Lecture 20: Solid Modeling

... a cubit on the one side, and a cubit on the other side Exodus 26:13

1. Solids

A *solid* is a 3-dimensional shape with two well-defined sides: an inside and an outside. Thus, for any point in space it is possible, at least in principle, to determine whether the point lies on the inside, or on the outside, or on the boundary of a solid.

Solid modeling allows us to represent more complicated shapes than surface modeling. With solid models, we can compute mass properties such as volume and moments of inertia; we can check for interference and detect collisions; we can also apply finite element analysis to calculate stress and strain for solid models.

Three types of solid modeling techniques are common in Computer Graphics: constructive solid geometry (CSG), boundary representations (B-Rep), and octrees. Constructive solid geometry relies heavily on ray tracing both for rendering and for analysis, so constructive solid geometry is closest to methods with which we are already familiar from surface modeling. Boundary representations often rely on parametric patches such as Bezier patches or B-splines, freeform surfaces which we shall study later in this course. In contrast, octrees are an efficient spatial enumeration technique, a method for approximating smooth 3-dimensional shapes with a simple, compact collection of rectangular boxes. In this lecture we shall give a brief survey of each of these solid modeling methods, and we shall compare and contrast the advantages and disadvantages of each approach.

2. Constructive Solid Geometry (CSG)

In constructive solid geometry, we begin with a small collection of simple primitive solids -- boxes, spheres, cylinders, cones, and tori (see Figure 1) -- and we build up more complicated solids by applying boolean operations -- union, intersection and difference (see Figures 2 and 3) -- to these primitive solids. For example, if we want to drill some holes in a block, then we can model this solid by subtracting a few cylinders from a box (see Figure 3).

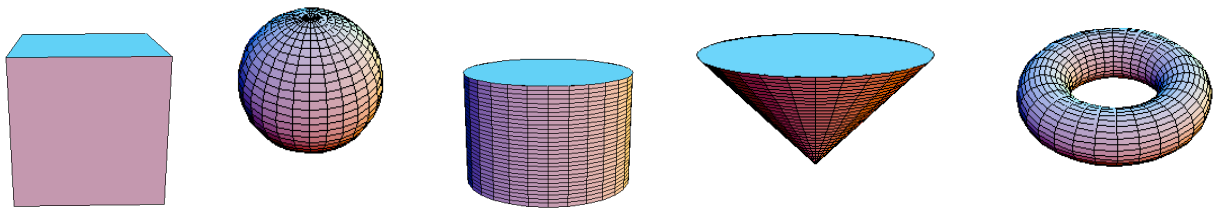

Figure 1: Primitives for a solid modeling system: a box, a sphere, a cylinder, a cone, and a torus.

Solids are stored in binary trees called *CSG trees*. The leaves of a CSG tree store primitive solids; the interior nodes store either boolean operations or nonsingular transformations. Nodes storing boolean operations have two children; nodes storing transformations have only one child. Boolean operations allow us to build up more complicated solids from primitive solids. Transformations allow us to reposition the solid from a canonical location to an arbitrary location or to rescale the solid from a symmetric shape to a more general shape.

Each node of a CSG tree defines a solid. The solid defined by a leaf node is the primitive solid stored in the leaf. The solids at interior nodes are defined recursively. If the interior node stores a transformation, then the solid corresponding to the node is the solid generated by applying the transformation in the node to the solid stored by the node's child. If the interior node stores a boolean operation, then the solid corresponding to the node is the solid generated by applying the boolean operation in the node to the solids stored in the node's two children. The solid corresponding to the root of the tree is the solid represented by the CSG tree (see Figures 2 and 3).

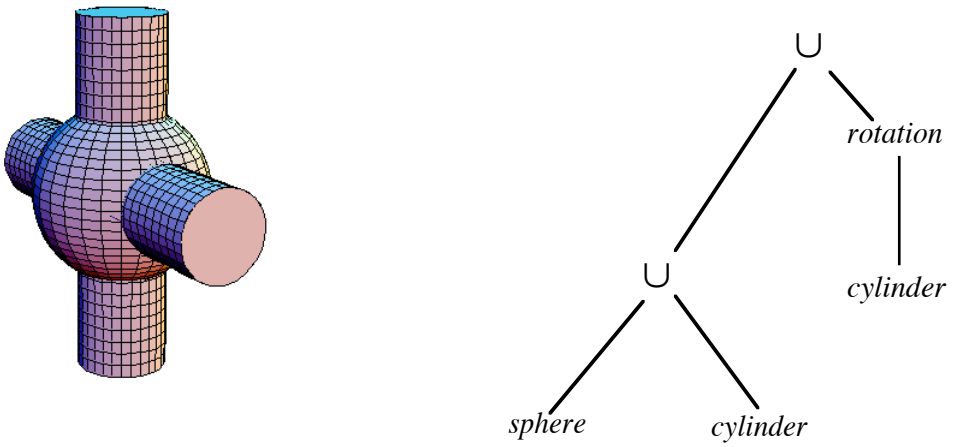

Figure 2: A spherical tank with two cylindrical pipes (left), modeled by a CSG tree consisting of the union of a sphere and two cylinders (right).

