Hierarchical Solid Boolean Modeling

Bernie Freidin, 1999-2000

Division III • Final Project
Chair: Lee Specter
Member: Ken Hoffman

Abstract

Here I present a data structure suitable for representing and constructing solid geometric models through Boolean operations. I call it Hierarchical Solid Boolean Modeling, or HSBM. An HSBM model is stored as a tree structure, where the nodes of the tree define convex polyhedra in a spatial hierarchy. The spatial hierarchy can be used to accelerate both rendering and construction. In addition, we present efficient algorithms for updating the boundary of the model during construction, allowing for interactive editing and viewing in real-time. I expect HSBM to be a useful representation for CAD/CAM applications, 3D architectural modeling, and game engine design.

Table of Contents

Abstract 1

Introduction 4

Chapter 1 – Solid Modeling 5

1.1 Polygons and Polyhedra 5
Figure 1-1: Simple and non-simple polygons 6
Figure 1-2: Simple polyhedra 6
Figure 1-3: Convex and non-convex polygons and polyhedra 7
1.2 Multishelled Objects 7
Figure 1-4: A multishelled polygon 7
Figure 1-5: Some allowable multishelled polygons 8
Figure 1-6: An invalid multishelled polygon 8
1.3 Solid Primitives 9
Figure 1-7: Typical primitive shapes 9
Figure 1-8: Some tessellated primitives 10
1.4 Boolean Modeling 10
Figure 1-10: Object built from solid primitives 10
Figure 1-11: Boolean operations between a cube (A) and a cylinder (B) 11
Figure 1-12: Degenerate region formed by intersection 11
Figure 1-13: Complex intersection forming degenerate “dangling” structures 12
1.5 Boundary Representation (b-rep) 12
1.6 Manifolds 13
Figure 1-14: Three polygons joined together, forming two T-vertices 13
Figure 1-15: A multishelled polygon with non-manifold vertices 14
1.7 Geometric Queries 14
Figure 1-16a: Point classification with respect to convex polygon –
Point lies inside all half-spaces, therefore point lies in inside polygon 15
Figure 1-16b: Point classification with respect to convex polygon –
Point lies outside last half-space, therefore point lies outside polygon 15
1.8 Spatial Hierarchies 16
1.8.1 Quadtrees and Octrees 16
Figure 1-17: Quadtree hierarchy (3 levels) 16
Figure 1-18: Regular grid vs quadtree partitioning 17
1.8.2 BSP Trees 18
Figure 1-19: A BSP tree 18
1.9 Constructive Solid Geometry (CSG) 19

Chapter 2 – HSBM 20

2.1 Definition of HSBM Object Representation 20
Figures 2-1: An HSBM object (a) and its underlying tree (b) 20
2.2 Complex HSBM Objects 21
Figure 2-2: A simple HSBM object with node outlines (a), and without node outlines (b) 21
Figure 2-3: A complex HSBM object (a) and its underlying tree (b) 21
Figure 2-4: A complex HSBM object without node outlines (a), and with circles (b) 22
2.3 Boundary and Volume Classification 22
Figure 2-5: Examples of b-, p- and n-points 22
Figure 2-6: Boundary of an HSBM object 23
2.4 Polygonalization of Boundary 23
Figure 2-7: An HSBM object with b-rep polygons 24
Figure 2-8: Effect of offsetting nodes on the boundary of an HSBM object 24
2.5 Incremental Construction of an HSBM Object 25
Figure 2-9: An HSBM object being built by incremental construction 25
2.6 Boundary Updates During Incremental Construction 26
Figure 2-10: Face groups 26
Figure 2-11: Face and b-rep groups 27
Figure 2-12: A very complex HSBM object with face and b-rep groups 28

Conclusion and Retrospective 29

Bibliography and References 37

Appendix A – HSBM Node Insertion/Removal Algorithm 38

GLOBALS: 39
INSERT_NODE[S, Snew, flag = {0: insert, 1: remove}] 39
ASCEND_TREE[S] 39
COPLANAR_FACE[S] 39
COPLANAR_OPPOSITE_FACE[S] 39
UPDATE_BREP[S, Fb, d] 39
CUT_BREP[F, X] 40

Appendix B – Image Scrapbook 41

Introduction

Creating 3D geometric models via computer is an important process in many applications, such as Computer-Aided Drafting (CAD), Computer-Aided Modeling (CAM), construction, architecture, mechanical engineering, and virtual reality. However, the process is not straightforward. One of the problems in modeling is that while the user (the one using the modeling program) may wish to to create a model by transforming and positioning simple primitive shapes (such as cubes, spheres, etc.) and placing them in 3D space, the resulting topology and geometric intersections between the placed primitives often becomes exceedingly complex. Usually, such resulting topology is a necessary output of the modeling program, but from a human point of view, it is not convenient to manipulate topology directly.

To address these concerns, the field of *solid modeling* introduces ways of "automating" the process to some degree. Specifically, a solid modeling program is aware of intersections between shapes, and has the ability to update the geometry and topology accordingly. There are many existing data structures which are used in solid modeling, such as BSP trees, b-reps, polyhedral representations, etc. My contribution is a structure which I call *Hierarchical Solid Boolean Modeling*, or HSBM – which is actually very similar to commonly-used multishelled polyhedral representations. The difference is that in my data structure, I restrict shells to being convex polyhedra, but allow for shells to be adjacent. Additionally, I define algorithms for updating the boundary of the model as the model is constructed. This is an important capability for real-time or near real-time visualization.

I will begin with a short introduction to the subject of solid modeling along with explanations of the various terminology and concepts used in the field of solid modeling, provided for the benefit of the inexperienced reader. I will follow with a thorough examination of the HSBM data structure and its associated algorithms in the following chapters. To develop an appreciation for the complexity and subtlety of the field of solid modeling, I will present some concepts in greater depth or detail than is immediately necessary for the understanding of HSBM.

Chapter 1 – Solid Modeling

In solid modeling, objects are treated as having three-dimensional volume<FOOTNOTE>. This is in contrast to surface modeling, where only the surfaces of objects are considered, and wireframe modeling, where only the edges of objects are considered. While surface modeling may suffice for topologically simple cases such as the representation of an airplane wing or the body of a car, it is severely limited in topologically complex cases such as mechanical construction and architecture. This limitation will be made clear after introducing the concept of Boolean modeling (Section 1.4), which is the primary means by which solid models are constructed and has no real counterpart in surface or wireframe modeling.

While the focus of this paper is on three-dimensional solid modeling, in many cases it is difficult to draw clear diagrams of three-dimensional objects that show all of the object's internal structure. For this reason, many of the figures will be drawn in two dimensions. Sometimes, two-dimensional figures may represent "slices" of three-dimensional objects. In such figures, vertices represent edges, edges represent faces, and faces represent volumes.

1.1 Polygons and Polyhedra

Polygons form the basic building block for most modeling programs. While the precise definition of a polygon may vary depending on whom one asks, we will define a polygon as a finite set of points (*vertices*) which lie on a plane, connected by a single closed loop of line segments (*edges*) which form the boundary, enclosing a well-defined finite area (called the *interior* of the polygon). Further, a *simple polygon* will have a boundary with no self-intersection. Non-simple polygons will be considered invalid for the purposes of our discussion. Figure 1-1 shows some examples of simple and non-simple/invalid polygons. For the remainder of this paper, all references to polygons will imply simple polygons.

[1] Here we assume the model is three-dimensional. However, the concept of solid modeling and HSBM can be extended to any higher dimension. Solid modeling could be restated as the treatment of objects as having the same dimensionality as that of the space they are embedded in.

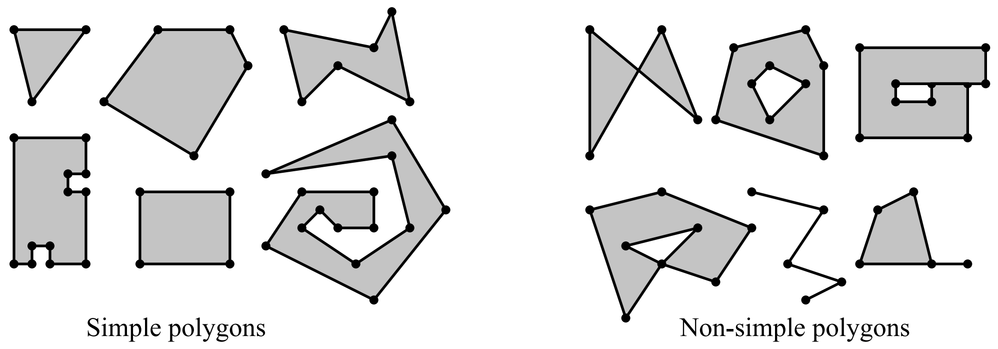


Figure 1-1: Simple and non-simple polygons

Similarly, we can define a polyhedron as a direct extension of the polygon concept into three dimensions. A polyhedron is a closed, connected, finite set of polygons (*faces*) in three-space, with no self-intersection, enclosing a well-defined finite volume. Some examples of valid polyhedra are shown in Figure 1-2. Notice that even though disjoint sets of polygons are disallowed, it is possible to create valid polyhedra with holes (a one-holed polyhedron is topologically equivalent to a donut, or *torus*).

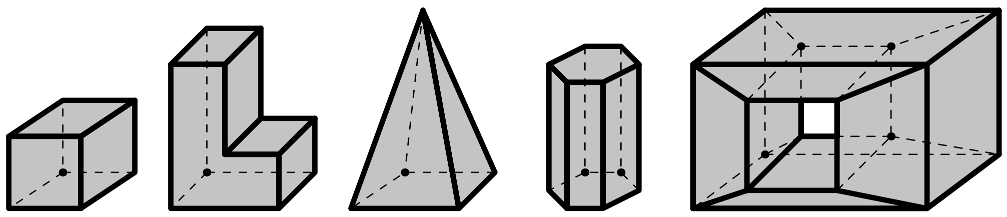


Figure 1-2: Simple polyhedra

In some situations, polygons and polyhedra may be further restricted to being convex, meaning that the interior angles between adjacent faces/edges are less than or equal to 180°. I will use the term *strictly convex* to mean the interior angles are strictly less than 180°. It is easy to see that any convex polygon may be trivially made into a strictly convex polygon by edge-merging, and similarly for polyhedra. Many geometric algorithms which operate on polygons and polyhedra are easier to implement for convex cases, so it is often desirable to decompose a non-convex object into a set of convex objects. Such decomposition is studied intensively in the field of computational geometry. One of the nice properties of convex objects is that their volume is the intersection of the half-spaces defined by their face planes. Some examples of convex and non-convex polygons are shown in Figure 1-3.

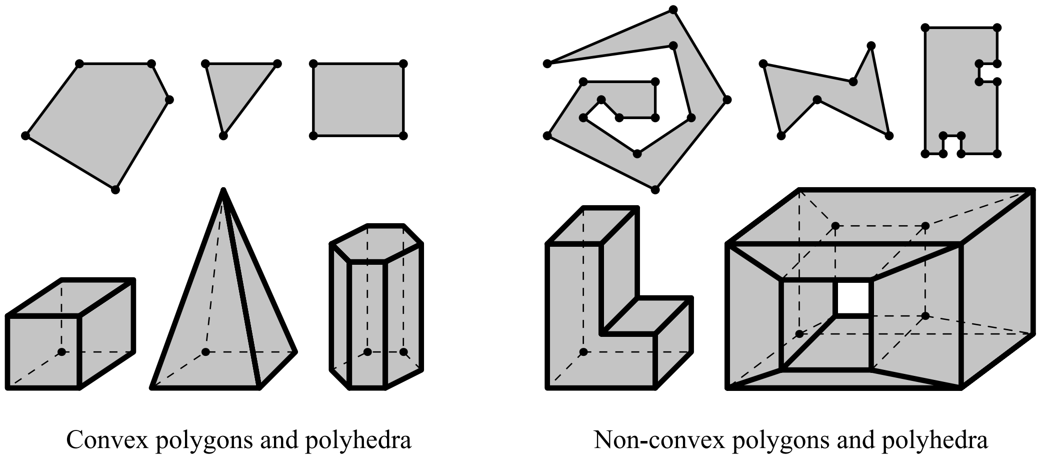

Convex polygons and polyhedra
Non-convex polygons and polyhedra

Figure 1-3: Convex and non-convex polygons and polyhedra

1.2 Multishelled Objects

A useful extension of the polygon/polyhedron concept is that of *multishelled* objects [CHRI89]. We'll use the term *object* to refer to either a polygon or a polyhedron, so that we don't have to repeat the same statements twice. Up until now we have been discussing single-shelled objects, i.e., objects without holes. A multishelled object can be thought of as an object with one or more holes, these holes possibly containing further multishelled objects, and so on. We can specify the *polarity* (to use my own terminology) of a shell in a multishelled object as being either *positive* or *negative,* indicating whether the shell contributes additional area/volume or removes existing area/volume. Some implementations require that the vertices of positive shells be listed in clockwise order while the vertices of negative shells (holes) be listed in counter-clockwise order, or visa-versa.

Figure 1-4 shows a multishelled polygon in which all of the shells are simple convex polygons and no shells are adjacent. However, allowing shells to be adjacent leads to a very useful way of representing complex geometric objects (including non-convex and non-manifold objects – see Section 1.6) as decompositions of simple convex shapes. The HSBM structures in Chapter 2 are based on this idea.

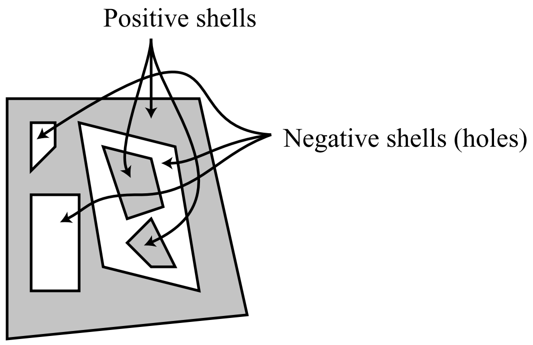


Figure 1-4: A multishelled polygon

Although a multishelled object may have adjacent shells (at least in this discussion), a multishelled object is valid only if none if its shells' interiors intersect partially. Another way of expressing this is that holes must be contained entirely within their parent shells, and likewise, the objects within the holes must be entirely contained within their respective holes. Figure 1-5 shows just a few examples of allowable multishelled polygons and their boundaries (we will discuss boundaries in more depth in Section 1.5).

