## 3.6 Triangle Rasterization

We often want to draw a 2D triangle with 2D points $\mathbf{p}_0 = (x_0, y_0)$, $\mathbf{p}_1 = (x_1, y_1)$, and $\mathbf{p}_2 = (x_2, y_2)$ in screen coordinates. This is similar to the line drawing problem, but it has some of its own subtleties. It will turn out that there is no advantage to integer coordinates for endpoints, so we will allow the $(x_i, y_i)$ to have floating point values. As with line drawing, we may wish to interpolate color or other properties from values at the vertices. This is straightforward if we have the barycentric coordinates (Section 2.11). For example, if the vertices have colors $\mathbf{c}_0$, $\mathbf{c}_1$, and $\mathbf{c}_2$, the color at a point in the triangle with barycentric coordinates $(\alpha, \beta, \gamma)$ is

$$\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2.$$

This type of interpolation of color is known in graphics as *Gouraud* interpolation after its inventor (Gouraud, 1971).

Another subtlety of rasterizing triangles is that we are usually rasterizing triangles that share vertices and edges. This means we would like to rasterize adjacent triangles so there are no holes. We could do this by using the midpoint algorithm to draw the outline of each triangle and then fill in the interior pixels. This would mean adjacent triangles both draw the same pixels along each edge. If the adjacent triangles have different colors, the image will depend on the order in which the two triangles are drawn. The most common way to rasterize triangles that avoids the order problem and eliminates holes is to use the convention that pixels are drawn if and only if their centers are inside the triangle, i.e., the barycentric coordinates of the pixel center are all in the interval (0, 1). This raises the issue of what to do if the center is exactly on the edge of the triangle. There are several ways to handle this as will be discussed later in this section. The key observation is that barycentric coordinates allow us to decide whether to draw a pixel and what color that pixel should be if we are interpolating colors from the vertices. So our problem of rasterizing the triangle boils down to efficiently finding the barycentric coordinates of pixel centers (Pineda, 1988). The brute-force rasterization algorithm is:

```
for all $x$ do
    for all $y$ do
```

```
compute $(\alpha,\beta,\gamma)$ for $(x,y)$
if $(\alpha \in [0,1]$ and $\beta \in [0,1]$ and $\gamma \in [0,1])$ then
    $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
    drawpixel $(x,y)$ with color $\mathbf{c}$
```

The rest of the algorithm limits the outer loops to a smaller set of candidate pixels and makes the barycentric computation efficient.

We can add a simple efficiency by finding the bounding rectangle of the three vertices and only looping over this rectangle for candidate pixels to draw. We can compute barycentric coordinates using Equation 2.32. This yields the algorithm:

```
$x_{\min} =$ floor $(x_i)$
$x_{\max} =$ ceiling $(x_i)$
$y_{\min} =$ floor $(y_i)$
$y_{\max} =$ ceiling $(y_i)$
for $y = y_{\min}$ to $y_{\max}$ do
    for $x = x_{\min}$ to $x_{\max}$ do
        $\alpha = f_{12}(x,y)/f_{12}(x_0,y_0)$
        $\beta = f_{20}(x,y)/f_{20}(x_1,y_1)$
        $\gamma = f_{01}(x,y)/f_{01}(x_2,y_2)$
        if $(\alpha > 0$ and $\beta > 0$ and $\gamma > 0)$ then
            $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
            drawpixel $(x,y)$ with color $\mathbf{c}$
```

Here $f_{ij}$ is the line given by Equation 3.3 with the appropriate vertices:

$$\begin{aligned} f_{01}(x,y) &= (y_0 - y_1)x + (x_1 - x_0)y + x_0y_1 - x_1y_0, \\ f_{12}(x,y) &= (y_1 - y_2)x + (x_2 - x_1)y + x_1y_2 - x_2y_1, \\ f_{20}(x,y) &= (y_2 - y_0)x + (x_0 - x_2)y + x_2y_0 - x_0y_2. \end{aligned}$$

Note that we have exchanged the test $\alpha \in (0,1)$ with $\alpha > 0$ etc., because if all of $\alpha$, $\beta$, $\gamma$ are positive, then we know they are all less than one because $\alpha + \beta + \gamma = 1$. We could also compute only two of the three barycentric variables and get the third from that relation, but it is not clear that this saves computation once the algorithm is made incremental, which is possible as in the line drawing algorithms; each of the computations of $\alpha$, $\beta$, and $\gamma$ does an evaluation of the form $f(x,y) = Ax + By + C$. In the inner loop, only $x$ changes, and it changes by one. Note that $f(x+1,y) = f(x,y) + A$. This is the basis of the incremental algorithm. In the outer loop, the evaluation changes for $f(x,y)$ to $f(x,y+1)$, so a similar efficiency can be achieved. Because $\alpha$, $\beta$, and $\gamma$ change by constant