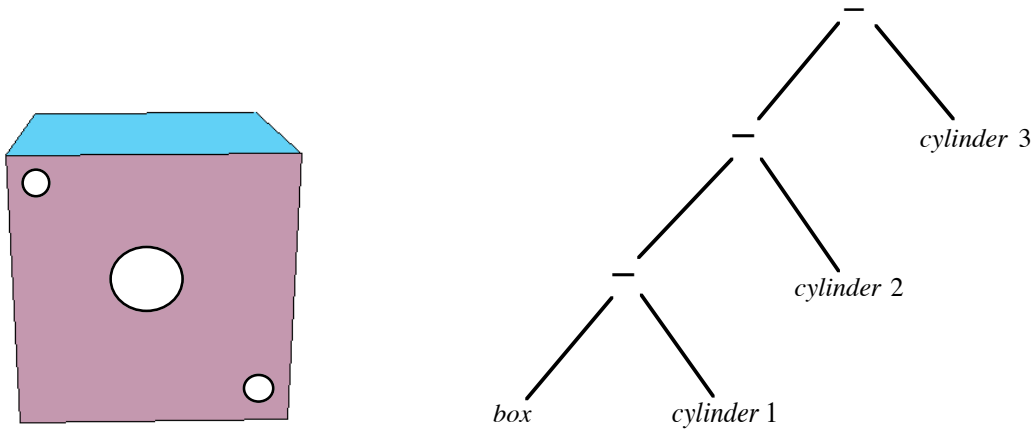

Figure 3: A solid block with three cylindrical holes (left), modeled by a CSG tree that subtracts three cylinders from a box (right).

Ray tracing can be applied to render solids represented by CSG trees. However, unlike ray tracing for surface models where only isolated intersection points are computed, ray tracing for solid models proceeds by finding the parameter intervals for which each line lies inside the solid. We need these parameter intervals for two reasons. First, even if a ray intersects a primitive solid in a leaf node, there is no guarantee that this intersection point lies on the boundary of the solid represented by the CSG tree, since we may in fact be subtracting this primitive from the solid. Keeping track of parameter intervals inside the solid instead of merely the intersection points on the surfaces bounding primitive solids allows us to keep track of which intersection points actually lie on the boundary of the solid. Second, later on we plan to use these parameter intervals to help compute mass properties such as the volume of a solid.

Thus to ray trace a solid, for each line from the eye through a pixel, we first compute the parameter intervals along the line for which the ray lies inside the solid. We then display the point corresponding to the parameter closest to the eye. Finding these parameter intervals for the primitive solids is usually straightforward, since these solids are typically chosen so that they are easy to ray trace. To find these parameter intervals for interior nodes, we must consider two cases. If the node stores a transformation T, then the parameter intervals for the node along the line L are the same as the parameter intervals for the line $T^{-1}(L)$ and node's child; if the node stores a boolean operation, then the parameter intervals can be found by applying the boolean operation at the node to the parameter intervals for the node's two children. Pseudocode for ray tracing a CSG tree is presented below.

Ray Tracing Algorithm for CSG Trees

```
For each pixel,
    Construct a line L from the eye through the pixel.
    If the solid is a primitive, compute all the intersections of the line with the primitive.
    Otherwise if the solid is a CSG tree:
        If the root stores a transformation T,
            Recursively find all the intervals where the line T^{-1}(L) lies inside the root's child.
        Otherwise if the root stores a boolean operation,
            Recursively find all the intervals in which the line intersects the left and right
            subtrees.
            Combine these intervals using the boolean operation at the root.
    Display the closest intersection point.
```

Ray casting can also be applied to compute the volume of solids represented by CSG trees. Instead of firing a ray from the eye through each pixel, fire closely spaced parallel rays at the solid. Each ray represents a solid beam with a small cross sectional area ΔA. If we multiply the length of

each interval I_k inside the solid by ΔA and add the results, then we get a good approximation to the volume of the solid. Thus

$$Volume \approx \sum_k Length(I_k)\Delta A.$$

Alternatively, we can apply Monte Carlo methods to compute the volume of a solid represented by a CSG tree. Monte Carlo methods are stochastic methods based on probabilistic techniques. Enclose the solid inside a rectangular box. If we select points at random, uniformly distributed inside the box, then some of the points will fall inside the solid and some of the points will fall outside the solid. The probability of a point falling inside the solid is equal to the ratio of the volume of the solid to the volume of the box. Thus

$$\frac{\#\,Points\ inside\ Solid}{\#\,Points\ Selected} \approx \frac{Volume(Solid)}{Volume(Box)}$$

so

$$Volume(Solid) \approx \frac{\#\,Points\ inside\ Solid}{\#\,Points\ Selected} \times Volume(Box).$$

Therefore to find the volume of a solid, all we need is a test for determining if a point lies inside or outside the solid.

