evaluated. The discussion is therefore restricted to the union and intersection operators.

The algorithm begins at the root of T with the entire boundary of B, and proceeds to traverse T in a recursive, pre-order fashion. The problem (A op B, R(v)) is partitioned into three subproblems: $P_{left} = (AopB, R(v.left))$, $P_{right} = (AopB, R(v.right))$, $P_{coplanar} = (AopB, R(v) \cap H_v)$. Since A is represented by the BSP tree rooted at v, the representations of A in subproblems P_{left} and P_{right} consist of the left and right children of v, respectively. The boundary of B lying in R(v) is partitioned with the hyperplane H_v. The portion in H_v^- is passed on to a recursive call for P_{left}, and the portion in H_v^+ to P_{right}. Any faces of the boundary of B found to lie in (coplanar with) H_v are kept at node v for later

```
procedure Incremental_Set_op ( op : set_operation ;
                               v : BSPTreeNode ;
                               B : set of Face ) returns BSPTreeNode

    If op = -* then
        B := Negate_B-rep ( B )
        op := ∩*

    If v is a leaf then
        case op of
            ∪* : case v.value of
                    in  : return v
                    out : return Build_BSPT ( B )
            ∩* : case v.value of
                    in  : return Build_BSPT ( B )
                    out : return v

    else
        <B_left, B_right, B_coplanar> := partition B with H_v

        If B_left has no faces then

            status := Test_in/out (H_v, B_coplanar, B_right)

            case op of
                ∪* : case status of
                        in  : discard_BSPT ( v.left )
                              v.left := new "in" leaf
                        out : do nothing
                ∩* : case status of
                        in  : do nothing
                        out : discard_BSPT ( v.left )
                              v.left := new "out" leaf
        else
            v.left := Incremental_Set_op ( op, v.left, B_left )

        If B_right has no faces then

            (* similar to above *)

        else
            v.right := Incremental_Set_op ( op, v.right, B_right )

        return v

end; (* Incremental_Set_op *)
```

Figure 24. Algorithm for Incremental Set Operations

processing (Section "Boundaries"). If T is not boundary-augmented, $P_{coplanar}$ need not be solved: faces of B lying in H_v are simply discarded. However, if T is boundary-augmented, $P_{coplanar}$ need only be partially solved, to the extent necessary to determine the *boundary* of A

TABLE 2. Expression Simplification Rules

op	left operand	right operand	result
$\cup^*$	S	in	in
	S	out	S
	in	S	in
	out	S	S
$\cap^*$	S	in	S
	S	out	out
	in	S	S
	out	S	out
$-^*$	S	in	out
	S	out	S
	in	S	$\sim^*$ S
	out	S	out

op B lying in H_v. This is due to the fact that a boundary-augmented BSP tree does not explicitly differentiate between regions of sub-hyperplanes that are in the interior or exterior of the polyhedron. Such regions would bound *in*-cells (or *out*-cells) on both sides of the sub-hyperplane.

The recursion terminates when one of the operands is homogeneous in the region R(v), defined by the current node v. This is the case when v is a leaf node, or if no part of B's boundary intersects R(v). First, consider the case in which no part of B's boundary intersects R(v). If v is not a leaf, the *In/Out Test* (presented in later in this Chapter) is evaluated to determine the status of R(v) with respect to B. This test is described in a subsequent section. As Figure 25 indicates, depending on the set operation and the outcome of this test, there are two basic outcomes for the set operation. Either the result is homogenous, with the value returned by the in/out test, or the result is identical to A. If the result is homogenous, the subtree rooted at v is replaced by a leaf of the appropriate value. Otherwise, it is the case that the subtree rooted at v already describes operand A within R(v), so the subtree is left unmodified.

Next, consider the case in which v is a leaf node and some portion of B's boundary lies in the corresponding cell R(v). As before, the result of the set operation will be either

homogeneous, or defined by the other operand, in this case, B. If the result is homogenous, the leaf retains its value and the portion of B's boundary lying in the cell is discarded. Otherwise, a new subtree rooted at v is constructed to describe B within R(v). This is done using the boundary-to-BSP conversion algorithm presented as procedure Build_BSPT in the previous chapter. The input to the algorithm is the boundary of B lying in R(v), and the output is a labeled-leaf BSP tree. This tree takes the place of the leaf v.

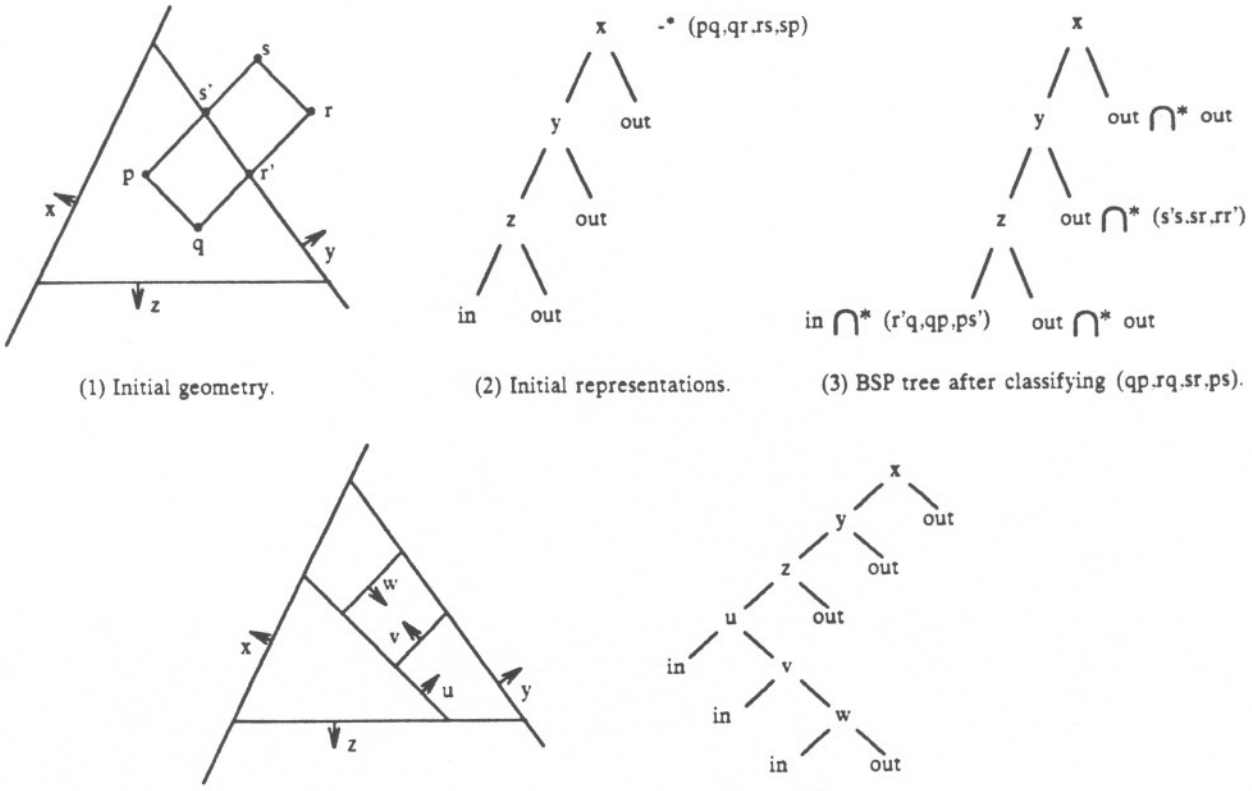
TABLE 3. Handling the Termination of Recursion in Incremental Evaluation

op	Classification of cell of T containing (part of) B's boundary		Result of in/out test for R(v)	
	in	out	in	out
∪*	discard	build tree	discard, replace with "in"	keep
∩*	build tree	discard	keep	discard, replace with "out"
	Disposition of B's boundary		Disposition of subtree rooted at v	

Finally, consider the case in which v is a leaf and no part of B's boundary lies in R(v). Both operands are homogeneous in R(v). If the set operation does not depend on B, the in/out test need not be run. (This is the case at an in-leaf for union and at an out-leaf for intersection.) Otherwise, the test is run and its result becomes the value of the leaf. Table 3 summarizes the handling of the base cases of this recursive algorithm.

Figure 25 illustrates an example of the algorithm in 2D. In (1) and (2), the initial geometry and representations are shown: a BSP tree representing a triangle A, and a set of edges (pq,qr,rs,sp) representing the (counterclockwise) boundary of a quadrilateral B. A difference operation is to be performed. As the first step, the orientation of each edge of the quadrilateral is reversed, and the set operation becomes an intersection of A and the quadrilateral (qp,rq,sr,ps).

Starting at node x, the edges are found to all lie in H_x^-. The in/out test would determine that H_x^+ is exterior to B. But, the test need not be run, since subtree x.right is an out-leaf, and the result of the intersection is therefore an out-leaf. The edges are then partitioned with H_y. This results in two sets of edges, (s's,sr,rr') and (r'q,qp,ps'). The edges



(1) Initial geometry. (2) Initial representations. (3) BSP tree after classifying (qp,rq,sr,ps).

(4) Resulting partitioning. (5) Final BSP tree.

Figure 25. Example of an Incremental Set Operation

(s's,sr,rr') lie in the right subtree of y, an out-cell, and so are discarded. The remaining edges are tested to H_z, and found to all lie in H_z^-. At this point, the algorithm reaches a leaf of the tree. The leaf's value is "in", and Table 2 shows that an intersection of an in-cell with another object is described by that other object. This second object is defined by the edges of B lying in the in-cell. Procedure Build_BSPT is then used to construct a BSP tree defining it, and this tree replaces the former leaf.

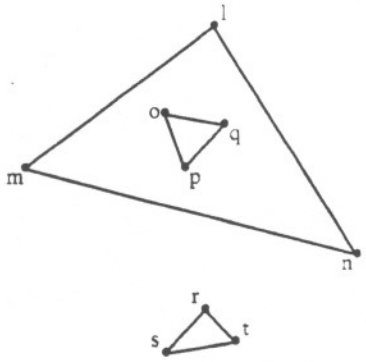
CSG Evaluation

A CSG evaluation problem P is a pair (τ, R), where τ is a CSG tree and R is a convex region of d-space on which τ is defined. The result is a BSP tree T such that *Denotes*(T) = *Denotes*(τ). Starting with the problem P = (τ, R), the algorithm chooses a hyperplane H to partition the problem into three sub-problems, $P_{left} = (\tau_{left}, R \cap H^{-})$, $P_{right} = (\tau_{right}, R \cap H^{+})$, and $P_{coplanar} = (\tau, R \cap H)$. Since H has partitioned the problem P = (τ ,R), we can solve each independently, and return the union of the results as the solution to P.

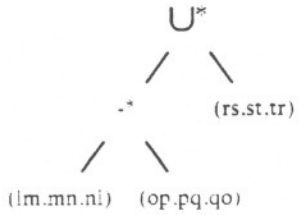
The algorithm returns a BSP tree whose root node is associated with the chosen hyperplane H, and whose left and right children are the results of the recursive evaluation of P_{left} and P_{right}, respectively. The recursion is terminated when the CSG tree τ can be evaluated directly. ($P_{coplanar}$ need not solved completely. Solving for *bd* $P_{coplanar}$ will usually suffice for a boundary-augmented BSP tree. This can be done at a later stage, once the BSP tree has been built (Section "Boundaries").)

The hyperplane H is chosen to embed some face of a primitive of τ. A heuristic examines some set of possible hyperplanes and chooses the "best" one. As is the case in the construction of BSP trees as a preprocessing step for visible surface rendering, the choice of partitioning planes, especially near the root of the tree, can greatly influence the resulting tree. In the case of CSG evaluation, it can also effect the efficiency of the evaluation itself. Since the number of possible trees is at least factorial in the number of faces, we take the standard approach of using heuristics to guide us in tree construction. The heuristics used are addressed in the discussion of the implementation of the algorithm, in Chapter VI.

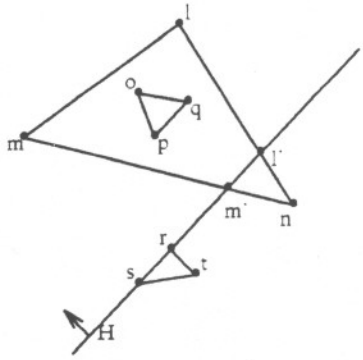
In dividing the problem into three sub-problems, the boundary representation of each primitive of the CSG tree is partitioned with the hyperplane. If any primitive has no part of its boundary on one side of the hyperplane, it is homogeneous in that region. Its position in



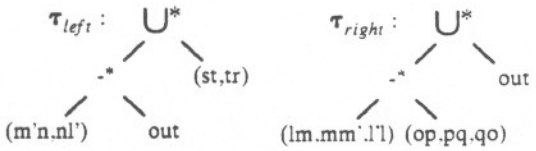
(1) Initial geometry



(2) initial CSG tree

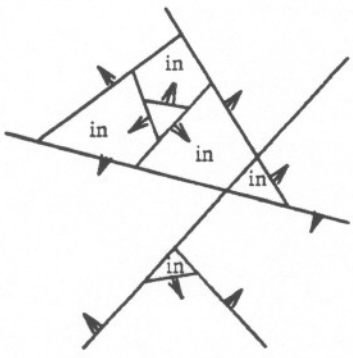


(3) partitioning by H

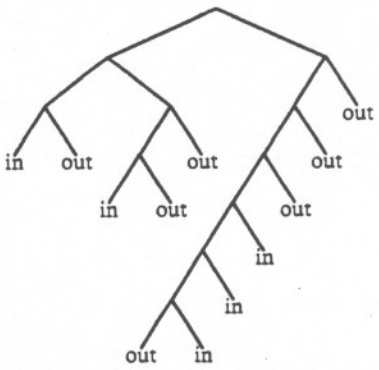


(4) subproblems before simplifying

(5) subproblems after simplifying



(6) final partitioning



(7) final BSP tree

Figure 26. An Example of CSG Evaluation

the CSG tree for that side (τ_{left} or τ_{right}) is replaced with a trivial BSP tree (an *in* or *out* leaf). This can be done because if $prim \cap R = \varnothing$ or $prim \cap R = R$, then either $Denotes(prim,R) = Denotes(out,R)$ or $Denotes(prim,R) = Denotes(in,R)$, respectively. The classification of this trivial BSP tree, *in* or *out*, depends on whether the region to that side of the hyperplane is in the interior or exterior of the primitive, respectively. The technique to determine this is presented in the next section.

The CSG tree τ_{left} (similarly, τ_{right}) is thereby derived from τ in such a way that $Denotes(\tau_{left}, R \cap H) = Denotes(\tau, R \cap H)$. The CSG tree is then simplified, using the rules given in Table 2. Note that this process also does not change the set represented by the tree. If the tree is pruned down to a single *in*- or *out*-leaf, the problem in that region has been completely solved. When all sub-problems are completely solved, the BSP tree represents the set defined by the CSG tree.

In Figure 26, (1) and (2) Show the input to the algorithm; (3),(4), and (5) depict the first partitioning into subproblems; (6) and (7) show the output of the algorithm. Figure 27 presents a pseudo-code description of the algorithm.

In/Out Testing

A fundamental step of both of the set operation algorithms presented in this chapter and the algorithm to construct a BSP tree from a B-rep (presented in Chapter III) is the detection and classification of regions homogeneous with respect to a (portion of a) B-rep. Detection is simple: a homogeneous region contains no part of the boundary of the B-rep. Classification requires determining whether the region lies wholly to to interior or exterior of the set represented by the B-rep. In the B-rep-to-BSP algorithm presented as Procedure Build_BSPT in Chapter III, classification was made trivial by restricting hyperplanes to those embedding a face of the (single) B-rep. This is not generally the case for the set operation

```
procedure Evaluate_CSG ( τ : CSGTree ) returns BSPTree

        f := choose a face of a primitive of τ
        v := new BSPTreeNode
        H_v := H_f
        <τ_left , τ_right> := Split_CSG ( τ, H_v )

        τ_left := Simplify_CSG ( τ_left )
        if τ_left represents ∅ then
                v.left := new "out" leaf
        else if τ_left represents U then
                v.left := new "in" leaf
        else
                v.left := Evaluate_CSG ( τ_left )

        (* similar code for τ_right *)

        return v
end; (* Evaluate_CSG *)


procedure Split_CSG ( τ : CSGTree;
                      H : plane_equation ) returns <CSGTree, CSGTree>

        if τ is not a primitive then
                τ_left := copy ( τ )
                τ_right := copy ( τ )

                <τ_left.left, τ_right.left> := Split_CSG ( τ.left )
                <τ_left.right, τ_right.right> := Split_CSG ( τ.right )
        else
                <τ_left, τ_right, τ_coplanar> := partition τ with H

                if τ_left = ∅ then
                        τ_left = Test_in/out ( H , τ_coplanar , τ_right )
                if τ_right = ∅ then
                        τ_right = Test_in/out ( H , τ_coplanar , τ_left )
        return <τ_left, τ_right>
end; (* Split_CSG *)
```

Figure 27. Algorithm for CSG Evaluation

algorithms, or for B-rep to BSP conversion using arbitrary hyperplanes.

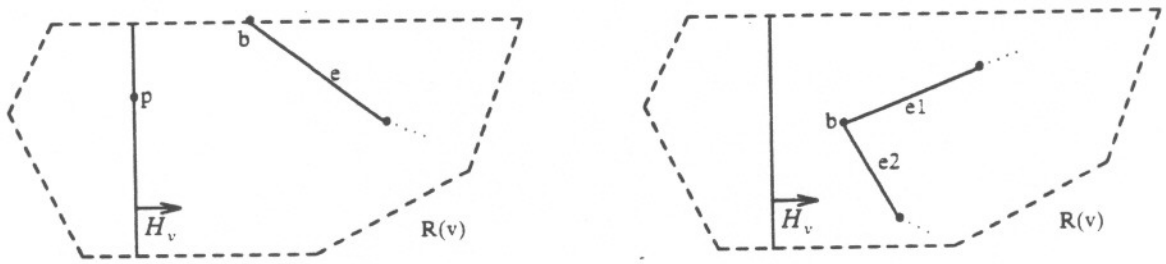
This section presents a method of classifying regions homogeneous with respect to a B-rep when no face of the B-rep lies in the "final" hyperplane. This "final" hyperplane is the one which partitions the current region of interest, resulting in the detection of homogeneity in one of the two resulting sub-regions.

Denote the portion of the boundary representation lying in R(v) by B_v. Note that faces of B lying entirely in *bd* R(v) are not included in B_v. Assume in the following discussion, without loss of generality, that B_v lies entirely in H_v^+, i.e., in R(v.right). The status of R(v.left) with respect to B must be determined. (See Figures 28 and 29.)

This test is relatively simple if restricted to boundary representations of convex polyhedra. A representative point in the (interior of the) sub-hyperplane of H_v could be tested for inclusion in the back half-spaces of all faces of B_v. If the representative point is "behind" all of these faces, then R(v.left) lies in the interior of B. If the representative is "in front" of any face, R(v.left) $\subset$ *ext* B. The representative point for a sub-hyperplane is simple to determine if each hyperplane embeds some face of a boundary representation. In the implementations, the centroid of three non-collinear vertices of this face is used as the representative.

A test is now presented that handles arbitrary polyhedra. The "ray test" [Laid86] could be used, counting intersections with *bd* B along a ray emanating from a representative point of the sub-hyperplane. However, this would either require testing the ray against the entire boundary of B, or testing the ray against B_v, and if it did not intersect B_v, repeatedly perturbing the ray until it did so. The test presented is not based on ray-casting, uses only B_v, and returns the answer on the first try. Let b be the vertex closest to H of B_v. The closest vertex is determined during the partitioning of B_v by H_v: each comparison of point to hyperplane returns the signed distance of the point from the hyperplane. Let p be a representative point on the sub-hyperplane of H_v.

In 2D, the situation is as pictured in Figure 28. Vertex b is either in *bd* R(v) or in *int* R(v). If b lies in *bd* R(v), then there is a single edge e in B_v incident with b. (Otherwise a second edge would lie in *bd* R(v) or *ext* R(v), which contradicts the definition of B_v.) If p lies in H_e^+, then R(v.left) lies in the exterior of B. Otherwise, R(v.left) is in the interior of



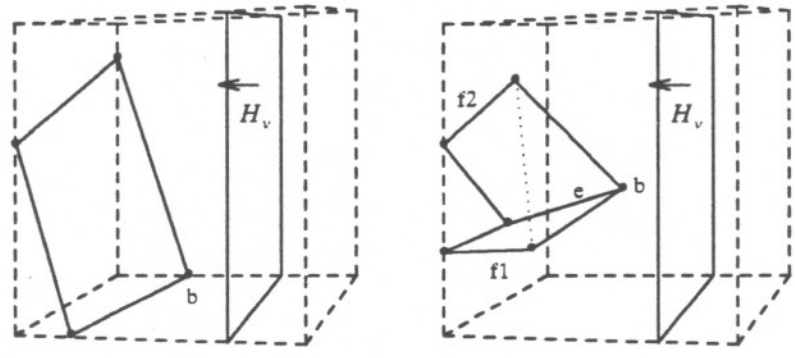
(a) closest vertex in boundary of R(v).

(b) closest vertex in interior of R(v).