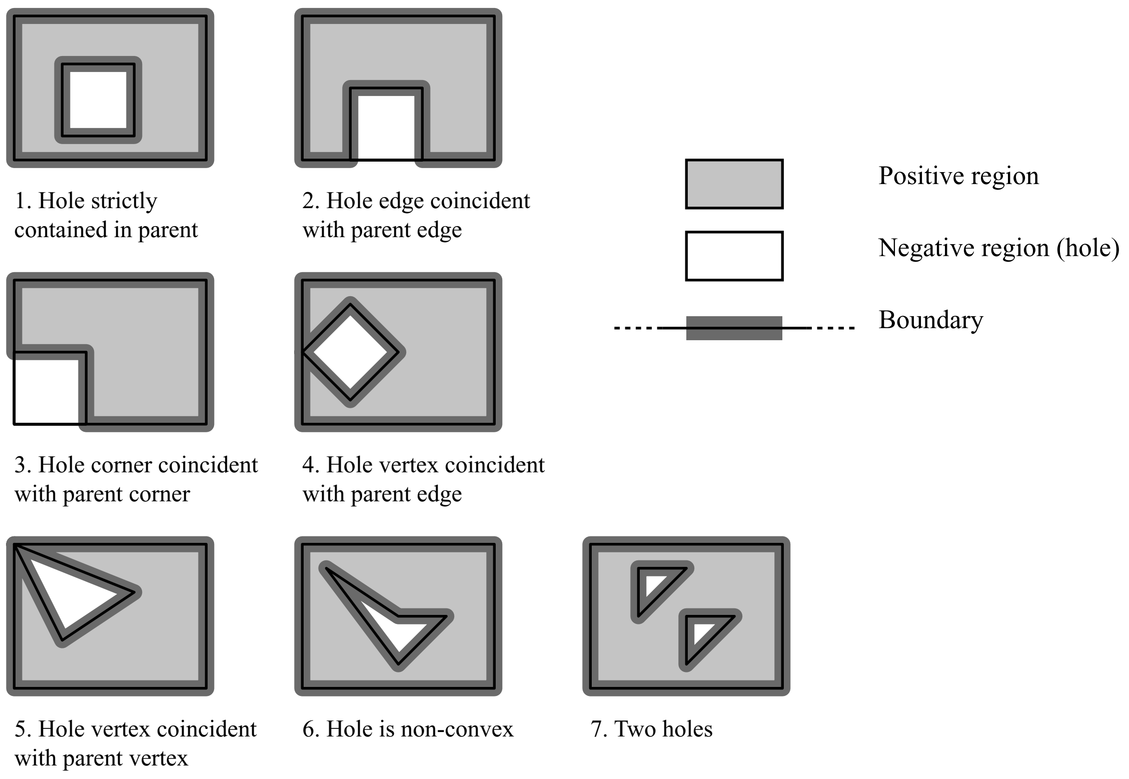

1. Hole strictly contained in parent
2. Hole edge coincident with parent edge
Positive region
Negative region (hole)
Boundary
3. Hole corner coincident with parent corner
4. Hole vertex coincident with parent edge
5. Hole vertex coincident with parent vertex
6. Hole is non-convex
7. Two holes

Figure 1-5: Some allowable multishelled polygons

Figure 1-6 shows an invalid multishelled polygon, where the hole pokes out of the parent shell's left side.

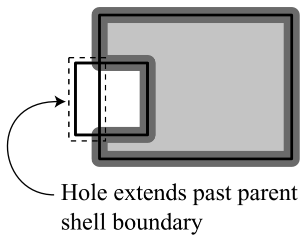


Figure 1-6: An invalid multishelled polygon

In situations where the boundaries of the holes are adjacent to the boundary of the containing object, the multishelled object could be represented as an equivalent non-convex single-shelled object (see Figure 1-5, examples 2 and 4). However, since many geometric algorithms are easier to implement on convex objects, multishelled objects with adjacent convex shells are still a useful representation.

1.3 Solid Primitives

A *primitive* is an object which cannot be better represented as a composition of simpler objects. Of course there is no precise mathematical definition for “better represented”, we use the term strictly in an informal/contextual sense. Solid modeling programs typically provide an assortment of solid primitives such as blocks, wedges, pyramids, cylinders, cones, toruses, and spheres, as well as various geometric transformations that can be applied to the primitives. Figure 1-7 shows some typical primitive shapes.

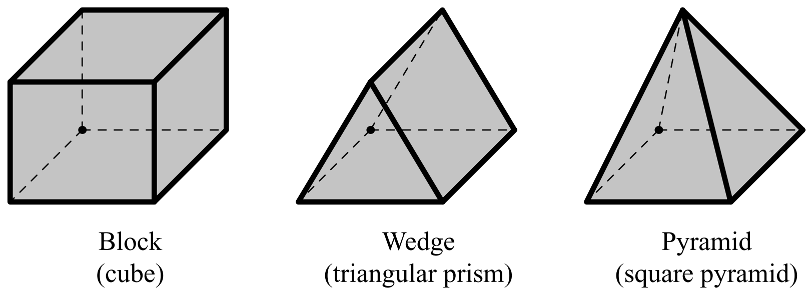


Figure 1-7: Typical primitive shapes

Note that shapes such as cylinders, cones, toruses, and spheres contain curved surfaces, which makes many solid modeling operations extremely difficult (but not impossible) to implement [CHRI89]. To simplify the implementation of Boolean modeling with curved primitives, some programs choose to approximate the curved surfaces with many small flat polygons as shown in Figure 1-8, a process known as *faceting* or *tessellating*. Even programs which operate directly on curved primitives usually tessellate the final output for rendering. Tessellated primitives and primitives without curved surfaces may be conveniently represented as polyhedra - non-tessellated curved shapes are usually represented by a combination of topological information and parametric or implicit curve equations.

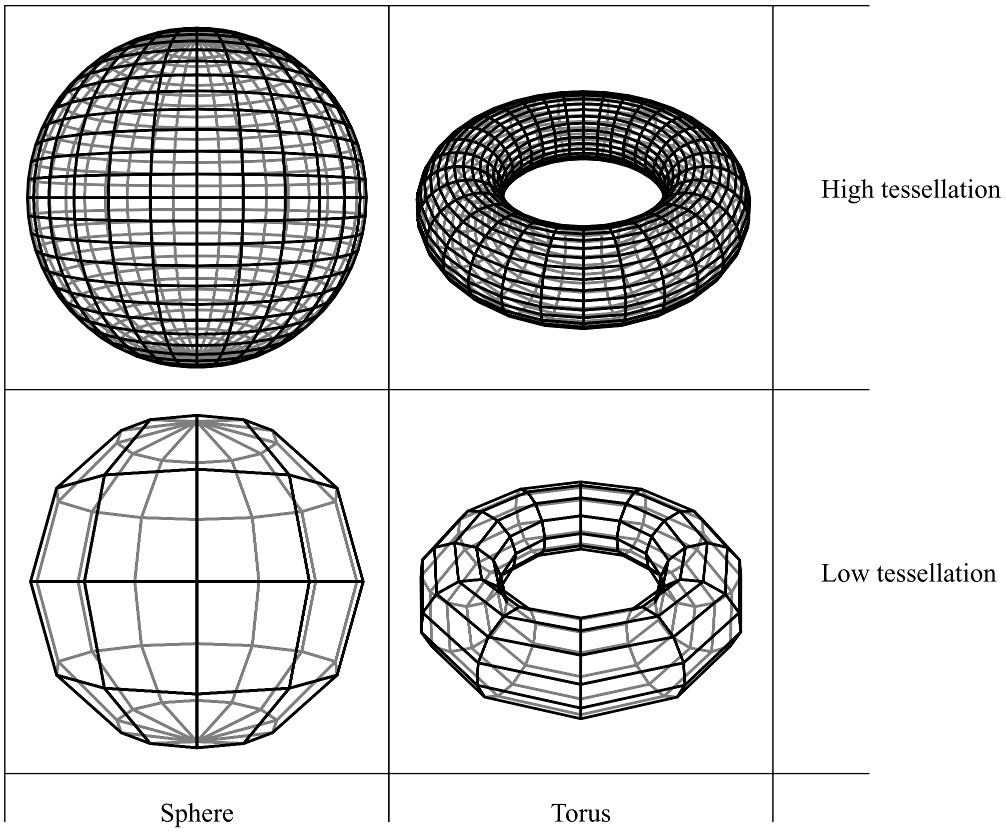


Figure 1-8: Some tessellated primitives

Custom primitives can also be constructed by means of *extrusion*, or *sweeping*, defined as the total volume covered by a polygon or solid primitive as it is moved continuously along a finite path.

1.4 Boolean Modeling

Given two solid objects, we can perform a Boolean operation on their volumes to produce a new solid object. This is a very useful tool, because it allows complex objects to be constructed out of simpler primitives, as shown in Figure 1-10. The Boolean operations most commonly used in solid modeling are *union*, *intersection*, and *difference* (denoted "$\cup$", "$\cap$", and "–" respectively). See Figure 1-11 for examples of each.

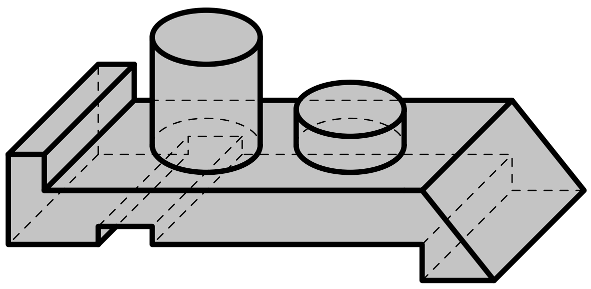


Figure 1-10: Object built from solid primitives

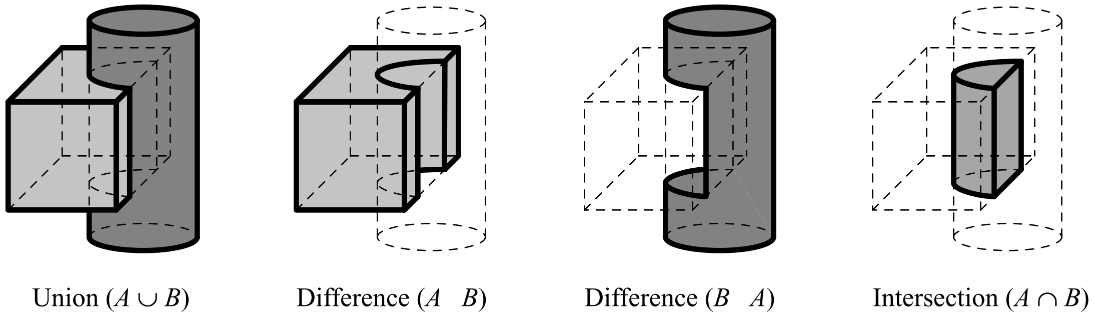


Figure 1-11: Boolean operations between a cube (A) and a cylinder (B)

Although the concept of Boolean modeling seems intuitive and easy, there are many cases where a naïve implementation may fail (or produce unintended results) due to oversimplification. Figures 1-12 and 1-13 show how degenerate structures may arise from the intersection of non-degenerate objects. With this in mind, it is helpful to define explicitly how Boolean operations behave: Given a solid object, we take the infinite set of all points which lie in the interior of the object or on its boundary. We may call this set the *point set* for the object. Now, given two objects and their associated point sets, we perform the set-theoretic Boolean operation on the two point sets, resulting in a new point set. The closure[2] of the resulting point set defines the new object.

Given the above definition, the intersection of two cubes which lie next to each other so that their faces are coincident will produce a flat two-dimensional square, not a solid three-dimensional object (Figure 1-12). This happens because we have included the points on the boundary of the objects in their point sets. If we change our definition of the point sets to exclude these boundary points, then we have an analogous problem when we take the union of two adjacent cubes; namely, the result will be an elongated cube with a square missing in the center. Sometimes, the result of a Boolean operation on two solid objects will be a solid object with "dangling" lower-dimensional structures (Figure 1-13). Clearly, we need to define a more robust set of Boolean operations on solids, one that will eliminate the lower-dimensional structures but retain the higher-dimensional result.

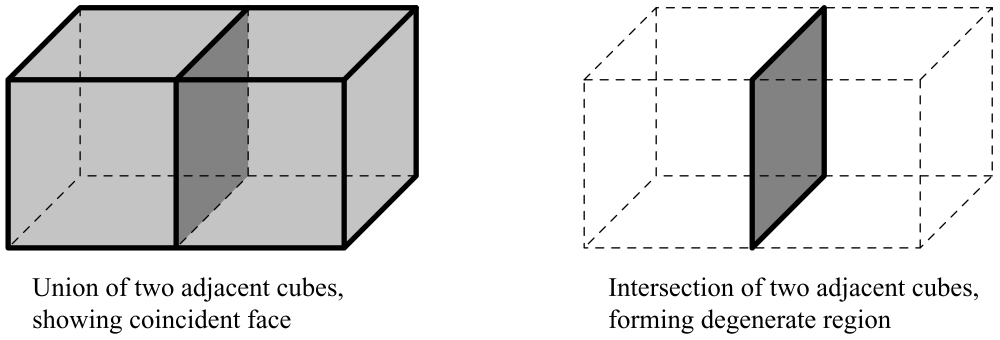

Union of two adjacent cubes,
showing coincident face
Intersection of two adjacent cubes,
forming degenerate region

Figure 1-12: Degenerate region formed by intersection

[2] We need to specify closure for the case of Boolean subtraction.

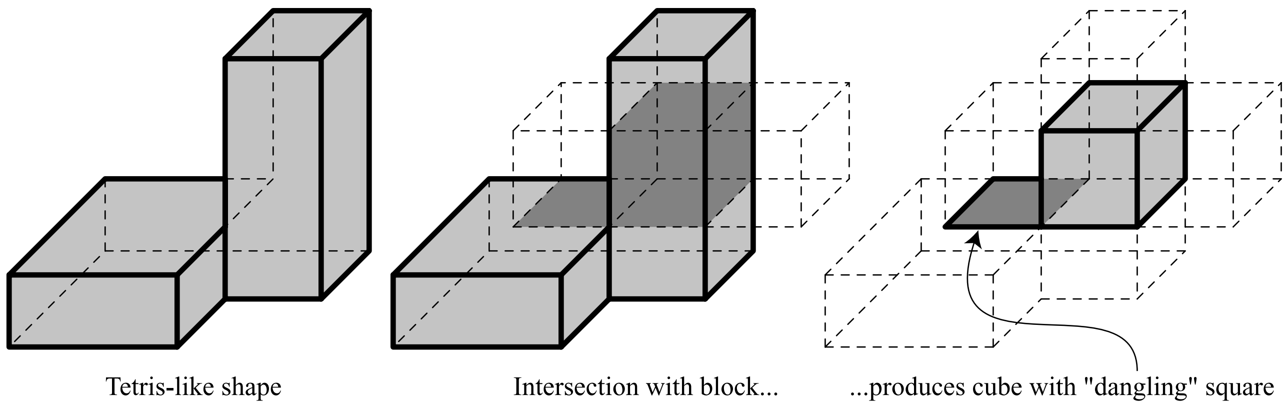

Tetris-like shape
Intersection with block...
...produces cube with "dangling" square

Figure 1-13: Complex intersection forming degenerate “dangling” structures

We define a new class of Boolean operations, called *regularized Boolean operations*, defined specifically with solid modeling in mind [CHRI89]. Regularized Boolean operations are denoted with an asterisk (*) next to the usual symbol. For example, regularized intersection is denoted as $\cap^*$. A regularized Boolean operation is defined informally as an ordinary Boolean operation followed by the removal of all dangling, lower-dimensional structures. The precise mathematical definition is more subtle.

Referring back to the representation of multishelled objects, the region enclosed by a multishelled object can be defined as the regularized difference of the outer shell and its holes, together with the region(s) enclosed by the multishelled objects within those holes, and so on. Alternatively, we might define the region as the regularized exclusive-or[3] of all of the shells taken individually.

