CS348a: Computer Graphics Geometric Modeling Stanford University Handout #31 Original Handout #25 Tuesday, 8 December 1992

Original Lecture #15: 19 November 1992

Topics: Manipulations with Quad-Edge Data Structures

Scribe: Mike Lenz and Michael Goldwasser

In the previous lecture we introduced the *quad-edge data structure*, in which each edge of a surface is represented by an edge record. The record contains pointers to the four neighboring edges *e Onext*, *e Lnext*, *e Dnext*, and *e Rnext*. Vertices and faces are represented by circular, linked lists of edges.

## 1 Quad-Edge Construction Operators

We introduce two operators for creating edge records and combining them to form representations of complex shapes. Both operators have the property that they always return valid quad-edge structures when given valid inputs. A valid quad-edge structure is one in which the axiom  $eRot\ Onext\ Rot\ Onext = e$  holds for each edge.

The *MakeEdge* operator creates an edge record; it takes no arguments. When a new edge is created, each of its four pointers must point to some edge, for the structure to be valid; since there is no other edge available, they must in fact point back at the newly-created edge. Figure 1 shows the two possible arrangements of pointers that arise in valid quad-edge data structures with a single edge. The subdivision that each arrangement represents is shown in Figure 2.

In the left-hand pictures, we have two vertices connected by a primal edge, and a single face. In the right-hand pictures, a single vertex is connected to itself by a primal edge that is a loop, separating the plane into two faces. Each diagram is the dual of the other. By convention, the first diagram represents the result of a call to *MakeEdge*: a topological sphere containing two points with an edge between them.

The second operator, Splice, is used to connect quad-edge structures together, to split them apart, and to rearrange them in other ways. Remember that vertices and faces are represented by circular lists which must be maintained for the structure to be valid. A group of circular lists has the property that, if you swap the outgoing pointers of two nodes, the result is again a group of circular lists. In particular, performing Swap[a,b] on nodes a and b gives the following result. If the nodes were in the same list, that list is split into two circular lists, one containing a and the other containing b. If the nodes were in different lists, those two lists are combined into a single circular list. In either case, a second swap restores the original structure.

The requirement for applying Splice[a,b] is that edges a and b must either both be primal, or both be dual. The algorithm for a call to Splice is as follows:

- 1.  $\alpha := a Onext Rot$
- 2.  $\beta := b \, Onext \, Rot$

2 *CS348a: Handout #31* 

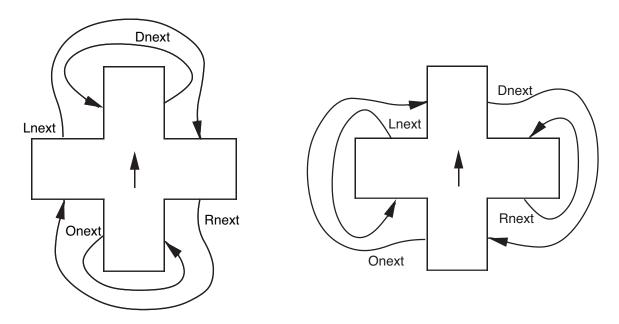


Figure 1: Two possible results of a call to MakeEdge

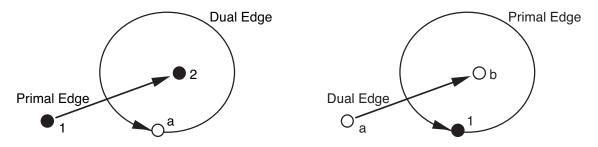


Figure 2: Geometric interpretations of each edge structure

- 3. Swap[a Onext, b Onext]
- 4.  $Swap[\alpha Onext, \beta Onext]$

The first exchange swaps the next edges in the origin rings of *a* and *b*. The second swap has a similar effect on the edges' left-face rings. To see what this does, consider the four possible cases.

**Case 1.** Edges a and b share the same origin but have different left faces. After splicing, the origin is split into two vertices, one now the origin of a and the other the origin of b. The edges' two left faces, on the other hand, are merged into a single face. See Figure 3.

Case 2. Edges a and b share the same left face but have different origins. The two origins are merged into a single vertex, but the face is split into two separate faces. This is also shown in Figure 3, but reading from right to left.

CS348a: Handout #31 3

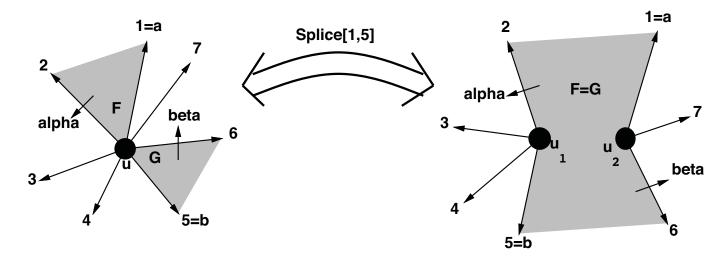


Figure 3: Effects of Splice[a,b] on single origin ring

Case 3. Edges a and b are in different origin rings and different left-face rings. After splicing, they share the same origin as well as the same left face. See Figure 4, in which the shape on the bottom is something like a barbell or a telephone receiver. The topological effect of this type of splice depends upon whether the edges a and b started out in the same connected component or not. If they started out in separate components, doing this splice pastes those two components together into a single component. If they started out in the same component, doing this splice adds a *handle* to that component, hence increasing the number of holes that component has. For example, if a and b both started out belonging to the same sphere-like component, that component will be torus-like after the splice.