Figure 28. In/Out Testing in 2D

B. If b is in *int* R(v), b is incident with two edges, e1 and e2. The region R(v.left) is in the exterior of B if e1 and e2 lie in each other's back halfspace, i.e., if $e1 \subset \bar{H}_{e2}$ and $e2 \subset \bar{H}_{e1}$. This is the case when b is a point of "local convexity" [Lay82] of B. Otherwise R(v.left) is in the interior of B (and b is a point of "local concavity").

In 3D, the cases are similar: either b lies in *bd* R(v) and is not shared by any other face of B_v, or b is shared by more than one face of B_v (and may lie in either *bd* R(v) or *int* R(v)). The test for the first case is identical: p is tested against the hyperplane of the single face containing b, with the same results. (Figure 29(a)) When b is is shared by more than one face of B_v, each pair of sharing faces defines an edge which shares b. (Figure 29(b).) Select the edge which forms the smallest angle with the plane H_v. This is the "closest" edge of B_v to H_v in the neighborhood of b. If, in a local region of b, f1 and f2 are in each other's back half-space, then R(v.left) is in the exterior of B (b is a local convexity). Otherwise, R(v.left) is in the interior of B. To determine whether or not the faces lie in each other's back half-spaces, it suffices to consider if a particular vertex of one of the faces lies in the other face's



(a) point b belongs to only one face. (b) point b is shared by more than one face.

Figure 29. In/Out Testing in 3D

back half-space.

Vertex b is connected by an edge to two vertices of f1. One of these lies in edge e, and hence in f2. The other vertex, not lying in f2, is tested against H_{f2}. If it lies in $\bar{H}_{f2}$, f1 is in f2's back half-space. Since face normals are consistently outward-pointing, and f1 and f2 share an edge, then it must also be the case that f2 is in f1's back half-space, and R(v.left) is then in the exterior of B. If f1 and f2 are both convex, then any vertex of f1 not lying in f2 can be used.

It may happen that there is a tie for closest vertex. In this case, the vertex that allows the simplest test can be chosen.

BSP Tree Reduction

Once a BSP tree has been constructed as the result of the evaluation of a set operation, it may be possible to *reduce* the tree by eliminating certain nodes. This reduction will not change the set represented by the BSP tree. (BSP tree reduction is similar to quadtree condensation [Hunt79].) Two cases in which this reduction is possible are

identified.

The first case occurs when the subtrees of a node the node can be replaced by a trivial BSP tree of that classification.

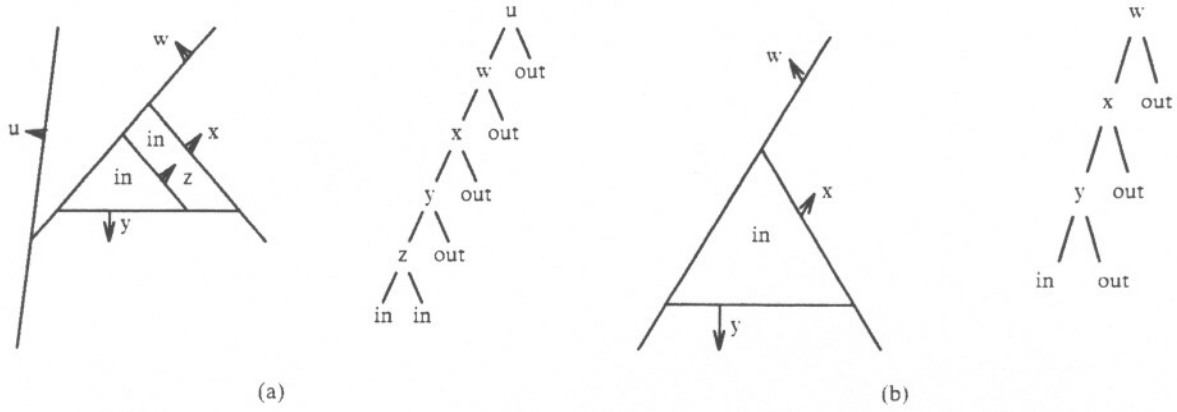


Figure 30. BSP Tree Before (a) and After (b) Reduction

(Node z in Figure 30(a).) This is detected during construction of the BSP tree. Note that no boundary polygons could lie in the sub-hyperplane of such a node. Such a hyperplane serves no purpose, since it partitions a homogeneous region.

It is also possible to remove a node that has a trivial BSP tree as one child and has no part of the boundary of the set in its sub-hyperplane. (Node u in Figure 30(a).) The non-trivial subtree replaces the node in the tree. Note that this may alter the sub-hyperplanes of descendants of the node removed, allowing those sub-hyperplanes previously bounded by the removed node's sub-hyperplane to extend into the region previously represented by the trivial BSP tree. (The sub-hyperplane of node w in Figure 30 is an example of this.) This second case is detected after processing each node for boundary polygons.

Boundaries

In the discussion of BSP tree algorithms for set operations, whenever a face was found to lie in a sub-hyperplane, it was appended to a list of faces lying in that sub-hyperplane. In this section, algorithms to process these faces are discussed. The algorithms yield faces lying in the boundary of the set represented by the BSP tree.

Since the boundary of the result of any regularized set operation is a subset of the boundaries of the operands [Requ78], the boundary in the sub-hyperplane will consist of a subset of the polygons on this list. Information is not maintained about which regions of each sub-hyperplane are "in" or "out": the subtrees of the sub-hyperplane's node are used, as was done for point classification. The evaluation of a node's subtrees must be completed before this can be done.

CSG Evaluation

Each list of polygons lying in H can be classified separately by applying the polygon classification algorithm of Chapter III. The resulting polygons with *on* classification are retained. It may be possible that more than one of these resulting boundary polygons overlap. To eliminate these, a 2D union operation can be performed on all coplanare boundary polygons of each orientation. This can be done with a 2D version of the CSG evaluation algorithm.

An alternative approach reduces the number of times polygon insertion is performed by increasing the amount of work done in the 2D phase. During set operation evaluation, a single list of coplanar boundary polygons from primitives is kept at each node. When a coplanar polygon is detected when partitioning a primitive's boundary representation, before adding the polygon to the list of coplanar polygons, it is checked to see if its normal is anti-parallel to the hyperplane's. If so, the polygon has its orientation reversed, giving all coplanar polygons the same orientation.

Once the CSG evaluation has produced a BSP tree, every sub-hyperplane with coplanar faces is processed in order to determine which polygons lie in the boundary. The technique for polygon classification presented in Chpater III is used, classifying each coplanar polygon with respect to both subtrees of the node. Begin by inserting the list of faces into v.left, and inserting a copy of the list, with the orientation of each polygon reversed, into v.right. Since any boundary polygon bounds one or more "in" cells on its back side, polygons from each insertion that are found to lie in "in" cells are kept.

This process eliminates any polygons, or portions of polygons, that lie in the exterior of the set. Removal of polygons in the interior and redundant polygons on the boundary is done with the "glueing" operation described below.

Incremental Evaluation

In the evaluation of an "incremental" set operation, A *op* B, not all sub-hyperplanes need be processed to determine the coplanar boundary polygons. The polygons lying in those sub-hyperplanes of A's tree that were found to lie entirely in the interior or exterior of B, and which are retained in the result, are already correct. (For example, the face in hyperplane x of Figure 31.) These nodes were either kept as is or deleted, along with their boundary polygons. The sub-hyperplanes of subtrees created when faces of B fell in cells of A are also already correct, their boundary polygons consisting of those faces of B. (E.g., those in hyperplanes u, v, and w.) The sub-hyperplanes in which additional work must be done are those that were coplanar with a face of B, or that split a face of B (e.g., lying in hyperplane y).

Whenever a subtree of node v is discarded, the boundary polygons which lie in the partitioning planes of the subtree are discarded as well. If it is also the case that no face of B is coplanar with H_v, the boundary polygons in H_v can also be discarded. This is so because if one side of H_v is homogeneous with respect to B, such that the subtree for that region is

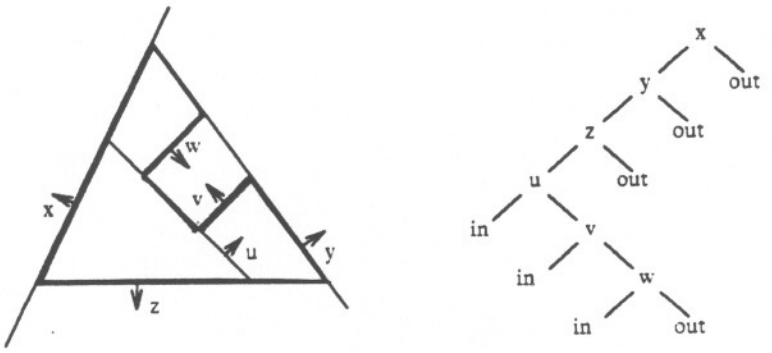


Figure 31. The Result of Incremental Evaluation

discarded, then the region is homogeneous in the result. If it is also the case that no part of B's boundary is in $SHp(H_v)$, the sub-hyperplane is also homogeneous with respect to B and thus homogeneous in the result.

If any face of B is coplanar with H_v, we retain it along with the boundary polygons of A lying in H_v. Actually, two lists are maintained at each node, for polygons oriented parallel and anti-parallel to the normal of H_v. Faces of B coplanar with H_v are added to the appropriate list.

Due to the fact that a single set operation is being evaluated, not all coplanar polygons need be inserted into both subtrees of v. Once v.left and v.right are evaluated, the semantics of the set operation are used to determine which of the two lists of polygons to classify with respect to which subtree, and which classification ("in" or "out") signifies that a

polygon is to be kept. For example, in the case of union, the cell(s) behind an outward-pointing face of either operand will be "in" in the result. Also, it is the case that a polygon is in the boundary when it bounds both an "in" cell and an "out" cell. Therefore, a (portion of a) face of either operand will be in the boundary of their union when the cell lying in front of the polygon is "out". So, to determine this, the faces oriented parallel to H_v's normal are inserted into v.right, and the anti-parallel faces into v.left. The polygons lying in "out" cells are kept. For intersection, the parallel faces are inserted into v.left, the anti-parallel into v.right, and the polygons lying in "in" cells kept.

This eliminates any polygons not in the boundary, but there may be regions of the boundary in which polygons overlap. This redundancy does not affect renderings of the object, other than to possibly increase time and space requirements. If desired, the redundancy can be eliminated by performing a "glueing" operation in 2D on the coplanar polygons. This is described below.

The 'Glue' Operator

As mentioned above, inserting faces into the subtrees on either side of their embedding hyperplane can leave redundant faces. The "glue" operator described below addresses this issue, as well as providing a tool to construct a B-rep from the BSP tree. In this latter sense, "glueing" is used to re-join faces that were split by hyperplanes in the course of building the tree. One can say that the resulting B-rep is, in this sense, minimal.

TABLE 4. Semantics of the Glue Operator

A	B	A⊕B
same-on	same-on	same-on
flipped-on	flipped-on	flipped-on
same-on	flipped-on	not-on
flipped-on	same-on	not-on
not-on	X	X
X	not-on	X

The 'glue' operator is a sort of "exclusive or" operator. Regions of the plane occupied by polygons of opposite orientation are replaced by the empty set. This occurs when the region bounds "in" cells on both sides of the sub-hyperplane. Regions occupied by more than one polygon, all of the same orientation, are replaced by a single polygon of that orientation. Regions with no polygons remain empty. (Table 4.) This is similar to the "regularization" operation in the algorithm of [Putn86].

The "glueing" operation is carried out in a manner similar to the CSG evaluation algorithm of section 3.4, but in 2D with a 2D BSP tree. Given a set of polygons in a hyperplane H_v, we construct a 2D BSP tree to evaluate the glue operator as follows. First, for computational convenience, the polygons are projected orthogonally into the coordinate plane $x_i=0$, where x_i is the largest coordinate of H_v's normal. This involves simply dropping the i-th coordinate. All computations (dot-products, edge-hyperplane intersections) are now carried out in 2D.

A partitioning hyperplane (line) is chosen to embed an edge of some polygon. This hyperplane is used to partition the problem into sub-problems, as was done in 3D. This process continues until the sub-problem can be solved directly. This can be done when either: (1) all polygons in the region are of the same orientation, and at least one of these covers the entire region, or (2) when the region is covered by two polygons of opposite orientation.

If the polygons in a region are replaced by a single polygon oriented in the same direction as H_v, the leaf is marked "same-on". If the result is a polygon with the opposite orientation as H_v, the leaf is marked "flipped-on". Otherwise the leaf is marked "not-on". It is relatively straightforward to maintain vertex-list representations of the regions of the 2D tree. The vertices of "same-on" and "flipped-on" regions must then be projected back into H_v. If the plane equation of H_v is $ax+by+cz+d=0$, and the projection was done along the z-axis,

then the coordinates of the projection of a 2D vertex (x',y') is $(x',y',(-ax-by-d)/c)$.

Minimizing The Boundary

The glue operator can also be used to produce a "minimal" boundary representation of the polyhedron. Essentially, a set of faces of dimension d-k that share faces of dimension d-k-1 are replaced by a (possibly concave) d-k face. This is done by applying a glueing operation while recursing on the dimension of the faces making up the boundary.

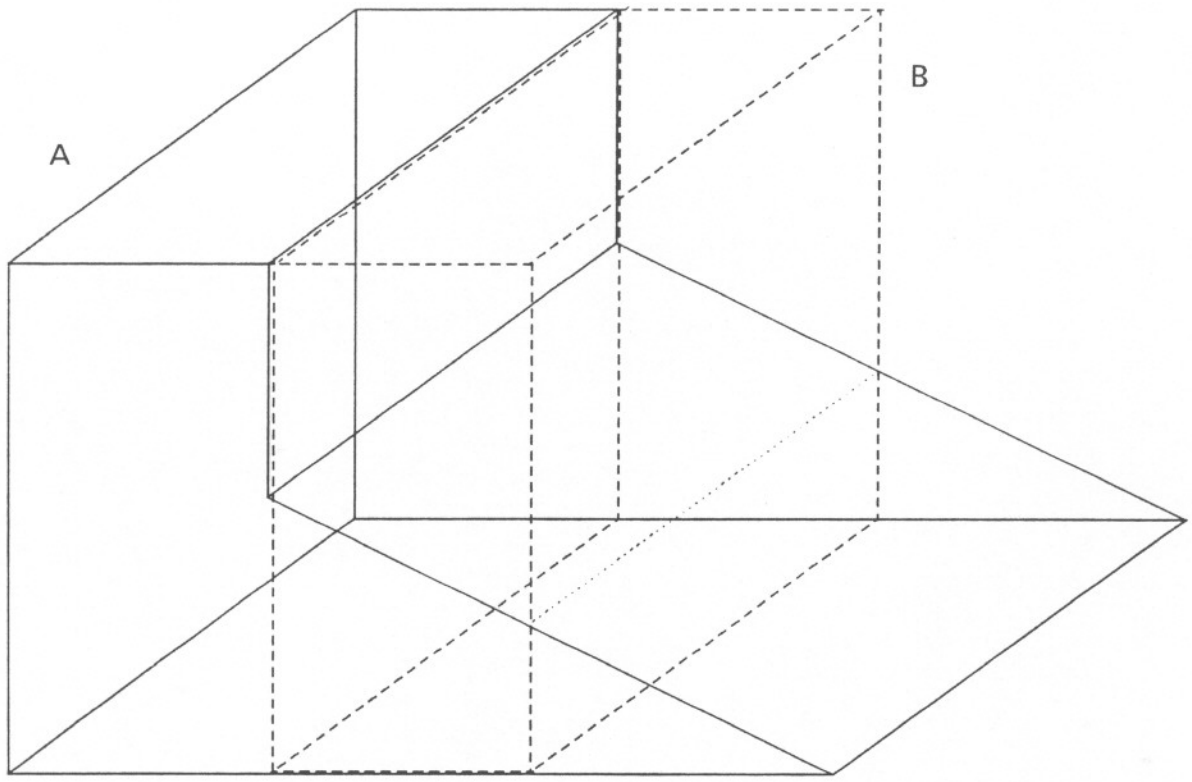


Figure 32. 3D Geometry of Objects A and B

The technique is presented by means of an example, seen in Figure 32. Two polyhedra, A and B, are to be combined via union using the CSG evaluation algorithm. This

results in a 3D BSP tree. Coplanar polygons are kept at the corresponding node of the 3D BSP tree until evaluation is complete, as usual. And, as before, they are then inserted into the subtrees of that node, and pieces landing in in-cells are retained. Assume the result of this insertion for the "front" side of the object in Figure 32 is the four polygons in Figure 33. All four polygons share the same orientation.

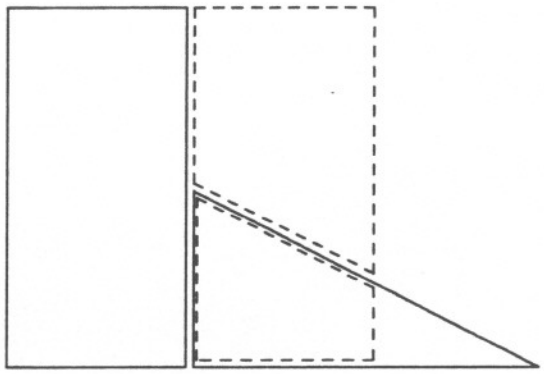


Figure 33. Polygons Lying in the Front Plane

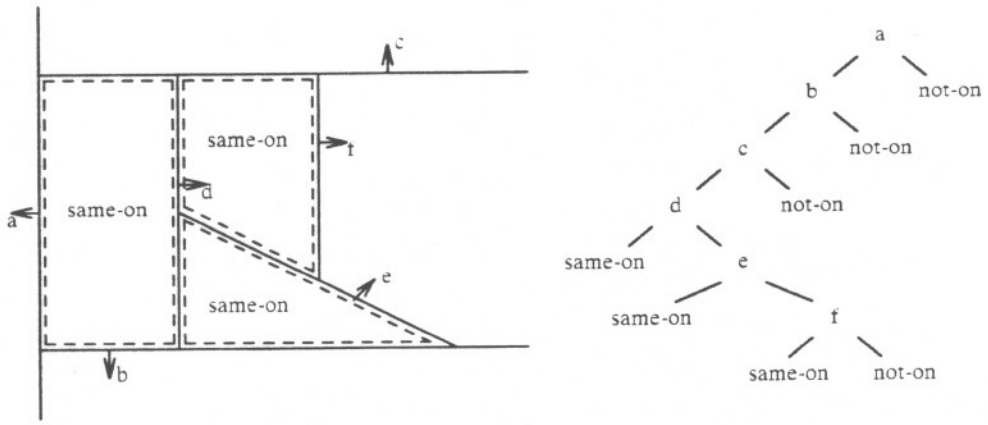


Figure 34. Result of 2D glue operation

The 2D glue operation is then performed, as before, using a 2D BSP tree. This removes the overlapping polygon (Figure 34). Now, instead of stopping once the glue has been performed, we associate each remaining polygon boundary (edge) with the internal node of the 2D tree corresponding to the partitioning line in which the edge lies.

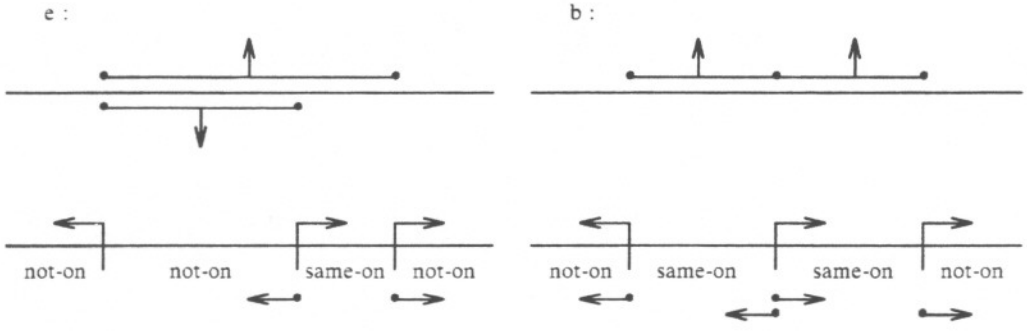


Figure 35. Examples of 1D glue operation

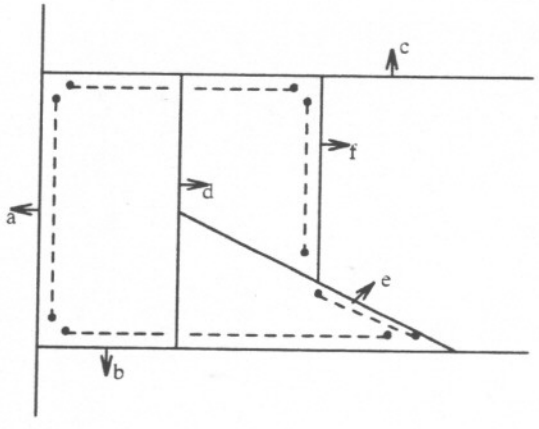


Figure 36. Fragments of the Boundary Remaining After Glueing

Next, a 1D glue operation is performed on the edges lying in each hyperplane of the 2D tree. A 1D BSP tree is constructed for each such hyperplane (line). Consider the hyperplane of node e (Figure 35). The region shared by oppositely oriented edges is cancelled out by the glue operation and is marked as "not-on," and the region spanned by the

left-over edges is marked "same-on." After glueing is complete, we associate boundaries (vertices) of the remaining "on" regions with the hyperplane of the 1D tree in which they lie.