There are two ways to perform this inside-outside test: by ray casting or by analyzing the CSG tree recursively. To apply ray casting, observe that a point lies inside a solid if and only if an arbitrary ray emanating from the point intersects the solid an odd number of times; otherwise the point lies outside the solid. Now the same interval analysis we applied in the ray casting algorithm for rendering solids represented by CSG trees can be applied to determine the number of times a ray intersects a solid and hence too whether a point lies inside or outside a solid.

However, for CSG trees built from simple primitives, it is easier to determine whether a point lies inside or outside a solid by analyzing the CSG tree recursively. To begin, one can easily determine whether a point lies inside or outside simple primitives like boxes, spheres, cylinders, and cones by using the appropriate distance formulas (see Exercises 1-3). For example, a point P lies inside a sphere if and only if the distance from the point P to the center C of the sphere is less than the radius R of the sphere -- that is, if and only if $|P-C|^2 < R^2$. For a solid such as a torus defined by an implicit equation $F(x,y,z)=0$, if the gradient $\nabla F = (\partial F/\partial x, \partial F/\partial y, \partial F/\partial z)$ represents the outward pointing normal, then a point P lies inside the solid if and only if $F(P) < 0$; if the gradient ∇F represents the inward pointing normal, then a point P lies inside the solid if and only if $F(P) > 0$ (see Exercise 4).

Once we have an inside-outside test for each of the primitives, we can generate an inside-outside test for each CSG tree recursively in the following fashion. If the root stores a

transformation T, then a point P lies inside the corresponding solid if and only if the point $T^{-1}(P)$ lies inside the solid represented by the root's child. If the root stores a boolean operation, then to determine if a point P lies inside the corresponding solid, first perform tests to determine if the point P lies inside the solids represented by each of the root's two children and then apply the boolean operation at the root to the outcome of these two tests. Pseudocode for this inside-outside test on a CSG tree is presented below.

Algorithm for Determining if a Point P Lies Inside or Outside a Solid Represented by a CSG Tree

If the solid is a primitive:

Use the appropriate distance formula or the implicit equation associated with the primitive to determine whether the point P lies on the inside or the outside of the primitive.

Otherwise if the solid is a CSG tree:

If the root stores a transformation T,

Recursively determine if the point $T^{-1}(P)$ lies inside or outside the root's child.

Otherwise if the root stores a boolean operation:

Recursively determine if the point lies inside or outside the root's two children.

There are three cases to consider:

Union

If P lies inside the child at either node, then P lies inside the solid.

Intersection

If P lies inside the children at both nodes, then P lies inside the solid.

Difference $(A - B)$

If P lies inside the child at the left node (A) and outside the child at the right node (B), then P lies inside the solid.

Otherwise P lies outside the solid.

Constructive solid geometry based on CSG trees has many advantages, The representation is compact, the data structure is robust, and the user interface based on boolean operations is simple and natural. Transformation nodes in the CSG tree permit parameterized objects based on canonical positions or canonical shapes. The analysis of CSG trees is also straightforward: ray casting can be used for rendering, calculating volume, and testing whether points lie inside or outside a solid.

Nevertheless, constructive solid geometry also has certain disadvantages. There is no adjacency information in a CSG tree. That is, there is no information about which surfaces are next to a given surface. But adjacency information is crucial in manufacturing, especially for NC machining. Also, in a CSG model there is no direct access to the vertices and edges of the solid. Thus, it is difficult for a designer to select specific parts of an object as referents, nor can a designer

tweak the model by adjusting the location of vertices and edges. Worse yet, it is hard to extract salient features of the solid such as holes and slots which are important for manufacturing. Finally, the CSG representation of a solid is not unique. After subtracting a primitive, there is nothing to prevent us from unioning the primitive back into the solid. Thus it is difficult to determine if two CSG trees represent the same solid. Therefore even though CSG representations are simple, natural, and robust, other types of representations for solids have been developed to overcome some of the shortcomings of constructive solid geometry.

3. Boundary Representations (B-Rep)

A boundary representation for a solid stores the topology as well as the geometry of the solid. Geometry models position, size, and orientation; topology models adjacency and connectivity. Constructive solid geometry models the geometry of a solid, but the topology of a solid is not present in a CSG tree. Therefore we shall concentrate our discussion here on topology, which is the novel component of the boundary file representation.

Topology consists of vertices, edges, and faces along with pointers storing connectivity information describing which of these entities are either on or adjacent to other entities. An edge joins two vertices; a face is surrounded by one or more closed loops of edges. Topological information is binary data: a vertex V is either on the edge E or not on the edge E, a face F_1 is either adjacent to another face F_2 or not adjacent to the face F_2. Topology allows us to answer basic topological queries such as: find all the edges adjacent to the edge E, or list all the faces surrounding the face F. This topological information is what is missing from CSG representations. Topology does not store any information about the position, size, and orientation of the vertices, edges or faces; this numerical data is stored in the geometry.

