Triangles and Perspective Correct Interpolation

Carl Henrik Ek

March 1, 2019

Abstract

In most types of computer graphics, especially when we are interested in real-time graphics, the geometry describing the world will be built up by triangles. Triangles are the simplest 2D surface that we can describe by only its vertices that is guaranteed to be flat. What this means is that every point on the surface of the triangle share the same normal. As most of the shading that we do is a function of the normal this is a very useful charateristic. What we will do here is to summarise a couple of triangle filling routines and also look at how we can interpolate perspective correct.

1 Filling Triangles

In Rasterisation we will first project down the vertices onto the image plane, each triangle in the three dimensional world will correspond to a triangle in the image plane. In the second stage we now want to fill in these triangles with data. This is something that your rasteriser will spend an a lot of its time doing so its worth to make sure that your filling routine is fast. There are many different methods to do this and we will here look at two different ones, the first one you will implement in the lab and the second one is something that is what your GPU is doing. If you are interested in making your code parallell I would suggest that you have a look at the second.

1.1 Span Tables

Span tables is a very simple method to fill a triangle. What you will do is to first draw the edges of the triangle and then fill each raster-line[1] from top to bottom. One way to do this quickly is to notice that each triangle needs at most to be converted into two "flat" triangles, where at least one side is along a raster-line Figure 1. In order to do so we need to be able to compute the coordinate u_4 on the triangle. This can be done easily by looking at equal triangles using the relationship,

$$\frac{\Delta v_3}{\Delta u_3} = \frac{\Delta v_4}{\Delta u_4},$$

Now as we want to split the triangle such that it becomes two flat ones we know that,

$$v_4 = v_2,$$

this means that we only have one unkown as,

$$\Delta u_4 = u_4 - u_1 = \frac{\Delta v_4}{\Delta v_3}\Delta u_3 = \frac{v_2 - v_1}{v_3 - v_1}(u_3 - u_1),$$

meaning that the unknown value can be calculated as,

$$u_4 = \frac{v_2 - v_1}{v_3 - v_1}(u_3 - u_1) + u_1.$$

Now we can convert a single triangle to two flat ones and render them separetly with a specialised routine that handles this.

[1] the lines on the screen

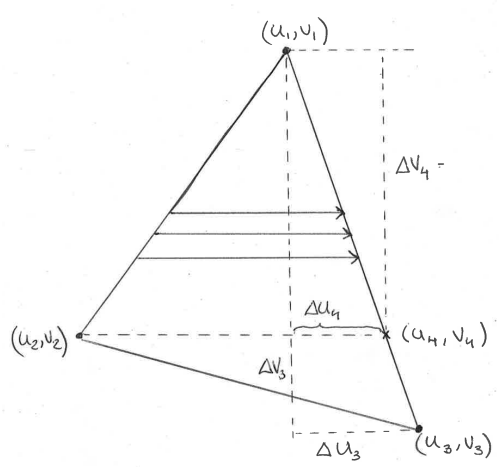

Figure 1: A general triangle can be converted into two triangles that have one side being parallell to a raster-line.

1.2 Triangle Halfplanes

Modern CPU/GPUs are parallell to their core and if we want to write fast code we really should think about how we can avoid doing serial algorithms where we wait for something to be updated. The spantable method is not very parallell as we fill one pixel after another. With the half-plane method what we will try to do is to derive a test function that says if the point is on the triangle or not. It would be silly to test all pixels so what we can do is first create a bounding box around a triangle, this is just the box defined by,

$$[(u_{min}, v_{min}), (u_{min}, v_{max}), (u_{max}, v_{min}), (u_{max}, v_{max})],$$

where,

$$u_{min} = \min[u_0, u_1, u_2] \tag{1}$$
$$u_{max} = \max[u_0, u_1, u_2] \tag{2}$$
$$v_{min} = \min[v_0, v_1, v_2] \tag{3}$$
$$v_{max} = \max[v_0, v_1, v_2]. \tag{4}$$

as shown in Figure 2.

Now we want to derive a simple test function that checks if a point is inside the triangle or not, we will do this with a half-plane method. What this does is that we will make three checks, for each line on the triangle we will test which side the point is on, if it is on the triangle side of all three we know that it is inside the triangle. One way to do this is to use cross products. Now remember the right-hand rule for the cross product Figure 3, now if you change the order of the vectors in the product what you are effectively doing is changing the sign of the resulting vector, this is what we will use to define our half-planes.

The first thing we will do is to pick a starting vertex on the triangle, we will then create a new vector from this point to the next vertex on the triangle. We will also create a vector from this point to the point that we want to check. Now if we take the cross product of these two vectors, if our z-coordinate is going "into the screen", and the point is outside the new vector will have a negative z-value. Importantly the other two dimensions will be zero as the two vectors are both in the (u, v)-plane so we only need to check the sign of the non-zero dimension. Now if the point would be inside instead the sign will be positive. We will call the cross product function our test function, $f_{01}(u, v)$ as can be seen in Figure 4.

