## 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:

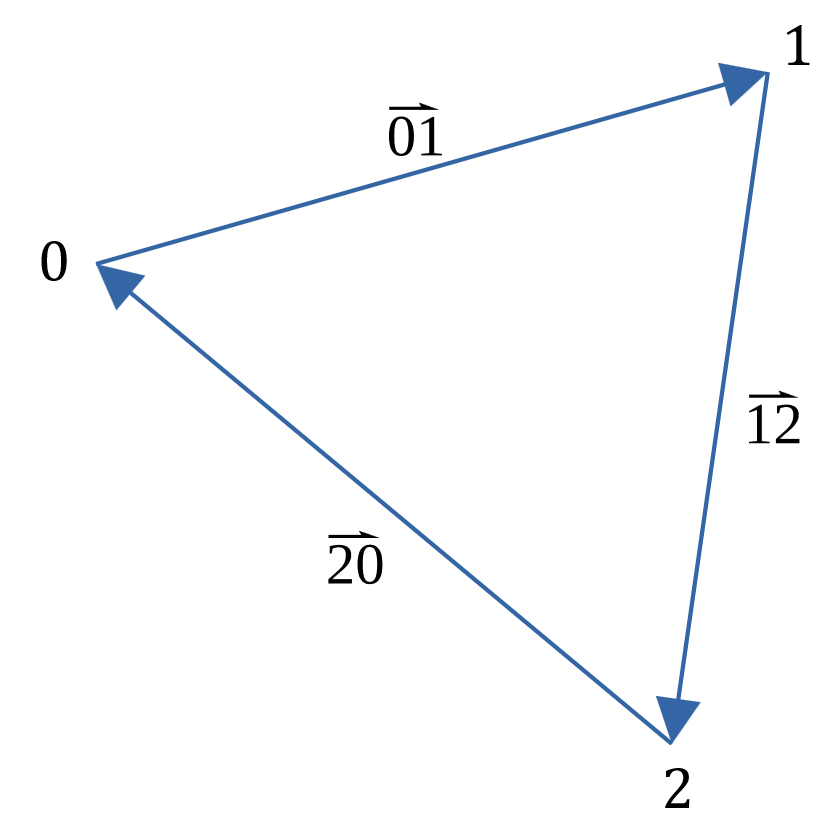
Following the same logic from before:

(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?

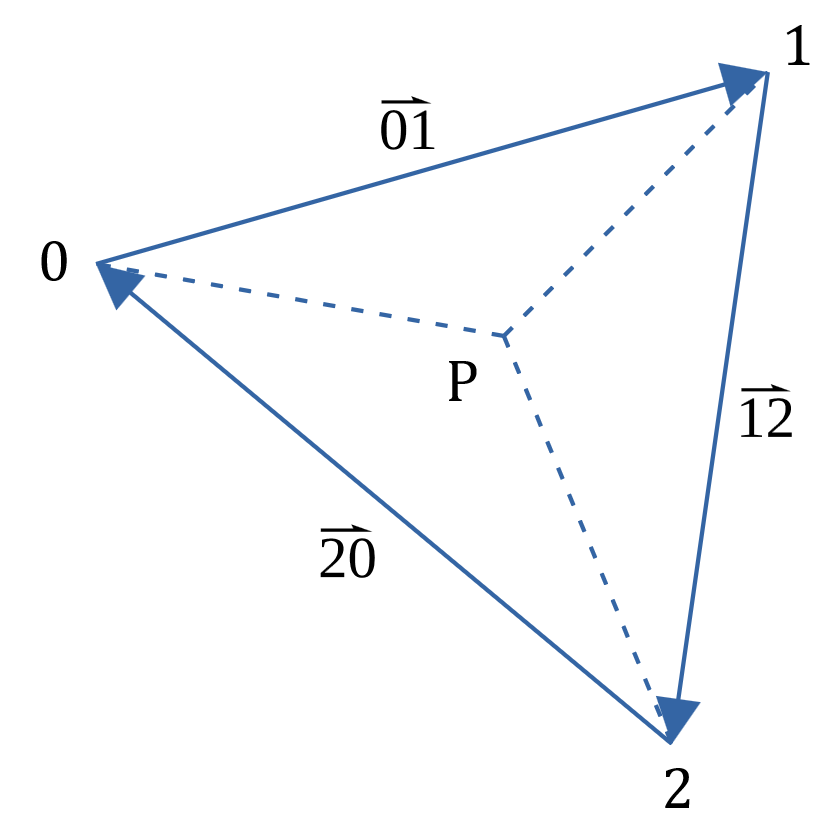
The triangle is represented by 3 vertices, , , and . Between these vertices, there are edges, , , and as the following image illustrates:



