## Bits of precision needed to determine area of 2D triangle

Area of a 2D triangle as a determinant:

Calculation of the determinant:

From here-on, we’ll ignore the absolute value and halving, because we most care about the sign bits (as it is the triangle formed by an edge of the triangle we’re trying to rasterize and the current point being considered for rasterization.)

In a notation where represents an integer with bits, the determinant might be computed as:

Where is the number of bits needed to represent the resulting determinant.

For each operation:

* Addition and subtraction, , in other words: addition and subtraction require an additional bit of precision on top of the maximum number of bits required amongst its two operands.
* Multiplication, , in other words: multiplication of two bit words requires a bit word to represent.

Now let:

we have:

And let:

We have:

Let:

Then:

And:

In other words, the number of bits needed for the result is two times the number of bits needed for the inputs, plus 3 additional bits. So if the inputs are 16 bits, we need 35 bits to contain the result without loss of precision.

We have 64 bits available:

So with 64 bits, we could support 31 bit inputs. Suppose for a moment our inputs are 4 bit sub-pixel resolution, this comes to bits of pixels. For purely the triangle rasterization, we could therefore support displays of up to 128 million pixels across (notwithstanding that we’ll still need to compute with it..)

## Evaluating it down to a steppable polynomial

One reason for pursuing integers is so we can “step” the integers from one pixel to the next incrementally, without loss of precision. Let’s use and as the coordinates we’d like to step with:

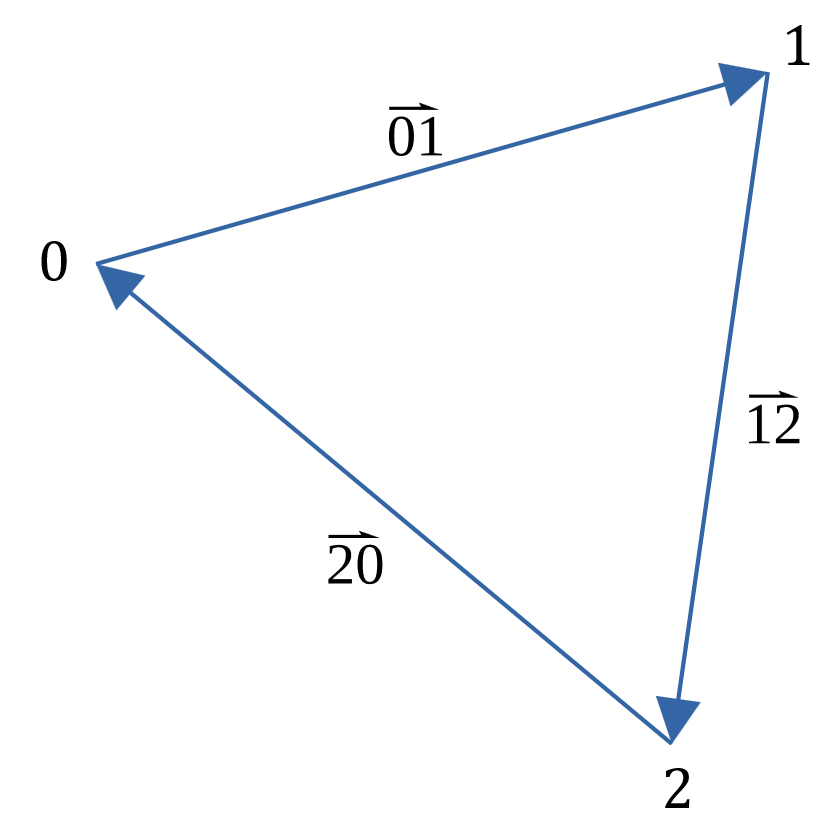
Following the same logic from before:

(where we’re taking a liberty with representing instead of a coordinate, and represents the equation.) Continuing with the logic from before:

So the result in terms of precision is the same.