**Case 4.** Edges a and b share the same origin and the same left face. After splicing, both their origins and their left faces are distinct. This is also shown in Figure 4, but reading from bottom to top. Topologically, this type of splice can either split one connected component into two or else reduce by one the number of handles that some component has.

## 2 Topology Review

The *Euler Characteristic*,  $\chi$ , of a surface is the number of vertices, minus the number of edges, plus the number of faces. It is also twice the difference between the number of connected components and the total genus of the surface.

$$\chi = V - E + F 
= 2(c - g)$$

Consider the result of a call to MakeEdge, a topological sphere with two vertices, one edge, and one face. Thus  $\chi$  is 2, which is consistent with 1 connected component of genus 0. Each

4 *CS348a: Handout #31* 

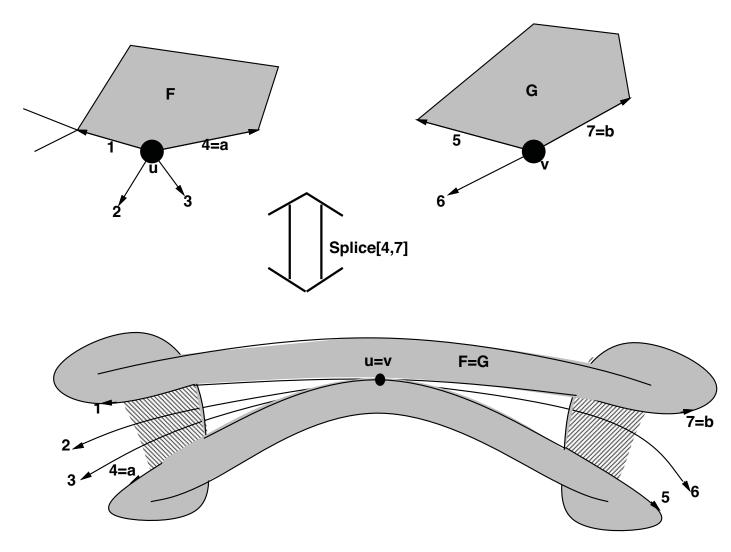


Figure 4: Effects of Splice[a,b] on separate rings

call to *MakeEdge* increases the total Euler Characteristic by 2, in effect adding a connected component (a new sphere) to the surface.

In the case of a torus, we may have a pair of edges from a single vertex that still partition the surface into only one face. In this case  $\chi$  is 0, since the genus is 1, indicating a surface containing one "hole" or "handle." Note that orientable surfaces have even, but possibly negative, values of  $\chi$ ; non-orientable surfaces have odd values of  $\chi$ .

For a call to *Splice*, case 1 increments the number V of vertices and decrements the number F of faces; case 2 does the opposite. In either case, the Euler Characteristic is unchanged. In case 3, both V and F are decremented, so  $\chi$  is decreased by 2. There are two ways this can happen: either two connected components are combined into one, or the genus is increased by one by adding a handle to the surface. In case 4, both V and F are incremented, so the Euler Characteristic is increased by 2. There are two ways that this can happen also: either

CS348a: Handout #31 5

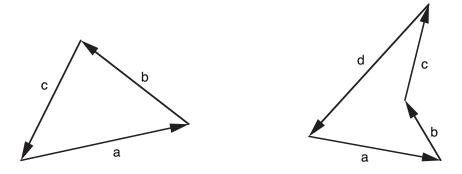


Figure 5: Two simple polygons with directed edges

one connected component is split into two, or the genus is decreased by one by removing one handle.

## **3** Polygons in the Plane

We turn to the problem of finding the interior of a simple (not self-intersecting) polygon in the plane. For each directed edge e, let us designate the half-plane on its left side by e. Consider the first polygon in Figure 5, made up of three convex corners (that is, at each corner, the polygon makes a counterclockwise turn). Its interior is abc, where multiplication represents logical "and", that is, intersection. The second polygon has a concave (clockwise) corner, and its interior is a(b+c)d, where addition stands for logical "or", that is, union.

Suppose we want to find a general formula for the interior of a simple polygon. The criteria are that we use each literal only once, the various sub-expressions being combined by either addition or multiplication. The tricky part is determining how to parenthesize the resulting expression.