Finally, a glue operation in 0D is performed, operating on coincident vertices at internal nodes of the 1D trees. (Figure 36) This will cancel out the vertices at the juncture of two adjacent "on" regions in the 1D tree, as in those for nodes b and c. The recursion now terminates.

It remains to re-join the constituents of the minimized boundary, shown in Figure 36. The degree to which these boundary elements are rejoined depends on the richness of the boundary representation being maintained. Assume each polygon is to be represented as a list of vertices. The representation is constructed while unwinding the recursion.

After glueing vertices in 0D, a traversal of each 1D tree yields a set of connected line segments as follows. The tree is traversed in an in-order fashion, such that leaves are encountered in an order corresponding to the geometric left-to-right order of the cells along the line. The order in which the children of a given node are visited is determined by evaluating the hyperplane equation of each node at $-\infty$. The child corresponding to the side of the hyperplane on which the point at $-\infty$ lies is traversed first. For example, in Figure 37, at node a, traversal first proceeds to the right child, then at node b to the right child again. The first leaf encountered should be a "not-on" cell if the polyhedron is bounded.

After the first child of a node has been traversed, upon return to the parent, the coincident vertex, if any, returned by the 0D glue is output. Traversal then proceeds to the other child. Pairs of vertices output by the traversal correspond to maximally connected line segments lying in that hyperplane of the 2D tree.

After generating the edges of the polygon in this fashion, pairwise comparison of vertices' locations can be used to join adjacent edges. This last step is not always necessary.

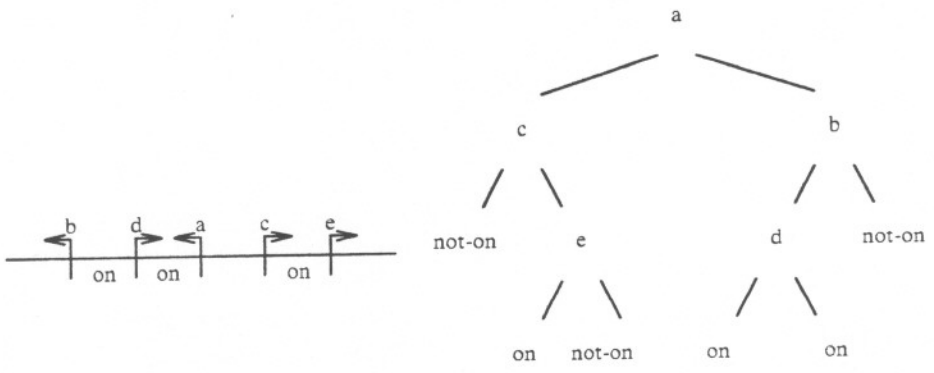


Figure 37. A 1D BSP Tree

A set of bounding edges for each polygon is actually the desired input for some scan-conversion algorithms [Fole83]. Figure 38 gives pseudo-code for the algorithms just discussed.

The above discussion does not discuss all possible opportunities for minimization, in that it does not address the situation in which coplanar faces lie in different sub-hyperplanes. In Figure 39, hyperplanes b and c contain edges AB and CD, which should properly be replaced with the single edge AD in a "minimal" boundary. This situation could be handled in the "Join_boundaries" step of the Minimize_boundary procedure. The hyperplane equations of the internal nodes of the tree should be examined to find those that are identical. This identity can be maintained during construction of the tree. Each set of such nodes should then be tested to see which have adjoining sub-hyperplanes. (A method for generating sub-hyperplanes in 3D is given in Chapter III.) Those sub-hyperplanes that do "touch" should then have their coplanar boundary elements glued at the juncture. This can be accomplished by performing the glue operation for all coplanar sub-hyperplanes at the

```
procedure Minimize_Boundary ( Dimension : integer;
                              BSPT : BSPTreeNode ) returns Boundary-rep

        for each internal node of BSPT do
                LowerDimensionalBSPT := Glue (coplanar faces at v)
                if Dimension > 0 then
                        Minimize_Boundary ( Dimension-1, LowerDimesionalBSPT )
        return Join_Boundaries ( Dimension, BSPT )

end; (* Minimize_Boundary *)


procedure Join_Boundaries ( Dimension : integer;
                            BSPT : BSPTreeNode ) returns Boundary-rep

        case Dimension of
                0: if BSPT is null then
                        return ∅
                   else return the vertex
                1: return Traverse_1D ( BSPT )
                2: return Join_incident_edges ( BSPT )
                3: return Join_incident_faces ( BSPT )
                etc.

end; (* Join_Boundaries *)


procedure Traverse_1D ( T : BSPTreeNode ) returns Boundary-rep

        if T is not a leaf then
                d := evaluate T's hyperplane equation at -∞
                if d < 0 then
                        first_child := T.left
                        second_child := T.right
                else
                        first_child := T.right
                        second_child := T.left
                Traverse_1D ( first_child )
                if there is a boundary vertex at node T then
                        append the vertex to the B-rep (of the edge)
                Traverse_1D ( second_child )

end; (* Traverse_1D *)
```

Figure 38. Algorithm for Boundary Minimization

same time.

Note that a "complete" boundary minimization that includes this last step produces a B-rep no longer associated with the BSP tree. In other words, a conversion from a BSP tree to a B-rep has been accomplished.

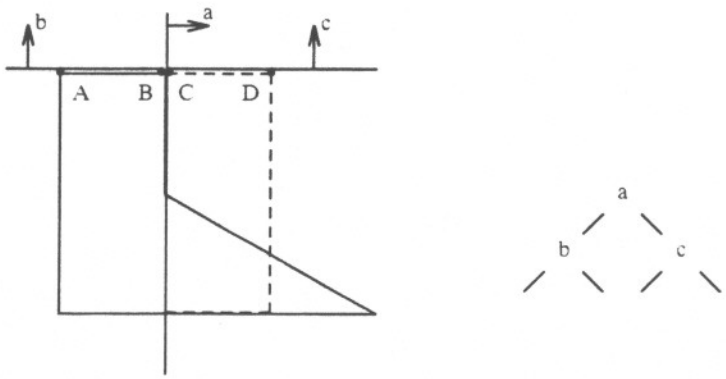


Figure 39. Adjacent, Collinear Edges in Different Sub-Hyperplanes

Set Operations on BSP Trees

In this section, an extension of the set operation algorithms presented in this Chapter are discussed. This extension involves the use of operands represented by specially augmented BSP trees. By showing how such a BSP tree can be split and tested for homogeneity, they can be easily incorporated into the previous algorithms.

The augmentation used is in addition to leaf-labels and boundaries. Each internal node of the tree is augmented with a boundary representation of the associated sub-hyperplane. In 3D, the sub-hyperplane is a convex polygon, and can be represented by an ordered list of vertices. An algorithm to determine such a representation of a sub-hyperplane was presented in Chapter III.

Splitting a BSP Tree

In both of the algorithms for set operations presented in this Chapter, a fundamental

step is the splitting of an operand. This occurs in the course of partitioning a problem (X, R(v)) with a hyperplane H_v, forming two, hopefully simpler, sub-problems. In the sequel, all BSP trees are augmented with explicit sub-hyperplanes, unless stated otherwise. To split a BSP tree augmented with explicit sub-hyperplanes, the following case analysis is used. Consider a BSP tree T rooted at node v. T is defined in $R(v)$. A hyperplane H is to split the tree T, returning two trees T^- and T^+, lying in $H^- \cap R(v)$ and $H^+ \cap R(v)$, respectively. Denote the sub-hyperplane associated with a hyperplane H by $SHp(H)$.

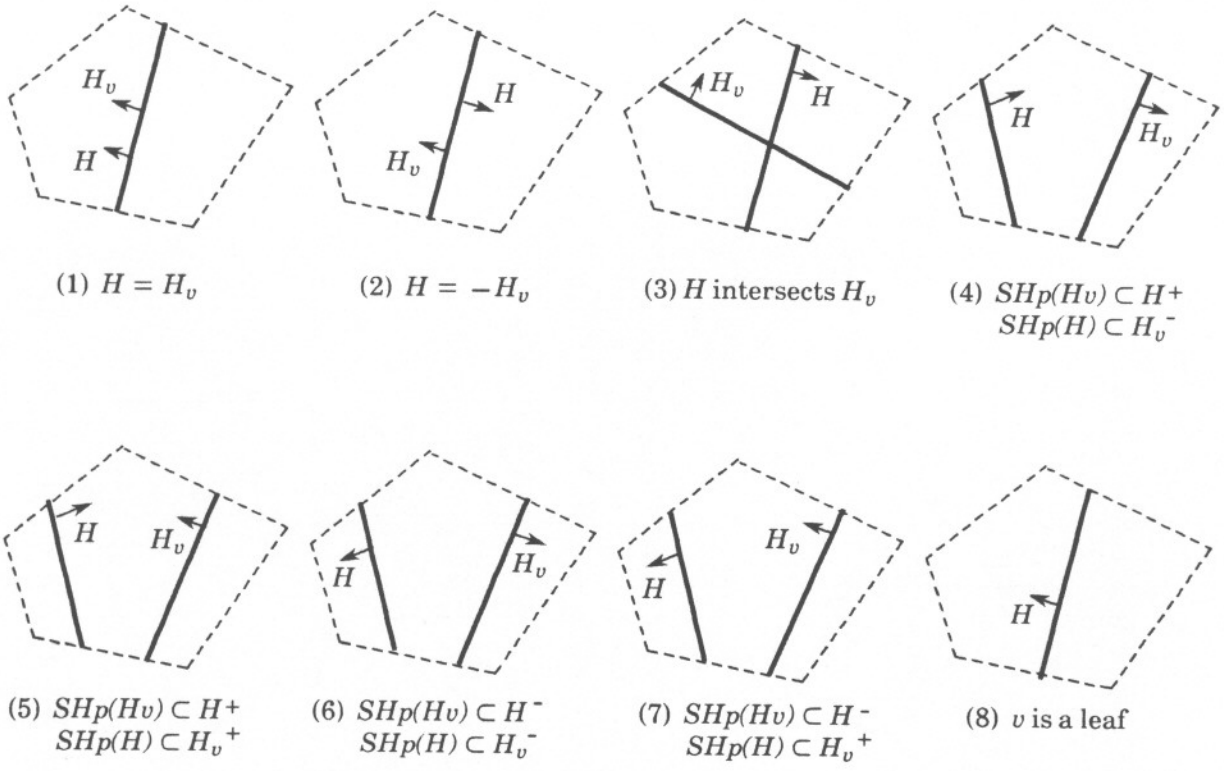


Figure 40. Case Analysis for BSP Tree Splitting

The first major division in the case analysis is to consider whether or not H intersects $SHp(H_v)$. If so, there are three possible ways in which this might occur. If $SHp(H_v)$ lies

entirely in H, then either $H_v = H$ or $H_v = -H$ (Cases 1 and 2, Figure 40). Otherwise, $SHp(H_v)$ lies in both half-spaces of H (Case 3). Note that $SHP(H_v)$ is an open set, and therefore if H intersects the boundary of $SHp(H_v)$, this is not considered and intersection with $SHp(H_v)$.

Four more cases of the analysis deal with the possibilities when $SHp(H_v)$ does not intersect H. (Note that H may intersect H_v in these cases, but that the intersection would occur outside of $R(v)$.) The first two cases handle $SHp(H_v)$ lying in H^+. Either $SHp(H)$ is in H_v^- (Case 4) or in H_v^+ (Case 5). An analogous pair of cases (Cases 6 and 7) exist when $SHp(H_v)$ lies in H^-.

The last case occurs when v is a leaf (Case 8). Figure 41 gives the algorithm.

The handling of each case is now discussed. In cases 1 and 2, T^- and T^+ are trivial to determine, since H and H_v are coincident, and H_v provides a ready-made partitioning of T. In case 1, $H = H_v$, so T^- = (the tree rooted at) v.left, and T^+ = v.right. In case 2, $H = -H_v$, and T^- = v.right, and T^+ = v.left.

In case 3, things are more complicated. H intersects $SHp(H_v)$, R(v.left) and R(v.right). Both subtrees of v are split with a recursive application of the algorithm to v.left and v.right. This returns four new BSP trees. Call the results of splitting v.left T_{left}^+ and T_{left}^-, lying respectively in H^+ and H^-. Similarly, splitting v.right yields T_{right}^- and T_{right}^+. $SHp(H_v)$ must also be split, using the algorithm of Chapter III. Call these results SHp^- and SHp^+. (Figure 42.)

These elements can now be combined to form the result trees T^- and T^+. The root of each tree has the same hyperplane equation, H_v. The subhyperplane of the root of T^- is SHp^-. The left and right subtrees of this root node are T_{left}^- and T_{right}^-. Similarly, the subhyperplanes of the root of T^+ is SHp^+, and its subtrees, T_{left}^+ and T_{right}^+.

```
procedure SplitBSP ( v : BSPTreeNode; H : hyperplane )
                returns < Left_Tree, Right_Tree : BSPTreeNode >

        if v is a leaf then     (* case 8 *)
                return < copy(v), copy(v) >
        else
                Evaluate H at all vertices of SHp(v)
                if all vertices evaluate to zero then
                        if H = H_v then
                                (* case 1 *)
                                return < v.left, v.right >
                        else
                                (* case 2 *)
                                return < v.right, v.left >

                else if there is at least one vertex on both sides of H then
                        (* case 3 *)

                        < left_minus, left_plus > ← SplitBSP ( v.left, H )
                        < right_minus, right_plus > ← SplitBSP ( v.right, H )

                        Left_Tree ← new BSPTreeNode
                        Right_Tree ← new BSPTreeNode

                        < SHp(Left_Tree), SHp(Right_Tree), dont_care > ← SplitPolygon ( H, SHp(v) )

                        Left_Tree.left ← left_minus
                        Left_Tree.right ← right_minus

                        Right_Tree.left ← left_plus
                        Right_Tree.right ← right_plus

                        return < Left_Tree, Right_Tree >
                else
                        d ← evaluate H_v at any point in H ∩* R(v)
                        if (d < 0) and all vertices of SHp(v) are in H^+ then
                                (* case 4 *)

                                < left_minus, left_plus > ← SplitBSP ( v.left, H )
                                v.left ← left_plus
                                return < left_minus, v >
                        else
                                (* cases 5, 6, and 7 are similar to 4 *)

end; (* SplitBSP *)
```

Figure 41. Algorithm to Split a BSP Tree

Cases 4 through 7 are all similar, so only case 4 will be discussed. In case 4, $SHp(H_v)$ lies in H^+, and $SHp(H)$ lies in H_v^- (Figure 43). To determine which side of H_v contains $SHp(H)$, a point in $SHp(H)$ is tested against H_v. Since no part of H intersects R(v.right),

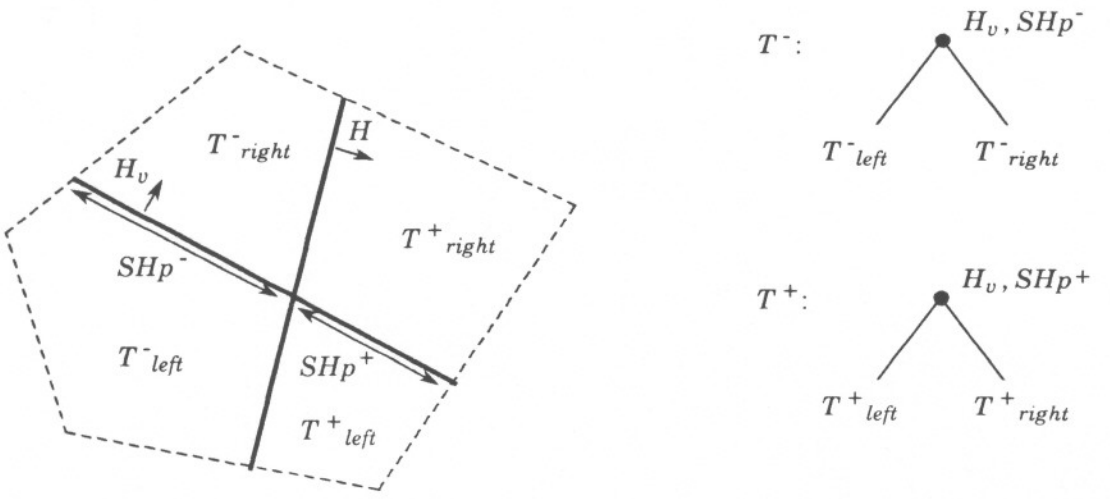


Figure 42. Subparts Generated in Case 3

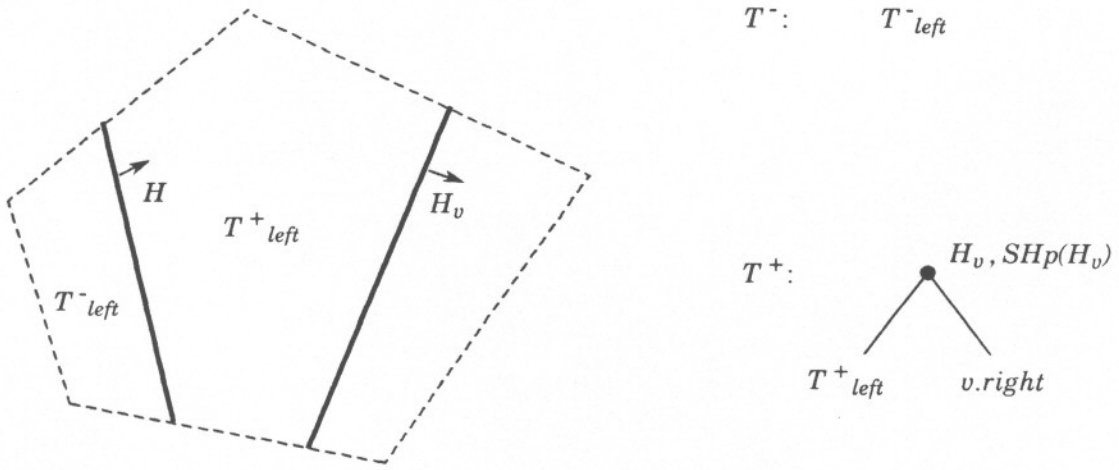


Figure 43. Subparts Generated in Case 4

that subtree need not be split. Only the tree rooted at v.left is split, with a recursive

application of the algorithm. The results are T_{left}^{-} and T_{left}^{+}. The result tree T^{-} is T_{left}^{-}. The result tree T^{+} is formed by replacing v.left with the root of T_{left}^{+}. $SHp(H_v)$ and v.right are left unchanged. The fact that v.right need not be considered in this case is yet another benefit derived from the structure provided by the BSP tree.

The final case, case 8, involves splitting a cell corresponding to a leaf v of T. This simply requires returning two leaf nodes, each with the same label as v.

Set Operations

Algorithms for set operations on BSP trees take the same basic form as those on B-reps. A partitioning process continues until a region homogeneous with respect to all operands (or all but one operand) is found. First consider the incremental algorithm. Both operands are represented with BSP trees augmented with sub-hyperplanes. As before, the second operand is inserted into the first. This involves a pre-order traversal of the first tree, splitting the second tree with the hyperplane of the current node at each stage of the traversal. Homogeneity of the first operand is determined as before, when a leaf of the first tree is encountered. To determine if one of the results returned by a splitting operation on the second operand is homogeneous, tree simplification is performed. A homogeneous result will simplify to a leaf.

A CSG algorithm on BSP trees is also a straightforward extension of CSG on B-reps. Partitioning hyperplanes are chosen as desired, splitting of BSP trees is performed as above, in/out testing is done by tree simplification, and CSG expression simplification is the same.

Boundary-augmented trees are maintained in a manner identical to that for the B-rep versions of the algorithms.

CHAPTER V

Ray-Tracing

And the principles governing reflections in mirrors and other smooth reflecting surfaces are not difficult to understand.

— Plato's Timaeus

Ray-tracing has been used to generate some of the most realistic images sythesized to date. The technique is based on principles from geometric optics, which models light as rays that travel in straight lines, and describes changes in direction caused by reflection and refraction. The rays are traced backwards from the viewpoint into the scene. (Figure 44.) Ray-tracing facilitates the use of realistic illumination models, has been extended to a wide range of primitives, and is conceptually straightforward. Its primary drawback is that large amounts of computation are required compared to other rendering algorithms. The bulk of this is spent in calculating the intersection of rays with the objects constituting the model. In the simplest version of the ray-tracing algorithm, the ray must be tested against all objects in the model, and the closest object chosen. This closest intersection point determines what is visible from the ray origin when looking in the direction of the ray. Even though the computation required is linear in the number of objects, the relatively high cost of computing the intersection of a ray with an object can make rendering unacceptably slow for complex scenes. This chapter addresses how the BSP tree can be used to speed up the ray-tracing process.