1.5 Boundary Representation (b-rep)

The *boundary representation,* or *b-rep,* of a three-dimensional object is a set of polygons (or curved surfaces) which constitute the boundary of the object [CHRI89, FOLE96]. Similarly, the b-rep of a two-dimensional object is the set of line segments or curves which constitute the boundary. For example, the b-rep of a single-shelled polygon or polyhedron is simply the collection of its edges or face polygons. However, more complex solid representations such as multishelled polyhedra with adjacent shells have non-trivial b-reps. It is often necessary to maintain or compute the b-rep of an object for the purpose of rendering, especially interactive rendering, because most rendering systems take single-shelled convex polygons as their input. The exception to this is in ray-tracing, where objects can be often be rendered without explicit knowledge of the object’s b-rep. Figure 1-5 in Section 1.2 shows some examples of multishelled polygons and their associated b-reps. Note that the non-trivial cases involve situations where the shells are adjacent (i.e., the boundaries of the shells overlap).

Although b-reps are often used as secondary data structures to facilitate rendering or other functionality, it is possible to completely define a solid object by its b-rep alone. It is even possible to do Boolean modeling on b-reps, but not as easily as with convex polyhedra.

[3] Here we borrow a term more often used in Boolean logic and give it set-theoretic meaning.

1.6 Manifolds

A *manifold* is a surface for which the local neighborhood around every point on the surface is topologically equivalent to a plane (or, in the two-dimensional case, a line) [CHRI89]. All simple polygons and polyhedra are manifolds. If we have a b-rep made out of polygons, the boundary is a manifold only if every edge is adjacent to precisely two polygons. Because of this property, it is easier to represent the connectivity between adjacent polygons in a manifold surface than in a non-manifold surface. There are many geometric algorithms which operate only on (or are easier to implement with) manifold surfaces.

Actually, the above statement regarding when a polygonal b-rep describes a manifold surface is an oversimplification. To see this, we introduce the concept of a *T-vertex*. A T-vertex is a vertex which belongs to two or more polygons and simultaneously lies on the edge of another polygon. T-vertices often result from splitting a polygon without similarly splitting adjacent polygons which share the edge or edges being split. Figure 1-14 shows an example of three polygons being joined together, forming two T-vertices. When a b-rep has T-vertices, it may contain edges which are adjacent to three or more polygons, even if the surface is a manifold. It is possible to remove T-vertices by introducing extra vertices in the offending polygons; doing so will cause some strictly convex polygons to become merely convex. Usually, this is not a problem. Removing T-vertices is often necessary for accurate rendering as well as manifold checking, because T-vertices can cause "cracks" to appear between polygons due to limited-precision approximations of polygon edges (a process called scan conversion that is used in rendering).

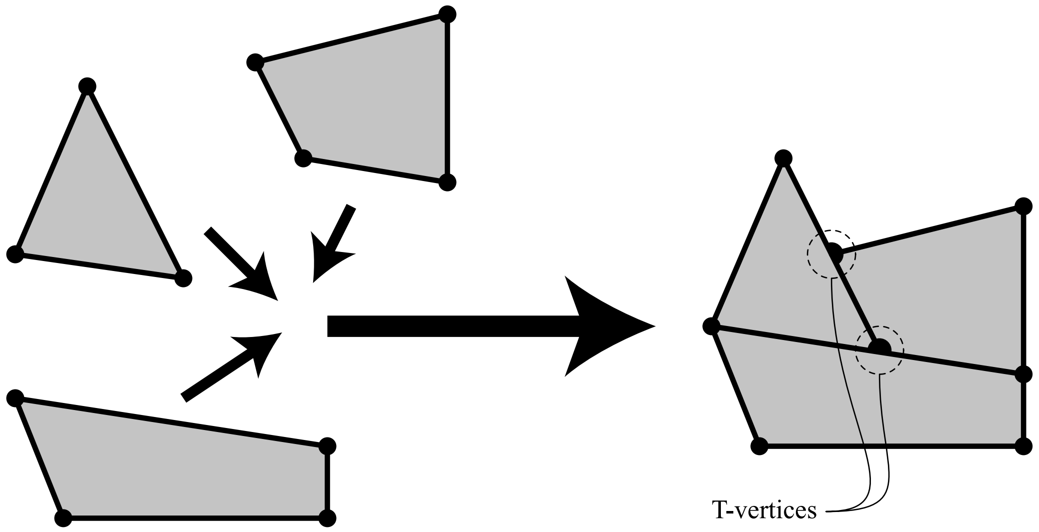


Figure 1-14: Three polygons joined together, forming two T-vertices

Let us compare the definitions of single-shelled polyhedra, multishelled polyhedra, and manifolds. A single-shelled polyhedron is always a manifold. However, a multishelled polyhedron (or rather, its boundary) is a manifold only if the edges of the shells aren't coincident, and similarly a multishelled polygon is a manifold only if the vertices of the shells aren't coincident. Figure 1-15 shows a multishelled polygon with a non-manifold boundary.

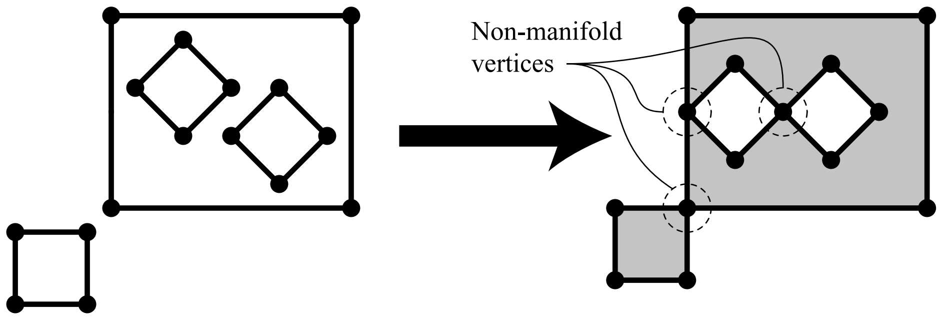


Figure 1-15: A multishelled polygon with non-manifold vertices

Regularized Boolean operations on manifold objects can produce non-manifold objects [CHRI89]. For example, the union of two cubes intersecting along an edge produces a surface with a non-manifold edge with four polygons adjacent to it. It is usually unreasonably restrictive to assume that all objects are manifolds in a modeling program. One way of viewing non-manifold structures is to disassociate the geometry from the topology; hence we might have features that are *geometrically coincident but topologically distinct* [CHRI89].

1.7 Geometric Queries

In dealing with solid models, several important querying operations tend to come into play regardless of the model representations used. One of these operations is known as *point/solid classification* [CHRI89, FOLE96], or sometimes point classification for short. Point classification is used to determine whether a given point is inside, outside, or on the boundary of a solid object. At first glance point classification may seem trivial, but there are subtleties involved which can lead to inconsistent or incorrect results when implemented in a naïve way. Overly-simplified point classification algorithms may return an indeterminate result when the point is on the boundary of the object, and some may outright fail when presented with some inputs unless certain special cases are carefully handled.

Let us consider the problem of point classification with respect to a convex single-shelled polyhedron. This is a rather easy case to visualize (point-classification with respect to a non-convex polyhedron is considerably more difficult). Since the volume enclosed by a convex polyhedron is the intersection of the half-spaces defined by its face planes, we begin by classifying the point with respect to each half-space. This involves a dot product for each classification; we assume the reader is familiar enough with elementary geometric operations so that further explanation is not required. If the point is outside any of the half-spaces, then we can conclude that the point is outside the polyhedron (Figure 1-16a). If the point is inside all of the half-spaces, then we can conclude that the point is inside the polyhedron (Figure 1-16b). If the point is precisely on one or more face planes and inside the remaining half-spaces, then we conclude the point is on the boundary of the polyhedron.

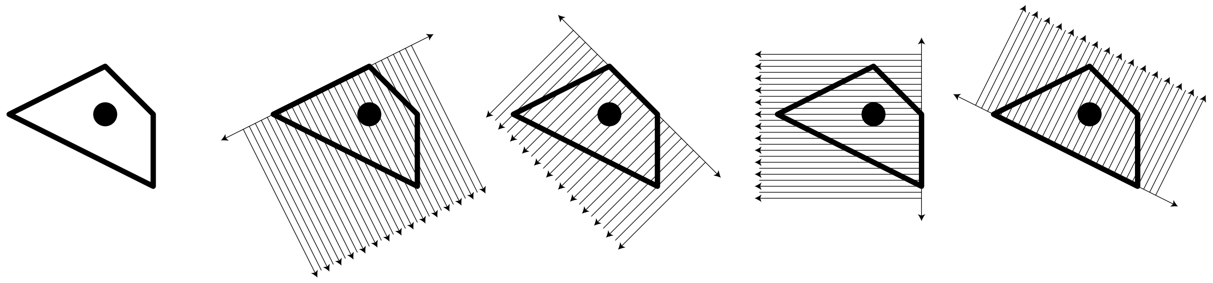


Figure 1-16a: Point classification with respect to convex polygon –
Point lies inside all half-spaces, therefore point lies in inside polygon

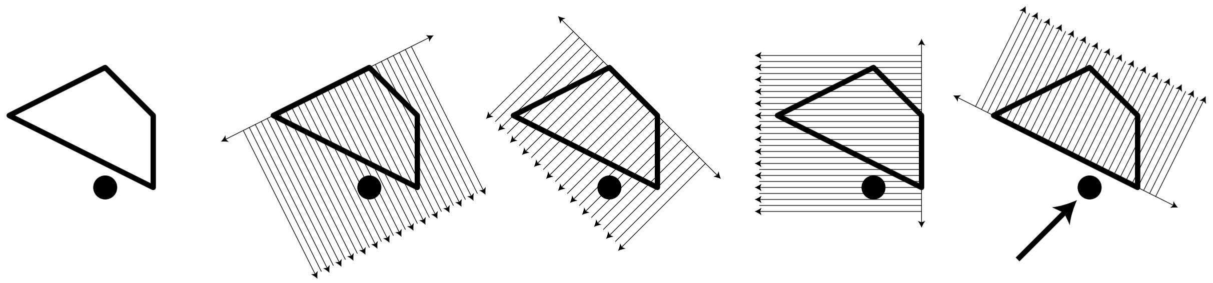


Figure 1-16b: Point classification with respect to convex polygon –
Point lies outside last half-space, therefore point lies outside polygon

In the case of the strictly convex polyhedron, we can derive even more information from this point classification approach. If the point is on the boundary of precisely two face planes and inside the remaining half-spaces, then the point is on the edge of the polyhedron formed by the intersection of the two face planes. If the point is on the boundary of three or more face planes and inside the remaining half-spaces, then the point is coincident with the corresponding vertex of the polyhedron.

We will revisit the problem of point classification with respect to specific object representation schemes as they are presented, and briefly outline algorithms for their solutions. Other similar geometric queries include line/solid classification, curve/solid classification, face/solid classification, solid/solid classification, etc. These are usually more complex than point/solid classification (they may return more complex results than just in/out/on), but often use point/solid classification as a building block. We won't discuss these in depth.

Another important but very different geometric querying operation is called *visible surface determination* (also called, *hidden surface removal*), more often used in rendering than in modeling per se [FOLE96]. Any interactive modeling program will by necessity include a rendering component, so the subject is of relevance. The underlying question asked by visible surface determination is, "Which parts of the boundary of an object are visible from a given viewing location and direction?" This is an important question, since the visible portion of the boundary of a large, densely-occluded object may be many orders of magnitude smaller than the entire boundary. Thus, efficient visible surface determination is key to achieving rapid interactive frame-rates when viewing a complex object.

1.8 Spatial Hierarchies

As the complexity of an object grows, the computational cost involved in manipulating, rendering, and interrogating its geometry becomes significant. *Spatial hierarchies* are used to dramatically reduce the computational cost of such operations. In general, a spatial hierarchy is a tree structure where the nodes of the tree define a subdivision of space. As the tree is traversed downwards, the spatial subdivision defined by the nodes progressively reduces and refines the subset of space represented by the sub-tree at each particular node. The benefit of a spatial hierarchy is that large portions of an object that are unaffected by a localized operation can be ignored (or *rejected*) by selectively traversing only the nodes which are to be affected by the operation. We assume the reader is already familiar with tree data structures and has a general intuition as to how trees are handled in computer science and mathematics.

The next two sections describe two of the most common hierarchical representations used in graphics and modeling; quadtrees and BSP trees. There are many variants on both of these representations, as well as a vast assortment of unrelated hierarchical object representations. Some examples include: *k-d trees*, *range trees*, *interval trees*, *segment trees*, *strip trees*, *prism trees*, *arc trees*, *cone trees*, *beam trees*, and *layered dags*. A thorough (although technically dated) analysis is presented in [HANA89].

1.8.1 Quadtrees and Octrees

One of oldest and most well-known spatial hierarchy representation is the *quadtree* (and its three-dimensional counterpart, the *octree*) [HANA89, CHRI89, FOLE96]. Each node of a quadtree represents a square area and has four children representing the four quadrants of the square. See Figure 1-17. Since the spatial subdivision of a quadtree is restricted to axis-aligned planes, quadtrees rarely define an object explicitly. More often they are used as an approximation of an object, or are superimposed on top of an object and used to accelerate various operations.

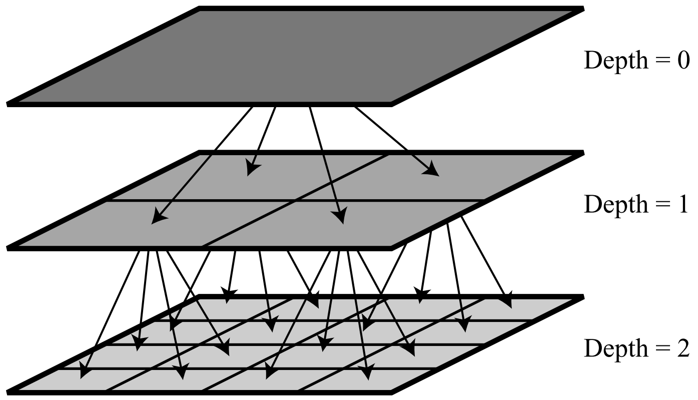
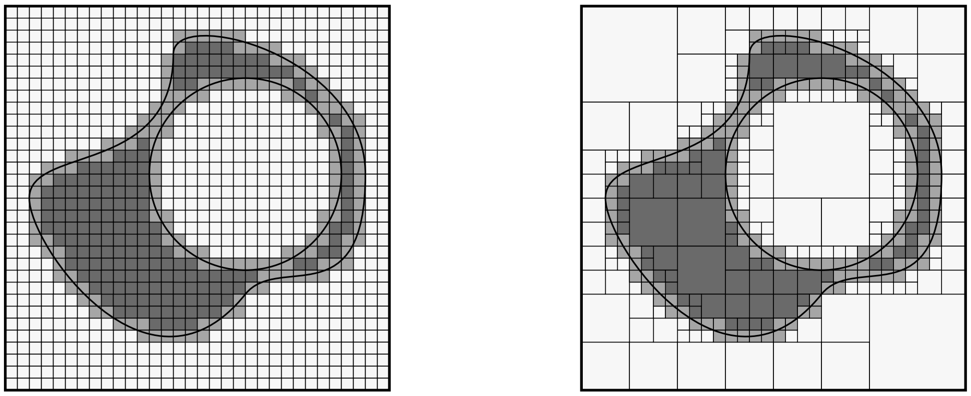


Figure 1-17: Quadtree hierarchy (3 levels)

Given a complex two-dimensional object, we might construct a quadtree such that every node which is entirely covered by the object is marked as an *in*-leaf. Similarly, nodes which don't intersect the object's interior at all are marked as *out*-leaves. Nodes which intersect the object partially are unmarked and have four children. Leaves don't have any children. Figure 1-18 shows an object with a superimposed quadtree constructed in this manner, where *in*-leaves are shaded dark gray and *out*-leaves are shaded white. In addition, the number of leaves at different depths are counted, and compared to the number of grid cells covered by the same object when superimposed on a regular grid.