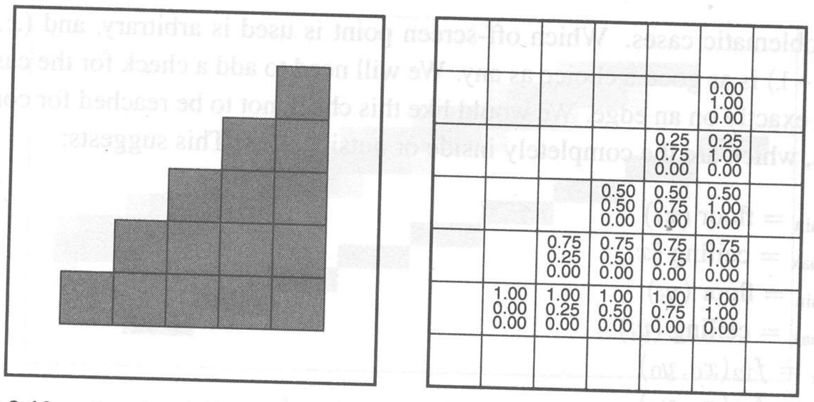

**Figure 3.10.** A colored triangle with barycentric interpolation. Note that the changes in color components are linear in each row and column as well as along each edge. In fact it is constant along every line, such as the diagonals, as well. (See also Plate III.)

increments in the loop, so does the color **c**. So this can be made incremental as well. For example, the red value for pixel $(x+1, y)$ differs from the red value for pixel $(x, y)$ by a constant amount that can be precomputed. An example of a triangle with color interpolation is shown in Figure 3.10.

### 3.6.1 Dealing With Pixels on Triangle Edges

We have still not discussed what to do for pixels whose centers are exactly on the edge of a triangle. If a pixel is exactly on the edge of a triangle, then it is also on the edge of the adjacent triangle if there is one. There is no obvious way to award the pixel to one triangle or the other. The worst decision would be to not draw the pixel because a hole would result between the two triangles. Better, but still not good, would be to have both triangles draw the pixel. If the triangles are transparent, this will result in a double-coloring. We would really like to award the pixel to exactly one of the triangles, and we would like this process to be simple; which triangle is chosen does not matter as long as the choice is well defined.

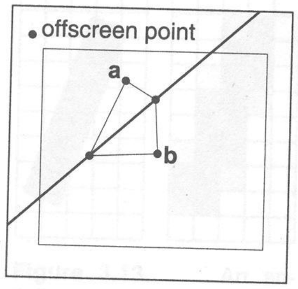

**Figure 3.11.** The off-screen point will be on one side of the triangle edge or the other. Exactly one of the non-shared vertices **a** and **b** will be on the same side.

One approach is to note that any off-screen point is definitely on exactly one side of the shared edge and that is the edge we will draw. For two non-overlapping triangles, the vertices not on the edge are on opposite sides of the edge from each other. Exactly one of these vertices will be on the same side of the edge as the off-screen point (Figure 3.11). This is the basis of the test. The test if numbers $p$ and $q$ have the same sign can be implemented as the test $pq > 0$, which is very efficient in most environments.

Note that the test is not perfect because the line through the edge may also go through the offscreen point, but we have at least greatly reduced the number of problematic cases. Which off-screen point is used is arbitrary, and $(x, y) = (-1, -1)$ is as good a choice as any. We will need to add a check for the case of a point exactly on an edge. We would like this check not to be reached for common cases, which are the completely inside or outside tests. This suggests:

```
$x_{\min} = \text{floor}\,(x_i)$
$x_{\max} = \text{ceiling}\,(x_i)$
$y_{\min} = \text{floor}\,(y_i)$
$y_{\max} = \text{ceiling}\,(y_i)$
$f_\alpha = f_{12}(x_0, y_0)$
$f_\beta = f_{20}(x_1, y_1)$
$f_\gamma = f_{01}(x_2, y_2)$
for $y = y_{\min}$ to $y_{\max}$ do
  for $x = x_{\min}$ to $x_{\max}$ do
    $\alpha = f_{12}(x, y)/f_\alpha$
    $\beta = f_{20}(x, y)/f_\beta$
    $\gamma = f_{01}(x, y)/f_\gamma$
    if ($\alpha \geq 0$ and $\beta \geq 0$ and $\gamma \geq 0$) then
      if ($\alpha > 0$ or $f_\alpha f_{12}(-1, -1) > 0$) and ($\beta > 0$ or $f_\beta f_{20}(-1, -1) > 0$)
      and ($\gamma > 0$ or $f_\gamma f_{01}(-1, -1) > 0$) then
        $\mathbf{c} = \alpha\mathbf{c}_0 + \beta\mathbf{c}_1 + \gamma\mathbf{c}_2$
        drawpixel $(x, y)$ with color $\mathbf{c}$
```

We might expect that the above code would work to eliminate holes and double-draws only if we use exactly the same line equation for both triangles. In fact, the line equation is the same only if the two shared vertices have the same order in the draw call for each triangle. Otherwise the equation might flip in sign. This could be a problem depending on whether the compiler changes the order of operations. So if a robust implementation is needed, the details of the compiler and arithmetic unit may need to be examined. The first four lines in the pseudocode above must be coded carefully to handle cases where the edge exactly hits the pixel center.

In addition to being amenable to an incremental implementation, there are several potential early exit points. For example, if $\alpha$ is negative, there is no need to compute $\beta$ or $\gamma$. While this may well result in a speed improvement, profiling is always a good idea; the extra branches could reduce pipelining or concurrency and might slow down the code. So as always, test any attractive looking optimizations if the code is a critical section.

Another detail of the above code is that the divisions could be divisions by zero for degenerate triangles, i.e., if $f_\gamma = 0$. Either the floating point error conditions should be accounted for properly, or another test will be needed.