Applying Space Partitioning to Ray-Tracing

The ray-tracing problem is essentially a searching problem. By partitioning space

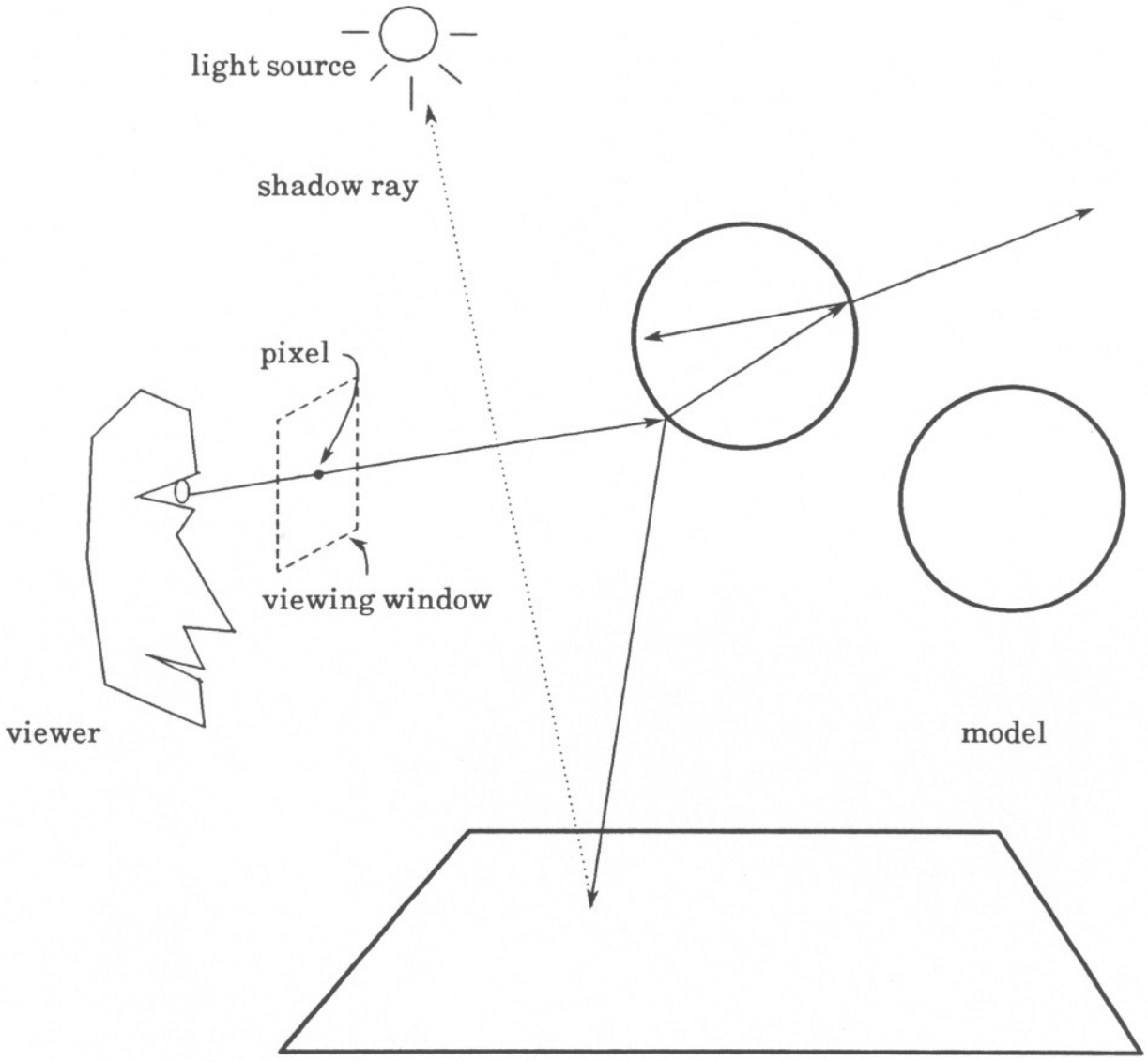


Figure 44. Illustration of the Basics of Ray-Tracing

with partitioning sets such as spheres or planes, the ray can be intersected with these partitioning sets in hopes of eliminating certain objects from consideration.

Since we are concerned with the closest object along each ray, we'd like to consider the partitions in visibility priority order. A straightforward method of determining the priority ordering is to find the ray's intersections with the space partitioning, sort them by distance, and then consider the ray partitions in the sorted order. However, we can do better. We can construct a representation of the space partitioning and use this to not only partition the ray, but also to obtain the priority ordering. As we shall see, if the representation is a BSP tree, the location of the ray origin determines uniquely this priority. (This property also holds for the octree partitioning. Note that any octree partitioning can be simulated with a BSP tree, and that the converse does not hold.)

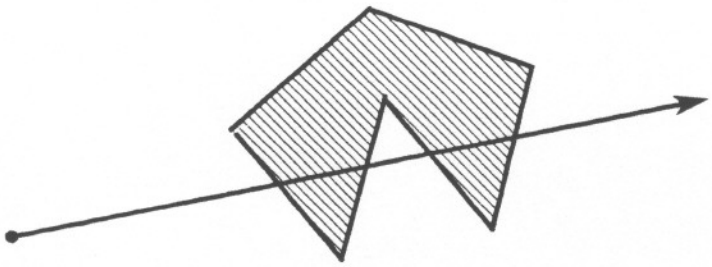


Figure 45. A Ray Intersecting a Non-Convex Partition

The ability to determine a priority ordering for the partitions intersected by any ray requires that all partitions are convex. If non-convex partitions are allowed, it is possible for a ray to enter partition A, then partition B, and then re-enter partition A again. (Figure 45.) If this occurs, any intersections with objects in A and B must be sorted to determine the closest intersection. The only partitioning set which can be used to produce convex partitions is a plane (or connected subsets of planes, as in the Voronoi diagram (see, for example[Prep85])). A single plane partitions a convex space into two convex sets, and by induction, all subsequent partitions will be convex. For each plane, the half-space lying on

the side of the plane in which the ray tail lies has priority over the other half-space.

Previous Work

The earliest and most common example of using space partitioning to improve performance is the use of *bounding volumes*. If objects are placed inside a simple volume, then a necessary condition for intersection of a ray with an object's surface is that the ray intersects that volume. Spheres [Whit80] and rectangular parallelipipeds [Rubi80, Roth82] have been employed for this purpose, and automatic selection and placement of 'tight-fitting' bounding volumes has also been studied [Wegh84]. However, bounding volumes do not produce a convex partitioning, nor are they generally disjoint.

There are problems inherent in the bounding volume approach. Bounding volumes may not be disjoint: the bounding volumes of nearby objects may intersect one another, even if the objects do not. If the ray enters the region where two bounding volumes intersect, the contents of both volumes must be tested for intersection with the ray. Another problem is that at least one concave partition will result. Concave partitions complicate priority determination. If all partitions are conves, all ray-partition intersections are convex (line segments). Determining the closest ray-object intersection then requires sorting only those intersections within the single (convex) partition. If concave partitions exist, this sorting step must consider all ray-object intersections in two or more partitions (those whose ray-partition intersections are interleaved). (However, this can be avoided if no objects are placed in the concave partition.)

Another approach is to use octrees [Glas84, Thom87] in which only axis-aligned partitioning planes are used (three at a time). It, like the BSP tree, forms a convex, disjoint partitioning. (Recall that any partitioning by an octree can also be formed by a BSP tree.) Computations involving axis-aligned planes are simpler than for arbitrary planes, but in certain applications, limiting planes to be axis-aligned may prove to be unacceptably

restrictive.

In [Dipp84], space is partitioned with 'generalized cubes.' These are topologically equivalent to the cube, but have relaxed constraints on face planarity and convexity. These relaxations allow the partitioning to be altered dynamically by moving one corner (shared by eight partitions) at a time. This structure is designed for use in a multiprocessor setting, with one processor assigned to each partition.

Glassner gives an algorithm that travels through the partitioning incrementally. The octree is first traversed from the root to determine the partition containing the ray tail. If the ray is found not to intersect any object in this initial partition, a point guaranteed to lie in the next partition along the ray is computed. This point is used to (again) traverse the octree from the root, in order to determine which partition that point lies in. The process continues until an intersection is found. In [Thom87], a more efficient algorithm is presented, that takes advantage of several properties to improve running time. One of these is based on *coherence*, the property that successive rays are usually only slightly perturbed from one another. This is due to the fact that as rays are traced from the viewing position, they are usually generated left-to-right accross each scan line. (Note that coherence does not apply to reflected or shadow rays.) At each level of the octree, the partitions entered by a ray will be unchanged from those entered by the previous ray if the ray exits each partition through the same face as the one the previous ray exited. Another technique used is to transform the planes of the octree by the perspective transformation defined by the viewing parameters. Rays from the viewpoint are then orthogonal to one coordinate axis, simplifying the ray-plane intersection calculations.

The BSP tree and octree eliminate the problems of overlapping and concave partitions experienced with the bounding volume hierarchy. This allows us to acheive the desired total visibility priority ordering on the partitions intersected by any ray. The techniques also seem

amenable to automatic partitioning construction. This has been studied for the octree[Glas84, Thom87] and for BSP trees in [Fuch80, Nayl81]. Automatic partitioning construction for polygonal CSG models is discussed in Chapter IV.

The BSP Tree Ray-Tracing Algorithm

Given a generic BSP-tree, it is augmented by associating a list of objects with each leaf. These objects are those constituents of the model that lie in the corresponding cell. Since a BSP tree provides both a partitioning of space and a priority ordering on those partitions, it is not only possible to restrict object-ray intersection calculations to volumes intersected by the ray, but to perform this in the order of the volumes' priority. Determining the volumes intersected by the ray and their priority is accomplished by traversing the tree with a modified "insertion" algorithm.

The search for an intersection of an object with the ray begins in the cell containing the ray's origin. The ray is first tested for intersection with the objects stored at the corresponding leaf of the tree. If no intersection is found, we proceed to the immediately adjacent volume intersected by the ray. This will be precisely the next volume (leaf) encountered in the traversal. The algorithm is presented as pseudo-code in Figure 46, and discussed below.

We represent the ray in parametric form, as two vectors, the ray direction D, and the ray origin T. The segment of the ray being considered is represented with two parameter values, t_{min} and t_{max}. Initially, $t_{min} = 0$, and $t_{max} = \infty$. The procedure for inserting a line segment, presented in Chapter III, is modified so that whenever the ray is split by a partitioning plane, recursion first proceeds to the subtree describing the halfspace that lies to the same side of the plane as the ray origin. At each node of the BSP tree, we find the intersection, t_{int}, of the ray with the plane at that node. (See the Appendix.) If $t_{min} < t_{int} < t_{max}$, we partition the ray segment at t_{int} (Figure 47(a)). To determine which

```
procedure FindClosestIntersection ( ray : RayRecord ; t_min, t_max : real; v : BSPTreeNode )
                returns Point;

var
        d_min , d_max , t_int : real;
        near, far : BSPTreeNode;
        p : Point;

        if v is not a leaf then

                d_min ← evaluate H_v at point corresponding to t_min
                d_max ← evaluate H_v at point corresponding to t_max

                if d_min > 0 or
                   (d_min = 0 and (d_max > 0)) then

                        near ← v.right;   far ← v.left;
                else
                        near ← v.left;   far ← v.right;

                t_Int ← parameter for intersection of ray and H_v
                        (cf. Appendix)

                if (t_min < t_Int < t_max) and (d_min ≠ d_max ≠ 0) then

                        p ← FindClosestIntersection ( ray, t_min, t_Int, near );
                        if p = ∅ then
                           p ← FindClosestIntersection ( ray, t_Int, t_max, far );
                        return p;
                else
                        return FindClosestIntersection ( ray, t_min, t_max, near );

        else  (* v is a leaf of the BSP tree, and is associated with
                        a representation of the contents of the corresponding
                        cell of the partitioning.  The below call will
                        determine if the ray segment has an intersection with
                        those contents. *)

                return ClosestObjectIntersection ( ray, t_min, t_max, v );

end;  (* FindClosestIntersection *)
```

Figure 46. Algorithm to Ray-Trace a Scene Partitioned with a BSP Tree

subspace to examine first, we must determine which side of the plane the point $D^* t_{min}+T$ lies on, i.e., the near side. If the point at t_{min} is in the partitioning plane, the near side is defined to be that which contains the point at t_{max}. If t_{max} also lies in the partitioning plane, we arbitrarily choose one side as the near side. Since we are interested in the nearest object along the ray, we first recurse on the near subspace with the segment $[t_{min}, t_{int}]$. If the recursion returns with an intersection, we are done. Otherwise, we recurse on the far halfspace with $[t_{int}, t_{max}]$.

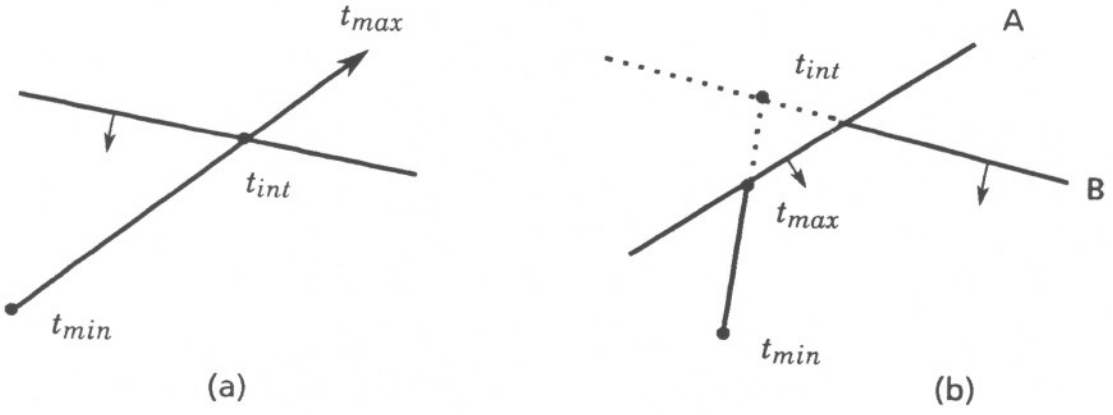


Figure 47. Segmenting a Ray

If the intersection with the partitioning plane does not lie within $[t_{min},t_{max}]$ (Figure 47(b)) we determine that the ray does not intersect the far subspace and therefore we will not search that subspace. Note that the ray must necessarily intersect the near subspace. It must intersect one of the subspaces or else the traversal would never have reached this point. Since the near subspace contains the point determined by t_{min} , we enter the near subspace, returning with the results of that recursion.

Upon reaching a leaf, the ray is tested against the contents of that volume. Since the contents may in general include objects that lie only partially within the volume, we must consider only those surface intersection points within the bounds of the volume. This avoids the potential problem shown in Figure 48. In the figure, object 1 is partially within the partition, but intersects the ray outside the partition. If we simply returned this intersection and halted our search, we would miss object 2, which is closer than object 1 along the ray. The partitioning of the ray has already been computed during BSP tree traversal, so it is a simple matter to test for object intersections lying within the bounds of the ray partition, and hence within the volume.

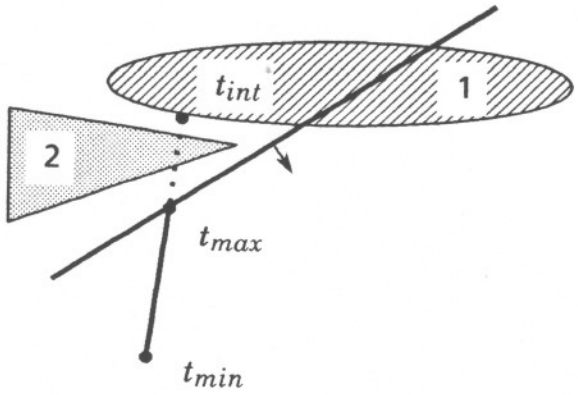


Figure 48. A Problem to Be Aware Of

Splitting objects at partitioning planes can eliminate this problem. This splitting is easily performed with polygons, but is more difficult, for example, with solids defined by quadric surfaces, or other non-linear primitives.

When the ray partition (ray segment) lies entirely in the partitioning plane, the algorithm only searches one subtree. This brings up several issues. The first is that it is possible for objects to intersect partitioning planes. This is a desirable property, since it is not always possible to pass a plane between two or more objects without intersecting any of them. (If one can guarantee no object intersects a partitioning plane, neither subtree need be searched in this situation.) Second, we are only concerned with intersections with the ray that intersect the interior of an object. Tangential intersections are therefore ignored. So, if searching the tree to one side of the partitioning plane yields an intersection, a search of the other tree should yield the same answer. This is the case because the only objects intersecting the partitioning plane that are contained in only one subtree are those that intersect the partitioning plane tangentially.

The choice of which subtree to search can cause aliasing. Noise can be introduced to

reduce the aliasing by flipping a coin to determine which subtree to search in this situation. If adaptive sampling [Mitc87] is being used, then this issue will be handled in the course of antialiasing.

We should expect the number of partitions pierced by a ray to approach the cube root of the number of partitions, as conjectured in [Dipp84]. Experimental results bear this out, as discussed in Chapter VI.

To automatically generate the BSP tree, given the objects in the scene, a technique similar to the median-cut technique discussed in Chapter III could be used. This should result in a nearly balanced tree.

Ray-Tracing of Polyhedra

If we use a BSP tree to represent polyhedral objects, we can use an extension to the above algorithm to provide us with an efficient means of ray-tracing these objects. The method of constructing a boundary-augmented BSP tree from an arbitrary set of polygons is used.

Ray-tracing proceeds as before, except that when returning to an internal node after searching for an intersection in the near subspace, we test the ray against the polygon(s) lying in the partitioning plane. Since the ray's intersection with the partitioning plane was determined when we first visited the node, we need only test that point for inclusion within the polygon's boundaries when we return to the node. If this test fails, we proceed to the far subspace. (We can use a 2-d BSP tree to represent the polygons in the hyperplane. Testing the intersection point for inclusion in the polygon(s) is then performed with the point classification algorithm presented in Chapter III.)

If we know that the polygons form a closed surface, we can use this method to represent polygonal solids. We classify segments of the ray as lying within the solid by counting the intersections with polygons along the ray. (Care must be taken to insure that

only one intersection is returned when the ray passes through the juncture of two or more polygons.) The polygonal solid can then be incorporated into a CSG modeler (discussed below) that performs solid evaluation during ray-tracing.

It is also possible to ray-trace polyhedra represented by labeled-leaf BSP trees, which were described in Chapter III. To ray-trace such a solid, we traverse the tree as before until the first *in*-cell is encountered. (This is assuming the ray origin is in an *out*-cell. Were it in an *in*-cell, the ray origin would be embedded in the solid. This is not an unreasonable situation, however, when considering propagation through light-transmitting objects. In that case, the tree is traversed until encountering the first cell of a different type than the one containing the ray origin.) The value of t_{min} at this point is then the parameter value for the visible surface, and the traversal can be terminated.

This is essentially the line classification procedure described in Chapter III, with the exception that the process always recurses to the near side first, and terminates when the first *in*-cell is found. In that algorithm, a segment lying entirely in a partitioning plane must be classified with respect to both trees. Here, however, we ignore *on* segements, as these reflect tangential intersections. As above, we arbitrarily choose one subtree to search.

Using BSP Trees in the Ray-tracing of CSG Models

CSG evaluation is often performed, not as a representation conversion, but during the rendering (ray-tracing) operation itself. Each ray is intersected with all primitives of the CSG tree, yielding a number of line segments along the ray. These line segments are then used to evaluate the set operations in 1-d for that ray. An example is illustrated in Chapter II. This is also called *ray-casting*.

BSP trees can be used to facilitate ray-tracing of CSG models in three fundamental ways. The first is where the partitioning planes are not part of the visible model. First consider a model composed of a set of objects each of which is modeled by a CSG tree

(DAG's can also be used for a more economical representation). If each object lies in one cell of the BSP tree, we can place their respective CSG trees at the leaves of the BSP tree. The "object" represented by the BSP tree is the (disjoint) union of the objects at its leaves. This also suggests that the BSP tree could serve as a "primitive" to a higher level CSG tree, which could in turn be at the leaf of a still higher level BSP tree, and so on. We call a BSP tree whose parent is a modeling operation an *embedded* BSP tree.

The second use is where a polyhedral primitive of a CSG tree is represented by a labeled-leaf BSP tree. This is, in fact, a special case of an embedded BSP tree. However, using the polyhedron as an argument to set operations (to be evaluated by ray-casting) prevents the termination of the search upon encountering the first intersection point with the primitive. In general, all intersections must be returned: this can be considered as the distinction between ray-casting (using intersections with a ray to solve the set operation in 1D) and ray-tracing (generating a visible surface rendering). There are a few exceptions to this: when the path from the root of the CSG tree to the embedded BSP tree consists only of union nodes and/or the left hand side of difference nodes. Then we know that the nearest intersection point is indeed the surface of the object represented by the CSG tree.

Thirdly, we can speed up ray intersections with a complex object represented by a CSG tree by partitioning it with a BSP tree and simplifying the CSG trees at the leaves. This requires determining which partitions (convex polyhedra) are completely full or empty with respect to some primitive(s), and then applying the simplification rules, as was done in Chapter IV.

CHAPTER VI

Implementations

CSG evaluation

The CSG evaluation algorithm has been implemented in a dialect of Pascal running under BSD 4.3 UNIX. A CSG expression is specified in a special-purpose language implemented with the UNIX compiler building tools **yacc** and **lex**. Figure 49 shows an example of an object description in the language. The operators union, intersection, and difference are indicated by the characters "|", "&", and "-". Transformations are preceded by the character "*". The user essentially defines a DAG (Directed Acyclic Graph) by defining named objects and referring to these from other objects. Primitive objects are filenames of files containing a simple boundary representation of the object, consisting of a list of polygons, each represented by a (counterclockwise ordered) list of vertices. Operations that can be used to construct more complex objects include: the transformations scaling, rotation, and translation; the assignment of a color to the object (as red, green, and blue components); the set operations union, intersection, and difference; binding an object definition to an identifier, and referring to a previously defined object by its identifier. A special identifier, "model," must be assigned, as it defines the root node of the DAG. The DAG so defined is expanded into a tree by applying the transformations, associated with each path to a primitive object, to the vertices of the boundary representation. This results in a tree whose internal nodes are set operations and whose leaves are boundary representations.

