4.1

T-Junction Elimination and Retriangulation

Eric Lengyel, Terathon Software

lengyel@terathon.com

Suppose that a scene contains two polygons that share a common edge as shown in Figure 4.1.1a. When two such polygons belong to the same object, the vertices representing the endpoints of the common edge are usually not duplicated. Instead, both polygons reference the same vertices to save space and bus bandwidth. Graphics hardware is designed so that when adjacent polygons use exactly the same coordinates for the endpoints of shared edges, the rasterizer produces pixels for each polygon that are precise complements of each other. Along the shared edge, there is no overlap between the pixels belonging to one polygon and those belonging to the other; and, more importantly, there are no gaps between the polygons.

A problem arises when adjacent polygons belong to different objects that have their own copy of the endpoint vertices for the shared edge. These vertices might greatly differ in each object's local coordinate space. For instance, when the vertices

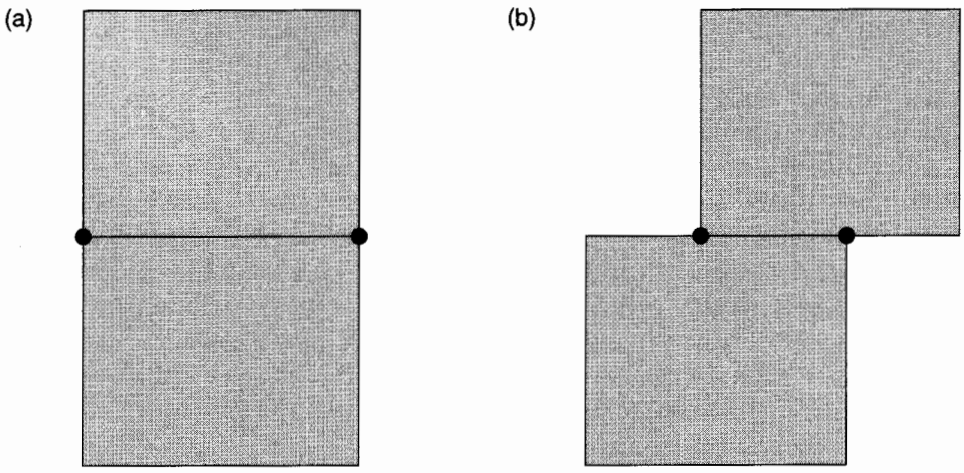

FIGURE 4.1.1 *(A) Adjacent polygons sharing a common edge. (B) Adjacent polygons with edges falling within the same line in space, but not sharing the same endpoints.*

are transformed into world space, floating-point round-off error could produce slightly different vertex positions for each object. Since the vertex coordinates are no longer equal, a seam might appear when the polygons are rasterized.

A greater problem occurs when two polygons have edges that fall within the same line in space, but they do not share the same endpoints, as illustrated in Figure 4.1.1b. In such a situation, a vertex belonging to one polygon lies within the interior of an edge belonging to the other. Due to the shape that the edges form, the location at which this occurs is called a *T-junction*. Because the adjacent edges do not share identical endpoints, T-junctions cause visible seams in any game engine that does not eliminate them.

This gem describes how to detect possible sources of these seams in complex 3D scenes and how to modify static geometry so that visible artifacts are avoided. Since T-junction elimination adds vertices to existing polygons (that are not necessarily convex), we also discuss a method for triangulating arbitrary concave polygons.

T-Junction Elimination

Given an immovable object A in our world, we need to determine whether there exist any other immovable objects possessing a vertex that lies within an edge of object A. We only consider those objects whose bounding volumes intersect the bounding volume of object A. Let object X be an object that lies close enough to object A to possibly have adjacent polygons. We treat both objects as collections of polygons having the greatest possible number of edges. We perform triangulation of these polygons *after* the T-junction elimination process in order to avoid the creation of superfluous triangles.