The geometry of a solid consists of points, curves, and surfaces along with numerical data describing the position, size, and orientation of these entities: for each point, coordinates; for each curve or surface, data for generating implicit or parametric equations. Geometry is not concerned with adjacency or connectivity; this information is stored in the topology. Topology and geometry are tied together by pointers: vertices point to points, edges to curves, faces to surfaces. The data structure that encompasses the geometry, the topology, and the pointers from the topology to the geometry is called a *boundary file representation* (*B-Rep*) of a solid.

The topology of a solid can be quite elaborate. There are nine potential sets of pointers connecting vertices (*V*), edges (*E*), and faces (*F*) (see Figure 4). Maintaining this complete data structure would be complicated and time consuming. Typically fewer pointers are actually stored; tradeoffs are made between the speed with which each topological query can be answered and the amount of space required to store the topology.

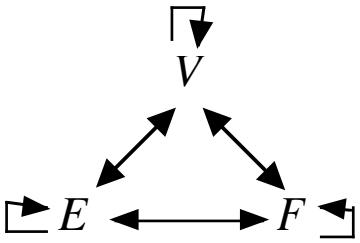

Figure 4: The nine potential sets of pointers for a topological data structure.

The *winged-edge* data structure is a standard representation for topology, which makes a reasonable compromise between time and space. Most of the pointers in a winged-edge data structure are stored with the edges. For solids bounded by 2-dimensional manifolds, each edge lies on two faces, and each edge typically connects two vertices. In the winged-edge data structure, edges are oriented and each edge points to:

2 Vertices -- Previous (*P*) and Next (*N*) (orientation),

2 Faces -- Left (*L*) and Right (*R*),

4 Edges -- P_R,N_R and P_L,N_L.

In addition, each vertex points to one edge that contains the vertex, and each face points to one edge that lies on the face (see Figure 5).

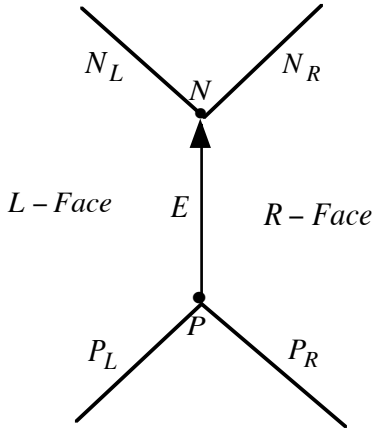

Figure 5: The winged-edge data structure.

With the winged-edge topology, we can answer all questions of the form: Is *A* on or adjacent to *B*. Typical queries are:

Find all the edges surrounding a face.

Find all the faces adjacent to a face.

Find all the vertices lying on a face.

Find all the edges passing through a vertex.

We leave it as a straightforward exercise to the readers to convince themselves that these queries can

all be answered quickly and easily from the data stored in the winged-edge topology (see Exercise 9). The main advantages of the winged-edge topology are that in addition to supporting fast retrieval of topological information, the winged-edge data structure is of moderate size and is relatively easy to maintain during boolean operations.

One problem with the winged-edge topology is that this data structure does not support faces with holes -- that is, faces defined by one exterior loop and one or more interior loops. To solve this problem auxiliary edges are typically added to the model in order to connect the inner loops to the outer loop. These auxiliary edges have the same face on both sides (see Figure 6)!

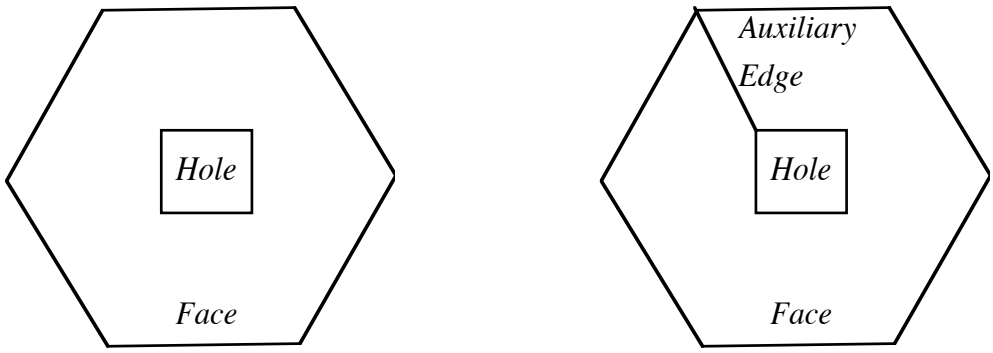

Figure 6: A face with a hole. The face is surrounded by two loops (left). An auxiliary edge connects the two loops (right). Notice that the auxiliary edge has the same face on both sides.

Boundary file representations are more complicated than CSG trees, so it is easy to make a mistake in the topology when updating a model during a boolean operation. One way to check the consistency of the topology in the boundary file is to verify Euler's formula.