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:

Assigning a few names to these:

### Thoughts on stepping a fraction

Let’s say we have some positive integer where at each step we increment by some fraction , where is also positive. We can simplify this down to:

Here, is the numerator (and is the denominator.) We increment by , if this causes then we should and for to retain an accurate value.

While this pretty much follows stepping the numerator over the denominator closely, the comparison is however not ideal, it would be better if we can check for if the numerator does not overrun, and if it does overrun (and should be incremented, and the stepper variable should be updated such that it is positive again.) The benefit here is that the sign bit can then be used for branchless selection. Let’s define the stepper variable as , and take a single step:

And when following that, if :

Let’s say the initial starting value of the numerator is in , then the stepper variable needs to be initialized with “the distance we need to go before we step for the first time” – this is not the same as the numerator:

The is necessary so we step at the right moment (e.g. when for ) while using as the condition and not . Let’s work an example:

We’ll interpolate stepping by at a starting position of . Intuitively, we’ll overflow into on the next step (), consequently, should reflect that. The initial value of :

Therefore, when taking a step:

Checking the condition:

Therefore we increment:

And now the “distance we need to go before we step again” is (again, taking into account we step at , so this is equivalent to saying the numerator is at .)

### Negative steps

The above handles the case where is positive. If is negative, things become arguably easier because counting a fraction down until the is, pretty much, exactly what we’ve been doing in a roundabout way for the positive case.

Consequently if we change to be:

Where is the “increment” value, and is for the positive case, and for the negative case, then we are good on the inner-loop. For initialization, we have (for the positive case):

And for the negative case, since the numerator in r already naturally decrements below 0:

### Changing directions

The above works for stepping a single dimensional space, e.g., we could use this to step along a single horizontal strip of fragments. For the rasterization of a triangle, however, we have two dimensions. Ideally, that second vertical dimension, from one row to the next, would also be a simple set of stepping functions.

Because in the case of interpolating the z-buffer values, the denominator is the same in both the horizontal and vertical directions, we can maintain the same numerator values without loss of precision. It is however possible that one direction is positive (ascending), and the other is negative (or descending). In that case, we need to convert the numerator to the alternative case, for this we can use the formula:

Where is the numerator stepper in the alternative dimension, is the stepper in the old direction and is the denominator shared by both directions. This operation is invertible:

It therefore doesn’t make a difference which of and is ascending or descending, but whether they are the same, or they are different.

### Applying steps to the z-buffer value

Applying the above to the z-buffer, let’s examine the case for the horizontal fractional step :

Note that we assume as otherwise we would have failed the backface test. Consequently, initialization of becomes:

Similarly the increment :

## Bits of precision needed analysis

We’ve previously discussed :

and determined that the bits of precision for is:

Where is the bits needed to represent given number of bits for any coordinate .

Determinants and at first glance appear to have a similar structure:

However, the introduction of a coordinate creates confusion because it does not necessarily have the same number of bits as the and coordinates. E.g. 16, 24 and 32 bits are all common z-buffer bit depths, these would be different from the range of and coordinates.

Let’s start with , and – in other words, will have bits, will have bits and will have bits. Given , and , what is ?

The bits of precision needed for representing the determinant given z-buffer bit depth and coordinate bit depth is therefore:

The next determinant we need to consider is :

We have , , , and – in other words, will have bits, and will each have bits, and will have bits. Given , and , what is ?

The bits of precision needed to represent the determinant given z-buffer depth and coordinate bit-depth therefore is:

While at first we’ll “blit” triangles out in a rectangular fashion, whichever way we do it, there is a point where we first put our pen down and at that have to compute . Equation from before:

Looking at just the numerator, we now have:

The bits of precision needed to represent the numerator of the value at the pixel is therefore:

So, for instance, suppose we’d like to suppose a canvas of pixels, with 4 bits of sub-pixel precision (), and with a 32-bit z-buffer , we would need:

A 73 bit register to represent the numerator of the pixel z-buffer value .