# CS 314: Barycentric Coordinates, 2D Transformations

Robert Bridson

September 9, 2008

## 1 Barycentric Coordinates

In the last installment we determined how to rasterize a triangle by reducing it to a point-in-triangle test based on special edge functions which implicitly define the lines containing the edges of the triangle. If the vertices of the triangle are at $(x_0, y_0)$, $(x_1, y_1)$, and $(x_2, y_2)$, and the point to be tested is $(x, y)$, the three functions to evaluate are:

$$\begin{aligned} F_{01}(x,y) &= (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0) \\ F_{12}(x,y) &= (x_2 - x_1)(y - y_1) - (x - x_1)(y_2 - y_1) \\ F_{20}(x,y) &= (x_0 - x_2)(y - y_2) - (x - x_2)(y_0 - y_2) \end{aligned}$$

If they all have the same sign, the point is inside. Remember that these functions were derived from the area of a triangle using the vertices of an edge and the point $(x, y)$.

We now will use these evaluations to define the **barycentric coordinates** of the point $(x, y)$ with respect to the triangle. The first fact to observe is that the sum of the edge functions is a constant:

$$F_{01}(x,y) + F_{12}(x,y) + F_{20}(x,y) = S \equiv \text{constant}$$

This is easy to verify just by expanding out the expressions: you should also be able to see it geometrically, at least when $(x, y)$ is inside the triangle, since up to signs the edge functions are measuring twice the areas of three triangles that can be combined to form the original triangle, independent of $(x, y)$. This quantity $S$ is, up to sign, twice the area of the original triangle.

It's also easy to both verify from the formulas, and see geometrically, that

$$S = F_{01}(x_2, y_2) = F_{12}(x_0, y_0) = F_{20}(x_1, y_1)$$

However, adding up the evaluations to get $S$ is cheaper and simpler.

We're now ready to define the barycentric coordinates $(\alpha, \beta, \gamma)$ of the point $(x, y)$:

$$\begin{aligned} \alpha &= \frac{F_{12}(x,y)}{S} \\ \beta &= \frac{F_{20}(x,y)}{S} \\ \gamma &= \frac{F_{01}(x,y)}{S} \end{aligned}$$

Because of the way we defined $S$, it's immediately obvious that they add up to one: $\alpha + \beta + \gamma = 1$. (In fact, in computation $\gamma$ is often equivalently defined as $1 - \alpha - \beta$ to avoid a third division.)

Since the barycentric coordinates are just constant rescalings of the edge functions, they can also be used to define the inside of the triangle: a point is inside the triangle if and only if all its barycentric coordinates have the same sign. Since the barycentric coordinates add up to 1, it's not hard to see this is equivalent to saying that they all lie in the range $[0, 1]$.

Remember each edge function, and thus its corresponding barycentric coordinate, is zero along the line containing the edge. A vertex of the triangle is a point on two edges, i.e. where two of these edge functions or barycentric coordinates are zero. Since the barycentric coordinates sum to one, the nonzero barycentric coordinate must be 1. Again, it's not hard to verify this:

- The barycentric coordinates of vertex $(x_0, y_0)$ are $(\alpha = 1, \beta = 0, \gamma = 0)$
- The barycentric coordinates of vertex $(x_1, y_1)$ are $(\alpha = 0, \beta = 1, \gamma = 0)$
- The barycentric coordinates of vertex $(x_2, y_2)$ are $(\alpha = 0, \beta = 0, \gamma = 1)$

This can be generalized to the following critical property of barycentric coordinates:

For any point $\vec{x}$ with barycentric coordinates $(\alpha, \beta, \gamma)$, we have $\vec{x} = \alpha\vec{x}_0 + \beta\vec{x}_1 + \gamma\vec{x}_2$

Together with the requirement they sum to 1, this can in fact be used as an alternative definition of barycentric coordinates. It says that every point in the plane is a weighted average of the triangle's vertices, with weights equal to the barycentric coordinates (which sum to 1 just like the weights in an average should do).

### 1.1 Linear Interpolation in 1D

Before making use of barycentric coordinates in 2D, let's quickly recall what **linear interpolation** means in 1D. Given two points on the real line, $x_0$ and $x_1$, with associated values $f_0$ and $f_1$, linear interpolation

draws the line between the points as an estimate of the graph of the function $f(x)$. For $x$ between $x_0$ and $x_1$, we can look up values on this line as a simple interpolation between the known function values.

It's not hard to work out the formula for this line, in classic slope-intercept form:

$$y = \left(\frac{f_1 - f_0}{x_1 - x_0}\right) x + \left(\frac{f_0 x_1 - f_1 x_0}{x_1 - x_0}\right)$$

This can be rearranged to:

$$y = \left(\frac{x_1 - x}{x_1 - x_0}\right) f_0 + \left(\frac{x - x_0}{x_1 - x_0}\right) f_1$$

Note that the coefficients of $f_0$ and $f_1$ in this form add up to 1, and so express the interpolated value $y$ as a weighted average of $f_0$ and $f_1$ with weights that depend on $x$. These weights are in fact the 1D line version of barycentric coordinates. They too satisfy, for any $x$, the property that

$$x = \left(\frac{x_1 - x}{x_1 - x_0}\right) x_0 + \left(\frac{x - x_0}{x_1 - x_0}\right) x_1$$

which makes sense if you realize that linearly interpolating the function $f(x) = x$ should be exact.