To do this, we recursively build a formula by splitting the boundary of the polygon into smaller and smaller polygonal chains. When combining the formulas for two smaller chains to get a formula for a larger chain, we use either "and" or "or" as the operator, depending upon whether the joint between the two subchains is convex or concave. For more about this, see Handout 33.

Homework 4 deals with a different formula for the interior of a simple polygon. That formula is more complicated in that it uses complement, intersection, and exclusive-or as its operators, instead of the two operators intersection and union. But it is simpler in that there are no tricky issues about parentheses.

## 4 How does the surface give you a solid?

In discussing the surfaced-based taxonomy of solid-modeling, we have left out an important question. Given a representation of a surface, how do we decide what points are in the solid

6 CS348a: Handout #31

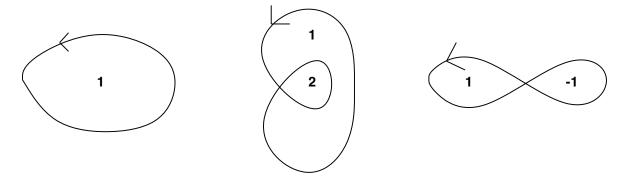


Figure 6: Winding numbers

"represented" by the surface. We have an intuitive notion of the solid being the "interior" of the surface, and this notion is formalized in two-dimensional topology by the Jordan Curve Theorem, which states that every closed boundary divides the space into 3 pieces, the unbounded exterior, the boundary itself, and the bounded interior. It is this interior, that we want to use as the solid defined by the boundary.

For simplicity, we consider this problem in a lower dimension. The concepts we discuss are easily generalized. Thus the question is, given a closed curve, C, in the plane, and a point p, how do we know if the point is "inside" the curve?

We introduce the concept of the winding number of point p with respect to curve C, denoted by  $\omega(p,C)$ . An intuitive notion of winding number is as follows: A viewer standing at point p, watches a car trace the curve from some arbitrary starting point until it reaches that point again. The winding number of p with respect to the curve is the number of full revolutions the viewer has made at the end of the process. For instance, let C be a circle. If you are at a point inside the circle, and watch the car around, you will end up making one full turn, and so your winding number will be 1 or -1 depending on the orientation of the circle. However if you are at a point outside of the circle, the winding number is 0. You might turn to the right or left as you watch, but at the end of the process, you have not made any full rotations from when you started. Figure 6 shows a few curves, and the winding number of the regions created by the curves.

Although we have defined this notion of winding number, we still have not answered the question of how to define the "interior" of the curve. There are two separate conventions for this.

- (p is inside C) if  $(\omega(p, C) \neq 0)$
- $(p \text{ is inside } C) \text{ if } (\omega(p, C) \text{ is odd})$

Notice that in the first and third curves of Figure 6, these conventions agree completely, however in the second curve, the region with winding number 2 is treated differently. The classification of this region is determined by the convention chosen.

Now that we have a use for the winding number, given a point and a curve, how do we efficiently compute the value. One school of thought is to divide up the curve into small

CS348a: Handout #31 7

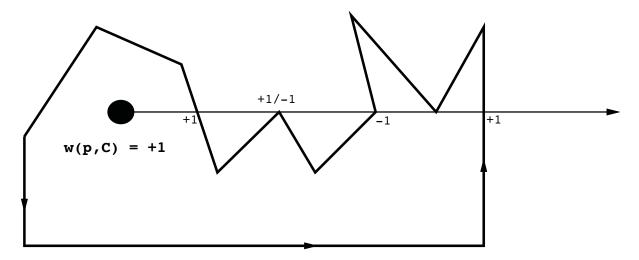


Figure 7: Ray Shooting to calculate winding numbers

pieces, and then for each piece, actually calculate the difference in angles between p and the two endpoints. The winding number is calculated by summing these angles for all the pieces. (To use this method, the pieces must be small enough so that no single piece winds completely around the point)

A second, and more popular, method of computing the winding number is by shooting out a half-infinite ray from the point, and then carefully counting the crossings between that ray and the curve. By counting the crossings in an *oriented* way, the result is the winding number. Whenever the ray crosses the curve, we add or subtract one from the value, with the sign based on whether we cross the curve from left-to-right, or from right-to-left.

As with most geometric algorithms, degeneracies cause some concern. We must be careful in cases where our ray crosses a curve at the intersection of two segments. If our ray hits the endpoint between two segments, does the ray intersect both segments? Does it intersect either segment? Fortunately, there is a solution to this problem. (Unlike most geometric algorithms). If we shoot a horizontal ray out from one side of the point, then we ignore all of the horizontal segments in the curve. For all of the other segments, we will say that every segment contains only the "higher" endpoint. (This notion of "higher" can easily be fixed relative to the horizontal ray) By using this method, we correctly compute the winding number, despite any degeneracies caused by our ray passing through endpoints. Figure 7 gives an example of using ray shooting to compute the winding number of a point.