Appendix: Steffen's Rasterization Method

[Note from Joe: this rasterization approach was first developed by Mike Steffen (link) when he was an ISU graduate student. The goal was to provide an algorithm that could traverse a triangle and interpolate all attributes without performing a full raster scan of the bounding box. A secondary consideration was to minimize the hardware resources required – this approach requires only one division operation and a handful of multipliers.]

Goal. Given a triangle in screen space, with coordinates (x_0, y_0), (x_1, y_1), and (x_2, y_2), and corresponding attributes p_0, p_1, and p_2 determine which other points (x, y) lie inside the triangle as well as their attribute values p.

Triangle Setup. We can use the 3D plane equation with the *z* dimension corresponding to the attribute to be interpolated, resulting in one plane equation per attribute. To solve for *p* at any arbitrary point, we can calculate the matrix determinant / surface normal:

$$\begin{vmatrix} x - x_0 & y - y & p - p_0 \\ x_1 - x_0 & y_1 - y_0 & p_1 - p_0 \\ x_2 - x_0 & y_2 - y_0 & p_2 - p_0 \end{vmatrix} = 0$$

It may help to first look at the more general 3x3 matrix determinant to simplify the variables somewhat:

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix} = 0 \leftrightarrow a \cdot (ei - fh) - b \cdot (di - fg) + \ c \cdot (dh - eg) = 0$$

Expanding this out for the triangle plane gets a bit messy but is still manageable:

$$\begin{aligned} &(x - x_0) \cdot \big((y_1 - y_0) \cdot (p_2 - p_0) - (p_1 - p_0) \cdot (y_2 - y_0)\big) \\ &- (y - y_0) \cdot \big((x_1 - x_0) \cdot (p_2 - p_0) - (p_1 - p_0) \cdot (x_2 - x_0)\big) \\ &+ (p - p_0) \cdot \big((x_1 - x_0) \cdot (y_2 - y_0) - (y_1 - y_0) \cdot (x_2 - x_0)\big) = 0 \end{aligned}$$

Note that $d = x_1 - x_0$, $e = y_1 - y_0$, $g = x_2 - x_0$, and $h = y_2 - y_0$, are a per-triangle constant, and $f = p_1 - p_0$ and $i = p_2 - p_0$ are a per-attribute constant. We can declare the following variables to make manipulating this equation simpler:

Per-Triangle Constants:
$C_1 = dh - eg = (x_1 - x_0) \cdot (y_2 - y_0) - (y_1 - y_0) \cdot (x_2 - x_0)$
$C_4 = \dfrac{1}{C_1}$

Per-Attribute Constants:
$C_2 = fh - ei = (p_1 - p_0) \cdot (y_2 - y_0) - (y_1 - y_0) \cdot (p_2 - p_0)$
$C_3 = di - fg = (x_1 - x_0) \cdot (p_2 - p_0) - (p_1 - p_0) \cdot (x_2 - x_0)$
$C_5 = C_2 \cdot C_4$
$C_6 = C_3 \cdot C_4$

Solving for the unknown attribute p in the determinant equation gives us an expression in terms of x, y, and the constants above:

$$(x - x_0) \cdot (ei - fh) - (y - y_0) \cdot (di - fg) + (p - p_0) \cdot (dh - eg) = 0$$

$$(p - p_0) \cdot C_1 = (x - x_0) \cdot C_2 + (y - y_0) \cdot C_3$$

$$p = \frac{(x - x_0) \cdot C_2 + (y - y_0) \cdot C_3}{C_1} + p_0$$

$$p = f(x, y) = (x - x_0) \cdot C_5 + (y - y_0) \cdot C_6 + p_0$$

Noting that in traversing a triangle in screen space, the change in this function in either the x or y dimension given a known starting point is well-defined. We can use this to minimize the number of multipliers required:

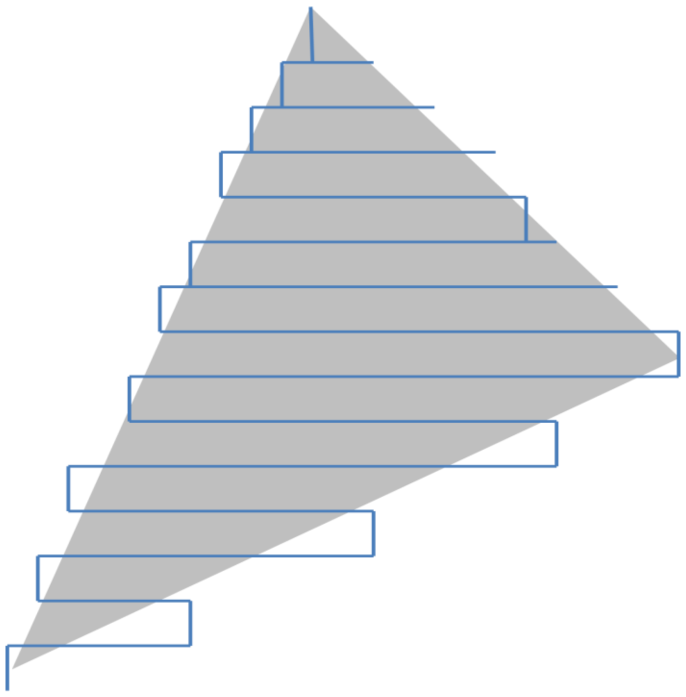
$f(x_0, y_0) = p_0$

$f(x_0 + \Delta x, y_0) = f(x_0, y_0) + \Delta x \cdot C_5 = p_0 + \Delta x \cdot C_5$

$f(x_0, y_0 + \Delta y) = f(x_0, y_0) + \Delta y \cdot C_6 = p_0 + \Delta y \cdot C_6$

If we can order the fragments such that the starting fragment is at a known solution p_0, and Δx or Δy is always 1, we can compute the C_5 and C_6 constants for each attribute and use addition to determine the interpolated attribute values for all fragments inside a triangle.

Generating Fragments. As described above, our goal is to generate fragments such that Δx or Δy = 1, and start at a known point. To accomplish this, a zig-zag scan will be used starting at the top-most vertex point:



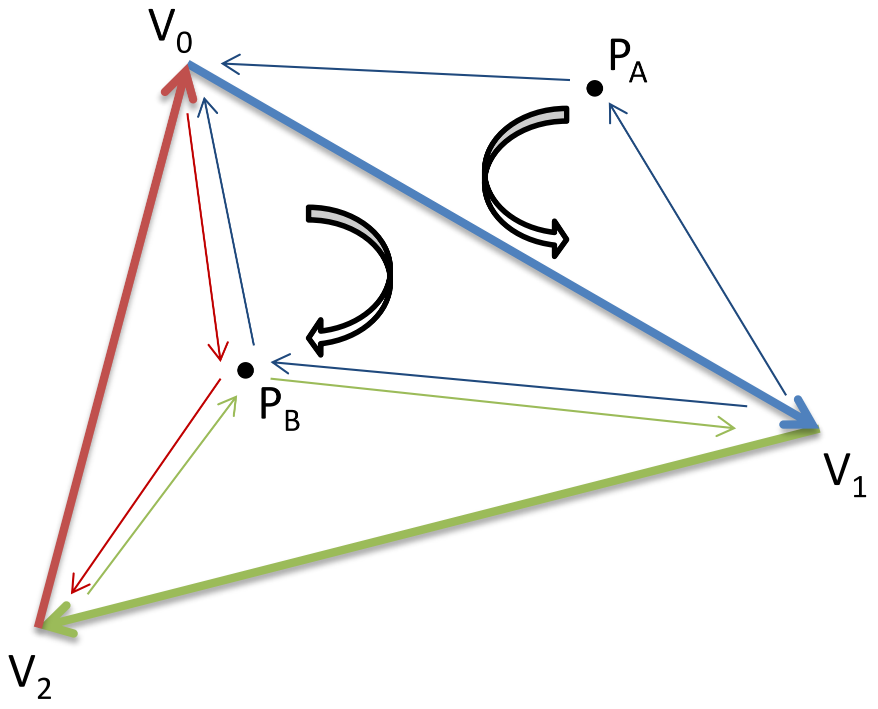
This may require sorting the vertex values for a triangle primitive before starting traversal, since it is not guaranteed that the triangle will be assembled with y_0 as the top-most of the three y-coordinate values.

Also, for this traversal, there are times when we may want to jump back to a known solution. For example, if we move down a scan line and start in the middle of the triangle, we need to go both directions. We can push the current value onto a single-entry stack and scan right. Once we hit the right limit, we can pop the stack and continue scanning left. Using a single-entry stack allows us to maintain the Δx or Δy of 1 while also allowing the traversal to skip large sections.

