Lecture 17: Solid Modeling

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

Who is on the LORD's side? Exodus 32:26

1. Solid Representations

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 the solid.

Solid modeling allows us to represent more complicated shapes than surface modeling. With solid models, we can check for interference and detect collisions; we can also compute mass properties such as volume and moments of inertia. Finite element analysis to calculate stress and strain can also be applied to 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 for rendering and analysis, so constructive solid geometry is closest to methods with which we are already familiar. Boundary representations often rely on general parametric patches such as Bezier patches or B-splines, surfaces which we plan to study later in this course. In contrast, octrees are an efficient spatial enumeration technique, a method for efficiently decomposing 3-dimensional smooth shapes into a collection 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, wedges, spheres, cylinders, cones, tori -- and we build up more complicated solids by applying boolean operations -- union, intersection and difference -- to these primitive solids. For example, if we want to drill a hole in a block, then we can model this solid by subtracting a cylinder from a box.

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 simpler solids.

Transformation allow us to reposition the location of a solid from a canonical location to an arbitrary location or to rescale the size from a canonical 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 to the solid stored by the child of this node. If the interior node stores a boolean operation, then the solid corresponding to the node is the solid generated by applying the boolean operation to the solids stored in the node's two children. The solid at the root of the tree is the solid represented by the CSG tree.

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 the line lies inside the solid. We need these parameter intervals for two reasons. First, even if a ray intersects a primitive solid, 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 intersection points on primitives 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 $L * T^{-1}$ 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 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.
        Display the closest intersection.
```

```
Otherwise if the solid is a CSG tree
    If the root stores a transformation $T$
        Recursively find all the intervals where the line $L * T^{-1}$ 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.
```

Ray tracing 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 a solid inside a box. If we select points at random 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

$$\frac{\#\,Points\ inside\ Solid}{\#\,Points\ selected} \approx \frac{Vol(Solid)}{Vol(Box)}$$

so

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

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; we can do this test by ray casting.

A point lies inside a solid if and only if an arbitrary ray through 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 here to determine the number of times a ray intersects a 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 quite 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 is a CSG tree. That is, there is no information about which surfaces are next to which surfaces. But adjacency information is important 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. Worse yet, it is hard to extract salient features of the solid such as holes and slots which are important for manufacturing. Nor can a designer tweak the model by adjusting the location of vertices and edges. 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 actually represent the same solid. Thus 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 CSG representations.

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 connectivity information. Topological data is not present in constructive solid geometry, so we shall concentrate our discussion here on the 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 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 by the geometry.

The geometry of a solid consists of points, curves, and surfaces along with floating point data describing the position, size, and orientation of these entities: for each point, coordinates; for each curve or surface, 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, edges, and faces (see Figure 1). 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.

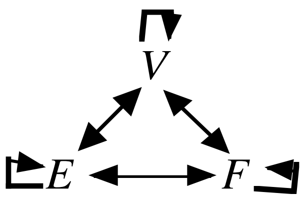

Figure 1: The nine potential sets of pointers for a topological data structure.

The winged-edge data structure is a standard approach to 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-manifolds, each edge lies on two faces, and each edge typically connects two vertices. In the winged-edge data structure, edge 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 2).

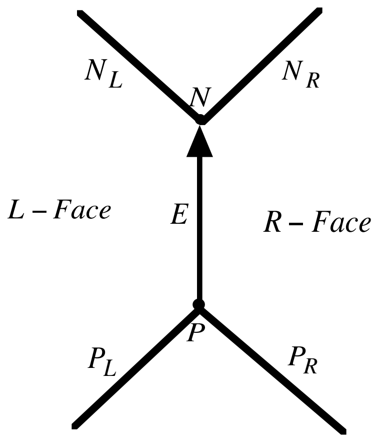

Figure 2: Winged-edge data structure.

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

Find all edges surrounding a face.
Find all faces adjacent to a face.
Find all vertices on a face.
Find all edges on 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 4). 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.

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 outer loop. These auxiliary edges have same face on both sides!

Boundary file representations are more complicated than CSG trees, so it is easy to make a mistake when updating a model. 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-manifolds asserts that

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

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.

Boolean operations, however, are much more difficult for boundary file representations than for CSG representations because the topology must be updated. Unlike CSG representation, when boolean operations are performed on boundary file representations of solids, the curves and surfaces of the first solid must be intersected with the curves and surfaces of the second solid to form the new vertices, edges, and faces for 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.

<u>Boolean Operations on Boundary File Representations</u>

```
*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 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 are generally 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 models are identical by comparing their boundary file representations. Finally, we can verify the correctness of the 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 difficult to handle. Hence, it is hard to maintain robust models 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 -- that is, 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 cube. Large cubes are used to model the interior of the solid, while small cubes are used to model the boundary of the solid. Octrees to model particular solids are generated by the following divide and conquer algorithm.

Octree Algorithm

```
Start with a large cube whose faces are oriented parallel to the coordinate planes.
Divide this large cube into 8 octants.
    Label each octant cube 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 descendent are labeled either $E$ or $F$, or a
        lowest level of resolution (cutoff depth) is reached.
```

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 occurs to capture the boundary of the solid.

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. Boolean operations such as union, intersection, and difference are also easy to perform on octrees. 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 must be aligned with the coordinate axes. Thus octree models are coordinate dependent. Therefore, it is difficult to perform arbitrary affine transformation on octree representations. 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 are not efficient.

5. Summary

For easy reference, in Table 1 we comp;are and contrast some of the main properties of CSG, B-Rep, and octree representations for solids.

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

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

2. 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.

4. 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.