This linear interpolation operation is so common and important in computer graphics that it often is abbreviated to the word **lerp**, as in "I lerped the colour from the two inputs". Often it's written in a slightly different form, with a parameter variable $t$:

$$\begin{aligned} t &= \frac{x - x_0}{x_1 - x_0} \\ y &= (1-t)f_0 + tf_1 \qquad \text{or} \qquad f_0 + t(f_1 - f_0) \end{aligned}$$

Due to round-off error in floating point arithmetic, the first form $(1-t)f_0 + tf_1$ is generally recommended, but you also sometimes see the second form.

### 1.2 Linear Interpolation on Triangles

We extend linear interpolation to 2D by using barycentric coordinates. If we have some function values at the corners of a triangle, $f_0$, $f_1$ and $f_2$, then we can interpolate between them in the triangle as:

$$f(\vec{x}) \approx \alpha f_0 + \beta f_1 + \gamma f_2$$

This works for vector-valued functions too, like RGB colour: the colour at a point inside a triangle can be interpolated from colours stored at the vertices using barycentric coordinates.

Rasterization of triangles usually provides this as an option, specifying colours at the vertices of triangles instead of just a constant colour for the entire triangle. This provides much smoother shading,

since the colour will now vary continuously across an edge from one triangle to the next. This is so important in graphics that it has a special name, "Gouraud" shading or interpolation, named for Henri Gouraud who introduced it. (In contrast, using a constant colour per triangle is sometimes called "flat shading".)

# 2 Transformations in 2D

We now have basic triangle rasterization sorted out; the next step back up the pipeline is to examine geometric transformations in 2D.

## 2.1 Scaling

Right now our rasterization expects the coordinates of the triangles to be given with respect to the size of the image in pixels. This is a bit inconvenient when building geometric models: for example, in a 2D drawing program that models a printed page it might make a lot more sense to specify the coordinates of triangle vertices in terms of millimetres or inches or points, and even for just displaying on the computer if the user wants to change the resolution of the image it would be nice to avoid having to also change all the coordinates.

This is solved by including a geometric transformation called **scaling**, which just means multiplying all coordinates by a scale factor just before rasterization. The idea is that the scale factor should take into account the desired image resolution, converting from whatever coordinates the user made their triangles with to image pixel coordinates.

The important thing to know in determining the scaling is how big the user wants the image to be in terms of their coordinates. For example, if the user is drawing triangles with coordinates in inches, and wants the image to match a standard $8.5" \times 11"$ piece of paper, the graphics system needs to know, say with a width variable $w = 8.5$ and a height variable $h = 11$. If this needs to be rendered to a $791 \times 1024$ image, for example, then the pixel dimensions are set to $m = 791$ and $n = 1024$. Finally, whenever a triangle specified in inches on the paper is to be rasterized in the image, the vertices $(x, y)$ are scaled to pixel coordinates $(x', y')$ by:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \frac{m}{w}x \\ \frac{n}{h}y \end{pmatrix}$$

The $x$ and $y$ scale factors in this case are nearly the same, but they needn't be of course.

We can express scaling a little more conveniently (when it comes to integrating with additional

transformations yet to come) by using a $2 \times 2$ matrix:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \frac{m}{w} & 0 \\ 0 & \frac{n}{h} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

In particular, once we have the scaling matrix $S$ calculated as

$$S = \begin{pmatrix} \frac{m}{w} & 0 \\ 0 & \frac{n}{h} \end{pmatrix}$$

then scaling points is just a matrix-vector multiply: $\vec{x}' = S\vec{x}$.

## 2.2 Rotation

Once we start using a diagonal matrix to encode the scaling transformation, we can think about what other $2 \times 2$ matrices might do. The most important is **rotation**. Some easy trigonometry should let you work out that if you rotate a point $(x, y)$ around the origin counterclockwise by $\phi$ radians, you get new coordinates $(x', y')$:

$$\begin{aligned} x' &= \cos(\phi)x - \sin(\phi)y \\ y' &= \sin(\phi)x + \cos(\phi)y \end{aligned}$$

(The Shirley textbook derives this by expressing the original point in polar coordinates, $(r\cos(\alpha), r\sin(\alpha))$, which makes rotation trivial.) This also can be written as a matrix-vector multiply:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}$$

This new transformation matrix, call it $Q$, is not diagonal but does have the very useful property of being orthogonal: its columns are orthogonal and have unit vector length, as do its rows, so that for example $Q^TQ$ is the identity matrix $I$.

If the user has a set of triangles they drew, but now wants to rotate them all around the origin, our graphics system can (before rasterization) multiply the points by the appropriate matrix and then by the scaling matrix to get pixel coordinates.

This is where the whole matrix thing starts to become really useful: instead of multiplying every point first by the rotation matrix $Q$ and then by the scaling matrix $S$, i.e. evaluating $\vec{x}' = S(Q\vec{x})$, we can form a single matrix product

$$M = SQ$$

and then much more efficiently transform points with a single multiply: $\vec{x}' = M\vec{x}$. This is an important optimization if you have a lot of points to transform.