Before we locate any T-junctions, we first want to find out if any of object A's vertices lie very close to any of object X's vertices. We must transform the vertices belonging to both objects into world space and search for vertices separated by a distance less than some small constant ε. Any vertex $\mathbf{V}_A$ of object A that is this close to a vertex $\mathbf{V}_X$ of object X should be moved so that $\mathbf{V}_A$ and $\mathbf{V}_X$ have the exact same world-space coordinates. This procedure is sometimes called *welding*.

Once existing vertices have been welded, we need to search for vertices of object X that lie within a small distance ε of an edge of object A, but which do not lie within the distance ε of any vertex of object A. This tells us where T-junctions occur. Let $\mathbf{P}_1$ and $\mathbf{P}_2$ be endpoints of an edge of object A, and let $\mathbf{Q}$ be a vertex of object X. The squared distance d^2 between the point $\mathbf{Q}$ and the line passing through $\mathbf{P}_1$ and $\mathbf{P}_2$ is given by

$$d^2 = (\mathbf{Q} - \mathbf{P}_1)^2 - \frac{\left[(\mathbf{Q} - \mathbf{P}_1) \cdot (\mathbf{P}_2 - \mathbf{P}_1)\right]^2}{(\mathbf{P}_2 - \mathbf{P}_1)^2} \tag{4.1.1}$$

If $d^2 < \varepsilon^2$, then we know that the point $\mathbf{Q}$ lies close enough to the line containing the edge of object A, but we still need to determine whether $\mathbf{Q}$ actually lies between $\mathbf{P}_1$ and $\mathbf{P}_2$. We can make this determination by measuring the projected length t of the

line segment connecting $\mathbf{P}_1$ to $\mathbf{Q}$ onto the edge formed by $\mathbf{P}_1$ and $\mathbf{P}_2$. As shown in Figure 4.1.2, this length is given by

$$t = \|\mathbf{Q} - \mathbf{P}_1\| \cos\alpha, \tag{4.1.2}$$

where α is the angle between the line segment and the edge. Using a dot product to compute the cosine, we have

$$t = \frac{(\mathbf{Q} - \mathbf{P}_1) \cdot (\mathbf{P}_2 - \mathbf{P}_1)}{\|\mathbf{P}_2 - \mathbf{P}_1\|}. \tag{4.1.3}$$

If $t > \|\mathbf{P}_2 - \mathbf{P}_1\| - \varepsilon$, then the point $\mathbf{Q}$ does not lie within the interior of the edge formed by $\mathbf{P}_1$ and $\mathbf{P}_2$. Otherwise, we have found a T-junction, and a new vertex should be added to the polygon of object A between $\mathbf{P}_1$ and $\mathbf{P}_2$, precisely at $\mathbf{Q}$'s location.