## Is a pixel inside the triangle?

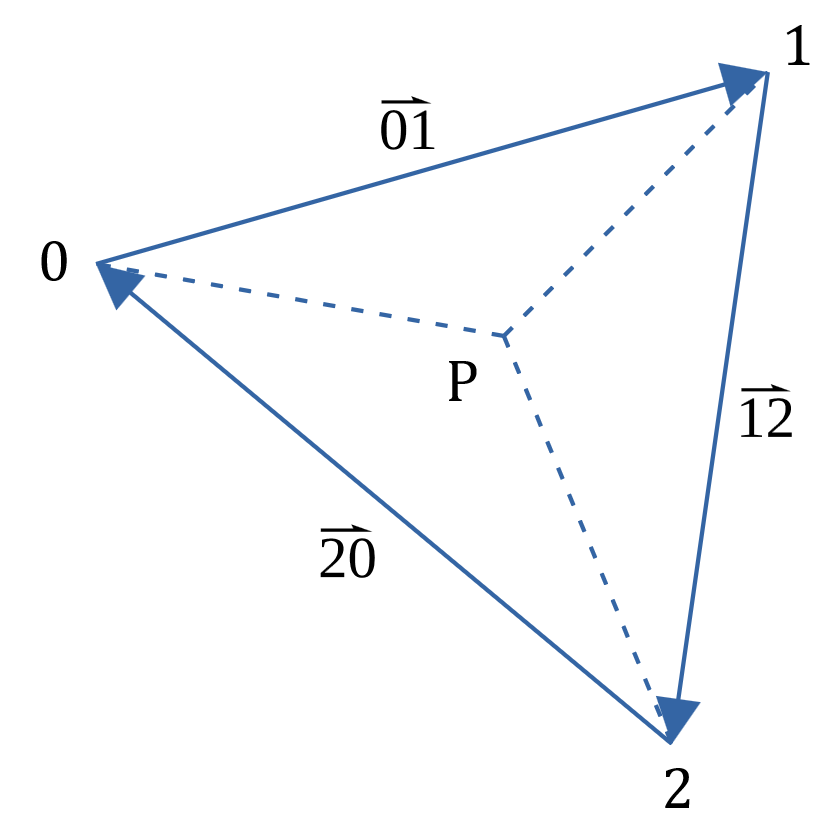
The triangle is represented by 3 vertices, , , and . Between these vertices, there are edges, , , and as the following image illustrates:



Assuming that we are in a screen coordinate space (where is positive to the right and is positive downwards,) then the determinant will be positive if the vertices are clockwise, and negative if they are counterclockwise:

In the Colinear case, the triangle should be rejected as no pixel will be inside it. (Important: if the is positive going upwards, the Clockwise and Counter-clockwise cases exchange.

For any pixel at point , we can split the triangle into 3 sub-triangles, as follows:



Here the triangles thus formed at the combination of P with each of the edges , , and , forming triangles , and , each having a corresponding determinant , and equal to double their area.

For P to be inside the triangle, all of , and have to be clockwise, and consequently, all of , and have to be positive values. Care needs to be taken with fill rules and the colinear case. For instance, it is permissible for to be equal to zero and still have the pixel lie inside the triangle, because, while edge is “top-left” here, at the adjacent triangle above it, it would not be “top-left” but “bottom-right.” Consequently, it meets the criteria for this triangle, but not the triangle above it, on that specific edge.

Note also that these triangles can also be used to form the barycentric coordinates where is the vertex whose weight we’re grabbing, by taking the opposing edge’s triangle divided by the overall triangle’s determinant:

And then some general barycentric properties:

Therefore:

So if we wish to interpolate a value defined at each vertex as , and at , we would evaluate:

In other words, if we take the integers , and , multiply them against the corresponding values, we get the value at point but still need to divide by . Were we to perform the division, we would lose accuracy due to the round-off and take a performance hit. It should however be possible to postpone taking the division by stepping from one pixel to the next (incrementing by the polynomial coefficients.)

*Potential simplification*: Suppose the value about is actually the z-buffer value, as we’re interpolating it over the triangle. Given that that z-buffer value is always positive (unsigned) – and is a combination of multiplying against determinants, and these determinants turn negative if we wish to reject the pixel as not in the triangle: could we then not simplify the evaluation to be z-buffer only? In this scenario, the polynomials simplify to tracking , , and . We would not need to manage the values separately from the values as the latter principally serves us to get to the “varying” attributes later.

Probably best if we keep the two separate as the values themselves are dependent on the projection matrix.

TODO:

Write out the long form of the determinants , , and . Convert these long form determinants to their polynomials. Show how the polynomial form can be “stepped” across the screen using integer addition only. Show how the polynomial form can be used to “fast-forward” from a current position to the starting edge of triangle (or ending edge for that matter.)

Show how the barycentric coordinates can be used to interpolate the Z-buffer value at each of the three vertices 0, 1 and 2, across the triangle. Given that this is a multiplication followed by a division by the triangle’s overall determinant, deduce a “DDA-ish” or “Bresenham-ish” way of doing the same thing, but without a division, or a one-over-determinant multiplication (that looses accuracy), but instead by running the fraction across numerator and denominator.

## Stepping the z-buffer value

The z-buffer value is different from varying attributes. One, it is already in screenspace whereas the other varying attributes are not (therefore no division is needed, we can interpolate linearly), and the other is the z-buffer contains integer values, not floating point. Consequently, we treat it completely differently and use it for early rejection (avoiding division.) Note however that, if any one of the four bundled fragments pass the z-buffer test, then the others need to enter the fragment shader as well (otherwise no mip-mapping etc.)

Suppose we are at pixel , then to interpolate the z-buffer value :

Where is the barycentric coordinate belonging to vertex .

This works great for floating point, but for integer values like the z-buffer, we must find a way to retain precision, and not divide by the determinant. Expanding the barycentric coordinates (again, definitions as before):

The interesting thing here is that both the numerator and the denominator are integer values, and the denominator is constant across all pixels. We should therefore be able to step them in a Bresenham-like DDA fashion without losing precision.

Let’s elaborate what we need to step:

Reinserting the expansions for , , and into the equation for , we get:

Simplifying to some new determinants (and re-ordering the columns a bit) reveals a beautiful pattern: