Triangle Rasterization

Keegan McAllister

October 23, 2007

1 Barycentric coordinates

Consider a 2D triangle with vertices $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2 \in \mathbb{R}^2$. Let $\mathbf{x} \in \mathbb{R}^2$ be any point in the plane. We can always find $\alpha, \beta, \gamma \in \mathbb{R}$ such that

$$\mathbf{x} = \alpha\mathbf{p}_0 + \beta\mathbf{p}_1 + \gamma\mathbf{p}_2 \quad \text{where} \quad \alpha + \beta + \gamma = 1.$$

We will have $\alpha, \beta, \gamma \in [0, 1]$ if and only if $\mathbf{x}$ is in the triangle. Intuitively, a triangle consists of all weighted averages of its vertices. We call (α, β, γ) the *barycentric coordinates* of $\mathbf{x}$. This is a non-orthogonal coordinate system for the plane. (Of course, none of this works if the triangle is degenerate, i.e. if the vertices lie along a common line.)

2 Rasterization

Now suppose you want to draw a triangle with vertices $\mathbf{p}_i = (x_i, y_i), i \in \{0, 1, 2\}$ onto a canvas of discrete pixels. This is called *triangle rasterization* and can be accomplished in many ways; what follows is one conceptually simple algorithm for triangle rasterization.

We assume that the $\mathbf{p}_i$ are *normalized device coordinates* (NDC); that is, the canvas corresponds to the region $[-1, 1] \times [-1, 1]$. This is what you get after applying all transformation matrices.

First, identify a rectangular region on the canvas that contains all of the pixels in the triangle (excluding those that lie outside the canvas). This could be the whole canvas, but for efficiency we'd like to deal with fewer pixels. It's not hard to calculate a tight bounding box for a triangle: simply calculate pixel coordinates for each vertex, and find the minimum/maximum for each axis, clipped to the canvas size.

Once we've identified the bounding box, we loop over each pixel in the box. For each pixel, we first compute the corresponding NDC coordinates (x, y). Next we convert these into barycentric coordinates for the triangle being drawn. To do this, let

$$f_{ab}(x, y) \stackrel{\text{def}}{=} (y_a - y_b)\, x + (x_b - x_a)\, y + x_a y_b - x_b y_a$$

for $a, b \in \{0, 1, 2\}$. We then have

$$\alpha = \frac{f_{12}(x, y)}{f_{12}(x_0, y_0)}$$
$$\beta = \frac{f_{20}(x, y)}{f_{20}(x_1, y_1)}$$
$$\gamma = \frac{f_{01}(x, y)}{f_{01}(x_2, y_2)}$$

You will probably want to compute the denominators here outside the loop. Also, watch out for divide-by-zero errors; these indicate a degenerate triangle which you can throw out entirely.

Once we have (α, β, γ) we can check if all of α, β, γ lie in $[0, 1]$. If not, the current pixel is outside the triangle and we're done with this loop iteration. Otherwise, we draw the pixel.

3 Interpolation

To draw pixels in a triangle, you need access to certain information for every pixel, as interpolated from the vertices. At minimum you'll need interpolated NDC z-coordinates to do depth buffering. Depending on the shading model you may also need interpolated colors or normals. Fortunately, the barycentric coordinates also serve as interpolation coefficients. Say we have values c_0, c_1, c_2 (vectors or scalars) at the corresponding vertices. Then the interpolated value for a pixel with barycentric coordinates (α, β, γ) is just

$$c = \alpha c_0 + \beta c_1 + \gamma c_2.$$

4 References

This document is based on pages 63-66 of the course text (Shirley *et al.*). The pseudocode given there (and the sample code given on the course website) additionally ensures that pixels along the edge between two adjacent triangles are drawn with exactly one of those triangles. This is not necessary for lab 3, and has caused trouble for students in the past. However, keep this in mind if you're ever rasterizing in a situation where the drawn pixels are somehow blended with data already in the color buffer.