Euler's formula for solids bounded by 2-dimensional manifolds asserts that

$$V - E + F - H = 2(C - G) \qquad (3.1)$$

where V = # vertices, E = # edges, F = # faces, H = # holes in faces, C = # connected components, and G = # holes in the solid (genus). For example, for a cube, $V = 8$, $E = 12$, $F = 6$, $H = 0$, $C = 1$, and $G = 0$. Therefore,

$$V - E + F - H = 2 = 2(C - G).$$

If Euler's formula is not satisfied, then there is a error in the model. Moreover, it is unlikely (though possible) that an invalid model will actually satisfy Euler's formula. Therefore Euler's formula provides a robust check for the consistency of the topological model.

Topology plays no essential role in rendering. Therefore, ray tracing for solids represented by boundary files is essentially the same as ray tracing for surface models: simply intersect each ray with each surface on the boundary of the solid and display the closest intersection point.

Volume for solids with boundary file representations can be computed in three different ways: by ray casting, by Monte Carlo methods, and by the divergence theorem. The ray casting method for boundary file representations is similar to the ray casting method for constructive solid geometry. Fire closely spaced parallel rays at the solid, but instead of using the CSG tree use the boundary file representation to determine the intervals I_k along each line that lies inside the solid by intersecting each line with the surfaces that bound the solid. Each ray represents a solid beam with a small cross sectional area ΔA, so if we multiply the length of each interval I_k inside the solid by ΔA and add the results, then we get a good approximation to the volume of the solid.

Alternatively, we can apply Monte Carlo methods to find the volume of a solid by enclosing the solid in a rectangular box. If we select points at random, uniformly distributed inside the box, then the probability of a point falling inside the solid is equal to the ratio of the volume of the solid to the volume of the box. We can use ray casting at a boundary file representation to determined if a point lies inside a solid: a ray emanating from a point lies inside a solid if and only if the ray intersects the surfaces bounding the solid an odd number of times.

The divergence theorem provides a novel technique for computing the volume of a solid with a boundary file representation which is not available for solids represented by CSG trees. If the solid is bounded by the surfaces $S = \{S_k\}$ with outward pointing normals $N = \{N_k\}$, then by the divergence theorem

$$Volume = \frac{1}{3} \oiint_S (P \bullet N)\, dS = \frac{1}{3} \sum_k \iint_{S_k} (P \bullet N_k)\, dS_k \,, \qquad (3.2)$$

where $P = (x, y, z)$ represents an arbitrary point on the bounding surfaces. If the surfaces bounding the solid are sufficiently simple, then we can compute these integrals analytically (see Exercise 10); otherwise we can use quadrature methods to approximate these integrals.

Boolean operations, unlike rendering or volume computations, are much more difficult for boundary file representations than for solids represented by CSG trees because for boundary file representations not only the geometry but also the topology must be updated. Unlike constructive solid geometry where boolean operations are performed simply by updating a CSG tree, in a boundary file representation the curves and surfaces of the first solid must be intersected explicitly with the curves and surfaces of the second solid to form the new vertices, edges, and faces for the topology of the combined solid. These intersection computations are much more difficult than ray-surface intersections, so boolean operations on boundary file representations are much more difficult than boolean operations on CSG representations. Pseudocode for boolean operations on boundary file representations is presented below.

Boolean Operations on Boundary File Representations

Input: Boundary file representations of two solids *A,B*.

Algorithm

Intersect each surface (face) of A with each surface (face) of B to form new curves (edges) on A and B. {Difficult Computations}

Intersect each new curve with existing edges on the old faces to form new vertices and edges on A and B.

Insert the new faces, edges, and vertices into the topology of A and B.

{Update Topological Data Structures}

Combine the boundary topologies based on the particular boolean operation:

$$\partial(A \cup B) = (\partial A - \partial A_{inB}) \cup (\partial B - \partial B_{inA})$$

$$\partial(A \cap B) = \partial A_{inB} \cup \partial B_{inA}$$

$$\partial(A - B) = (\partial A - \partial A_{inB}) \cup \partial B_{inA}$$

$$\partial(B - A) = (\partial A_{inB}) \cup (\partial B - \partial B_{inA})$$

While this pseudocode may seem straightforward, the actual computations can be quite difficult. Computing accurate surface-surface intersections is hard; maintaining correct boundary files is time consuming as well as susceptible to numerical errors because to speed up the computations the intersections are typically calculated using floating point arithmetic. Many man years are generally required to code robust boolean operations on boundary file representations.

The main advantage of boundary file representations is that adjacency information is readily available. This adjacency information is important for manufacturing operations such as NC machining. Adjacency information also permits the designer to modify the model by tweaking vertices and edges, and to extract important features for manufacturing such as holes and slots. Moreover, unlike CSG representations, the boundary file representation is unique. Thus it is possible to decide whether two solids are identical by comparing their boundary file representations. Finally, we can verify the correctness of the topological model by checking Euler's formula.