Triangle Testing. This approach is only helpful if we are generating valid fragments, that is the fragments that are actually inside of the triangle. Conceptually, to test if a fragment is inside the triangle, we can consider the three individual line segments – if the fragment is on the correct side of all three lines, the point is inside the triangle.

To do this we can break the triangle into three lines. We then test the fragment against each line to see what side of the line the fragment is on. If the fragment is on the correct side of all the lines, the point is inside the triangle. "Correct" is a relative term, and for our purposes, we use the triangle orientation, either in Clockwise (CW) or Counter-Clockwise (CCW) direction. A triangle created from the testing fragment would have the same direction as the original triangle, if the fragment is inside the triangle. Otherwise, the new point would result in a triangle that is oriented in the opposite direction. OpenGL refers to this direction/orientation as the triangle's "Winding order", and your applications can configured to skip rendering triangles that face the wrong direction.

The figure below illustrates a triangle $V_0 \rightarrow V_1 \rightarrow V_2$ which has a CW orientation. When considering fragment P_A, the resulting triangle $V_0 \rightarrow V_1 \rightarrow P_A$ has a CCW orientation. It is clearly outside the triangle. In contrast triangle $V_0 \rightarrow V_1 \rightarrow P_B$ has a CW orientation (as well as the other triangles that include P_B), and consequently fragment P_B is inside the triangle.



Computing Triangle Direction. Similar to how we interpolated the fragment attributes using the plane equation, we can compute a triangle's direction using the equation for the area of a triangle given three points in a coordinate plane:

$$A = \frac{x_0 \cdot (y_1 - y_2) + x_1 \cdot (y_2 - y_0) + x_2 \cdot (y_0 - y_1)}{2}$$

Manipulating this equation a bit give us:

$$A = \frac{(x_0 - x_2) \cdot (y_1 - y_2) - (x_1 - x_2) \cdot (y_0 - y_2)}{2}$$

Note that A can be either positive or negative, which is why one usually takes the absolute value of this equation when calculating the area. If $A > 0$, the triangle made by these three points has a CCW orientation, otherwise if $A < 0$, it has a CW orientation. If $A = 0$, the three points are co-linear and this is not a valid triangle. Specifying an $A = 0$ triangle is undefined in OpenGL and so we can ignore this case for our purposes.

We can then define the area of a triangle containing V_0, V_1, and new point P, to obtain:

$$A_P = \frac{(x_0 - x_P) \cdot (y_1 - y_P) - (x_1 - x_P) \cdot (y_0 - y_P)}{2}$$

Just like before, we can then define a function that allows us to calculate this value ($2A$) in terms of P:

$$f(P) = 2 \cdot A_P = x_0 \cdot y_1 - x_1 \cdot y_0 + y_P \cdot (x_1 - x_0) + x_P \cdot (y_0 - y_1)$$

When $P = V_2$, then $f(P) = 2 \cdot A$ for our original triangle. Note also that given this construction, $f(V_0) = f(V_1) = 0$. As before, when traversing the triangle, each change in either the x or y dimension given a known starting point is well-defined, which can minimize the amount of multiplications required:

$f(x_0, y_0) = \{0\}, \{0\}, \{2 \cdot A\}$

$f(x_0 + \Delta x, y_0) = f(x_0, y_0) + \Delta x \cdot (y_0 - y_1) = f(x_0, y_0) + \Delta x \cdot (y_0 - y_1) = f(x_0, y_0) + \Delta x \cdot C_7$

$f(x_0, y_0 + \Delta y) = f(x_0, y_0) + \Delta y \cdot (x_1 - x_0) = f(x_0, y_0) + \Delta y \cdot (x_1 - x_0) = f(x_0, y_0) + \Delta y \cdot C_8$

Since we are doing a zig-zag scan starting at the top-most vertex (enforcing that Δx or Δy is always 1), we can check if the fragment is inside or outside the triangle by performing a simple addition and checking the sign bit of the resulting area.

This process can be wrapped into the Triangle Setup operation – in addition to calculating attribute interpolation constants C_5 and C_6 (which are defined per-attribute), we can also calculate initial value $2 \cdot A$, area interpolation constants C_7 and C_8 (which are fixed per-triangle), as well as the initial triangle orientation. Subsequently when traversing the triangle to determine fragment attribute values using C_5 and C_6, we can also determine if the fragment should be transmitted to the next stage of the graphics pipeline using just C_7 and C_8.