To prevent the construction of the quadtree from continuing forever, we impose a limit on the depth of the nodes. Nodes which reach this limit and still are only partially covered by the object are marked as indeterminate leaves, and are shaded light gray.



Regular grid partitioning   Quadtree partitioning

□ **OUTSIDE** leaves and cells
▨ **INDETERMINATE** leaves and cells
■ **INSIDE** leaves and cells

```
OUTSIDE LEAVES.................................3 / 17 / 37 /   76   (total = 133, cells = 688)
INSIDE LEAVES.....................................0 /   3 / 19 /   54   (total =   76, cells = 178)
INDETERMINATE LEAVES....................0 /   0 /   0 / 158   (total = 158, cells = 158)
QUADTREE NODES..............................122
QUADTREE TOTAL (nodes + leaves)......489
REGULAR GRID TOTAL (cells)..............1024
QUADTREE MEMORY USAGE..............47.7%
```

Figure 1-18: Regular grid vs quadtree partitioning

We can perform rough point classification on a quadtree in the following manner. Given a point, we start at the root node, and determine which of the four children the point lies in and repeat the process with that node, stopping when a leaf is encountered. The leaf's marking (either *in*, *out*, or *indeterminate*) is then used to classify the point. Note that if the result is indeterminate, then additional computations must be performed in order to correctly classify the point. Computing the quadtree to higher depths result in lower chances of encountering an indeterminate leaf given a random input, but for most objects it is impossible to completely eliminate the indeterminate leaves. Also, points on the boundary of the object won't be properly classified by this method. However, if local geometry is stored in the indeterminate leaves of a quadtree, more accurate point classification is possible.

There is a large body of literature on the subject of quadtrees, and numerous variations on the quadtree concept designed to address particular requirements. Briefly, we have PM quadtrees (including PR1, PR2, and PR3 subvariants), MX quadtrees, PR quadtrees, PMR quadtrees, point quadtrees, line quadtrees, and many more [HANA89].

1.8.2 BSP Trees

Another well-known spatial hierarchy representation is the *binary space partitioning tree*, or *BSP tree* [FOLE96]. Each node of a BSP tree represents a splitting plane, not necessarily axis-aligned. Each node has two children, representing the remaining portion of the object on each side (half-space) of the splitting plane. Leaves of the BSP tree are marked either *in* or *out*. Figure 1-19 shows an example of a two-dimensional BSP tree. If we descend a BSP tree starting at the root node and take the intersection of all the half-spaces we encounter, the result will be convex polyhedra (or convex polygons in the case of a two-dimensional BSP tree) at the *in*-leaves representing the inside region of the object. Hence the BSP tree implicitly defines a collection of non-intersecting, but possibly adjacent, convex objects. If the object being represented by the BSP tree contains no curved surfaces, then the BSP tree completely defines the object. Otherwise, as is usually the case with a quadtree/octree, the BSP tree can only provide an approximation. Also note that BSP tree leaves can represent infinite (unbounded) polygons or polyhedra, if the object has such features.

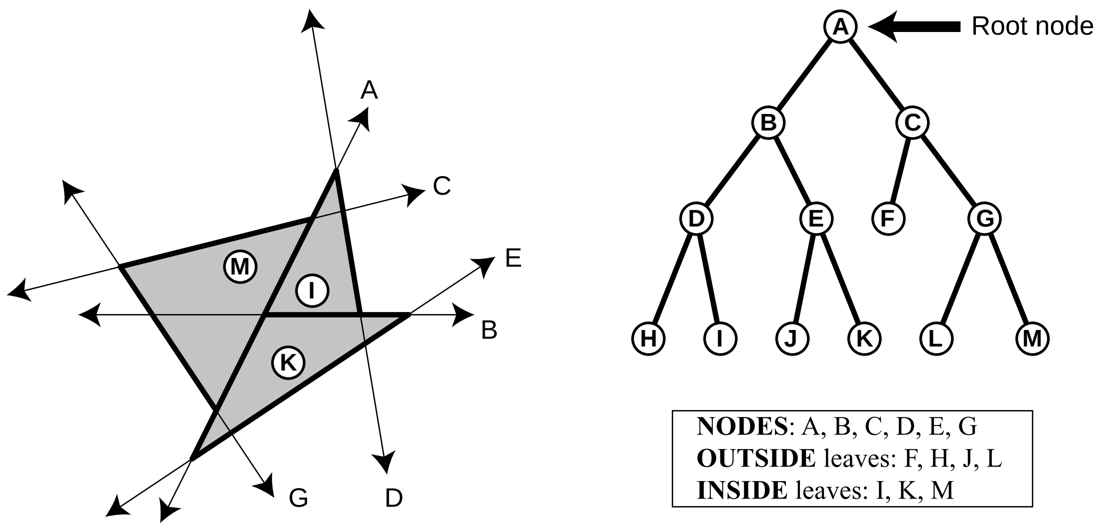


Figure 1-19: A BSP tree

Point classification with a BSP tree proceeds as follows. Starting at the root node, the point is classified with respect to the node's splitting plane. If the point lies on one side of the splitting plane, then the process continues with the appropriate child. If a leaf is encountered, then the point is classified according to the leaf's marking. Otherwise, if a point lies directly on the splitting plane and , then the point must be classified with respect to each of the children and the results compared; if the results are the same then the point is classified accordingly, if they are different then the point is classified as being on the boundary of the object.

Notice that point classification with a BSP tree can accurately determine when a point lies on the boundary of the object, but that the situation is handled rather delicately. With an ordinary quadtree this situation cannot be so easily detected.

1.9 Constructive Solid Geometry (CSG)

Constructive solid geometry, or *CSG*, is a commonly used representation for solid objects [CHRI89, FOLE96]. A CSG representation consists of a binary tree, however it does not define a spatial hierarchy. Instead, it defines an *operational hierarchy* where nodes represent regularized Boolean operations ($\cup$, $\cap$, or –) and leaves represent transformed solid primitives. The strict definition of CSG actually restricts the allowable primitive types to the block (cube), triangular prism, cylinder, torus, and sphere, but this is merely a historical limitation. CSG can be thought of as more of a "description" of how to construct an object rather than an actual description of an object's features. We can say that CSG is an *unevaluated representation*, whereas a b-rep is an *evaluated representation*.

Point classification on a CSG object is generally more difficult than it is with BSP trees or other evaluated or semi-evaluated representations. Consider first the naïve approach: we might try classifying the point with respect to each solid primitive (assuming that this is trivial) and propagating the results upward through the tree, combining them at each node according to the specified Boolean operation. This method works when the point can be classified as either in or out with respect to each solid primitive. However, difficulties arise when the point lies on the boundary of more than one primitive. The general solution is to propagate not only point classifications upwards through the tree, but additional geometric information which describes the neighborhood of the object around the point. The details of neighborhood classification are beyond the scope of this paper.

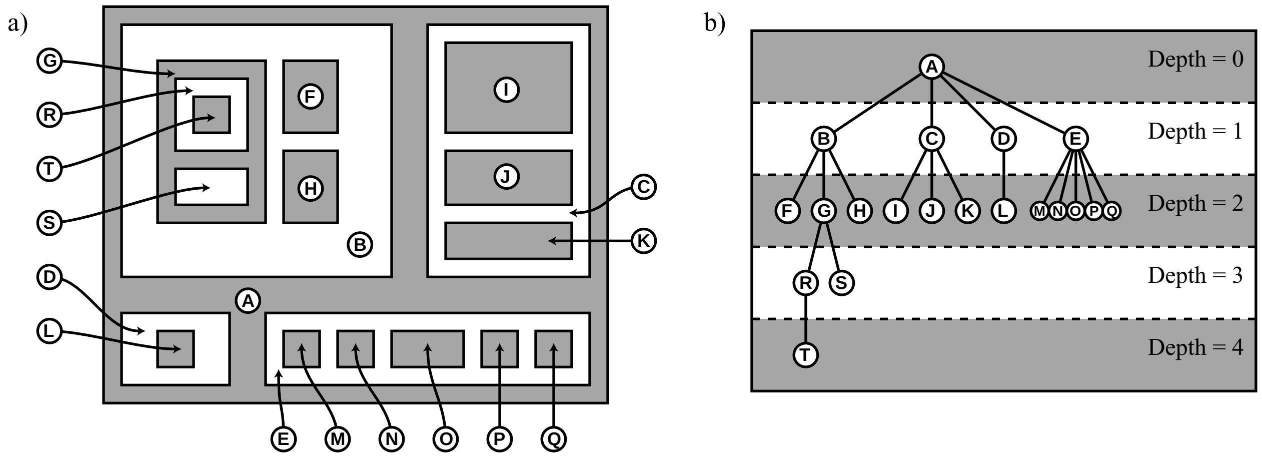
Chapter 2 – HSBM

2.1 Definition of HSBM Object Representation

An HSBM object is stored as a tree hierarchy, where nodes represent strictly convex, non-degenerate polyhedrons. Nodes may have zero, one, or more children (a node with zero children is technically a *leaf*, although no particular distinction is made between nodes and leaves in our discussion). We indicate nodes as *S* (both in the context of the HSBM tree and in the geometric sense), and faces of node polyhedra as *F* (these are *face polygons*, which differ from *b-rep polygons* described later). An HSBM object has the following properties:

1. Each child node is spatially contained within its parent node; i.e., $S_{\text{parent}} \cap^* S_{\text{child}} = S_{\text{child}}$.
2. Children of a particular node do not intersect; i.e., $S_{\text{child-1}} \cap^* S_{\text{child-2}} = \varnothing$. Children may be adjacent, however.

Figure 2-1 shows an HSBM object and its underlying tree. For the sake of clarity, I have chosen to use two-dimensional representations for all HSBM-related figures. These can be thought of as slices of a three-dimensional objects, where the vertices in the figures represent edges, the edges represent polygons, and so on. Shaded regions represent solid space, while unshaded regions represent empty space. Most HSBM figures are drawn with square or rectangular nodes for convenience and clarity.



Figures 2-1: An HSBM object (a) and its underlying tree (b)

2.2 Complex HSBM Objects

Despite the restriction that HSBM nodes be convex, it is easy to construct HSBM objects which are themselves non-convex, and even non-manifold. This is possible because children of a particular node may be internally adjacent to that node's boundary (i.e., faces of children and faces of parents may be coincident). Figure 2-3 shows a simple case with two nodes, the child's right face being coincident with the parent's right face. Figure 2-4 shows the same object without the node boundaries (faces) being outlined – this way it is easier to see the object itself rather than its underlying structure. The object itself is a squared-off "C" shape. Throughout the remainder of this paper, most HSBM figures will be drawn with the node boundaries outlined. This is done for the sake of clarity. It is important to develop the skill of seeing the object through the outline, however.

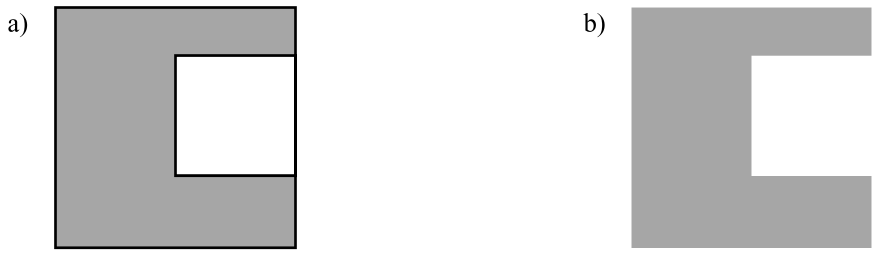


Figure 2-2: A simple HSBM object with node outlines (a), and without node outlines (b)

Figures 2-3 and 2-4 show a much more complex HSBM object with many adjacent nodes and one non-manifold vertex. In such an example, it may be difficult to see at first glance where the nodes themselves actually are – it may seem like a jumble of lines and regions. To help clarify where the nodes are, Figure 2-4 (b) shows the same object with a circle around each node.

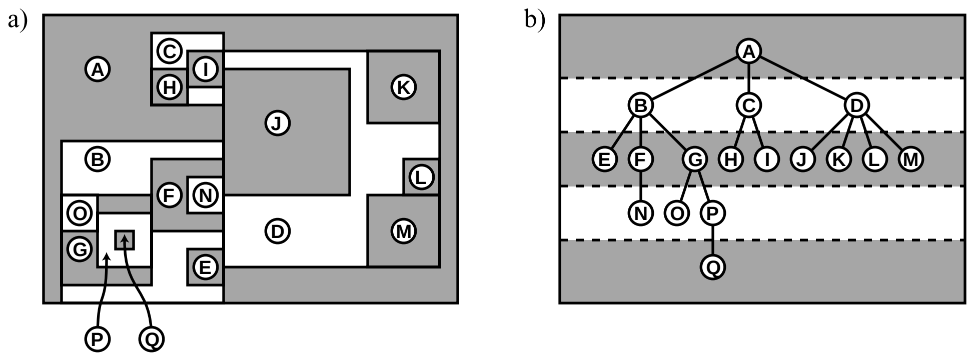


Figure 2-3: A complex HSBM object (a) and its underlying tree (b)

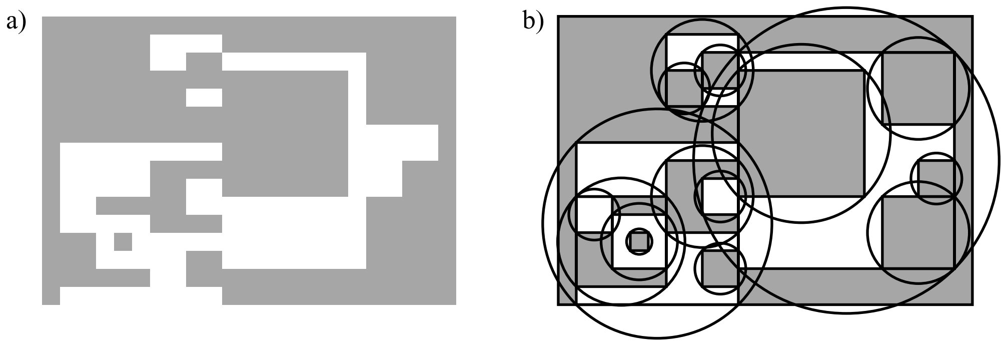


Figure 2-4: A complex HSBM object without node outlines (a), and with circles (b)

2.3 Boundary and Volume Classification

Let us explicitly define the boundary and internal volume of an HSBM object, so that no ambiguities arise later. The following definition involves classifying all points (that is, the infinity of points in three-space, not just the vertices of the object) as *b*-points, *p*-points, or *n*-points. All points which lie on the boundary of *any* node (i.e., within any face polygon *F*) are classified as *b*-points. All other points are classified as either *p*-points (positive) or *n*-points (negative), depending on whether they are inside an odd (*p*-) or an even (*n*-) number of nodes. Note that points entirely outside the HSBM object are trivially classified as *n*-points, since they are in 0 nodes.

Equivalently, we might classify points by recursively traversing the HSBM tree in the following way. Starting at the root node, we test whether the point under consideration is on the boundary of the node (resulting in a *b*-point classification and termination of the search) or strictly inside the node. If it is found to be strictly inside, we test each of the node's children in the same manner. The search terminates when the point is found to be a *b*-point, or it is found to be inside a given node but not any of its children. In the latter case we classify the point according to the depth[4] of the terminating node, modulo 2 (0 yields a *p*-point, 1 yields an *n*-point).

Figure 2-5 shows an HSBM object and various *b*-, *p*-, and *n*-points in it. Figure 2-11 shows the same object with the complete boundary outlined in darker gray.

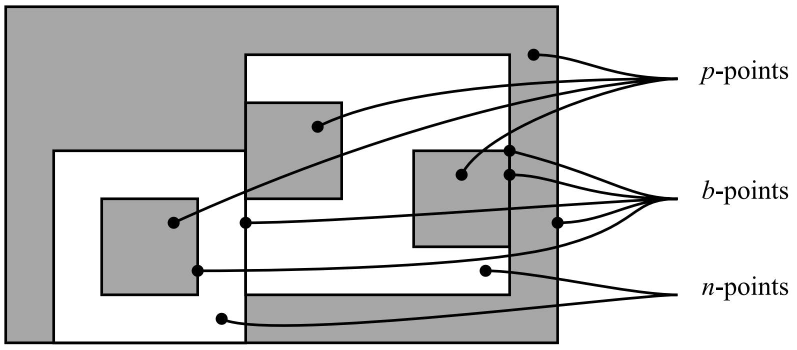


Figure 2-5: Examples of *b*-, *p*- and *n*-points

[4] The depth of a node is defined as the number of steps one must take to reach that node, starting at the root node. The root node has depth 0 by default, its immediate children have depth 1, and so on.

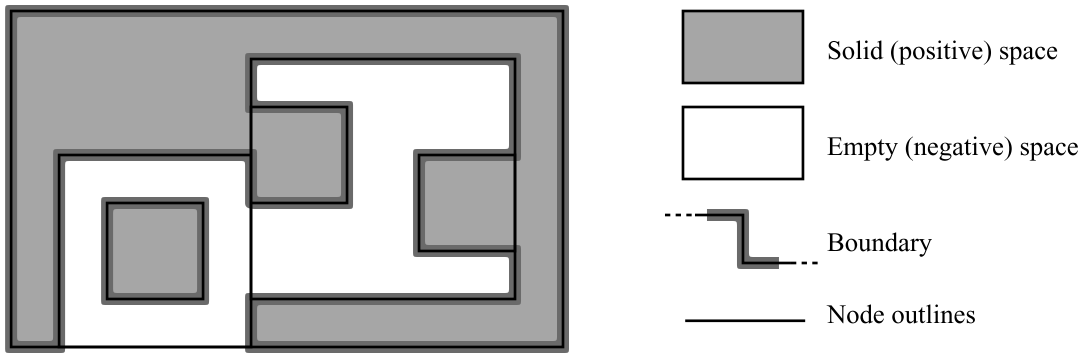


Figure 2-6: Boundary of an HSBM object

Given the above classification of points, we can define the boundary and internal volume of the HSBM object:

1. The **boundary** is the subset of *b*-points whose local neighborhood contains both *p*-points and *n*-points.
2. The **internal volume** is the subset of *b*- and *p*-points whose local neighborhood contains no *n*-points.

Since it is quite impractical to consider an infinity of points in a computer program, we prefer to treat the boundary of an HSBM object as a collection of convex polygons. This is a more traditional b-rep form, which we now turn to.

2.4 Polygonalization of Boundary

We construct the b-rep of an HSBM object from the convex face polygons via regularized Boolean operations. Since these operations may result in non-convex polygons, we choose to break these up into smaller convex polygons[5]. These polygons will be referred to as *b-rep polygons*, denoted as *P*. B-rep polygons have some obvious properties: the set of all b-rep polygons covers the boundary of the HSBM object exactly, and no two b-rep polygons overlap (share a non-degenerate intersection).

To exploit the advantages of the HSBM hierarchy (e.g., hierarchical rejection during construction and rendering), we assign each b-rep polygon *P* to a particular face polygon *F* which spatially contains it. In cases where multiple face polygons overlap and a b-rep polygon might be assigned to any of the overlapping face polygons, we choose the face polygon belonging to the node at the greatest depth in the tree. It is easy to see that there will never be an ambiguous situation where a b-rep polygon is coincident with two face polygons belonging to nodes at the same depth in the tree and no face polygons belonging to nodes at a deeper level.

Figure 2-7 shows an HSBM object with the b-rep polygons drawn as shaded arcs. Since the figure represents a two-dimensional slice, b-rep "polygons" would ordinarily be drawn as lines. I have chosen to curve them inwards towards the node of the face that they are assigned to, so that the relationships are clear.

[5] Actually, there is no reason why the algorithms described here require this decomposition into convex polygons, other than to make the implementation and the diagrams more clear. If non-convex polygons are broken up into multiple convex polygons, there are a multitude of ways to do this. The field of computational geometry treats this type of problem extensively – but for the purposes of our discussion here, the method of decomposition is irrelevant.

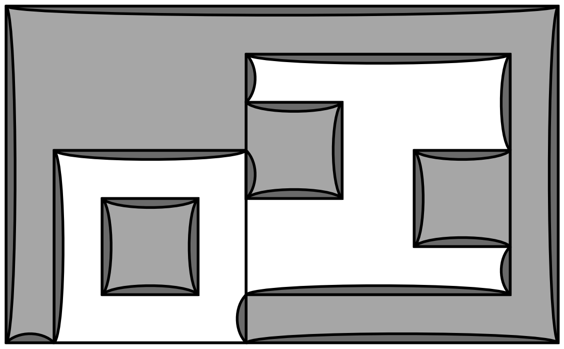


Figure 2-7:   An HSBM object with b-rep polygons

Figure 2-8 shows how simply offsetting the spatial location of the nodes can drastically alter the b-rep of an HSBM object. We start with the same object as used in Figure 2-1, and then offset the nodes (keeping children contained within their parents). The resulting b-rep changes, while the HSBM tree hierarchy stays exactly the same. Note that the face polygons <u>do not change</u> by this operation. This exemplifies the importance of treating face polygons and b-rep polygons separately.

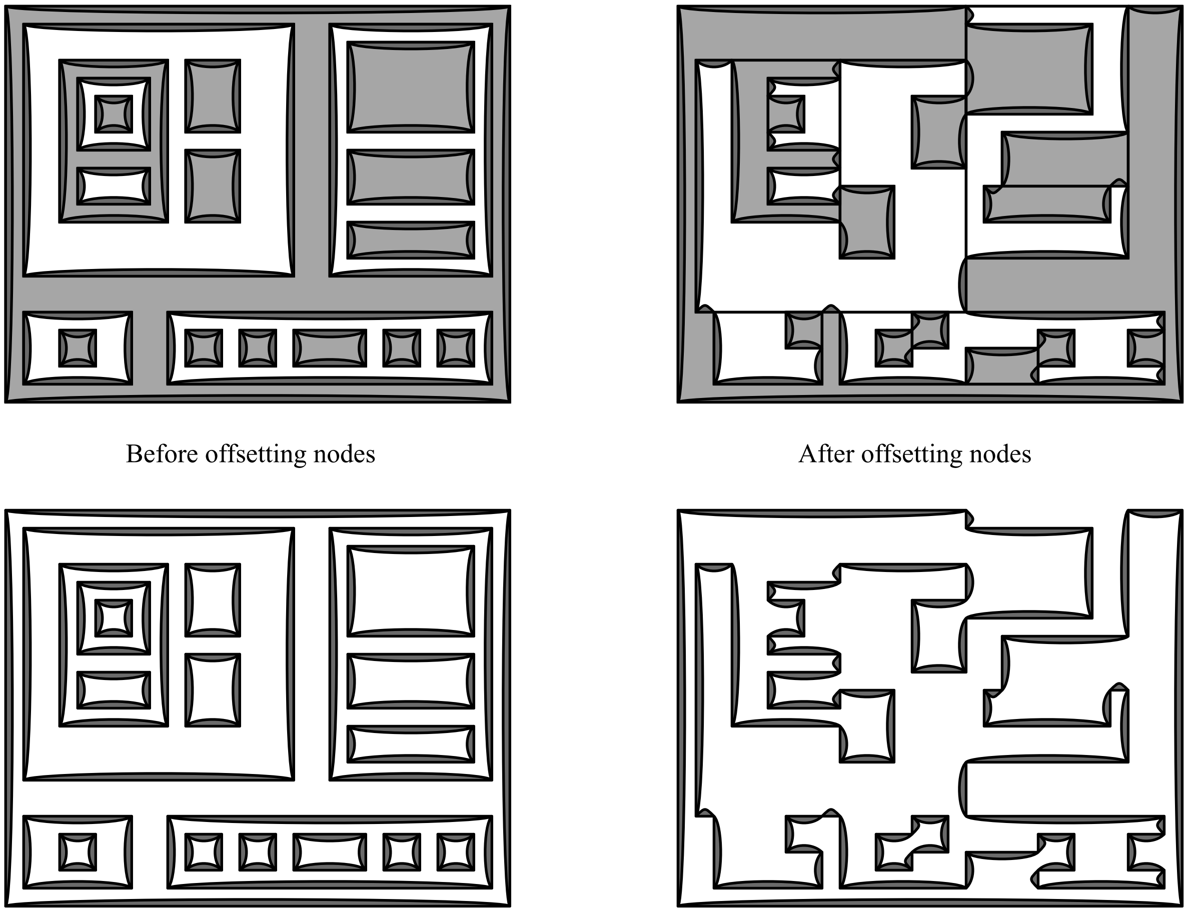


Figure 2-8:   Effect of offsetting nodes on the boundary of an HSBM object

2.5 Incremental Construction of an HSBM Object

Note that we have made the statement that the set of b-rep polygons may be constructed from the set of face polygons, but we have not explicitly defined how this is done. Actually, we left this process intentionally vague – in practice (that is, in a computer application), we don't need to construct the b-rep "from scratch". Instead, we rely on *incremental construction*, where the object is built up node-by-node, and the b-rep is maintained and updated after each node is inserted. It is assumed that during incremental construction, a node is inserted into the tree only after its parent has been inserted (the exception being the root node, which has no parent). Nodes inserted at even depths in the tree add additional volume to the object, while nodes inserted at odd depths remove existing volume. This is analogous to performing Boolean union and subtract operations between the existing partially-constructed object and the newly-inserted node. This "node insertion" process operates at a lower level than the usual solid modeling operations; we will describe how to perform solid modeling with these methods later. Figure 2-9 shows an HSBM object being built by incremental construction.

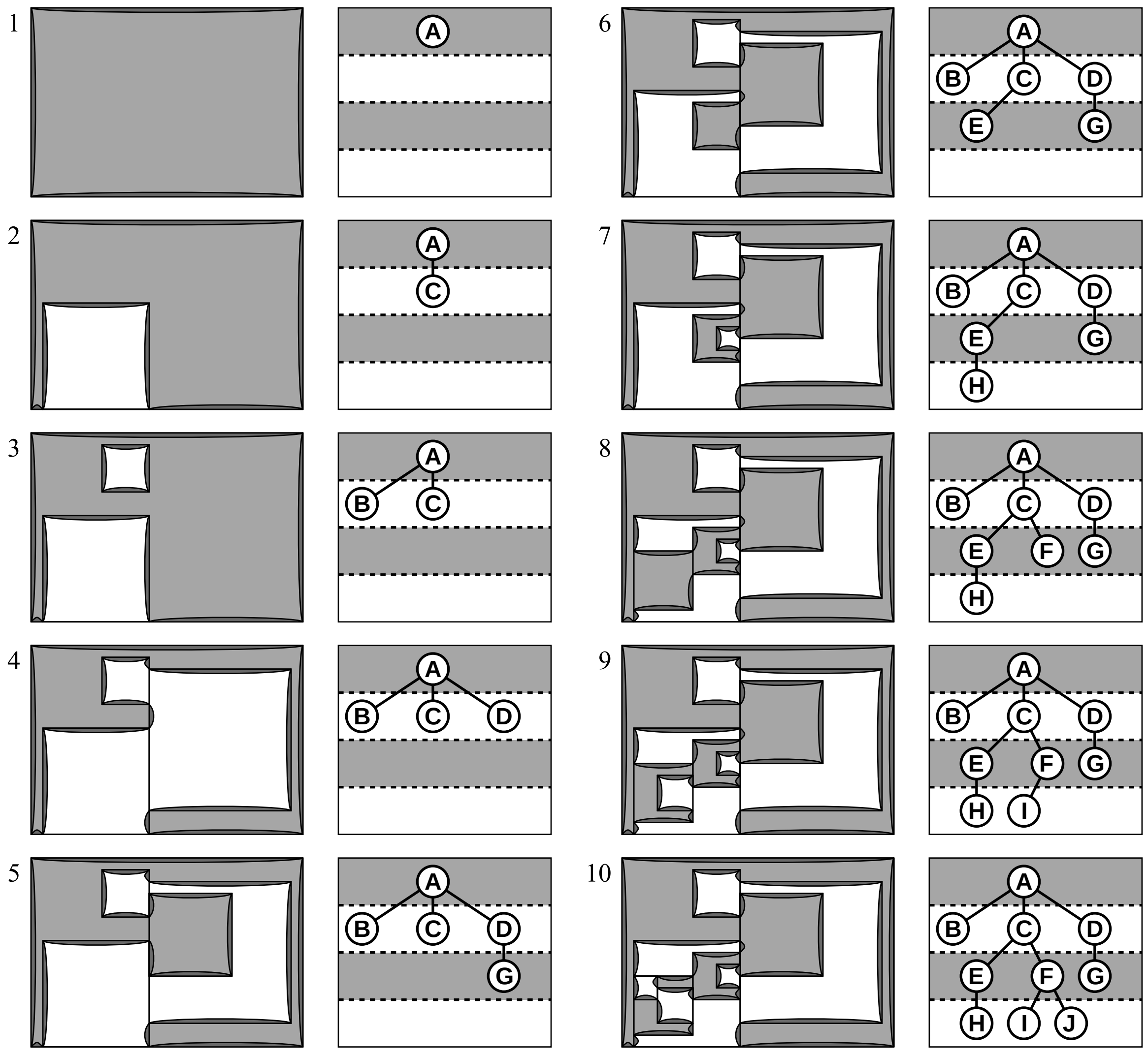


Figure 2-9: An HSBM object being built by incremental construction

2.6 Boundary Updates During Incremental Construction

At any given point during incremental construction, we have a partially-constructed HSBM object with a valid set of b-rep polygons that describe the boundary of the partially-constructed object. The next node to be inserted we will call the *newly-inserted node*; let's say its depth in the HSBM tree is *d*. Nodes may be inserted into the tree in any order (provided that nodes are inserted after their parents); this will not affect the b-rep's validity. Our task is to update the b-rep to reflect the changes made by the newly-inserted node.

All b-rep updates will obviously take place within the face-planes of the newly-inserted node. Therefore, we restrict our attention to existing polygons which are coincident with these face-planes. Further, we may proceed to update the b-rep plane-by-plane, since updates within one plane will not affect another.

A key concept in attacking this problem is categorizing sets of face polygons which lie in a given plane and belong to nodes at a particular depth in the HSBM tree. To better understand this we turn to Figure 2-10, which shows an HSBM object with many face polygons overlapping on a central plane. Since our figure is a two-dimensional slice, it is very difficult to highlight all these face polygons directly and organize them into coherent groups without complicating the figure. Therefore, we choose to draw vertical boxes (face-boxes) on both sides of the object to represent face polygons, with horizontal lines showing which nodes they belong to. The horizontal spacing of the boxes is further organized so that the groups (at each depth in the tree) are fully apparent.

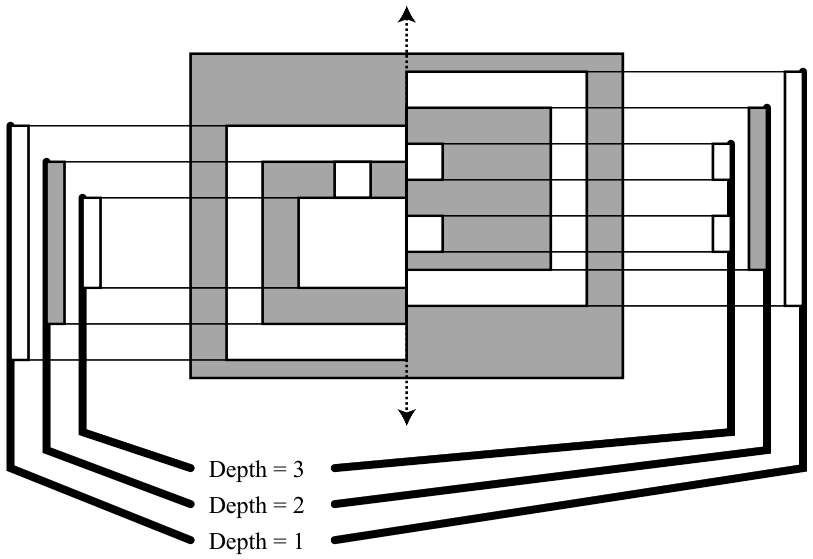


Figure 2-10: Face groups

The figure can be further enhanced by drawing the b-rep polygons which lie in the plane under consideration in the face-boxes. Figure 2-11 shows the same object with the b-rep polygons included. Dotted lines are used to aid the eye in connecting the b-rep polygons in the face-boxes with the b-rep polygons in the object itself.

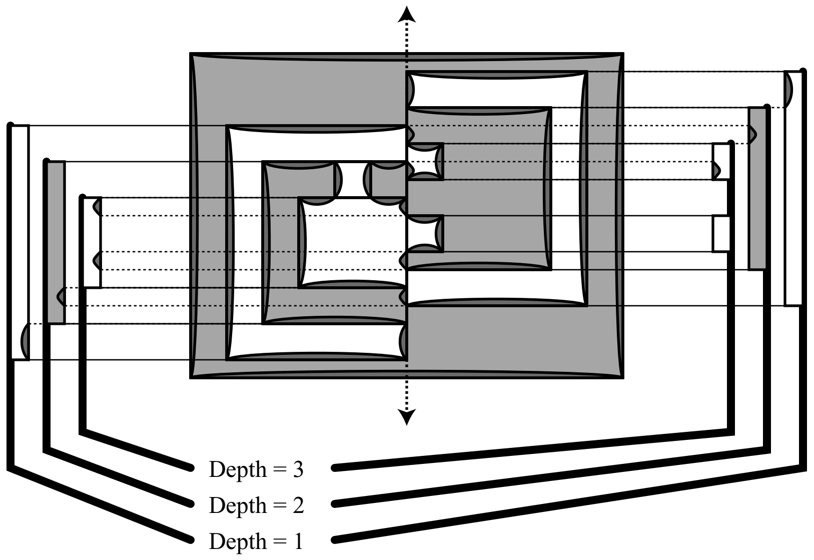


Figure 2-11:   Face and b-rep groups

Note that in Figure 2-11, on the left side is a face which contains two b-rep polygons. On the right side is a face group that contains two face polygons. Figure 2-12 below shows a much more complicated HSBM object with polygons groups. In this figure we omit the b-rep polygons which don't lie in the plane of interest, as they tend to clutter the drawing.

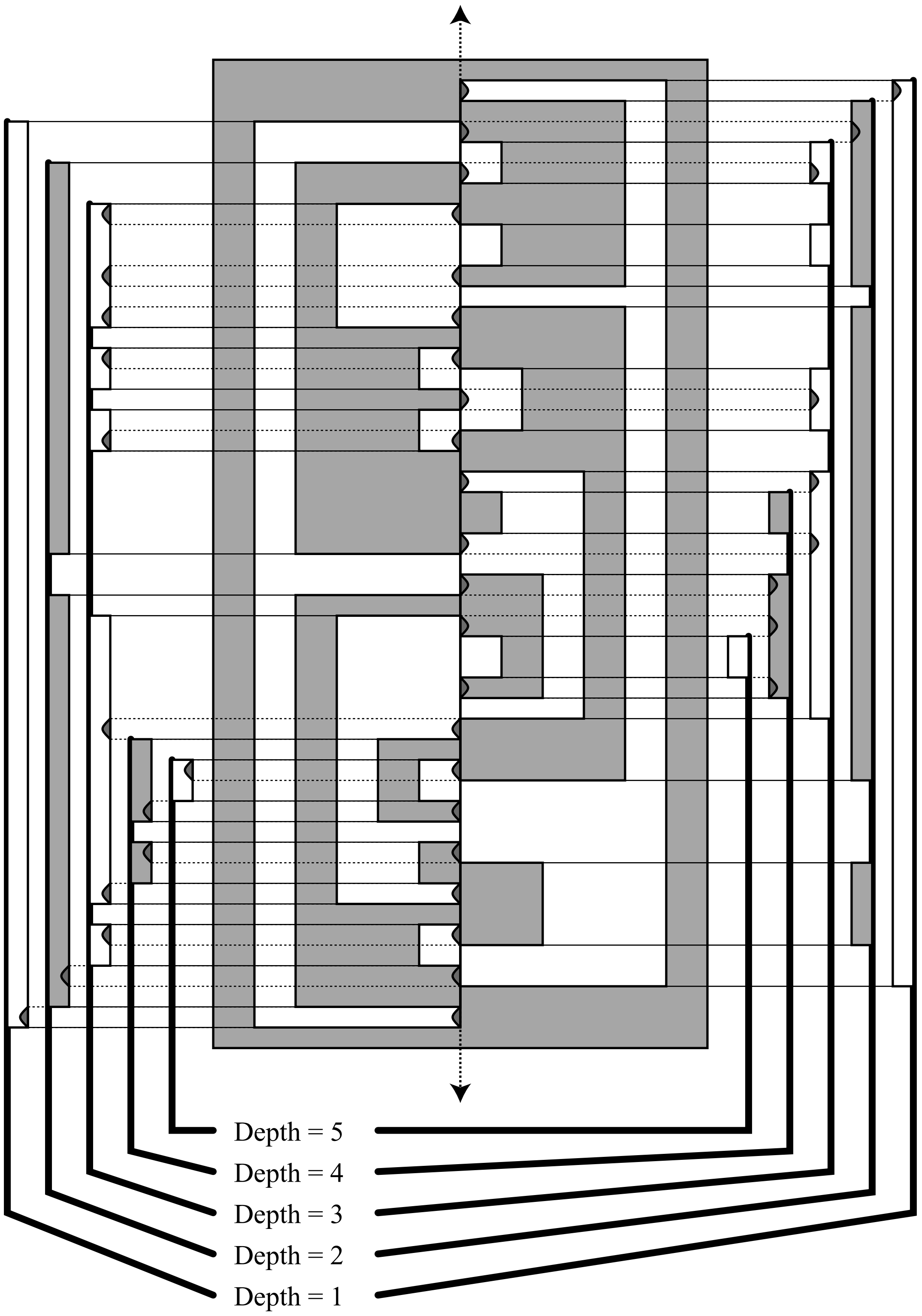


Figure 2-12: A very complex HSBM object with face and b-rep groups

Conclusion and Retrospective

When I began this project back in early Spring of 1999, my goal was to create a 3D game engine in the style of Quake and Unreal, but using hyperbolic space instead of ordinary Euclidean space. My inspiration was a short piece of computer animation called “Not Knot”, which depicted a computer-rendered flythrough of a dodecahedral tiling in hyperbolic space. The idea intrigued me for a number of reasons. One of these was that hyperbolic space behaves more or less equivalently to Euclidean geometry in a small local region, but very differently “overall”. This is a difficult concept to describe non-mathematically. I figured that because the space behaves in a familiar way in small regions, navigating through it in a game environment wouldn’t be too uncomfortable (this is in contrast to, say, four-dimensional space, where the local behavior is so unfamiliar that gameplay would likely be impossible). Another thing that attracted me to three-dimensional hyperbolic space was the fact that transformations could be modeled using the same homogeneous 4x4 matrices that are often used for ordinary three-dimensional space. I learned this after reading a paper entitled “Visualizing Hyperbolic Space: Unusual Uses of 4x4 Matrices”, co-written by the producer of Not Knot, and also by reading various sources on the Web.

Since I realized that creating a game engine in hyperbolic space would be an immense challenge, I decided to split the project into four stages. The first stage would be to develop a library of hyperbolic transformation function in C++ that I could use as a foundation for all the hyperbolic engine work to follow. This was relatively straightforward because all of the equations were readily available on the Web and in textbooks. The second stage was to use the transformation library to create two-dimensional hyperbolic tilings – I had seen many of these in books and on the Web, but most of them looked very crude. This second stage I also completed successfully, and produced some impressive PostScript images. The third stage would be to write an interactive two-dimensional “maze game” in hyperbolic space. This would be something much less complex than a Quake-style game. The fourth stage (which I never started) was to be a Quake-style game engine with “portals” into a more abstract hyperbolic universe, which the player could traverse. I spent at least a month on the third stage of the project, and eventually did more or less complete it. However, for a number of reasons which I will describe, I didn’t feel that continuing on to create a full 3D game engine in hyperbolic space would be feasible, so after completion of the interactive hyperbolic maze game I decided to switch gears and work exclusively on 3D solid modeling.

I am somewhat proud of the hyperbolic maze game, so I will describe it in a little more detail. First of all, it was based on a square tiling of hyperbolic space, where five squares meet at every vertex. I used a Poincaré projection model (in which angles are treated in a Euclidean fashion but straight lines are projected as circular arcs), and subdivided each edge into four segments – I found this level of subdivision was sufficient for visual smoothness. I also added a radial “fog” effect using an alpha channel, and later I added texture mapping (tiles being subdivided into 32 triangles), thickened pseudo-3D walls, a Klein projection mode and a Poincaré-Klein morphing mode, persistent objects that could be dropped by the player, collision-detection (sliding along the walls), and I spent some time attempting to do smooth interpolations between hyperbolic transformations (but never got it exactly right).

While working on the hyperbolic maze game, I was attempting to collaborate with a friend of mine, Jeff Brown. He was genuinely interested in the idea of using hyperbolic space in a graphics project, and he was the only person I was in contact with at the time (besides of course Ken Hoffman, my advisor and Division III member) who would understand the mathematics involved. While we did have many inspiring and productive discussions on the subject, we didn't end up collaborating in the actual development of the program. Jeff went on to complete an NS Division I using a simpler hyperbolic 3D engine which he created himself.

In playing with the Poincaré and Klein projection modes, I found that in Poincaré mode the scene appeared to be flat, but the individual walls were curved. In Klein mode, the scene appeared to be curved (almost as if it were on a sphere), but the walls were straight. Also, in Klein mode, I still had to subdivide the tiles in order to make the texture mapping smooth – for this, I used subdivision in Klein space for the vertices coupled with subdivision in Euclidean (i.e., screen) space for the texture coordinates. Doing this allowed me to use ordinary square texture maps (which I ripped from Quake) in the hyperbolic world.

Another strange effect of the hyperbolic space that I found was that when moving around the maze, it would appear as if I were moving in tight circles, coming back to the same tiles over and over again. In reality, however, I would be moving along the outer perimeter of the maze, never re-visiting the same tile twice. The underlying reason for this is that in hyperbolic space, cumulative translations cause actual *rotations*, so as one moves around the maze via translation, the view tends to spin. The maze was constructed so that between any two tiles, there would be exactly one path – therefore it was impossible to go in a continuous "loop". Additionally, I added a feature where the tiles would be highlighted when crossed – the highlight would remain active for the duration of the game. Both of these features prove that one is *not* going in loops, however it is difficult to adjust to the visual appearance. I don't think this effect would be evident in a three-dimensional version (played from first-person perspective).

One of the most interesting parts of writing the hyperbolic maze game was figuring out how to represent the maze geometry. Since the projection of hyperbolic space maps all points onto the unit disc (or the unit sphere in three-space), creating a large hyperbolic object and storing the vertices in their projected state leads to severe precision problems, even when implemented in 64-bit floating point. To see this, imagine two points which are close together in hyperbolic space but very far from the origin – they get projected to two points near the boundary of the unit disc. But since their coordinates are represented with finite precision, when they are transformed back near the origin they won't necessarily be the same (hyperbolic) distance apart. The only way to avoid this problem that I was able to think of was to impose a tiling on the hyperbolic space and store all geometry as projected points in the coordinate frame of the tiles they reside in. This meant creating a tiling datastructure without geometry (not very difficult – each tile points to four neighboring tiles) and associating geometry with the tiles. This approach worked perfectly in the game, I was able to create arbitrarily large mazes without any distinguishable geometric instabilities near the edges of the maze. To render the maze, I would recursively concatenate tile-to-tile transformations (there are four of these, which I pre-calculated and stored in a table) and render the tiles, stopping when a tile was completely off the screen.

On a slightly different note, I began this project using the 3Dfx Glide API (using a Voodoo2). It was around this time that I began to realize that 3Dfx was not going to be the best way to do graphics for much longer, so I devoted a weekend to learning OpenGL and porting my engine to an SGI Onyx that I had temporary access to. The performance of the $100,000+ machine was nearly four times that of my outdated PC (Voodoo2, PPro200MHz). Still, I found that when zoomed out in the Poincaré view, the sheer number of on-screen triangles caused the engine to run fairly slowly. Sure, I could get 30-60 frames per second even on my PC when the view was zoomed in a little, but I could only imagine that the performance problems in the two-dimensional game would be an order of magnitude more severe in a three-dimensional game. This is the primary reason why I decided to stop development of the three-dimensional hyperbolic game engine.

The other reason I decided to stop development was that I was becoming more aware of how difficult it would be to create interesting architecture for *any* game engine, let alone a hyperbolic one. The maze game's architecture had been limited to a fairly simple geometric embellishment of the tile edges (I had to do some fancy polygon work to make the corners look good). I had originally envisioned a Quake-style environment with some sort of "portals" into a more abstract hyperbolic universe, but now I began to wonder how I would make the Quake-style environment in the first place, let alone the hyperbolic universe. True, I could have decided to just stick with the hyperbolic space project in three-dimensions in a more pure form, but that wasn't as inspiring to me. I could pretty much visualize what a three-dimensional version of the existing two-dimensional hyperbolic maze game would be like – what I really wanted was to be able to run around in a Quake-style environment, and jump in and out of a hyperbolic universe. The *transition* between the two was what I was truly interested in. In retrospect I now realize that this was far too great a vision to realize in a Division III project, but I did learn a lot while trying.

As soon as it became clear to me that my original vision would be infeasible with the resources at my disposal, I switched my Division III theme from the hyperbolic graphics engine to a more traditional Quake-style engine. I found that I was more inspired to develop just the Quake-style environment (for which I would need a solid modeling program) than I was to develop just a three-dimensional version of what I already had. Lee Spector, my Division III chairperson, was not happy about this – he viewed the hyperbolic project as being genuinely innovative since essentially nothing of the sort already existed. However, this "Quake-style" project would be just one of thousands of similar projects being worked on in the game industry and in the field of computer graphics. While I agree with his reasoning, this wasn't enough to deter me from following my inspirations wherever they might lead.

By the time I made the transition away from the hyperbolic project, it was already summer. I was living in Berkeley with my fiancé Sharyn Case, and my outdated PC had been somewhat smashed by UPS shipping. I was able to get it partially running, enough to continue programming work. I began by researching the Quake and Unreal engines and the available modelers, as well as the formats in which the architectural geometry for these games was stored. I already knew that both of these games used BSP trees heavily, but I didn't want to use BSP trees in my own engine. (I don't know why, but I have a strange aversion to BSP trees, perhaps because I've never actually used them – although I think I understand them fairly well). My intention was to convert the Quake or Unreal file formats into whatever format I needed – this would likely be some sort of polyhedral/portal format in which convex volumes and their inter-connectivity are stored, and rendering takes place by recursively visiting polyhedral volumes and refining the view frustum to match the projected boundary of the portals. Such techniques I had already used in my previous 3D engines, and I knew were used in many existing games such as Descent, Marathon, Duke Nukem 3D, the upcoming Prey, and to a limited extent Quake2 and Unreal.

What I found was that the Unreal format was completely obscure – no information was available on the Web. The Quake format, on the other hand, was well-documented and easily parsed – but unfortunately, after a few days of trying, I was fairly convinced that conversion of the Quake BSP format into a polyhedral/portal format would be less than trivial. In fact, I began to think that it would be about as difficult as writing my own solid modeler program, and since the existing Quake editors seemed clumsy and difficult to use, I reasoned that writing my own solid modeler with an interface specifically suited towards my own needs would be the way to go.

With my trusty but aging 3Dfx card, I began work on a solid modeler. This was now the theme of my Division III – I knew that I wouldn't have a whole engine ready by the end of the year, but at least I could have a program that I could use to create architectural geometry and move around in it. Despite my attempts at researching existing techniques for solid modeling operations, I found very little information was available. (More recently, however, I have found a wealth of information on solid modeling – I can't really account for my inability to find information over the summer, except that information-finding isn't my greatest skill.) Anyway, without any useful references, I was forced to come up with a framework entirely on my own.

The first attempts I made at solid modeling involved placing polyhedra in three-dimensional space, finding their intersections, cutting them up, and discarding the intersecting portions so that what was left was a collection of non-overlapping polyhedra which constituted the union of the original polyhedra. Such a process involved writing code for splitting a polyhedron along a given plane, testing for interference between two polyhedra and between a polyhedron and a face, and other basic functionality. Luckily, most of the steps were either very straightforward or similar enough to processes I had coded in the past (for example – clipping polygons to the viewing frustum) that I was able to finish this part of the project without much difficulty.

At this point, all my viewing was in wireframe. Rendering opaque polygons was of little use, since it is impossible to tell the difference between a collection of overlapping shapes and their union when viewed from the outside. When viewed from the inside, rendering opaque polygons would simply obscure the view. What I needed was a way of computing the *boundary* of the set of polyhedra. Even though I had succeeded in forming the union from a volumetric point of view, the boundary required additional computations.

I had two ideas for how to compute the boundary. First of all, I knew (from mental visualization) that the boundary would consist of all the faces of the polyhedra, with the overlapping portions of the faces subtracted (or *cut out*). I also knew that the Glide API needed convex polygons to render – so the problem I was faced with was how to take a convex polygon and cut out other convex polygons from it, and split up the result into convex polygons. I could see several possible approaches to this task. For example, I thought I might take the triangulation approach, where I treat everything as triangles, and only have to deal with the subtraction problem between two triangles. Another approach I thought of using was to form *triangle strips* by walking an edge down two connected paths at once, and branching where needed. This approach seemed too complex, although I was tempted because it would avoid an artifact known as *T-vertices* which can cause tiny pixel-wide cracks to appear between polygons during rendering. The approach I finally settled on was the split up the polygons much like I was already splitting up the polyhedra – that is, when a cutting polygon intersected a boundary polygon, I would take one of the cutting polygon's edges and split the boundary polygon along that edge, producing two boundary polygons. Repeating this process and discarding the final overlapping boundary portion (which would of course be completely covered by the cutting polygon). I liked this for several reasons – one was that I could use the same or similar code between the polyhedron intersection and cutting and the polygon intersection and cutting, thereby reducing my debugging efforts. Secondly, I could use the original face-planes of the polyhedra as splitting planes for the boundary polygons, instead of having to resort to projecting the polygons into two-dimensions and re-projecting them back, or using Pluecker coordinates. In fact, the only core geometric operations in my solid modeler were a function that returned the distance between a point and a plane, and a function which found the intersection between a line and a plane.

After implementing the boundary-generating code, I finally had something that I could "fly around in". My first models were mostly collections of cubes, since I didn't have an interface to inputting data into the modeler and had to resort to hard-coded geometry (still, this was an order of magnitude easier than manually computing vertices of all the polygons!) And since my geometry was all axis-aligned, I didn't run into the infamous robustness issues involved in solid modeling that I now know so well. But that was soon to change.

To further test my modeler, I tried rotating my cubes by a random amount (at this point I had the program creating architecture by randomly placing 30 or so cubes into the scene, with sufficient density so that they would form roughly into single cluster). At once, I noticed that there were problems – in certain configurations, the solid modeling operations would simply "fail", producing weird invalid output. I figured this was due to finite precision floating-point math, which of course was correct – but I didn't think about it in too much depth. What I was unaware of was that perfectly robust solid modeling algorithms are still, *to this day*, unknown. So I merely implemented a little fudge-factor into the function that returned the distance between a point and a plane so that if the distance were sufficiently close to zero (around 10^-6), the function would return zero. This helped, but did not eliminate all errors.

My next problem was to implement aggressive visible surface determination. At the time, I believed that visible surface determination (the process by which a program determines which surfaces of an object are visible from a given location) was the be-all-and-end-all problem in 3D game engines. Now, I have a somewhat more balanced view I think – I see visible surface determination, collision physics, solid modeling interface, solid modeling robustness, and many other issues as being terribly interrelated and important to any game engine. But with visible surface determination in mind, I decided to attack this problem in full force before I had polished my existing solid modeler.

The first technique I tried to use for visible surface determination was portals. Portal polygons would simply be the intersections between adjacent face polygons of the polyhedra, so they were easy to calculate. The recursive rendering process was similarly easy to implement. However, what I found was that aggressive use of portals *slowed down* my engine! To give a rough example, a scene consisting of 4000 polygons ran at a tolerable 20 frames per second on my old hardware. When using portals, even though the number of visible polygons on-screen and the number of rendered pixels was significantly reduced, the performance would drop down to 2-3 frames per second! This I attributed to the way portals cut up the scene – if a polyhedron is visible through two separate portals, it is rendered twice. Even though only the pixels within the projected portals are drawn (so no pixel is drawn twice, well, that would be true if I used polygonal portals, except that I used rectangular pixels so the statement isn't quite accurate) and only the polygons within the projected portals are drawn, the number of polygons *considered* by the algorithm is still doubled. This only gets worse when another polyhedron is seen through two portals in the polyhedron which was considered twice. The problem exponentiates.

So I dropped the portals. I still think portals are a great idea, and very practical, but they are to be used sparingly. Realistically, they should be used to join two large areas of complex geometry that are connected by a small opening – for example, one might use a portal to join a large chamber in a castle to the hallway leading into it – when the camera is in the chamber and the portal is out of view, the hallway can be ignored (and similarly when the camera is in the hallway, the chamber can be ignored). If one examines how Quake2 and Unreal use portals, Quake2 uses them very sparingly and only when explicitly placed by the map designer. Similarly, Unreal doesn't seem to use portals in the conventional sense, but they have something called "zones" which are large convex polyhedra that partition the world into large chunks, and one might assume that the engine uses portals to render between zones.

Without portals to satiate my need for aggressive visible surface determination, I turned to spatial hierarchies. Pretty much every 3D game engine I know uses spatial hierarchies on some level – most of them use BSP trees which are a form of spatial hierarchy. I didn't want to use BSP trees in particular, because I was still hung up on the notion of convex polyhedral regions (I still am, to some extent). So for me the natural spatial hierarchy was a multishelled polyhedron, where a single world-polyhedron is cut up by polyhedra within it, and then those polyhedra have smaller solid polyhedra within them, and so on. This was the idea that eventually led to my set of algorithms that I call HSBM (for Hierarchical Solid Boolean Modeling) and the current form of my 3D engine.

Actually, I did some more work before switching to the multishelled polyhedron approach. Namely, I implemented solid differencing (in addition to the solid unions). Also, I began tinkering in collision detection physics. I found that flying through a complicated architectural model was nowhere near as fun as running around on stairways and jumping across ledges, looking over balconies, etc. (as one might do in a game such as Quake). All that interaction with the architecture was impossible without collision-detection. Let it be clear that when I say "collision detection", I mean collision detection and response, that is to say, what happens to the player object as it collides against a surface (or multiple surfaces). This problem is severely difficult, whereas straight collision "detection" is fairly trivial and well-known. Remember that I had implemented some collision detection in my hyperbolic maze game (which was, actually, the first tolerable collision detection I had ever coded – earlier efforts usually involved stopping the camera movement at the point of impact rather than allowing it to slide). The hyperbolic maze collision had been a very simple case, however, because all the walls were fairly long and a single impulse never caused the player object to collide against multiple walls.

Sitting in the Espresso Roma café in Berkeley for five hours a day drinking cappuccino after cappuccino for a month and a half later, I still had not solved the collision detection problem. I still hadn't even found a promising way or pursuing it – everything I came up with seemed to work at first and then I'd find some common case where it wouldn't work. Visualizing it in three-dimensions was difficult as well – it seemed that a lot of algorithms that would “sort of” work in two-dimensions would outright fail in three-dimensions, although I still held to the notion that a “perfect” algorithm in two-dimensions would be perfect in any dimensional space. The notion of “perfect” collision detection is of course not a pure science (the goals of game-engine collision detection are not the same as, say, in mechanical simulation) – I have a set of rules I made up which such an algorithm would have to follow, but that it beyond the scope of this retrospective. Let it suffice to say that it’s a yucky problem.

Back to the solid modeler itself, once I decided to go with the multishelled polyhedron approach I had to re-write my whole engine, mostly from scratch. This took a while. Around this time I also built myself a new, much more powerful computer, and switched everything to OpenGL. I found that computing the boundary of the multishelled polyhedron is a whole lot more complicated than computing the boundary of a set of non-overlapping single-shelled polyhedrons, so it was back to the Espresso Roma café for a while. This time, however, the caffeine worked.

When I got back to school, I continued working on the HSBM algorithms to make them faster. I also began working on a geometry scripting language which was implemented on top of my solid modeler, something text-based that would allow me to create much more complex geometry (for example, I included a transformation stack, loops, named shapes, extrusion, and other features). I knew that I wouldn’t have time to make a graphic interface, so my idea was that the text-based interface would be both functional, and also a bridge between the modeler and whatever graphic interface I might make in the future. Also, I was careful to integrate the computation of the boundary into the modeler as an immediate process after each polyhedron was inserted, rather than as a post-process. This allowed for future interactive editing.

To satisfy the requirements of my “advanced educational activities”, I took a UMASS graduate class in computer graphics with my friend James Beattie. Both of us were really too advanced for it, but we found it enjoyable to work on ridiculously simple problems (such as rendering a Mandelbrot set) and make them really, really complicated. Some of the work I did for that class I am actually fairly proud of – for example, my Mandelbrot renderer included a feature known as “orbit traps” which I independently stumbled on, as well as progressive sampling and a polished interface. Overall, we both spent a huge amount of time working on the assignments for that class, and I have included the programs that I developed as part of my Division III (whether it counts or not!) I even managed to finish the class with an “A” despite the fact that I never finished the last two projects – I showed the professor the current state of my Division III project and he was sufficiently impressed.

While taking the UMASS class, I was also working extensively on collision detection. At this point, I was more interested in collision detection than in solid modeling. I figured out a method of doing collision detection that in theory should work (I still believe it will work), however is plagued by numerical instabilities. I spent a lot of time trying to refine the code and the equations to make this algorithm work – I was able to make a two-dimensional version work “most of the time”, and I am including this sub-project in my Division III. I studied the Quake collision detection more closely (both by playing the game and by researching), and recently the Quake source code was released so of course I studied that. Even more recently, the Quake3 source code (or at least parts of it) has been released, so I have even more material to digest. I was also examining the collision detection in games like Descent (for which the source is available) and Crystal Space. From all the source I've seen, what I've found is that everyone seems to do it quite differently – and everyone gets very different results. Sometimes, even professional game programmers can't get it right, and I've seen games with downright *bad* collision detection.

During the last few months of my Division III, I was forced to write a paper (this paper) by my committee. This meant no more time for coding, and no more time for collision detection. In so doing I learned a few useful skills, including Adobe Illustrator and Adobe PageMaker. I also learned that my HSBM algorithms were wrong – there were certain cases where they would fail (not due to robustness issues). I fixed the algorithms, but not the code. The pseudo-code algorithm presented in Appendix A *should* work, but I stress the word *should*. I also learned that it's almost impossible to clearly communicate how my HSBM algorithms works (to Ken Hoffman) or even my opinion that it's remotely interesting (to Lee Spector). To these ends, I have given up. What you see is what you get.

I can imagine that someone reading my final paper here would be quite disappointed, if they even made it past the textbook-material first chapter. This is because the paper never finishes what it starts – I introduce the concept of HSBM modeling, and begin to talk about how one might attack the problem of updating the boundary, but I never get into the details (except in the pseudo-code). Another thing that I don't do is explain how to bridge solid modeling, which includes such Boolean operations as union, difference, and intersection – and HSBM, which includes such operations as insertion, splitting, and removal of nodes in a tree. There is in fact a fairly elegant connection, but I haven't the time, the space, the money, or the motivation to pursue elaborating on it.

Another thing that decreases my interest in this project is the current direction which game engines are heading. The next big thing (or rather, the current big thing) in game engines is *curved surfaces*, which are extremely ugly to implement in an HSBM context. I have found that current professional CAD/CAM packages (such as AutoCAD and IronCAD) use third-party solid modeling kernels such as ASIC and Parasolid. I researched these kernels, and after obtaining the documentation for both of them (the ACIS docs consist of over 60,000 files and 500 megabytes), I can't really see HSBM being of use in the industry.

I see the whole HSBM project as being a learning experience – certainly, I learned a lot about solid modeling and gained a much more in-depth appreciation for the subtleties and complexities in the field. However, I am determined to continue (at least I think I am determined). For starters, I would like to get myself a faster video card, buy Quake3, and immerse myself in understanding every scrap of information available on Quake3. The official editor for Quake3, called Q3Radiant, is available, as well as much of the Quake3 source code. Doing this I feel would get me back on track, and then I would have to see where that track leads me.

Bibliography and References

CHRI89: Hoffmann, C.M., *Geometric and Solid Modeling*, Morgan Kaufmann, San Mateo, CA, 1989.

HANA89: Hanan, Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, 1989

FOLE96: Fole, J.D. et. al., *Computer Graphics Principles and Practice*, Addison-Wesley, 1996.

RAPP97: Rappoport, A. and Spitz, S., "Interactive Boolean Operations for Conceptual Design of 3-D Solids", *SIGGRAPH 97*, p269-278.

Appendix A – HSBM Node Insertion/Removal Algorithm

We wish to update the b-rep in the region occupied by a particular face (call it the **newly-inserted face**) belonging to the newly-inserted node. Since the newly-inserted node has no children, we have the advantage of knowing that the polarity of the region on the side of the newly-inserted face is pure (the polarity of the newly-inserted node). However, the polarity on the other side of the newly-inserted face is not pure.

Let's begin by examining the simpler cases. If the newly-inserted face is not coincident with any of the containing node's faces (**containing node** refers to the parent of the newly-inserted node) or any of the containing node's children's faces (excluding of course the newly-inserted node itself, as it is a child of the containing node), then no work is to be done – the b-rep associated with the newly-inserted face will cover the whole newly-inserted face and the existing b-rep won't be changed. However, if the newly-inserted face is coincident with one of the containing node's faces or one (or more) of the containing node's children's faces, then the situation becomes more complex. We must start by finding the coincident faces, then decide how to update the b-rep's associated with each coincident face. For performance's sake, we wish to execute this process as efficiently as possible.

We recognize that if the newly-inserted face is coincident with one of the containing node's faces, then clearly it is NOT coincident with any of the containing node's children's faces. Similarly, if a coincidence is found between the newly-inserted node and a face of one of the containing node's children, then we can be sure that no other faces of that particular child will be coincident, but that doesn't preclude other children's faces from being coincident (unless we were to incorporate some sort of occlusion process whereby the "remaining portion" of the newly-inserted face would be tracked as coincidences are enumerated – this may or may not accelerate the process, depending on the nature of the geometry being created). We observe these things to be true because the nodes are convex. Therefore, we should check for coincidence with the containing node's faces first. If a coincidence is found (call it the **containing face**), then we have a similar situation to the one before: the containing face might be coincident with a face of the parent of the containing node, or if not, then it might be coincident with a face of one or more of the children of the parent of the containing node. Therefore, we again check for coincidence against the parent of the containing node first, and then against its children only if no coincidence is found.

The above process naturally leads to an iterative upward ascension of the tree. As the tree is ascended, we mark each containing face as being coincident. The ascension stops when either we reach the top of the tree (in which case the newly-inserted face is coincident with the outer convex boundary of the entire object) or when we reach a node which strictly contains the newly-inserted face (i.e., the newly-inserted face is not coincident with any of the node's faces). In the latter case, we now must begin a recursive descent of the tree, marking coincident children's faces. Obviously, if the ascension reached the top of the tree, then we cannot make a descent – there are no children.

In the descent process, we test each child node for coincidence with the newly-inserted node. Note that in this stage, we are actually checking **opposite-coincidence**, because the newly-inserted face's plane will separate the child node from the newly-inserted node. Once again, if we find a coincident child face, we can stop checking the faces of that particular child, but we must still check the remaining children. If we find a coincident child face, we then must also check that child's children, and so on. In this way, we descend the tree recursively until no more coincidences are found. Note that we may descending past the depth of the newly-inserted node itself – this is necessary for the full enumeration of coincident faces and the correct updating of the b-rep.

Now let's examine how a face coincidence changes the b-rep. If the newly-inserted face was coincident with one of the containing node's faces, then clearly no portion of the b-rep associated with the containing face can exist in coincidence with the newly-inserted face. So, we **cut** the b-rep by the newly-inserted face. Moving on, we might check the other faces marked during the upwards ascension. But note that their associated b-reps have already been cut by their coincident children, so there is no work to be done on their associated b-reps….

GLOBALS:

```
$F_{\text{aux-1}}$ = face of newly-inserted node (pointer)
$F_{\text{aux-2}}$ = coincident face of parent node (pointer)
```

INSERT_NODE[S, S_{new}, flag = {0: insert, 1: remove}]

```
If $flag = 0$, Append $S_{\text{new}}$ to $S \rightarrow^{\text{sub-nodes}}$
Else,
    Set $S = S_{\text{new}} \rightarrow^{\text{parent}}$ // if not already set
    Unlink $S_{\text{new}}$ from $S \rightarrow^{\text{sub-nodes}}$
For $F_{\text{aux-1}}$ = each face of $S_{\text{new}}$,
    Set $S'$ = **ASCEND_TREE**[$S_{\text{new}}$]
    If $S' = \varnothing$, Continue
    If $flag = 0$,
        **CUT_BREP**[$F_{\text{aux-2}}$, $F_{\text{aux-1}}$]
        Set $F_{\text{aux-2}} = F_{\text{aux-1}}$ // set pointer
    Set $d = S' \rightarrow^{\text{depth}} - S \rightarrow^{\text{depth}} + flag$
    For $S''$ = each sub-node of $S'$,
        **UPDATE_BREP**[$S''$, $\varnothing$, $d$]
If $flag = 1$, Discard $S_{\text{new}}$
```

ASCEND_TREE[*S*]

```
Set $S' = S$
Repeat,
    Set $S' = S' \rightarrow^{\text{parent}}$
    If $S' = \varnothing$ || **COPLANAR_FACE**[$S'$] = $\varnothing$, Break
    If $S' = S \rightarrow^{\text{parent}}$, Set $F_{\text{aux-2}} = F$
Return $S'$
```

COPLANAR_FACE[*S*]

```
For $F$ = each face of $S$,
    If $F$ is coplanar with $F_{\text{aux-1}}$, Return $F$
Return $\varnothing$
```

COPLANAR_OPPOSITE_FACE[*S*]

```
For $F$ = each face of $S$,
    If $F$ is coplanar and opposite to $F_{\text{aux-1}}$, Return $F$
Return $\varnothing$
```

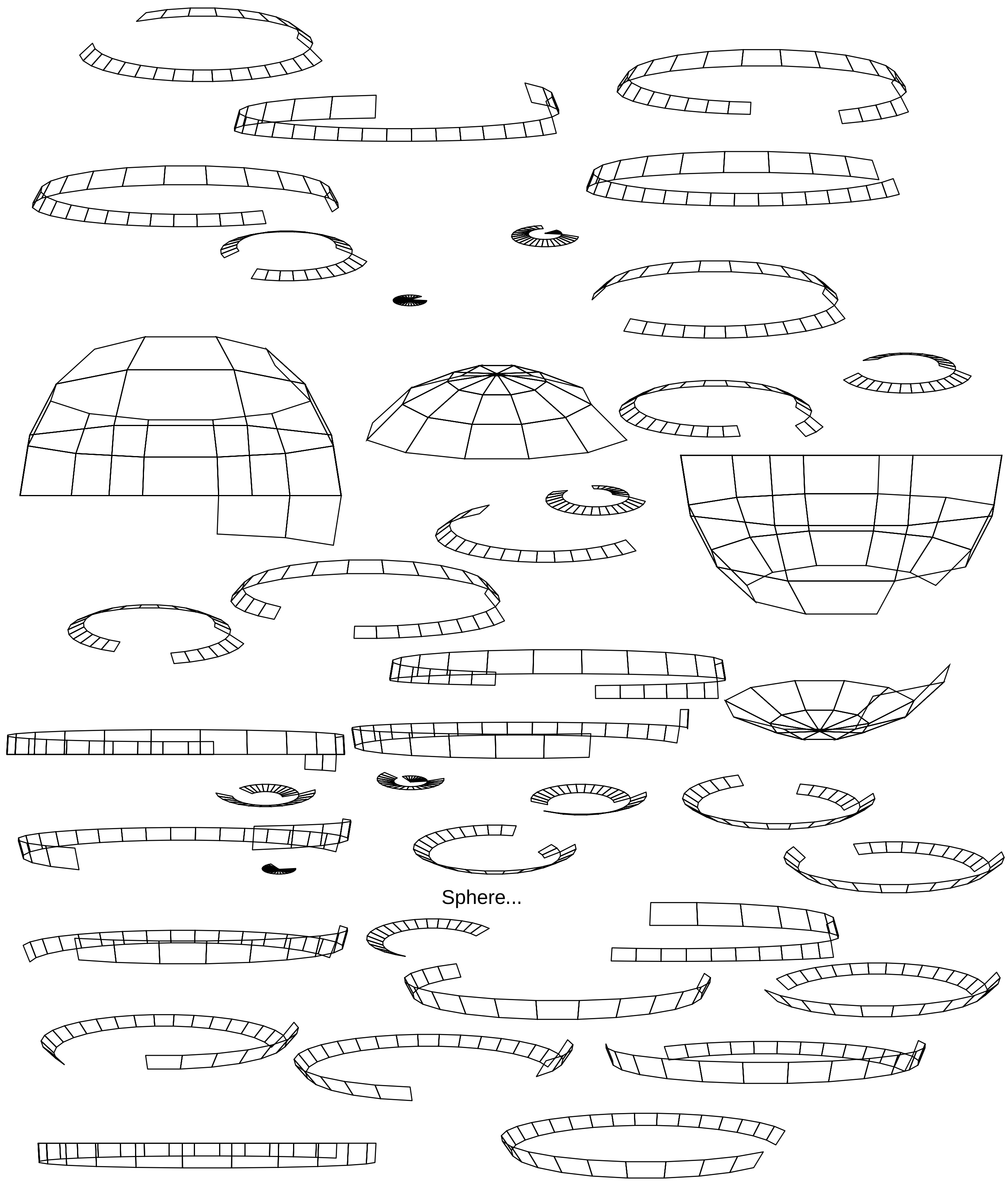
UPDATE_BREP[S, F_b, d]

```
Set $F$ = **COPLANAR_OPPOSITE_FACE**[$S$]
If $F = \varnothing \mid\mid (F_{aux-1} \cap^* F) = \varnothing$, Return
If $F_b \neq \varnothing$, **CUT_BREP**[$F_b$, $F_{aux-1}$]
If $d$ is even,
    Set $F' = F_b' = \varnothing$
    If $d \geq 0$, **CUT_BREP**[$F$, $F_{aux-1}$]
Else,
    Set $F_b' = F_b' \rightarrow^{b-rep} = F_{aux-1} \cap^* F$
    If $d \geq 0$, Set $F' = F$
    Else, Set $F' = F_{aux-2}$
For $S'$ = each sub-node of $S$,
    **UPDATE_BREP**[$S'$, $F_b'$, $d$+1]
Append $F_b' \rightarrow^{b-rep}$ to $F' \rightarrow^{b-rep}$
If $F_b' \neq \varnothing$, Discard $F_b'$
```

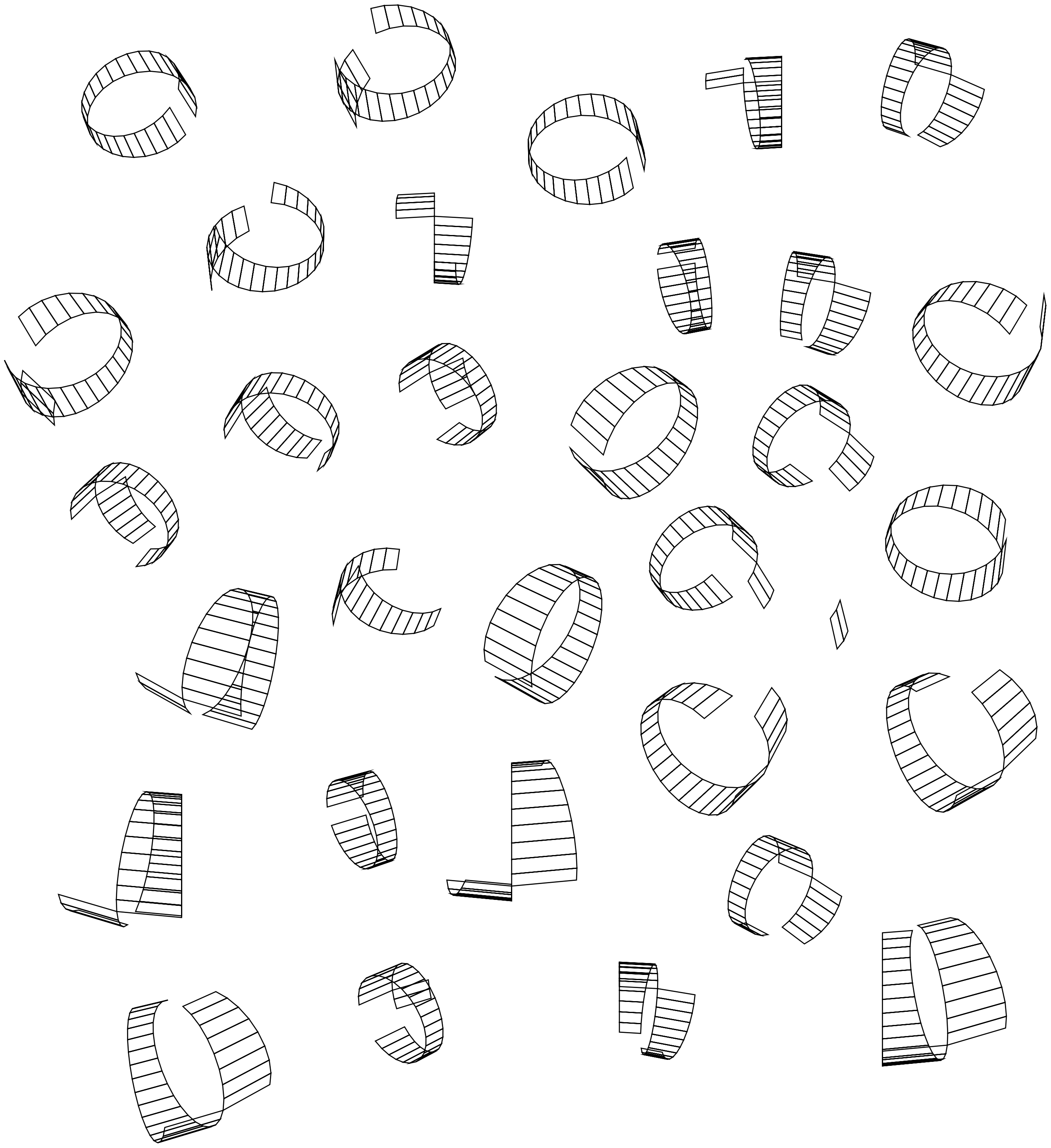
CUT_BREP[F, X]

```
If $X \cap^* F = \varnothing$, Return
If $X \cap^* F = F$, Remove all of $F \rightarrow^{b-rep}$
Else, For each $P \in F \rightarrow^{b-rep}$,
    If $X \cap^* P = \varnothing$, Continue
    If $X \cap^* P = P$, Remove $P$ from $F \rightarrow^{b-rep}$
    Else,
        Unlink $P$ from $F \rightarrow^{b-rep}$ and rename as $P_{in}$
        For $E_x$ = each edge of $X$,
            If $E_x \cap^* P_{in}$,
                Split $P_{in}$ along $E_x = \{P_{out}, P_{in}\}$
                Append $P_{out}$ to $F \rightarrow^{b-rep}$
        Discard $P_{in}$
```

Appendix B – Image Scrapbook



Scrap #1: Sphere tesselation rendered as PostScript (individual quads), converted to PDF via Acrobat Distiller, re-converted to Illustrator native format, and pulled apart.



Scrap #2: Torus tesselation, like Scrap 1