Now to test a point we just have to "walk" around the triangle and test the sign in order, importantly you can "bail" out of the test as soon as you find a negative value Figure 5.

This method can be parallellised very easily as each test is completely independent from the other,

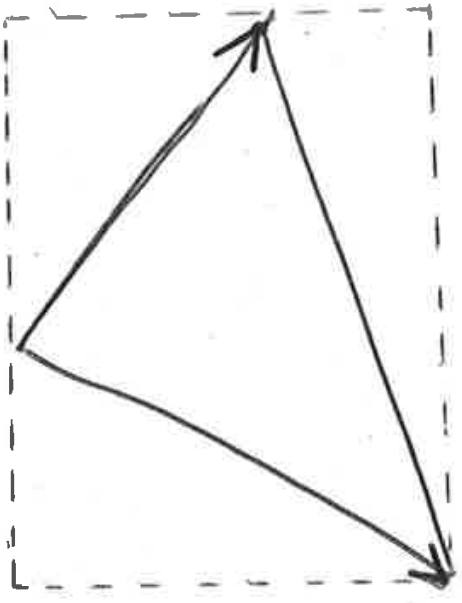

Figure 2: A triangle with its bounding box.

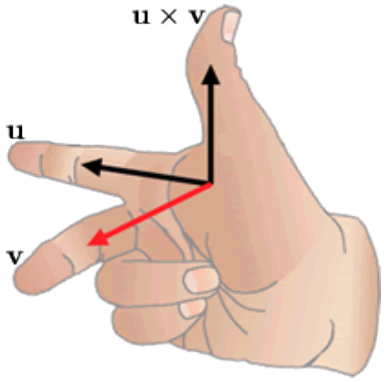

Figure 3: Right hand rule of the cross product

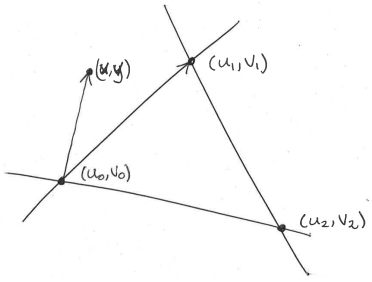

Figure 4: Test function for the first vertex

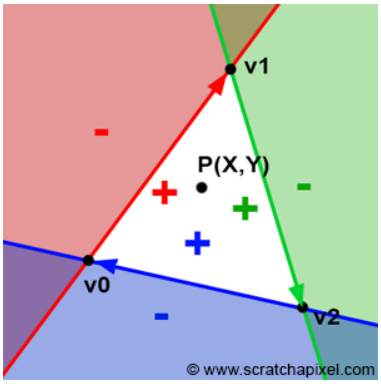

Figure 5: Test function applied to all half-planes

furthermore what is really useful is that we can use the test-function to specify a coordinate on the triangle using something called Barycentric coordinates there is a really good explanation of them here [2].

2 Perspective Correct Interpolation

Now when we have an idea of how to fill a triangle we want to fill it with something interesting. The idea behind rasterisation was that we should try and do as much as possible in screen space rather than doing everything is world space as with the raytracer. Now to make this looks right we need to take perspective into account. Look at Figure 6 what you can see here is that the mid-point on the screen plane does not correspond to the mid-point in the world. Therefore if we interpolate only in the screen space we will get something called a perspective error. What we want to do to avoid this is to do what is called *perspective correct interpolation*. What this means is that we want to interpolate in screen space but importantly move correctly in world space, for each pixel that I jump I want to know which part of the world it corresponds to. Sounds confusing, think about it and look at the image and hopefully it makes sense.

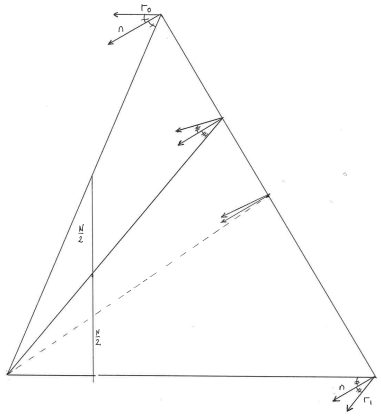

Figure 6: Perspective difference after projection

What we want to do is to write a loop in screen space where we go pixel by pixel, now lets call the variable that we iterate over q, now we want to see how this corresponds to an iteration over a variable t that interpolates along the 3D surface that has been projected Figure 7. To make things a little bit easier we are going to do this for a 2D projection to a 1D line, the extension to 3D is trivial.