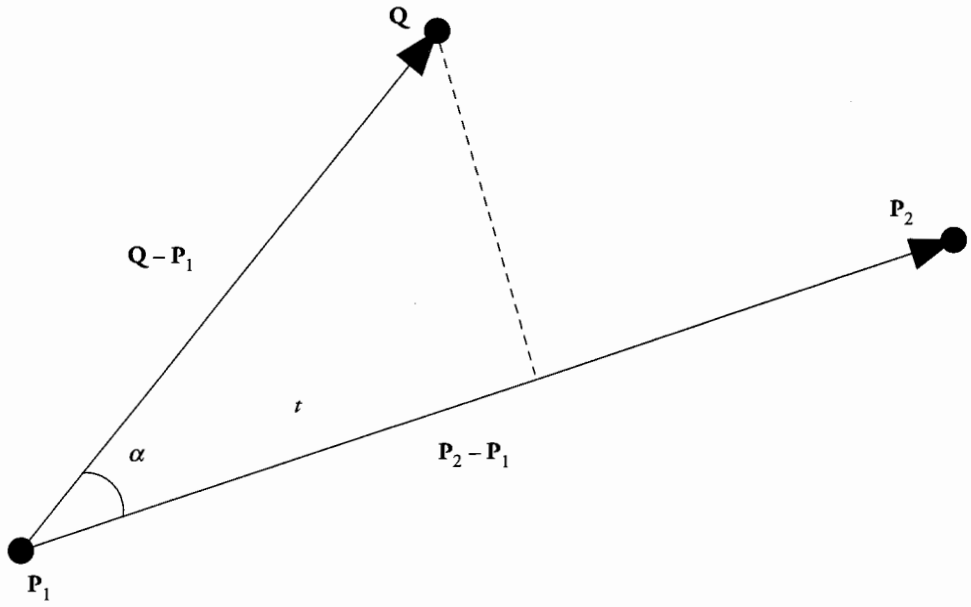

FIGURE 4.1.2 *The length* t *is equal to the distance from* $\boldsymbol{P}_1$ *to the projection of point* $\boldsymbol{Q}$, *onto the edge between* $\boldsymbol{P}_1$ *and* $\boldsymbol{P}_2$.

Retriangulation

After all the static world geometry has been processed, we must triangulate the resulting polygons so that they can be passed to the graphics hardware. Any vertex added to a polygon to eliminate a T-junction is collinear (or at least nearly collinear) with the

endpoints of the edge for which the T-junction occurs. After all T-junctions have been eliminated for a single polygon, its edges might contain several vertices that fall in a straight line. This prevents us from using a simple fanning approach that might ordinarily be used to triangulate a convex polygon. Instead, we are forced to treat the polygon as concave.

The algorithm that we describe takes a list of n vertices wound in a counterclockwise direction as input and produces a list of $n - 2$ triangles. At each iteration, we search for a set of three consecutive vertices for which the corresponding triangle is not degenerate (not wound in the wrong direction) and does not contain any of the polygon's remaining vertices. Once such a set of three vertices is found, the middle vertex is disqualified from successive iterations, and the algorithm repeats until only three vertices remain.

In order to determine whether a set of three vertices is wound in a counterclockwise direction, we must know beforehand the normal direction $\mathbf{N}_0$ of the plane containing the polygon being triangulated. Let $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ represent the positions of the three vertices. If the cross-product $(\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)$ points in the same direction as the normal $\mathbf{N}_0$, then the corresponding triangle is wound counterclockwise. If the cross-product is near zero, then the triangle is degenerate. Thus, two of our three requirements for a triangle are satisfied only if

$$\left(\mathbf{P}_2 - \mathbf{P}_1\right) \times \left(\mathbf{P}_3 - \mathbf{P}_1\right) \cdot \mathbf{N}_0 > \varepsilon \tag{4.1.4}$$

for some small value ε (typically, $\varepsilon \approx 0.001$).

Our third requirement is that the triangle contains no other vertices belonging to the polygon. We can construct three inward-facing normals, $\mathbf{N}_1$, $\mathbf{N}_2$, and $\mathbf{N}_3$, corresponding to the three sides of the triangle, as follows.

$$\begin{aligned}
\mathbf{N}_1 &= \mathbf{N}_0 \times \left(\mathbf{P}_2 - \mathbf{P}_1\right) \\
\mathbf{N}_2 &= \mathbf{N}_0 \times \left(\mathbf{P}_3 - \mathbf{P}_2\right) \\
\mathbf{N}_3 &= \mathbf{N}_0 \times \left(\mathbf{P}_1 - \mathbf{P}_3\right)
\end{aligned} \tag{4.1.5}$$

As shown in Figure 4.1.3, a point $\mathbf{Q}$ lies inside the triangle formed by $\mathbf{P}_1$, $\mathbf{P}_2$, and $\mathbf{P}_3$ if and only if $\mathbf{N}_i \cdot (\mathbf{Q} - \mathbf{P}_i) > -\varepsilon$ for every $i = 1,2,3$.

Since we have to calculate the normals given by Equation 4.1.5 for each triangle, we can save a little computation by replacing the condition given by Equation 4.1.4 with the equivalent expression

$$\mathbf{N}_1 \cdot \left(\mathbf{P}_3 - \mathbf{P}_1\right) > \varepsilon \tag{4.1.6}$$

This determines whether the point $\mathbf{P}_3$ lies on the positive side of the edge connecting $\mathbf{P}_1$ and $\mathbf{P}_2$.