A textual representation of this tree is input to the program, along with a number of parameters that specify the heuristic to use in hyperplane selection. The *candidate set* is

```
object cube is file /usr/tebo/part/data/prims/cube end;
object cyl is file /usr/tebo/part/data/prims/cyl20 end;
object cone is file /usr/tebo/part/data/prims/cone20 end;

object boundingbox is
        cube * (scale 16 16 1 about 0 0 0; translate -8 -8 -.5)
end;

object outeredge is
        cyl * (scale 7.5 7.5 1 about 0 0 0)
end;

object blank is
        boundingbox & outeredge
end;

object mounting_holes is
        cyl * (translate -4 -4 0)
        |cyl * (translate -4 4 0)
        |cyl * (translate 4 -4 0)
        |cyl * (translate 4 4 0)
end;

object intake is
        cone * (translate 1 0 -2.3)
        |cone * (translate -1 0 -2.3)
end;

object clutchplate is
        blank color 1 0 0
        - (mounting_holes color 0 1 0 | intake color 0 0 1)
end;

object model is
        clutchplate  * (scale 0.2 0.2 0.2 about 0 0 0)
end
```

Figure 49. An Example of the CSG Object Description Language

defined to be those faces in the current partition that are to be considered for generating partitioning planes. The *test set* consists of the entities (individual faces or spheres enclosing entire primitives) against which each candidate plane is tested. Each set is specified by a selection strategy that is to be applied to each primitive in the CSG tree of the current (sub)problem. The selection strategy specifies if faces of the primitive are to be chosen at random or in the order they appear on the list of faces of the boundary representation. In

either case, the number of faces of the primitive to be considered is specified as either a number or as a percentage.

The possible outcomes of testing a candidate plane to a member of the test set are "front", "back", or "split", according as the test set entity lies to the front of the plane (in the positive halfspace), in back of the plane, or is intersected by the plane. The heuristic is a function of the number of outcomes of each type that occurred when a candidate was tested against the members of the test set. The candidate chosen is that member of the candidate set that maximizes the heuristic. Four heuristic functions were investigated:

$$Heur_1\ (front,back,split) = (-|back - front|) - w_{split} * split$$

$$Heur_2\ (front,back,split) = (front * back) - w_{split} * split$$

$$Heur_3\ (front,back,split) = front - w_{split} * split$$

$$Heur_4\ (front,back,split) = - split$$

The weight w_{split} is a parameter that allows "tuning" of the heuristics. In all heuristics, the number of test set elements split by the candidate plane are weighted negatively. This is because the splitting of a polygon requires both extra work and extra storage. The first two heuristics try to balance the sizes of the two sides of the candidate. The third tries to maximize the number of polygons in the front halfspace of the candidate. The fourth simply tries to minimize the amount of splitting at each stage. The third heuristic is motivated by the fact that the front halfspace of a face of a convex primitive will be in the exterior of the primitive. Therefore, the primitive will reduce to a 0 or 1 in the sub-problem for the front halfspace, thereby allowing us to simplify the tree. Choosing a plane that maximizes the size of the test set that lies to the front of the plane attempts to maximize the extent to which the pruning will be helpful.

Figures 50, 51, and 52 show the objects used in the test runs. The tests were run on a VAX 8650 at AT&T Bell Laboratories. In all runs, the candidate set consisted of 5 faces

chosen at random, and the test set consisted of 100% of the current set of faces. The program was run on each object using each heuristic, with $w_{split} = 0,2,4,\ldots,20$. (For $Heur_4$, only one run was made per object, since w_{split} is not used.) Statistics on four quantities were collected: the cpu time required by the program, and the size, height, and number of polygons in the output BSP tree. The first data point in each graph contains a significant amount of noise.

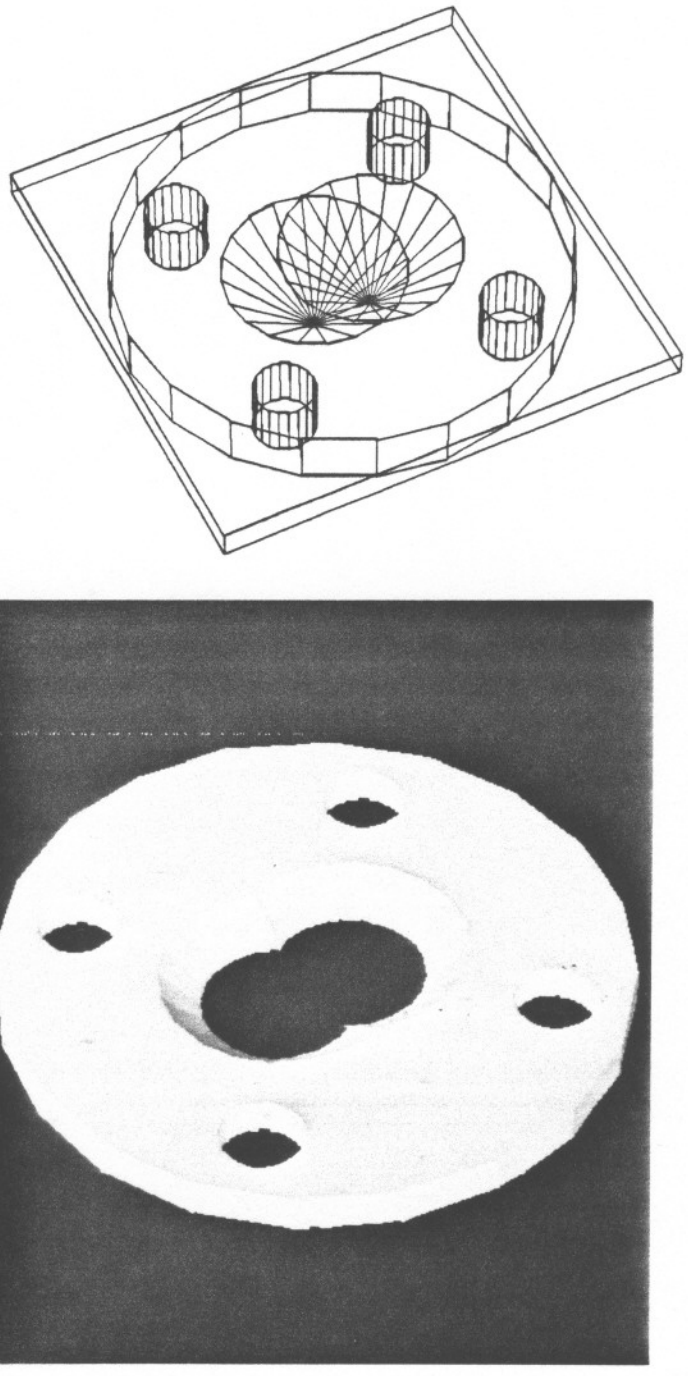


Figure 50. Test Object 'Clutchplate': 8 Primitives, 158 Polygons

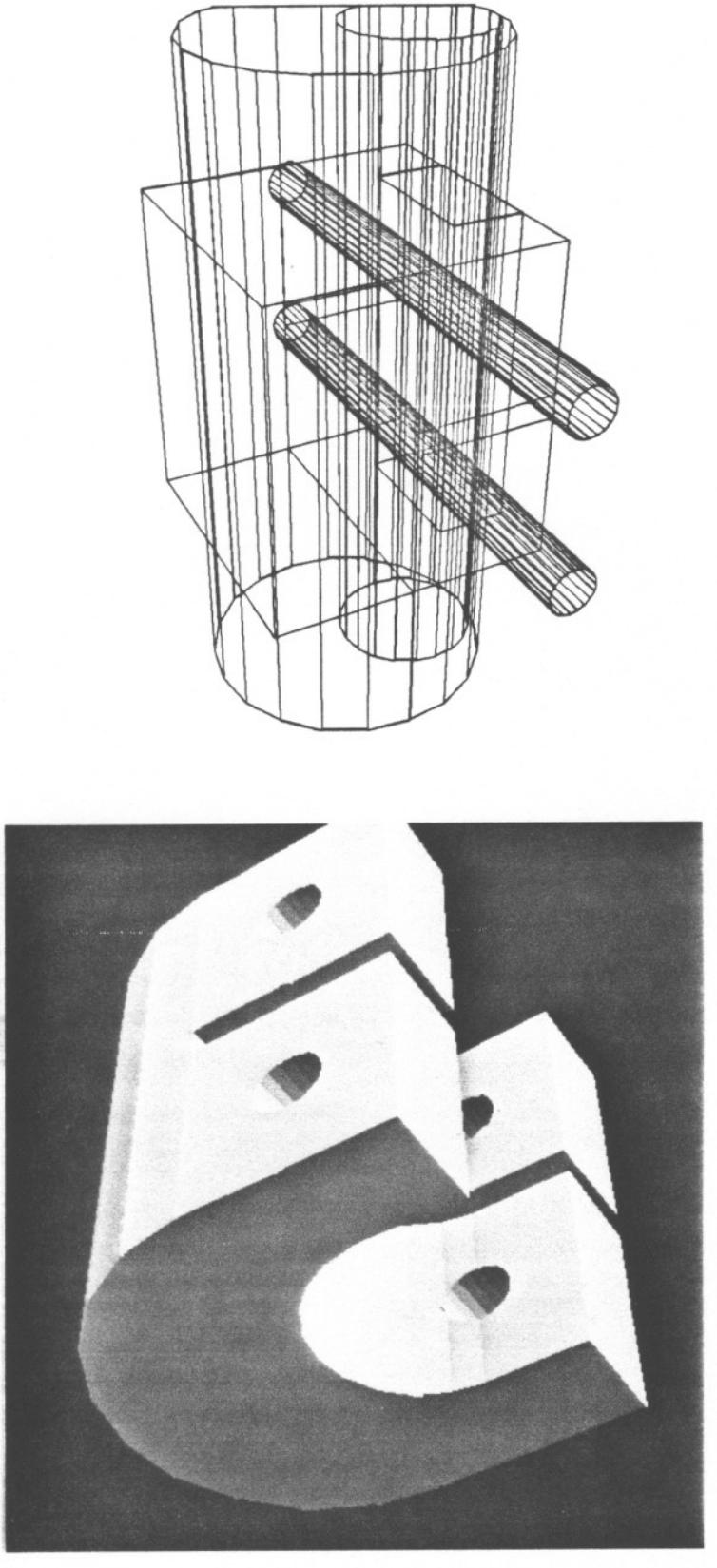


Figure 51. Test Object 'Bracket': 7 Primitives, 106 Polygons

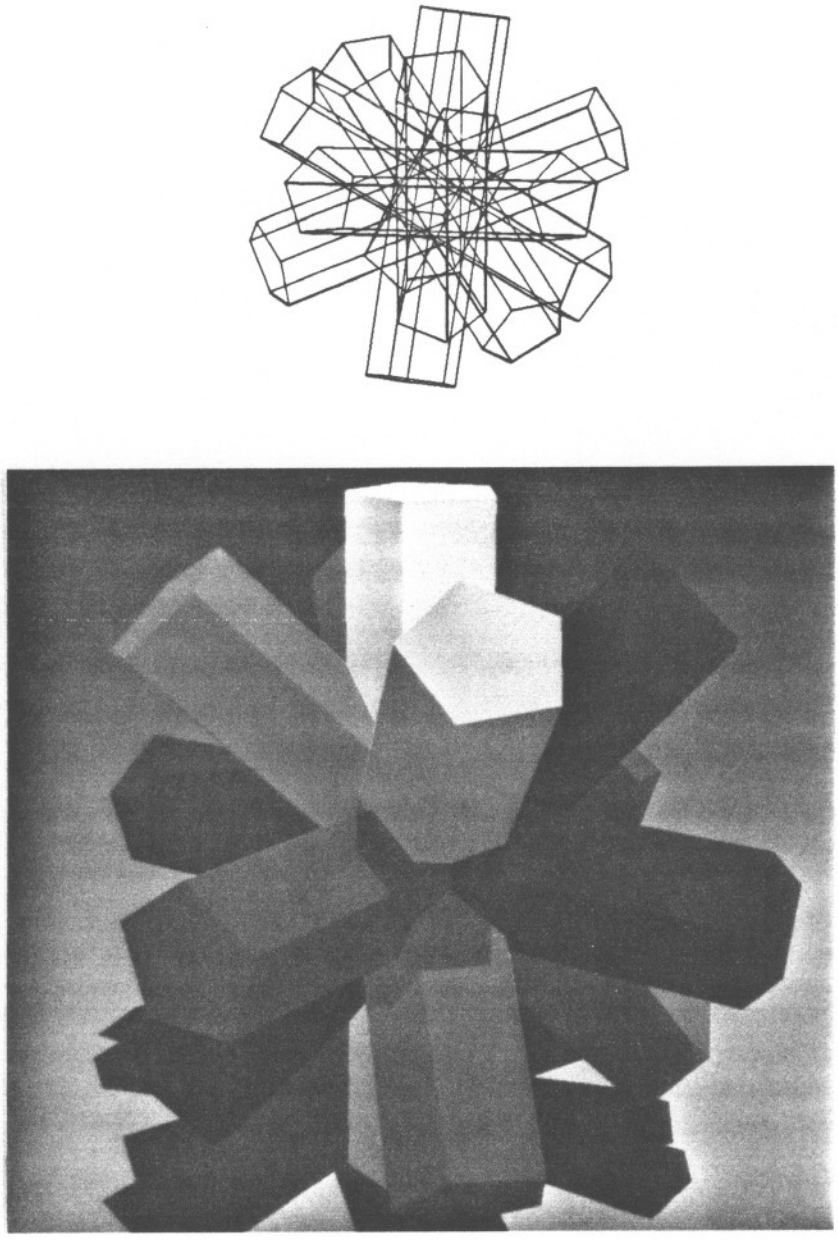


Figure 52. Test Object 'Brush': 7 Primitives, 49 Polygons

The graphs in Figure 53 show the CPU time (in CPU seconds) required by the program for a range of values of w_{split}. $Heur_2$ performs best on all test objects.

Figure 54 shows how the tree size, in number of nodes, varies with heuristic and w_{split}. $Heur_4$ performs best here, simply by choosing planes that minimize the number of split polygons.

Figure 55 shows the height of the BSP trees produced. Here, $Heur_2$ gives the shortest trees.

The results for the number of boundary polygons are less clear (Figure 56). $Heur_2$ performs worst out of all heuristec on the clutchplate, and best (for some values of w_{split}) on the bracket. $Heur_4$ performs well in this respect on all objects.

This early experience shows that $Heur_2$ is the best in terms of cpu time. $Heur_2$ sometimes produces trees with a larger number of nodes, but with less CPU time than is required by the other heuristics for the same objects. We speculate that using the product (*left* * *right*) in $Heur_2$ favors equal sized sub-problems at early stages of the evaluation. This improves the time requirements, but results in more split polygons, thereby increasing the size of the BSP tree.

It is also the case that the largest variation in the statistics occurred for low values of w_{split} (0 through 4). This indicates that the "control" exerted by the negative weighting of split polygons does not occur unless the coefficient w_{split} exceeds a threshold of about 4, and increasing it past this point exerts relatively less control.

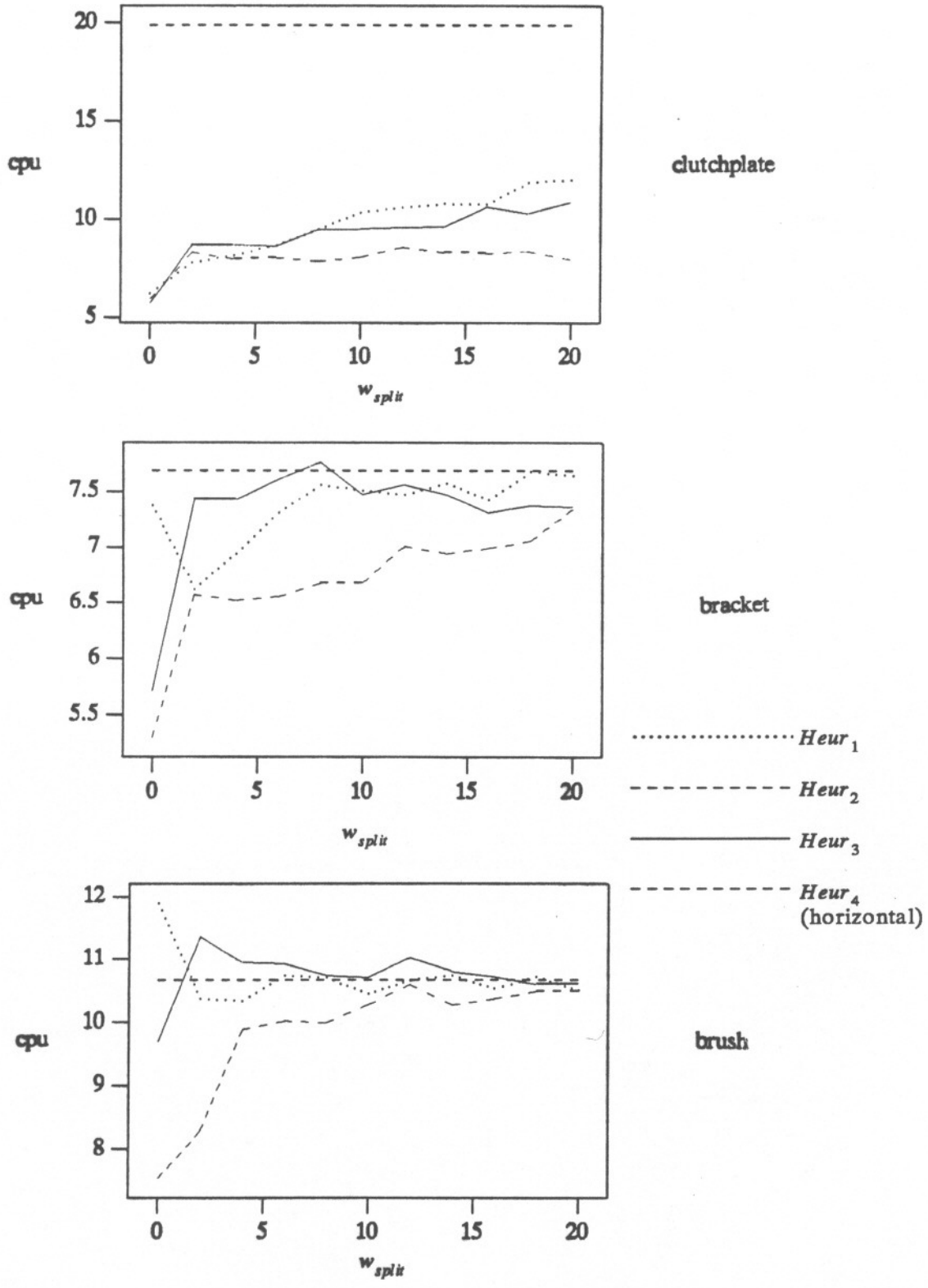


Figure 53. CPU Seconds vs. w_{split}

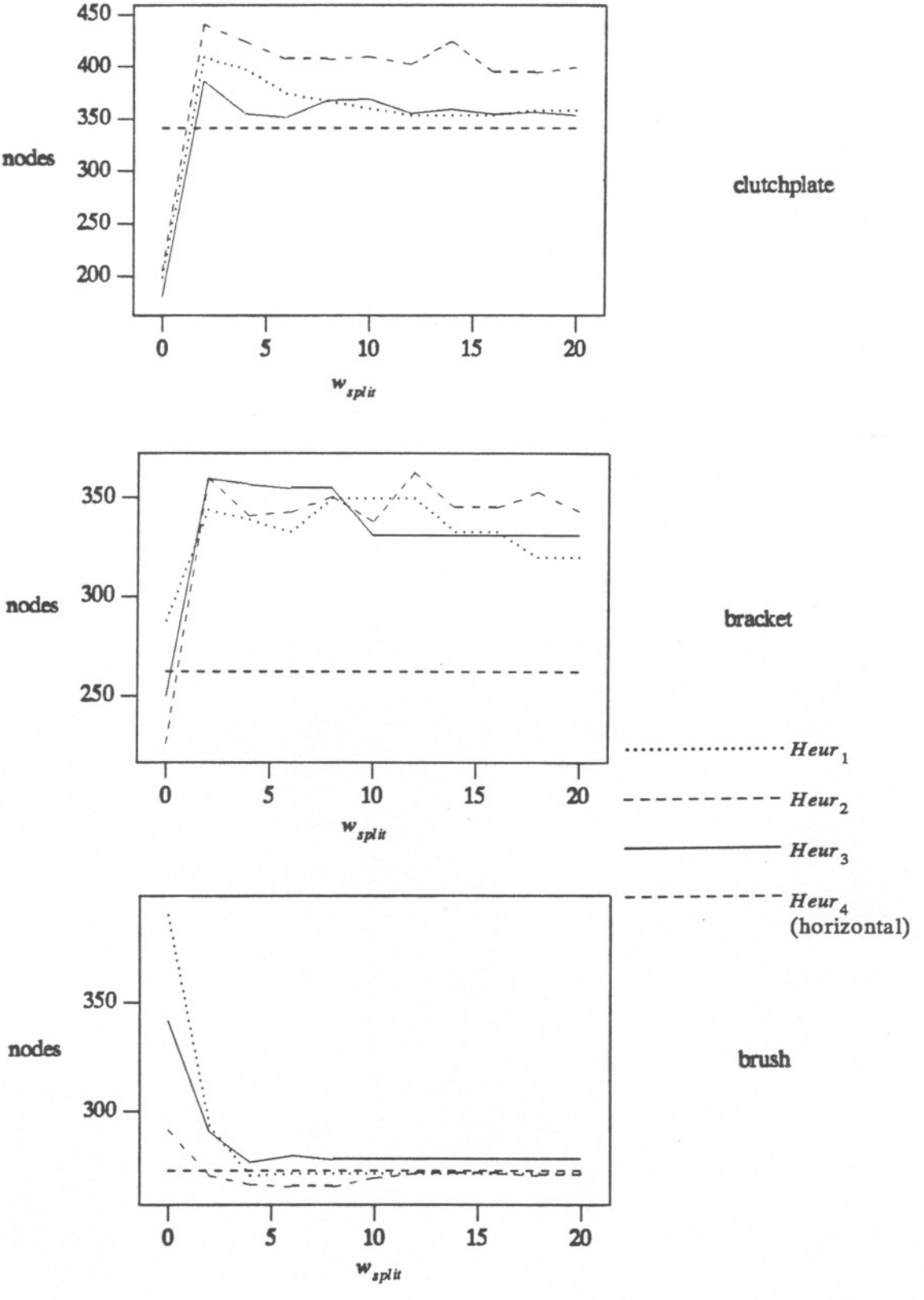


Figure 54. Nodes in BSP Tree vs. w_{split}

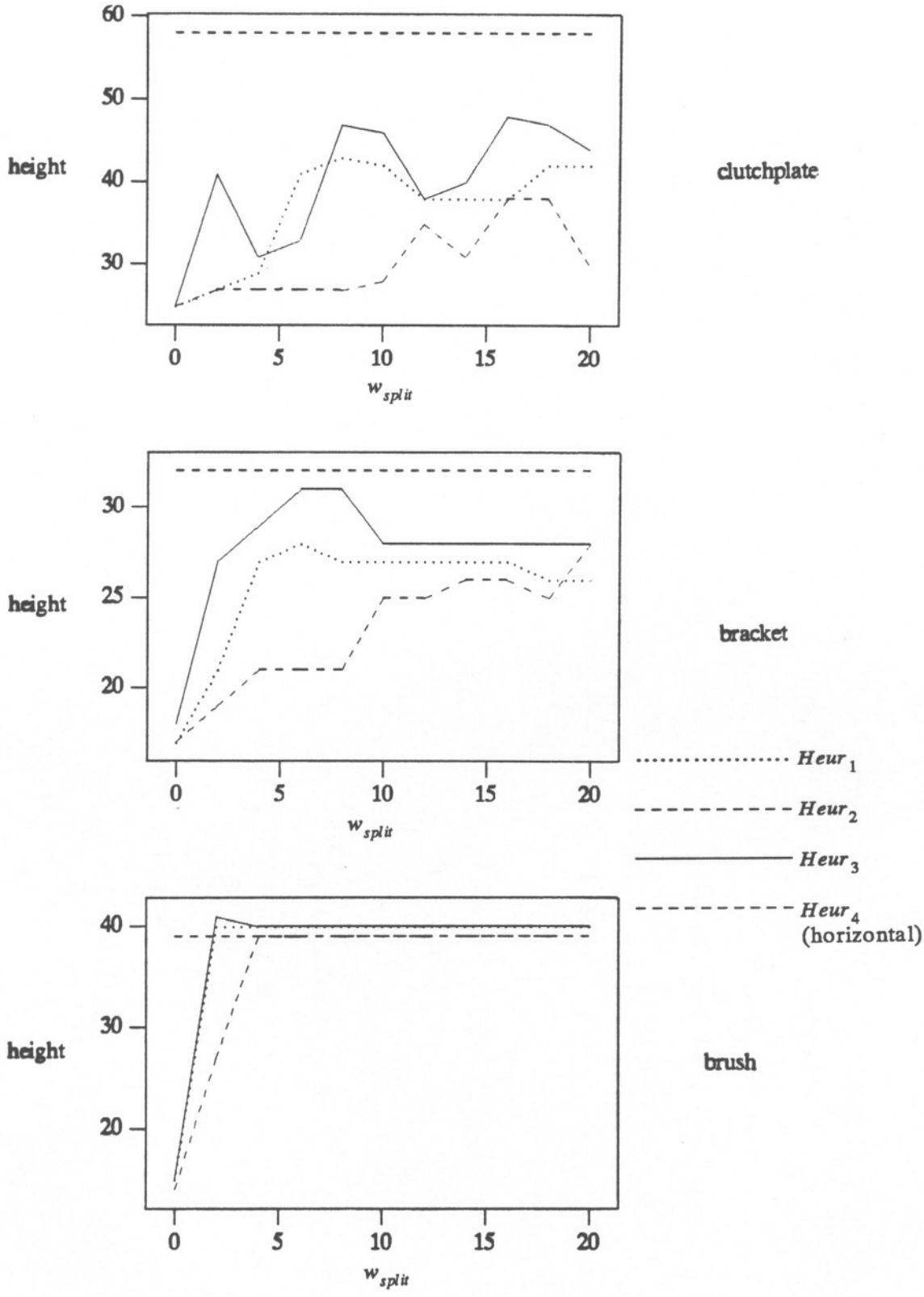


Figure 55. Height of BSP Tree vs. w_{split}

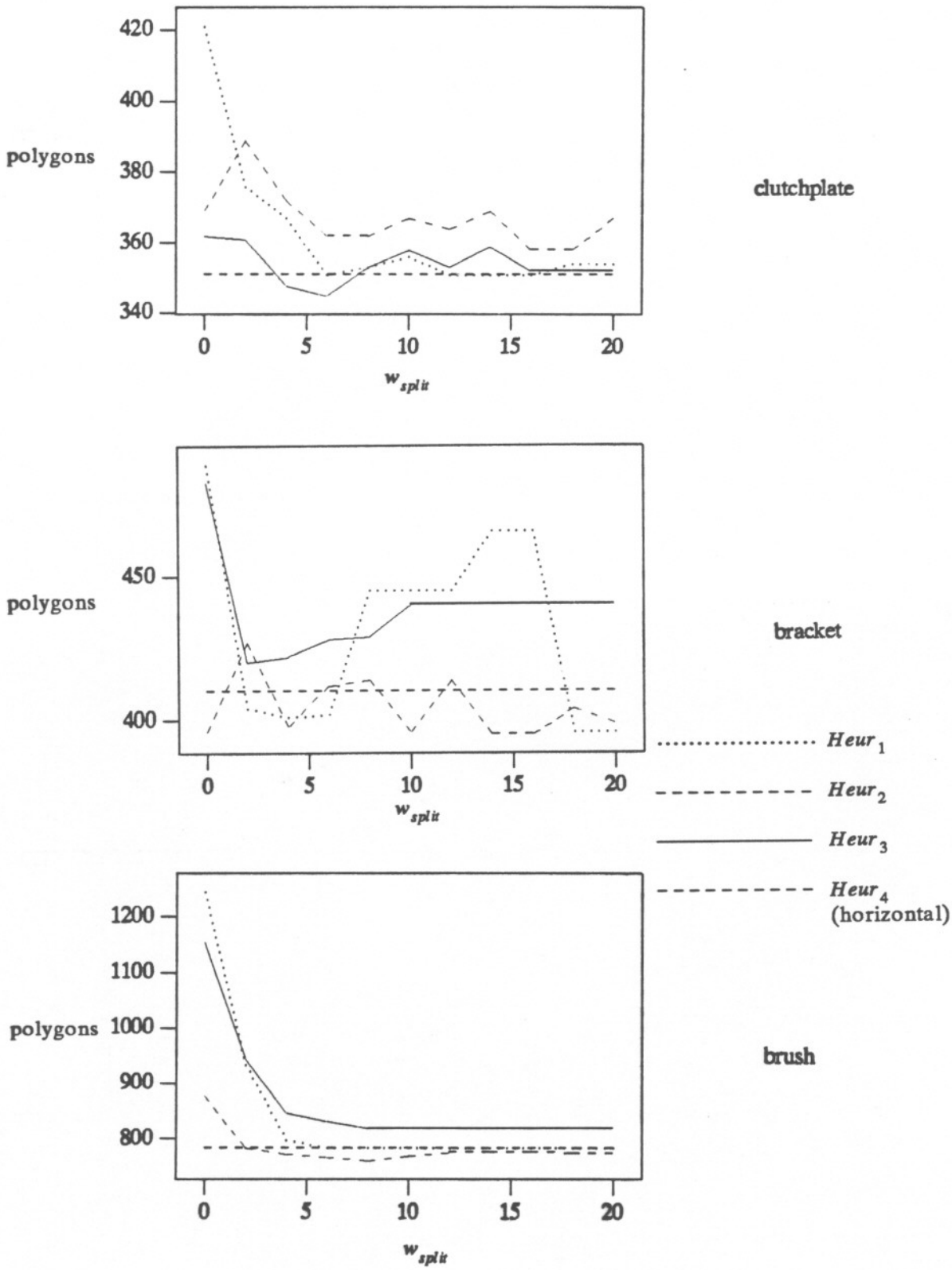


Figure 56. Number of Boundary Polygons in BSP Tree vs. w_{split}

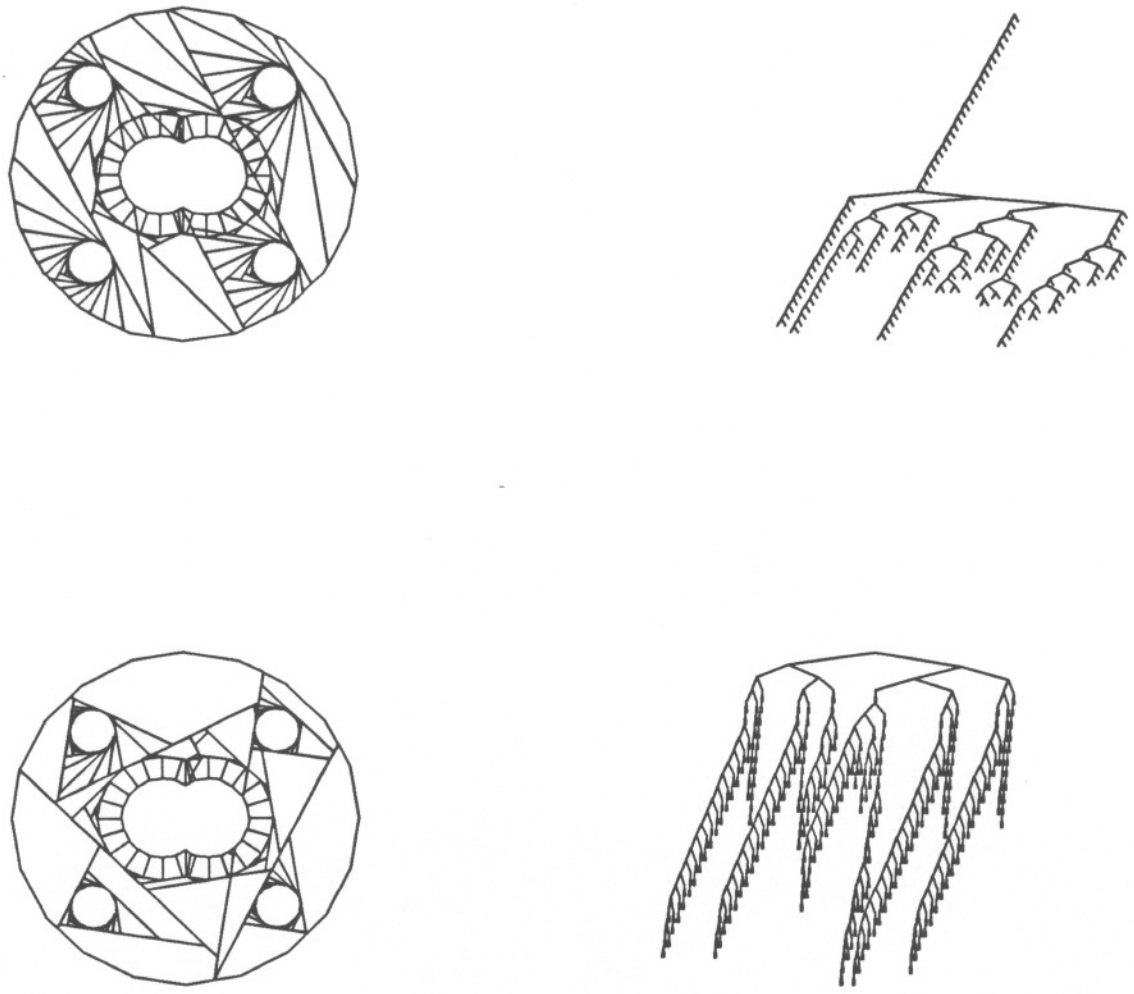


Figure 57. Two Different BSP Trees Describing the 'Clutchplate'

To illustrate how different the BSP trees describing identical sets can be, Figure 57 shows two different BSP tree representations for the "clutchplate" object. (The binary trees in the figures in this chapter were drawn using the algorithm of Reingold and Tilford[Rein81].) This first tree resulted from selecting the first polygon in the leftmost primitive of the CSG tree at each stage; no attempt was made to find the "best" hyperplane. In other words, the candidate set was of size 1, and the test set of size 0. The long string of nodes at the root of the tree resulted from choosing hyperplanes that bound the object, and define a convex set containing the entire object. This was due to the fact that the primitives

defining the outer boundary of the clutchplate occupied the leftmost positions in the tree. The second tree was produced by using $Heur_2$, with maximal test and candidate sets. (That is, all polygons were tested against all others at each stage.) This tree is more well-balanced than the first, as would be expected. Also note how the use of the two "balancing" heuristics serves to localize the primitives into regions relatively high in the tree. The splitting caused by the faces of each primitive is then localized.

In the above example, the use of heuristic $Heur_2$ produced a BSP tree with fewer boundary polygons (366 polygons) than the tree produced by simply choosing the "next" face (434 polygons). In the example in Figure 58, however, this is not the case. By simply choosing the "next" face at each stage, a tree with a fewer number of polygons results (at the top of the figure, with 247 polygons) than if heuristic $Heur_2$ is used (at the bottom of the figure, with 314 polygons). This may be due to the particular geometry of the brush object. Most of the splitting of polygons occurs near the center of the object, where all of the primitives intersect. Choosing a single primitive at a time to contribute partitioning planes encloses a portion of this region, creating a sort of "firewall" across which no plane may split other polygons. This serves to localize the effects of splitting early in the process, reducing the total number of (split) polygons.

The worst case number of polygons in a BSP tree is $\Omega(n^d)$, where n is the number of input faces and d is the dimension. However, the results obtained, even for relatively "bad" cases such as the bracket and brush, are not as large as the worst case result. The brush object, which contains a high number of intersecting primitives, has about a factor of 16 increase in the number of polygons when a heuristic is used. This performs the worst of the objects studied. The bracket also has a fair amount of intersecting primitives, but shows only a factor of 4 increase. The clutchplate has relatively few intersecting primitives, and the number of polygons increases by a factor of 2.3.

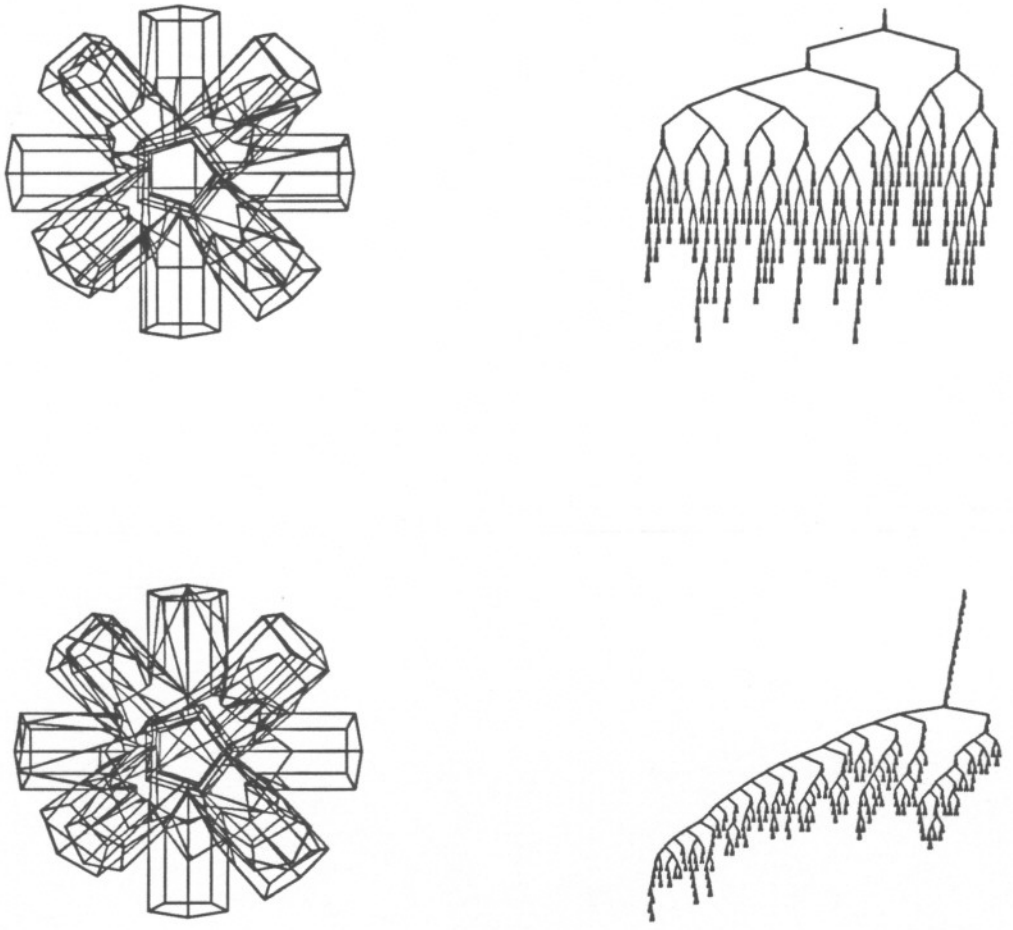


Figure 58. Two BSP Trees Describing the 'Brush'

The same increase as that experienced with the clutchplate (2.5) was experienced as a practical worst case in converting a B-rep to a BSP tree (no set operations) in our implementation of that algorithm. The same practical bound was experienced by Fuchs[Fuch83]. This suggests that the degree to which the primitives in a CSG representation intersect is a major factor in the increase in the number of polygons.

Incremental evaluation

The algorithm for incremental set operations has been implemented on a Silicon

Graphics IRIS workstation. The configuration used has a 68020 CPU running a 16MHz, a floating-point accelerator board, and 2 Mb of memory. The system runs a version of UNIX based on System V. The IRIS uses eight "Geometry Engine" chips organized in a pipeline that performs transformation and clipping of polygons. A separate graphics processor draws (scan-converts) polygons into the frame buffer. This all makes for an extremely effective polygon drawing system.

In the implementation, the user can interactively control the position of the viewpoint and a "tool" object with the mouse. While positioning the tool object, the user is appraised of its current position by evaluating a "cheap" union of the current object with the tool at its current position, and displaying a visible surface rendering of the result. This is "cheap" in the sense that we do not bother to re-evaluate boundary polygons in nodes of the tree that split or are coplanar with faces of the tool object. For union, this re-evaluation would only serve to eliminate faces in the interior or redundant faces on the boundary of the result of the union. As these faces are obscured during the process of writing front-facing polygons in back-to-front order (yielding the visible surface), and writing polygons to the frame buffer is relatively inexpensive on the IRIS, this serves to reduce the amount of time needed to show the current position of the tool.

Tree simplification is not performed during tool positioning. This is to prevent the deletion of any part of the workpiece's BSP tree until the actual set operation is performed. A list is kept of the leaves of the workpiece tree that are replaced by subtrees generated from faces of the tool. Since the operation is set union, this will happen only at out-leaves of the workpiece tree. When the user moves the tool again, this list is used to prune these subtrees, replacing them with out-leaves. The "cheap" union is then performed with the tool at its new position.

Once the tool is positioned, the user chooses a set operation to be performed. Having

done so, the result is evaluated, and the tree is displayed. Performing a set operation creates a new tree; the old tree is kept in case the user decides to undo the operation.