The main disadvantage of the boundary file representation is that maintaining the topology requires maintaining a large and complicated data structure. Therefore boolean operations on boundary file representations are slow and cumbersome to compute. Moreover, floating point inaccuracies can cause disagreements between geometry and topology; tangencies are particularly hard to handle. Hence, it is difficult to maintain robust boundary file representations for solids bounded by curved surfaces.

Thus even though boundary file representations facilitate the solution of several important problems in design and manufacturing, boundary files also introduce additional problems of their own. Therefore, next we shall investigate yet another, much simpler type of representation for solids that has some advantages over boundary file representations.

4. Octrees

Octrees are an efficient spatial enumeration technique. In naive spatial enumeration, space is subdivided into a large collection of small disjoint cubes all of the same size and orientation -- usually with faces parallel to the coordinate planes -- and each solid is defined by a list of those cubes lying inside or on the boundary of the solid. The accuracy of spatial enumeration for solids bounded by curved surfaces depends on the size of the cubes used in the spatial enumeration, so there are tradeoffs between the accuracy of the model and the space necessary to store the model.

To save on storage, octrees vary the size of the cubes so that lots of small cubes get coalesced into larger cubes. Large cubes are used to model the interior of the solid, while small cubes are used to model the boundary of the solid. The cubes in an octree are called *voxels*. In an octree representation of a solid the number of voxels is typically proportional to the surface area; most of the subdivision of large cubes occurs to capture the boundary of the solid. Octrees to model solids are generated by the following divide and conquer algorithm. Figure 7 illustrates a 2-dimensional analogue of an octree, a quadtree for the area enclosed by a planar curve.

Octree Algorithm

Start with a large cube containing the solid. The faces of the large cube should be oriented parallel to the coordinate planes.

Divide this large cube into 8 octants.

Label the cube in each octant as either *E* = empty, *F* = full, or *P* = partially full, depending on whether the octant is outside, inside, or overlaps the boundary of the solid.

Recursively subdivide the *P* nodes until their descendents are labeled either *E,* or *F*, or a lowest level of resolution (cutoff depth) is reached.

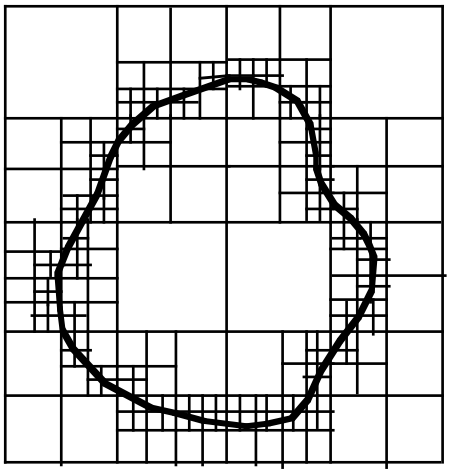

Figure 7: A quadtree model for the area enclosed by a planar curve. Notice that most of the subdivision of the large squares into smaller squares occurs along the bounding curve.

Since an octree is just a collection of different size cubes, many algorithms for analyzing solids represented by octrees reduce to simple algorithms for analyzing cubes. For example, to render an octree by ray tracing, simply intersect each ray with all the *F* voxels and accept the nearest hit. To find the volume of an octree, simply sum the volume of the *F* voxels. Simple pseudocode for boolean operations on octrees is presented below.

Boolean Operations for Octree Representations

Input

Octree representations of two solids *A,B*.

Algorithm

Apply the boolean operation to corresponding nodes in the octree representations for *A,B*.

There are three cases to consider:

Union

If either node is labeled *F*, label the result with *F*.

Otherwise if both nodes are labeled *E*, label the result with *E*.

Otherwise, label the new node as *P* and recursively inspect the children.

Intersection

If either node is labeled *E*, label the result with *E*.

Otherwise if both nodes are labeled *F*, label the result with *F*.

Otherwise, label the node as *P* and recursively inspect the children.

Difference ($A - B$)

If the *A*-node is labeled *E* or the B-node is labeled *F*, label the result with *E*.

Otherwise, if the *A*-node is label *F* and the *B*-node is labeled *E*, label the result with *F*.

Otherwise, label the node as *P* and recursively inspect the children.

If all the children are labeled *E* (*F*), then change the label *P* to the label *E* (*F*).

The main advantages of octree representations are that they are easy to generate and they are much more compact than naive spatial enumeration. Moreover, octree representations can achieve any desired accuracy simply by allowing smaller and smaller voxels. Many analysis algorithms for octrees, such as ray tracing, volume computations, and boolean operations, reduce to simple algorithms for analyzing cubes. Scaling along the coordinate axes and orthogonal rotations around coordinate axes are also easy to perform simply by scaling and rotating the voxels. Finally, octrees facilitate fast mesh generation for solids.

The main disadvantage of octree representations is that the voxels are usually aligned with the coordinate axes. Thus most octree models are coordinate dependent. Therefore, it is difficult to perform an arbitrary affine transformation on an octree representation of a solid. Also, when highly accurate models are required, octree representations will use lots of storage to model curved boundaries, so for highly accurate models, octree representations can be extremely inefficient.

5. Summary

There are three standard techniques in Computer Graphics for modeling solids: CSG trees, boundary files (B-Rep), and octrees. Each of these representations have certain advantages and disadvantages. For easy reference, in Table 1 we compare and contrast six of the main properties of these three representations for solids, including (*i*) the accuracy of the model, (*ii*) the size of the domain of the solids that are readily represented by the model, (*iii*) the uniqueness of the model, (*iv*) the ease of verifying the validity of the model, (*v*) whether or not the model is closed under boolean operations, and finally (*vi*) the compactness of the model.

	CSG	B–REP	Octree
Accuracy	Good	Good	Mediocre
Domain	Large	Small	Large
Uniqueness	No	Yes	Yes
Validity	Easy	Hard	Easy
Closure	Yes	No	Yes
Compactness	Good	Bad	Mediocre

Table 1: Comparison of different solid modeling methods.

Exercises:

1. Consider a box with a corner at Q, edges parallel to the unit vectors a,b,c, with length, width, and height given by the scalars l,w,h (see Figure 8). Show that a point P lies inside the box if and only if the following three conditions are satisfied:

   i. $0 < (P-Q) \bullet a < l$

   ii. $0 < (P-Q) \bullet b < w$

   iii. $0 < (P-Q) \bullet c < h$

Interpret these formulas geometrically.

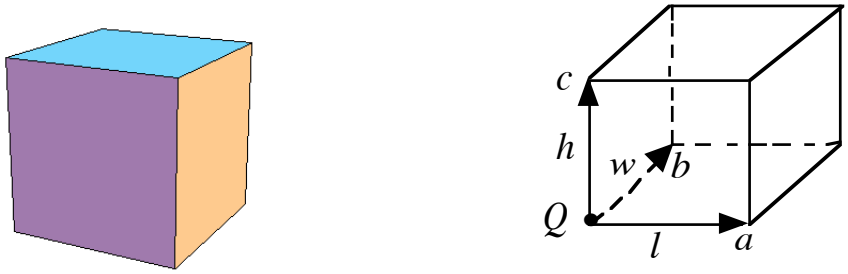

Figure 8: A box with a corner corner at Q, edges parallel to the unit vectors a,b,c, with length, width, and height given by the scalars l,w,h.

2. Consider a bounded cylinder of radius R and height H with an axis line L determined by a base point Q and a unit direction vector A (see Figure 9). Show that a point P lies inside the cylinder if and only if the following two conditions are satisfied:

i. $(P-Q)\bullet(P-Q)-\big((P-Q)\bullet A\big)^2 < R^2$

ii. $\left|\left(P-(Q+\frac{H}{2}A)\right)\bullet A\right| < \frac{H}{2}$

Interpret these formulas geometrically.

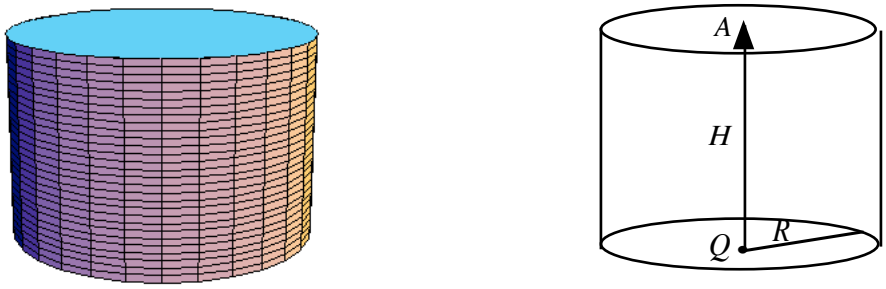

Figure 9: A bounded cylinder of radius R and height H with an axis line L determined by a base point Q and a unit direction vector A.

3. Consider a bounded cone with vertex V, cone angle α, height H, and an axis line L parallel to a unit direction vector A (see Figure 10). Show that a point P lies inside the cone if and only if the following two conditions are satisfied:

i. $\big((P-V)\bullet A\big)^2 > |P-V|^2\cos^2(\alpha)$

ii. $0 < (P-V)\bullet A < H$

Interpret these formulas geometrically.