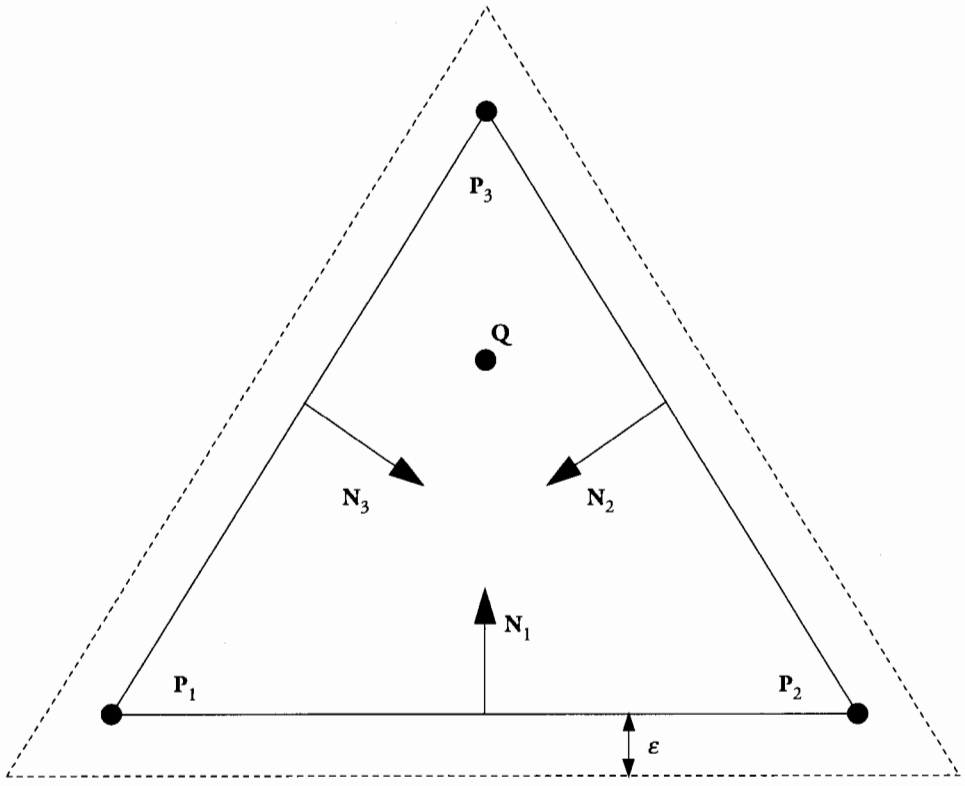

FIGURE 4.1.3 *A point* **Q** *lies inside a triangle if and only if it lies on the positive side of each of the three edges of the triangle.*

Implementation

The source code provided on the CD-ROM demonstrates an implementation of the retriangulation algorithm. This particular implementation maintains a working set of *four* consecutive vertices and, at each iteration, determines whether valid triangles can be formed using the first three vertices or the last three vertices. If only one of the sets of three vertices forms a valid triangle, then that triangle is omitted, and the algorithm continues to the next iteration. If both sets of three vertices can produce valid triangles, then the code selects the triangle having the larger smallest angle. In the case that neither set of three vertices provides a valid triangle, the working set of four vertices is advanced until a valid triangle can be constructed.

The method presented by the source code was chosen so that the output of the algorithm would consist of a series of triangle strips and triangle fans. Such triangle structures exhibit excellent vertex cache usage on modern graphics processors. The

implementation also includes a safety mechanism. If a degenerate, self-intersecting, or otherwise nontriangulatable polygon is passed to it, then the algorithm terminates prematurely to avoid becoming stuck in an infinite loop. This happens when the code cannot locate a set of three consecutive vertices that form a valid triangle.

Conclusion

Rendering artifacts such as seams between adjacent objects can be avoided by welding nearly-coincident vertices and performing T-junction elimination.

When these operations are performed as a preprocessing step, the resulting set of polygons may contain three or more collinear vertices. Fortunately, these polygons can be triangulated using a simple but robust algorithm that emits a single triangle at a time and recursively triangulates smaller sub-polygons.