The IRIS's three-button mouse is used for all user input. There are two basic modes of interaction, one, "Viewing mode," in which the mouse position controls the viewing position, and the other, "Tool mode," in which the mouse controls the x-y position of the tool. The rightmost button is reserved for use by the window manager **mex**. Depressing this button causes a "pop-up" menu to appear under the current cursor position. While keeping the right mouse button down, the cursor can be moved over the menu. When the cursor is positioned over the desired operation, releasing the mouse button causes the corresponding action to be performed. Main menu items include:

View	-- enter Viewing mode
Tool	-- enter Tool mode
UNION	-- perform a union operation with the tool at the current position
INTERSECT	-- perform an intersection operation
SUBTRACT	-- perform a subtraction (workpiece -* tool)
drawtree	-- draw the BSP tree as a binary tree in a subwindow
fix faces	-- perform a glue operation in all hyperplanes in hopes that some 2-d tree reduction will take place, reducing the number of polygons (usually *increases* their number)
Save	-- save the current BSP tree on disk
Quit	-- exit the program

The "View" and "Tool" menu items have sub-menus that are activated by moving the cursor off the edge of the menu while keeping the right mouse button depressed. The submenu items for "View" toggle characteristics of the visible surface rendering. These

include:

edges -- highlighting of polygons edges

light -- recalculate shade of each polygon as the model is rotated beneath a fixed light source

filled -- draw filled polygons (edges=on and filled=off generates wireframe drawings)

color -- draw filled polygons with their individual colors (color=off generate monochrome renderings)

rotate -- rotate the model at a constant rate when in tool mode

While in "Tool" mode, one of several different sub-modes is in effect. In all sub-modes, the mouse position controls the x and y position of the tool. The action that is associated the left and middle mouse buttons depends on the current sub-mode:

"z-translate" -- the default, in which the left and middle mouse buttons control the z position of the tool.

"rotate"-- in which the buttons control rotation of the tool about the x and z axes

"scale-all", "scale-x", "scale-y", "scale-z" -- where the buttons control scaling about x, y, z, or all axes.

Other submenu items reachable from the "Tool" entry in the main menu include the options :

"new" prompts for a new tool polyhedron

"newcolor" prompts for a new color for the tool

"vis" toggles the visibility of the tool

"constant"	prompts for a set operation. Subsequently, that set operation is performed for each new position of the tool. "union" makes the tool like a 3-d brush. "diff" makes it like a cutting tool. Selecting "constant" a second time turns it off.
"init"	initializes the tool to unit scaling, no rotation, and positioned at the origin.

For simple objects, moving the tool and generating visible surface renderings is done at 15-30 frames per second. For the most complex object tested (about 2000 polygons), the system generates frames at about 1 frame per second, most of this in the visible surface phase. Evaluation of a set operation on this object takes about 6 seconds.

A 6 minute video tape was made demonstrating the IRIS implementation[Thib87a]. The tape includes segments from live interactive sessions.

Ray-tracing

The ray-tracer is written in a dialect of Pascal. It is based on a simple ray-tracer written by Bill McAllister. Extensions to support CSG evaluation (along rays), arbitrary quadric primitives, color, texture, and BSP trees were made.

The image in figure 59 was generated by ray-tracing a labeled-leaf BSP tree output by the CSG evaluation program. The images in Figures 60 and 61 were generated by ray-tracing labeled-leaf BSP trees. The multiple (botanical) trees in Figure 61 were generated by performing transformations (rotations and translations) to a single boundary-augmented BSP tree representing a single botanical tree[Aono84].

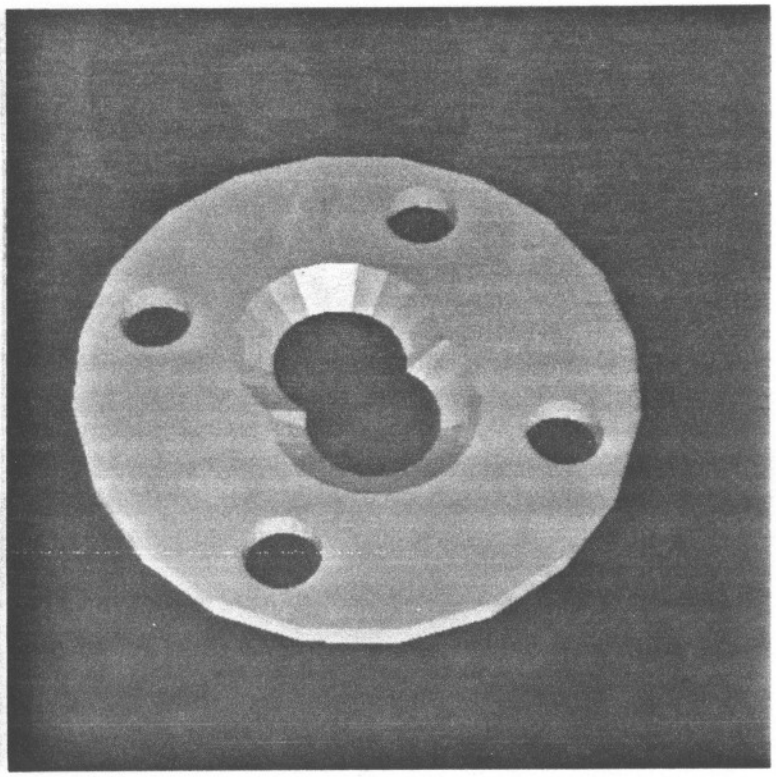


Figure 59. Ray-Traced Clutchplate

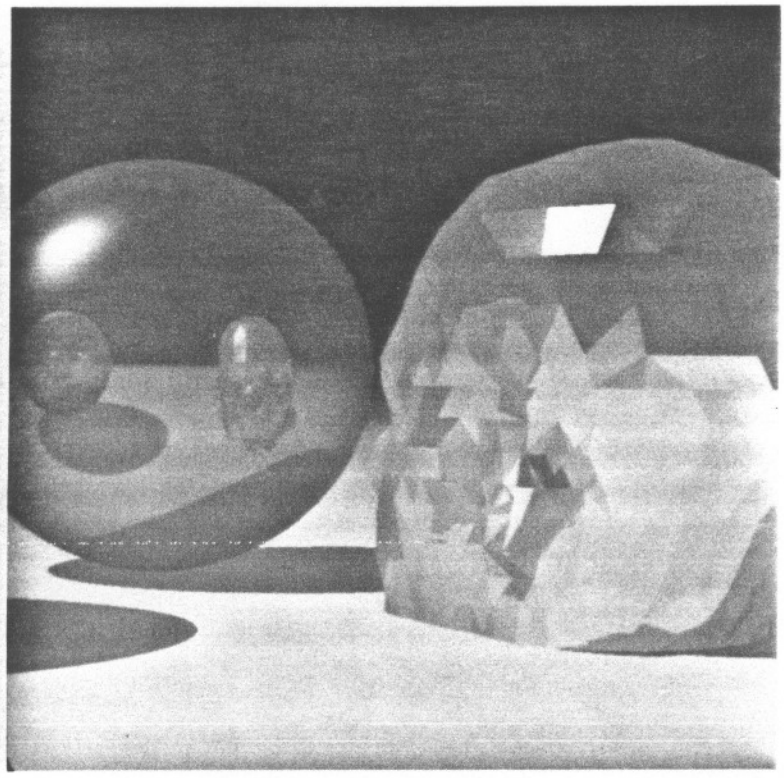


Figure 60. Ray-Traced Head

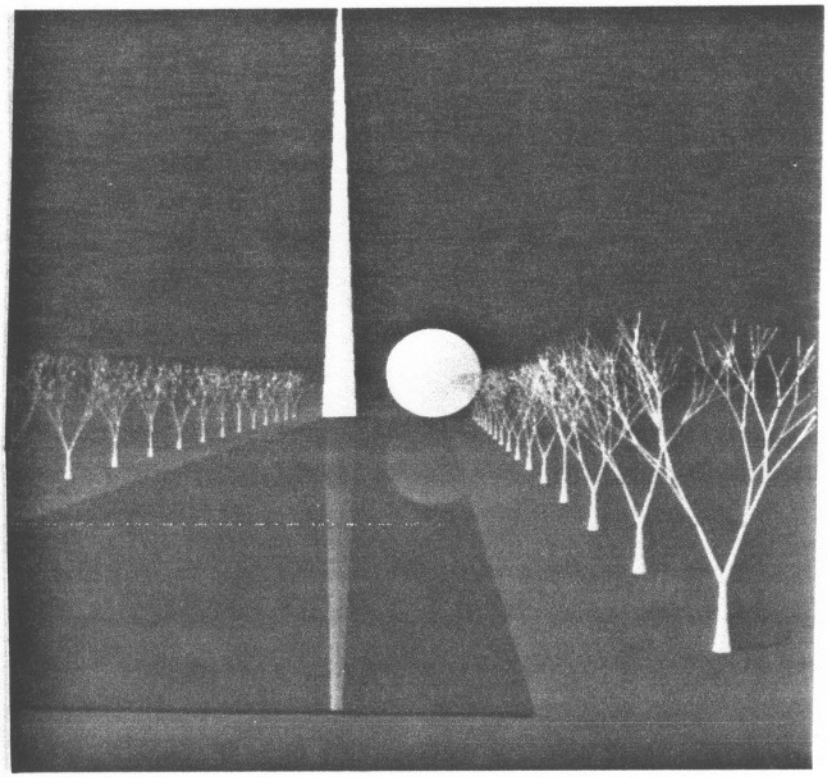


Figure 61. New York World's Fair

Performance

A series of tests were run to demonstrate that the BSP tree can yield performance improvements over ray-tracing without any type of partitioning. The results are summarized in Table 5. The BSP tree was constructed in an octree-like fashion, cycling through the dimensions, dividing along one dimension with an axis-aligned plane at each level of the tree. The objects used in the tests were spheres, five of of which were placed randomly within each cell of the partitioning. As can be seen from the Table, running time increases linearly with the number of objects when no partitioning is used, while the time requirement increases more slowly when the BSP tree is used. Any reasonable implementation of a ray-tracer must include some form of space partitioning in order to render any but the simplest scenes. The simplest and most common approach is to use a hierarchy of bounding volumes, usually spheres or ellipsoids. A comparison of the various existing techniques with the BSP tree is major undertaking, left for future investigation.

TABLE 5. Run Times for Ray-Tracing an Unpartitioned Scene versus a Scene Partitioned with a BSP Tree

number of partitions	number of objects	with BSP tree (cpu minutes)	without BSP tree (cpu minutes)
8	40	8	30
16	80	14	66
32	160	18	119
64	320	21	237
128	640	30	-
256	1280	39	-
512	2560	46	-
1024	5120	55	-

Set Operations

Set operations can be evaluated for each ray by *ray-casting*, as outlined in Chapter II. This is done by combining the intersections of the ray with the objects in the scene. With this technique, we can evaluate set operations on any primitives for which we have a method of determining their intersection with the ray. In this implementation, this evaluation was performed using a finite state automaton.

The image in Figure 62 shows a complex object evaluated by ray-casting. The space shuttle is represented with a boundary-augmented BSP tree. The impressions in each block were made with a subtraction operation (*shuttle -* block*).

The input to the automaton consists of the intersection points of the ray with the two objects, sorted by distance from the ray endpoint. Each intersection point is marked to indicate which operand it belongs to. The automaton outputs a 1 on a state transition to indicate that the current intersection point is an intersection point with respect to the result of the set operation, and a 0 to indicate that the current point is not in the result. The result is then also a sorted list of intersection points, which could be used as input to another evaluation.

States of the automaton reflect the classification of each line segment induced on the ray by the intersection points with the two sets. ("On" classifications are not considered for these line segments: an arbitrary decision is made to consider such tangent intersections as lying on the exterior of the set.) Transitions correspond to intersection points.

Specifically, a state consists of three bits. The first bit indicates the classification of the line segment with respect to the result of the set operation. For example, a 1 in the first bit indicates the line segment is in the interior of the result. The setting of this bit is determined by the particular set operation and the settings of the two remining bits. These two bits indicate the classification of the line segment with respect to each operand. There

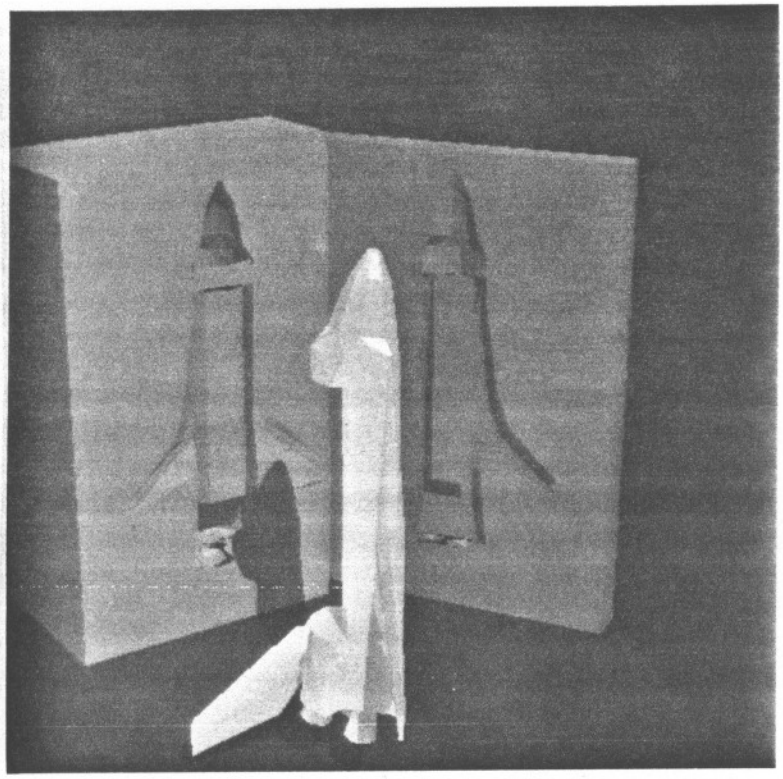


Figure 62. Shuttle Mold

are four states in each automaton, reflecting the fact that there are four possible ways to

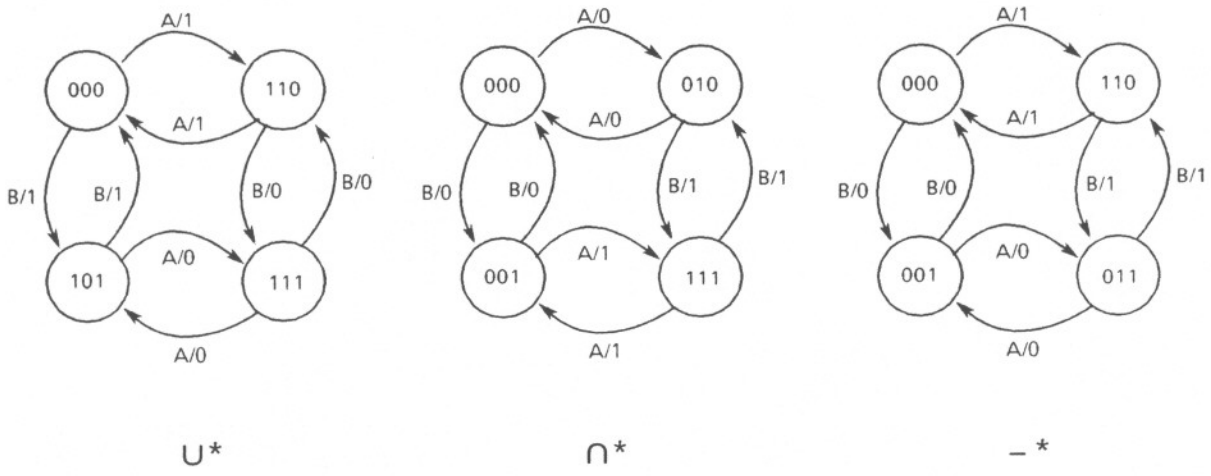


Figure 63. Automata for Evaluating 1D Set Operations

combine the two operands. The starting state for a particular evaluation is chosen based on the location of the ray tail with respect to the two objects.

Figure 63 shows the automata for the (regularized) operations union, intersection, and difference. Consider the automaton for union, for example. Suppose the ray tail is in the exterior of both operands. This means the automaton starts in state 000, signifying that we begin outside of both operands (the two rightmost bits) and outside of their union (the left bit). If the first intersection along the ray belongs to operand A, we transition to state 110 (in A $\cup^*$ B, in A, out of B), and output a 1, indicating that this first point is an intersection of the ray with A $\cup^*$ B.

If the ray intersects both objects at the same point, the automaton may output both points (for intersection and difference). To handle this, the list of intersection points output by the automaton should have any coincident points deleted.

Construction of BSP Trees with Hyperplanes that do not Embed Faces

As mentioned in Chapter III, it is possible to construct BSP trees with hyperplanes that do not embed faces of the object. One reason for doing so is to attempt to construct balanced trees. This can be accomplished with a "median-cut" algorithm, similar to that used to construct balanced k-d trees for point sets [Bent79]. At each stage of the process, a hyperplane is chosen that is orthogonal to the i-th coordinate axis. The axis to split at each stage is chosen either by cycling through the d dimensions, as in the k-d tree[Bent79], or by choosing the axis along which the current set of faces have the longest extent. The hyperplane intersects this axis at the median value of the i-th coordinates of the vertices of the current B-rep. (This median is found with the algorithm SELECT on pages 97-99 of [Aho74].) A threshold is defined in terms of the number of faces in the current B-rep. Below this threshold, hyperplanes that embed faces are used, as in procedure Build_BSPT. Figure 64 shows two BSP trees representing a 50-face approximation of a sphere, one built with the naive "face-at-a-time" approach, and the other with the median-cut algorithm with a threshold of 8.

Another reason for using non-face-embedding hyperplanes is to construct a (convex) bounding volume about the set represented by the BSP tree. The bounding volume can be used to quickly determine when an object lies entirely outside of the set represented by the BSP tree. This involves constructing a BSP tree describing the bounding volume, and rooting the tree of interest at what would be the *in*-cell of the tree for the bounding volume. Note that such hyperplanes would be removed by the BSP tree reduction rule that removes nodes whose hyperplanes do not contain part of the boundary and have a trivial BSP tree (leaf) as a child. Support for bounding such bounding volumes is implemented in the incremental set operation system by simply disallowing tree reduction on the top 6 levels of the tree (used to define an axis-aligned bounding box).

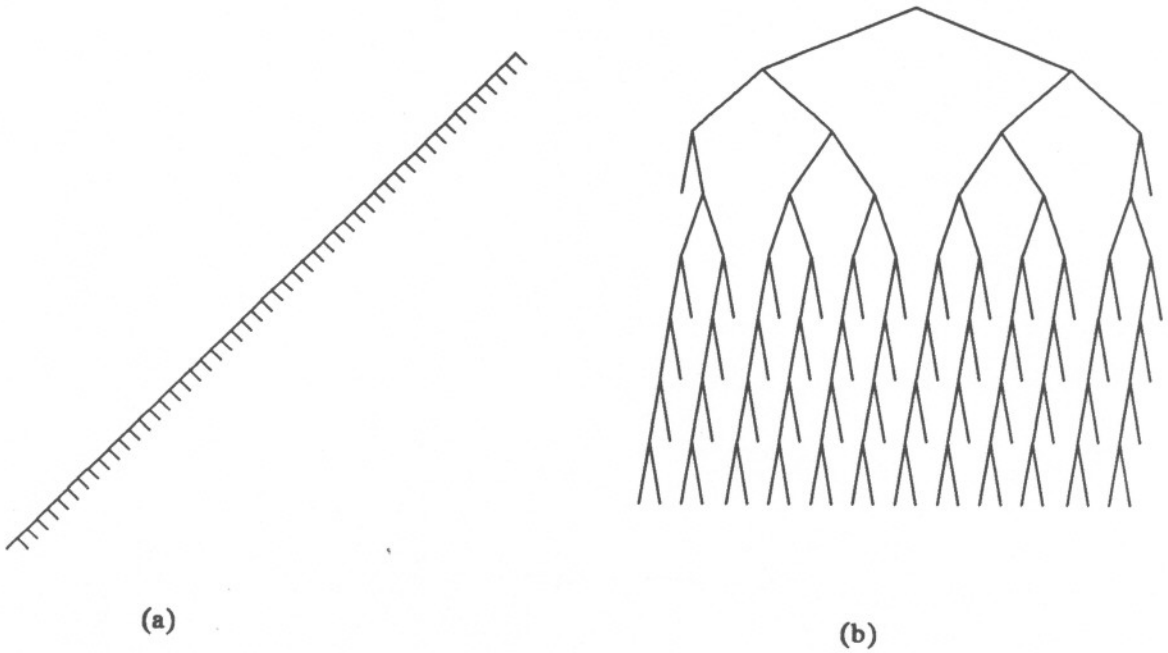


Figure 64. BSP Trees Built Using (a) Build_BSPT, and (b) the Median-Cut Algorithm

Information about the placement of polygons can be used to reduce the amount of splitting that occurs. If it is known that the polygons are arranged in "rings," such that no polygon crosses a given axis at certain values, a partitioning plane can be placed at that position along the given axis without splitting any polygon. This is often the case for polyhedra constructed from "slices," as is provided by most medical imaging systems such as CT scans, as well as for parametrically described surfaces, such as the sphere in Figure 65. For such objects, the first levels of the BSP tree can partition along the "ringed" axis without introducing any additional polygons resulting from splitting. In the Figure, splitting happens to only a few of the original retangular polygons, and a well-balanced tree is obtained. (The top of the tree constitutes a bounding box about the sphere, two planes of which embed the polar polygons (explaining why they are not split).)