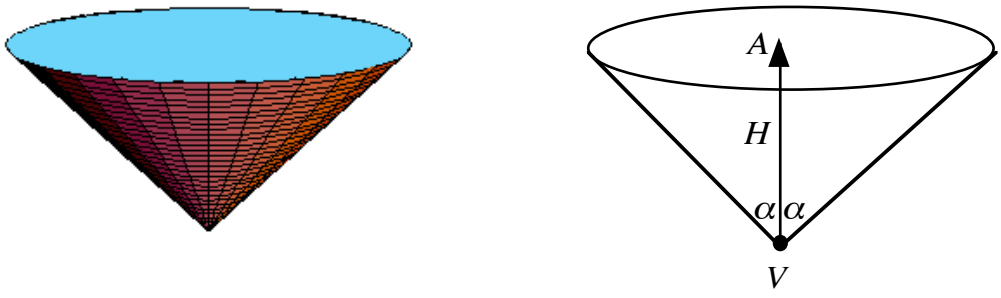

Figure 10: A bounded cone with vertex V, cone angle α and height H, with an axis line parallel to the unit direction vector A.

4. Consider the torus with center at the origin and axis parallel to the z-axis, with a generator of radius d and a tube radius a. Recall from Lecture 19, Section 6 that this torus is described by the implicit equation:

$$F(x,y,z) \equiv \left(x^2 + y^2 + z^2 - d^2 - a^2\right)^2 + 4d^2z^2 - 4a^2d^2 = 0\,.$$

Show that:

a. $\nabla F = \left(\dfrac{\partial F}{\partial x}, \dfrac{\partial F}{\partial y}, \dfrac{\partial F}{\partial z}\right) = \left(4xG(x,y,z), 4yG(x,y,z), 4z(G(x,y,z) + 2d^2)\right)$ is the outward pointing normal, where $G(x,y,z) = x^2 + y^2 + z^2 - d^2 - a^2$.

b. A point $P = (x,y,z)$ lies inside this torus if and only if $F(P) < 0$.

5. Consider a torus with center C and axis parallel to the unit vector A, with a generator of radius d and a tube radius a (see Figure 11). Show that a point P lies inside this torus if and only if

$$\left((P - C) \bullet A\right)^2 + \left(\sqrt{|P - C|^2 - \left((P - C) \bullet A\right)^2} - d\right)^2 < a^2.$$

Interpret this result geometrically.

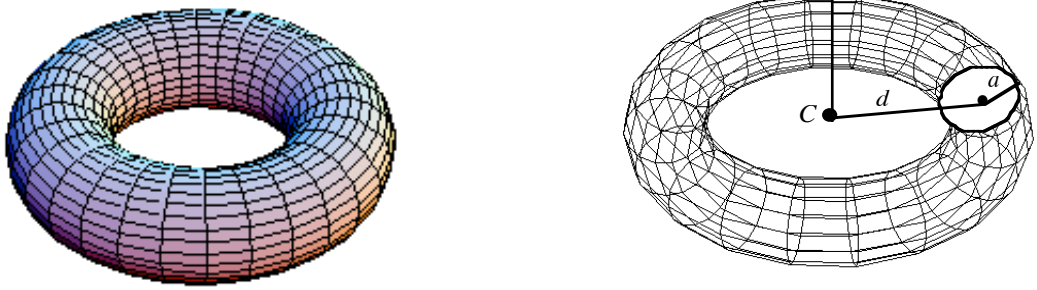

Figure 11: A torus with center C and axis parallel to the unit vector A, with a generator of radius d and a tube radius a.

6. Verify Euler's formula for the cube, the tetrahedron, and the octahedron.

7. Verify Euler's formula for a cube with a rectangular hole passing through the top face and ending at the center of the cube.

8. Verify Euler's formula for a cube with:
   a. A rectangular hole from top to bottom.
   b. Two rectangular holes: one from top to bottom and one from front to back.
   c, Three rectangular holes: top to bottom, front to back, and side to side.

9. Develop algorithms based on the winged-edge topology to answer the following queries:
   a. Find all the edges surrounding a face.
   b. Find all the faces adjacent to a face.
   c. Find all the vertices on a face.
   d. Find all the edges passing through a vertex.

10. Let T be a polyhedron bounded by planar polygonal faces $\{S_k\}$. Let Q_k be a fixed point on S_k and let N_k be the outward pointing unit vector normal to S_k. Using the divergence theorem (Equation (3.2)), show that

$$Vol(T) = \frac{1}{3}\sum_k (Q_k \bullet N_k) Area(S_k).$$

11. Develop an efficient algorithm to determine if a given point lies inside or outside of a solid represented by an octree.

Programming Projects:

1. *Constructive Solid Geometry*
   Implement a solid modeler based on constructive solid geometry in your favorite programming language using your favorite API.
   a. Include the following primitive solids:
      - Boxes and Wedges
      - Spheres, Cylinders, and Cones
      - Tori
   b. Incorporate the following boolean operations:
      - Union
      - Intersection
      - Difference
   c. Develop algorithms to render these solids.
   d. Develop algorithms to compute the volume of these solids.

2. *Octrees*
   Implement a solid modeler based on octrees in your favorite programming language using your favorite API.
   a. Incorporate the following boolean operations:
      - Union
      - Intersection
      - Difference
   b. Develop algorithms to render these solids.
   c. Develop algorithms to compute the volume of these solids.