[2] https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-rendering-a-triangle/barycentric-coordinates

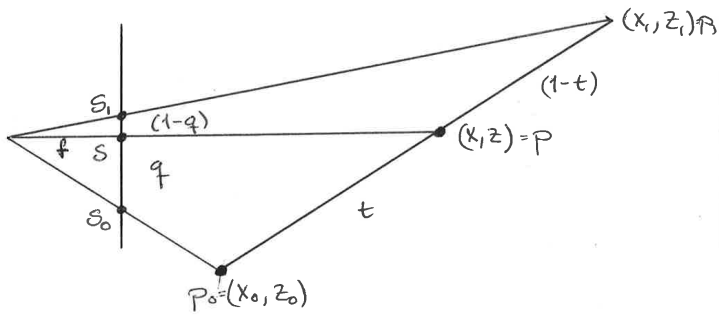

Figure 7: Perspective difference after projection

First lets write up all the points on the screen plane,

$$s = s_0(1-q) + s_1 q = s_0 + q(s_1 - s_0),$$

and all the points on the surface,

$$[x, z]^{\mathrm{T}} = \begin{bmatrix} x_0 \\ z_0 \end{bmatrix}(1-t) + \begin{bmatrix} x_1 \\ z_1 \end{bmatrix} t = \begin{bmatrix} x_0 \\ z_0 \end{bmatrix} + t\left(\begin{bmatrix} x_1 - x_0 \\ z_1 - z_0 \end{bmatrix}\right).$$

Now we know the relationship between the points in the screen plane and the points in the world, its a simple projection as,

$$\left[\; s = \tfrac{x}{z}, \quad s_0 = \tfrac{x_0}{z_0}, \quad s_1 = \tfrac{x_1}{z_1} \;\right].$$

Now lets re-write a general point z in terms of the interpolating quantitites,

$$z = \frac{x}{s} \tag{5}$$

$$z_0 + t(z_1 - z_0) = \frac{x_0 + t(x_1 - x_0)}{s_0 + q(s_1 - s_0)} \tag{6}$$

$$= \frac{s_0 z_0 + t(s_1 z_1 - s_0 z_0)}{s_0 + q(s_1 - s_0)}. \tag{7}$$

Now we are going to manipulate the expression above to try and find an expression of t in terms of q. First lets multiply with the denominator,

$$z_0 s_0 + z_0 q(s_1 - s_0) + t s_0(z_1 - z_0) + t q(z_1 - z_0)(s_1 - s_0) = s_0 z_0 + t(s_1 z_1 - s_0 z_0,$$

and then collect the terms including q and t,

$$t\left(s_0(z_1 - z_0) + q(z_1 - z_0)(s_1 - s_0) - (s_1 z_1 - s_0 z_0)\right) = -q z_0(s_1 - s_0.$$

Now looking at the above we can see that we can break out $(s_1 - s_0)$ from the left-hand side,

$$t(s_1 - s_0)(-z_1 + q(z_1 - z_0)) = -q z_0(s_1 - s_0) \tag{8}$$

$$t(z_1 - q(z_1 - z_0)) = q z_0. \tag{9}$$

This means we can now write the value t which interpolates the 3D space as a function of q which interpolates in the screen space as,

$$t = \frac{q z_0}{z_1 - q(z_1 - z_0)}.$$

Now we are nearlly there, rather than re-writing t in terms of q what we want to do is write z the perspective inducing variable as a function of the interpolation in screen space.

$$z = z_0 + t(z_1 - z_0) \tag{10}$$
$$= z_0 + \frac{qz_0}{z_1 - q(z_1 - z_0)}(z_1 - z_0) \tag{11}$$
$$= z_0 + \frac{qz_1z_0 - qz_0^2}{qz_0 + z_1(1-q)} \tag{12}$$
$$= \frac{qz_0^2 + z_0z_1(1-q) + qz_1z_0 - qz_0^2}{qz_0 + z_1(1-q)} \tag{13}$$
$$= \frac{z_0z_1}{qz_0 + z_1(1-q)} \tag{14}$$
$$= \frac{1}{\frac{qz_0}{z_0z_1} + \frac{z_1(1-q)}{z_0z_1}} \tag{15}$$
$$= \frac{1}{\frac{q}{z_1} + \frac{1-q}{z_0}} \tag{16}$$
$$= \frac{1}{\frac{1}{z_1} + q(\frac{1}{z_1} - \frac{1}{z_0})} \tag{17}$$

Now we are nearly there, lets just do some cleaning up to get the final results,

$$z = \frac{1}{\frac{1}{z_1} + q(\frac{1}{z_1} - \frac{1}{z_0})} \tag{18}$$
$$\Rightarrow \frac{1}{z} = \frac{1}{z_0} + q\left(\frac{1}{z_1} - \frac{1}{z_0}\right). \tag{19}$$

So now we have written the inverse of the depth value z as a linear function of the interpolating value q. What this means is that we can do all our work in screen space and as long as we know the depth value at the vertices we can get the depth value by linearly interpolating its inverse.