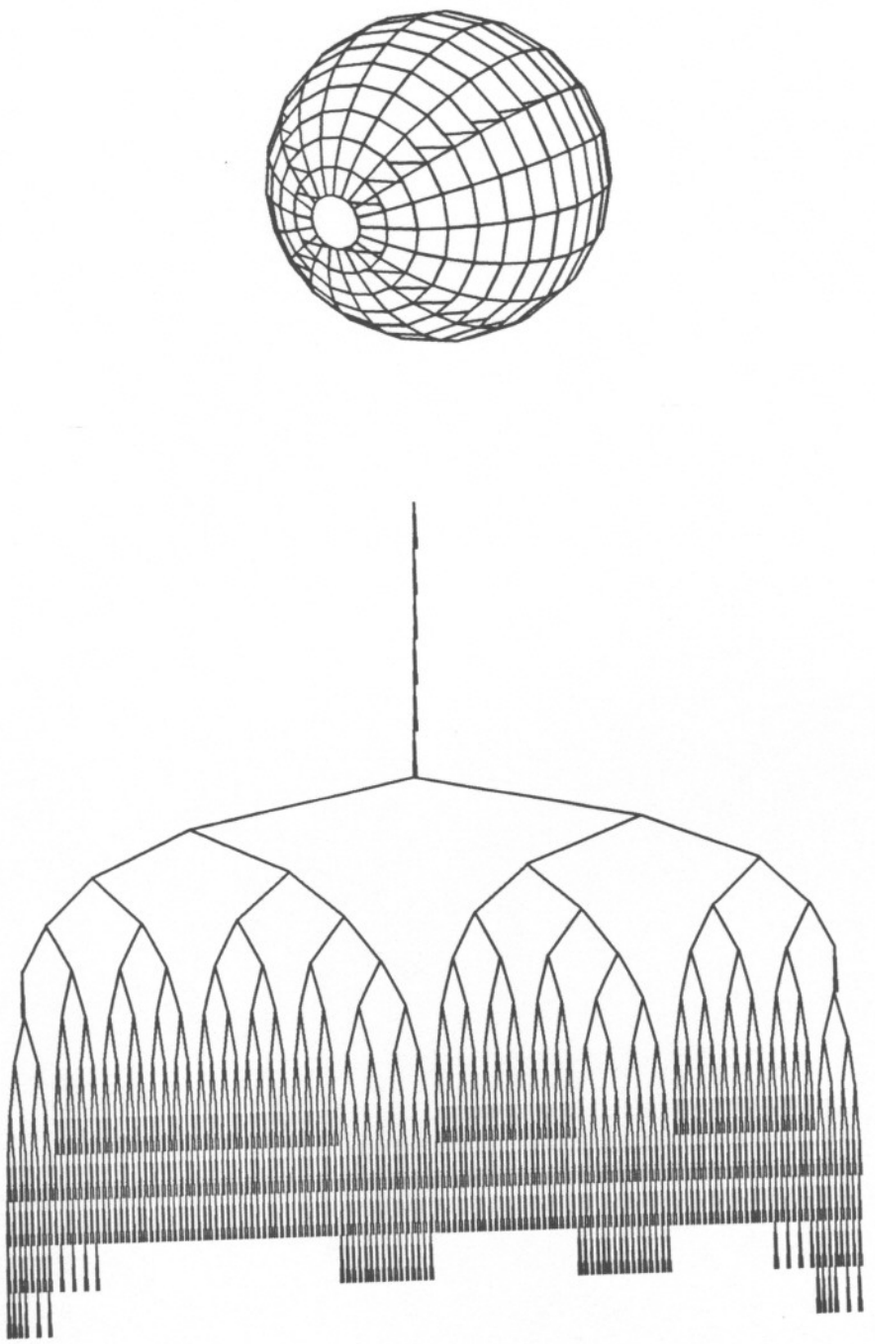


Figure 65. A Polyhedral Approximation to a Sphere and Its BSP Tree

CHAPTER VII

Conclusions

Overview

The Binary Space Partitioning tree has been presented as a novel representation for polyhedra, and has been shown to be effective for several useful operations. The representation allows for efficient point inclusion testing, volume and center-of-mass calculations. Algorithms that use the BSP tree to evaulate set operations on polyhedra were presented, and their efficacy shown in working implementations. Visible surface renderings are easily generated from BSP tree representations. The unification of three important aspects of any useful geometric modeling system in the BSP tree makes it an attractive representation for use as an auxiliary representation (for user interaction) in general-purpose solid modeling systems. These aspects are: modeling of and set operations on polyhedra, providing a structure for efficient geometric searching, and visible surface rendering.

Directions for Future Work

The BSP tree can be applied to other, more specific applications. One such is "beam-tracing," [Heck84] a technique similar to ray-tracing in which "bundles" of rays of polygonal cross-section are reflected and refracted through the scene. This has also been proposed [Dado82] as a technique for modeling propogation of "sound beams" in acousitical environments. By modeling the environment in question with a BSP tree, and the beam as a prism, a set intersection operation will determine the possible surfaces hit by the beam, where this can be resolved by performing a visible surface operation on the result.

Issues in the automatic generation of BSP trees to partition scenes for the above

technique as well as for classical ray-tracing deserve study.

The representation also promises to be useful in robotics, for problems of collision detection. The obstacles can be modeled as a BSP tree, and the moving parts of the robot as polyhedra. A non-null intersection result signals an intersection. For dynamic collision avoidance, where a proposed motion is to be tested for possible collision, a polyhedron modeling the volume swept out by the moving actuator is tested for intersection with the environment. The BSP tree provides the additional benefit of easily generating visible surface renderings of the simulated geometry.

Another straightforward application of the BSP tree technology is in medical applications. One specific problem where it could be useful is in radiation treatment. Directing a narrow beam of high intensity radiation at specific internal organs must be concerned with avoiding other healthy organs to prevent possibly damaging them. This would require a means of converting the cross-sections from CT-scans or other techniques into a BSP tree. The beam would then be modeled as a polyhedron, and intersections with organs to be avoided could be tested. This would allow the radiologist to plan the beam placement for the most effective treatment. One such system [Mosh86] requires visual identification of intersections with targeted tissues.

Work remains to be done on many interesting and important theoretical issues concerning BSP trees. Average-case analysis of tree size and running times for the various algorithms should provide more useful information than the worst-case results in[Nayl81]. Such an analysis would be based on the geometry of polyhedra, and should serve to better characterize the trees actually realized.

The BSP tree itself can be generalized to use partitioning sets other than hyperplanes: arbitrary functionals that define halfspaces can be used. For example, ellipsoids or other closed volumes could be used to organize space in a manner similar to that obtained in

bounding-volume schemes for ray-tracing. A drawback of this generalization would be the loss the convexity properties of cells and sub-hyperplanes maintained by use of hyperplanes. The use of more general partitioning sets should not invalidate the basic algorithms for classification and set operations, however, other than complications due to the loss of convexity.

APPENDIX

Finding the Intersection of a Line Segment and a Hyperplane

A point $\bar{x}$ is a dx1 column vector. The normal to a hyperplane H is a 1xd row vector $\bar{a}$. The hyperplane is

$$\{\bar{x} : \bar{a} \bullet \bar{x} = a_{d+1}\}$$

Using the parametric form, the line segment $\overline{pq}$ is

$$\{\bar{x} : \bar{x} = \bar{p} + t\bar{r},\ 0 \leq t \leq 1\},$$

where $\bar{r} = \bar{q} - \bar{p}$.

Substituting gives

$$\bar{a} \bullet (\bar{p} + t\bar{r}) = a_{d+1}$$

Solving for t yields

$$t = \frac{a_{d+1} - \bar{a} \bullet \bar{p}}{\bar{a} \bullet \bar{r}}$$

The coordinates of the intersection point are obtained by substituting this value for t into the parametric form for $\overline{pq}$. Note that if t is not in the interval (0,1), then the relative interior of the line segment does not intersect H.

BIBLIOGRAPHY

[Aho74] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Reading, Mass.: Addison-Wesley, 1974.

[Ambu86] Ambum, Phil, Eric Grant, and Turner Whitted. "Managing Geometric Complexity with Enhanced Procedural Models." *COMPUTER GRAPHICS* **20**, no. 4 (August 1986).

[Aono84] Aono, Masaki, and Tosiyasu L. Kunii. "Botanical Tree Image Generation." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (May 1984): 10-34.

[Athe83] Atherton, Peter R. "A Scan-line Hidden Surface Removal Procedure for Constructive Solid Geometry." *COMPUTER GRAPHICS* **17**, no. 3 (July 1983): 73-82.

[Ayal85] Ayala, D., P. Brunet, R. Juan, and I. Navazo. "Object Representation by Means of Nonminimal Division Quadtrees and Octrees." *ACM TRANSACTIONS ON GRAPHICS* **4**, no. 1 (January 1985): 41-59.

[Baum74] Baumgart, B. G. "Geometric Modeling for Computer Vision." REPORT NO. AIM-249, STAN-CS-74-463, Stanford Artificial Intelligence Laboratory, Stanford Univeristy, Stanford, California, October 1974.

[Bent79] Bentley, Jon Louis, and Jerome H. Friedman. "Data Structures for Range Searching." *COMPUTING SURVEYS* **11**, no. 4 (December 1979): 397-409.

[Brai80] Braid, I. C., R. C. Hillyard, and I. A. Stroud. "Stepwise Construction of Polyhedra in Geometric Modeling." In *Mathematical Methods in Computer Graphics and Design*, ed. K. W. Brodlie, 123-141. London: Academic Press, 1980.

[Brai75] Braid, I. C. "The Synthesis of Solids Bounded by Many Faces." *COMMUNICATIONS OF THE ACM* **18**, no. 4 (April 1975): 209-216.

[Carl85] Carlbom, Ingrid, Indranil Chakravarty, and David Vanderschel. "A Hierarchical Data Structure for Representing the Spatial Decomposition of 3-D Objects." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (April 1985): 24-31.

[Carl87] Carlbom, Ingrid. "An Algorithm for Geometric Set Operations Using Cellular Subdivision Techniques." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (May 1987): 44-55.

[Cohe79] Cohen, Jacques, and Timothy Hickey. "Two Algorithms for Determining Volumes of Convex Polyhedra." *JOURNAL OF THE ACM* **26**, no. 3 (July 1979): 401-414.

[Coxe63] Coxeter, H. S. M. *Convex Polytopes*. New York: Macmillan, 1963.

[Dado82] Dadoun, Norm, David G. Kirkpatrick, and John P. Walsh. "Hierarchical Approaches to Hidden Surface Intersection Testing." *GRAPHICS INTERFACE '82* (1982): 49-56.

[Dipp84] Dippe, Mark, and John Swensen. "An Adaptive Subdivision Algorithm and Parallel Architecture for Realistic Image Synthesis." *COMPUTER GRAPHICS* **18**,

no. 3 (July 1984): 149-158.

[Doct81] Doctor, Louis J., and John G. Torborg. "Display Techniques for Octree-Encoded Objects." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (July 1981): 29-38.

[East79] Eastman, Charles, and Kevin Weiler. "Geometric Modeling Using the Euler Operators." *PROCEEDINGS 1ST ANNUAL CONFERENCE ON COMPUTER GRAPHICS IN CAD/CAM SYSTEMS, MIT* (April 1979).

[Edah84] Edahiro, Masato, Iwao Kokubo, and Takao Asano. "A New Point-Location Algorithm and Its Practical Efficiency -- Comparison with Existing Algorithms." *ACM TRANSACTIONS ON GRAPHICS* **3**, no. 2 (April 1984): 86-109.

[Fole83] Foley, James D., and Andries Van Dam. *Fundamentals of Interactive Computer Graphics*. Reading, Massachusetts: Addison-Wesley, 1983.

[Four82] Fournier, Alain, Don Fussell, and Loren Carpenter. "Computer Rendering of Stochastic Models." *COMMUNICATIONS OF THE ACM* **25**, no. 6 (June 1982): 371-384.

[Fuch83] Fuchs, Henry, Gregory D. Abram, and Eric D. Grant. "Near Real-Time Shaded Display of Rigid Objects." *COMPUTER GRAPHICS* **17**, no. 3 (July 1983): 65-72.

[Fuch80] Fuchs, H., Z. Kedem, and B. Naylor. "On Visible Surface Generation by a Priori Tree Structures." *COMPUTER GRAPHICS* **14**, no. 3 (June 1980).

[Full75] Fuller, Buckminster R. *Synergetics: Explorations in the Geometry of Thinking*. New York: Macmillan, 1975.

[Gigu85] Gigus, Ziv. "Binary Space Partitioning for Previewing UNIGRAFIX Scenes." REPORT NO. UCB/CSD 86/280, Computer Science Division (EECS), University of California, Berkeley, California 94720, December 1985.

[Glas84] Glassner, Andrew S. "Space Subdivision for Fast Ray Tracing." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (October 1984): 15-22.

[Heck84] Heckbert, Paul S., and Pat Hanrahan. "Beam Tracing Polygonal Objects." *COMPUTER GRAPHICS* **18**, no. 3 (July 1984): 119-127.

[Hunt79] Hunter, Gregory M., and Kenneth Steiglitz. "Operations onImages Using Quad Trees." *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE* **PAMI-1**, no. 2 (April 1979): 145-153.

[Jack80] Jackins, Chris L., and Steven L. Tanimoto. "Oct-Trees ans Their Use in Representing Three-Dimensional Objects." *COMPUTER GRAPHICS AND IMAGE PROCESSING* **14** (1980): 249-270.

[John84] Johnson, Robert H. *Solid Modeling: A State-of-the-Art Report*. Chestnut Hill, Massachusetts: CAD/CAM ALERT, Management Roundtable, Inc., 1984.

[Kirk83] Kirkpatrick, David. "Optimal Search in Planar Subdivisions." *SIAM JOURNAL OF COMPUTING* **12**, no. 1 (February 1983): 28-35.

[Laid86] Laidlaw, David H., W. Benjamin Trumbore, and John F. Hughes. "Constructive Solid Geometry for Polyhedral Objects." *COMPUTER GRAPHICS* **20**, no. 4 (August 1986): 161-170.

[Lass83] Lasserre, J. B. "Volume of a Convex Polyhedron in R^n." *JOURNAL OF*

OPTIMIZATION THEORY AND APPLICATIONS **39**, no. 3 (March 1983): 363-377.

[Lay82] Lay, Steven R. *Convex Sets and Their Applications*. New York: John Wiley and Sons, 1982.

[Lee82] Lee, Yong Tsui, and Aristides A. G. Requicha. "Algorithms for Computing the Volume and Other Integral Properties of Solids. I. Known Methods and Open Issues." *COMMUNICATIONS OF THE ACM* **25**, no. 9 (September 1982): 635-641.

[Levi76] Levin, Joshua. "A Parametric Algorithm for Drawing Pictures of Solid Objects Composed of Quadric Primitives." *COMMUNICATIONS OF THE ACM* **19**, no. 10 (October 1976): 555-563.

[Levi79] Levin, Joshua Zev. "Mathematical Models for Determining the Intersections of Quadric Surfaces." *COMPUTER GRAPHICS AND IMAGE PROCESSING* **11** (1979): 73-87.

[Mant84] Mantyla, Martti. "A Note on the Modeling Space of Euler Operators." *COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING* **26** (1984): 45-60.

[Mant82] Mantyla, Martti, and Reijo Sulonen. "GWB: A Solid Modeler with Euler Operators." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (September 1982): 17-31.

[Mant83] Mantyla, Martii, and Markku Tamminen. "Localized Set Operations for Solid Modeling." *COMPUTER GRAPHICS* **17**, no. 3 (July 1983): 279-288.

[Meag82] Meagher, D. "Geometric Modeling using Octree Encoding." *COMPUTER GRAPHICS AND IMAGE PROCESSING* **19** (June 1982).

[Mitc87] Mitchell, Don P. "Generating Antialiased Images at Low Sampling Densities." *COMPUTER GRAPHICS* **21**, no. 4 (July 1987): 65-72.

[Mort85] Mortenson, Michael E. *Geometric Modeling*. New York: Wiley, 1985.

[Mosh86] Mosher, Charles E. Jr., George W. Sherouse, Peter H. Mills, Kevin L. Novins, Stepher M. Pizer, Julian G. Rosenman, and Edward L. Chaney. "THe Virtual Simulator." *1986 WORKSHOP ON INTERACTIVE 3D GRAPHICS* (Oct 23-24 1986).

[Nava86] Navazo, I., D. Ayala, and P. Brunet. "A Geometric Modeller Based on the Exact Octtree Representation of Polyhedra." *COMPUTER GRAPHICS FORUM* **5** (1986): 91-104.

[Nayl81] Naylor, Bruce F. "A Priori Based Techniques for Determining Visibility Priority for 3-D Scenes." PH.D. THESIS, University of Texas at Dallas, May 1981.

[Nayl86] Naylor, Bruce F., and William C. Thibault. "Application of BSP Trees to Ray-Tracing and CSG Evaluation." TECHNICAL REPORT GIT-ICS 86/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332, February 1986.

[Prep85] Preparata, Franco P., and Michael Ian Shamos. *Computational Geometry: An Introduction*. New York: Springer-Verlag, 1985.

[Putn86] Putnam, L. K., and P. A. Subrahmanyam. "Boolean Operations on n-Dimensional Objects." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (June 1986): 43-51.

[Rein81] Reingold, Edward M., and John S. Tilford. "Tidier Drawings of Trees." *IEEE TRANSATIONS ON SOFTWARE ENGINEERING* **SE-7**, no. 2 (March 1981): 223-228.

[Requ85] Requicha, Aristides A. G., and Herbert B. Voelcker. "Boolean Operations in Solid Modeling: Boundary Evaluation and Merging Algorithms." *PROCEEDINGS OF THE IEEE* **73**, no. 1 (January 1985): 30-44.

[Requ80] Requicha, Aristides A. G. "Representations for Rigid Solids: Theory, Methods, and Systems." *COMPUTING SURVEYS* **12**, no. 4 (December 1980): 437-464.

[Requ83] Requicha, A. A. G., and H. B. Voelcker. "Solid Modeling: Current Status and Research Directions." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (October 1983): 25-37.

[Requ78] Requicha, Aristides A. G., and Robert B. Tilove. "Mathematical Foundations of Constructive Solid Geometry: General Topology of Closed Regular Sets." TM-27A, Production Automation Project, University of Rochester, Rochester, New York 14627, June 1978.

[Roge85] Rogers, D. F. *Procedural Elements for Computer Graphics.* New York: McGraw-Hill, 1985.

[Roth82] Roth, Scott D. "Ray Casting for Modeling Solids." *COMPUTER GRAPHICS AND IMAGE PROCESSING* **18** (1982): 109-144.

[Rubi80] Rubin, Steven M., and Turner Whitted. "A 3-Dimensional Representation for Fast Rendering of Complex Scenes." *COMPUTER GRAPHICS* **14**, no. 3 (July 1980): 110-116.

[Same84] Samet, Hanan. "The Quadtree and Related Data Structures." *ACM COMPUTING SURVEYS* **16**, no. 2 (June 1984): 187-260.

[Sarr83] Sarraga, Ramon F. "Algebraic Methods for Intersections of Quadric Surfaces in GMSOLID." *COMPUTER GRAPHICS AND IMAGE PROCESSING* **22** (1983): 222-238.

[Schu69] Schumacker, R. A., R. Brand, M. Gilliland, and W. Sharp. "Study for Applying Computer-Generated Images to Visual Simulation." AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, 1969.

[Thib87] Thibault, William C., and Bruce F. Naylor. "Set Operations on Polyhedra Using Binary Space Partitioning Trees." *COMPUTER GRAPHICS* **21**, no. 4 (July 1987).

[Thib87a] Thibault, William C., and Bruce F. Naylor. *Set Operations on Polyhedra Using Binary Space Partitioning Trees: The Video.*(February 1987) .

[Thom87] Thomas, D., D. S. Fox, and A. N. Netravali. *Efficient Octree Building and Tracing Techniques for High Speed Ray Tracing.* AT&T Bell Laboritories, 1987.

[Tilo84] Tilove, Robert. "A Null-Object Algorithm for Constructive Solid Geometry." *COMMUNICATIONS OF THE ACM* **27**, no. 7 (July 1984).

[Wegh84] Weghorst, Hank, Gary Hooper, and Donald P. Greenberg. "Improved Computational Methods for Ray Tracing." *ACM TRANSACTIONS ON GRAPHICS* 3, no. 1 (January 1984): 52-69.

[Weil85] Weiler, Kevin. "Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments." *IEEE COMPUTER GRAPHICS AND APPLICATIONS* (January 1985): 21-40.

[Whit80] Whitted, Turner. "An Improved Illumination Model for Shaded Display." *COMMUNICATIONS OF THE ACM* **23**, no. 6 (June 1980).

[Yann79] Yannakakis, M. Z., C. H. Papadimitriou, and H. T. Kung. "Locking policies: safety and freedom for deadlock." *PROCEEDINGS 20TH ANNUAL SYMPOSIUM ON FOUNDATIONS OF COMPUTER SCIENCE* (1979): 286-297.

[Yau83] Yau, Mann-May, and Sargar N. Srihari. "A Hierarchical Data Structure for Multidimensional Digital Images." *COMMUNICATIONS OF THE ACM* **26**, no. 7 (July 1983): 504-515.

VITA

William Thibault was born on December 18, 1957, in New Orleans, Louisiana. He received the B.S. in Computer Science at the University of New Orleans in 1981. Upon entering Georgia Tech, he was awarded a President's Fellowship, and received the M.S. in Information and Computer Science in 1985. He was a Consultant at AT&T Bell Laboratories in Murray Hill, NJ for nine months during 1986 and 1987. He is a member of the Association for Computing Machinery and SIGGRAPH.