Clipping Space

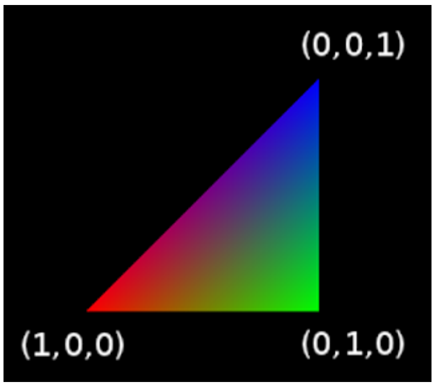
SUNDAY, SEPTEMBER 11, 2011

Single-Pass Wireframe Rendering Explained and Extended

I got a pretty good followup in my previous post with how to implement "Single-Pass Wireframe Rendering". I thought I'd take a second to briefly explain how the edge detection actually worked.

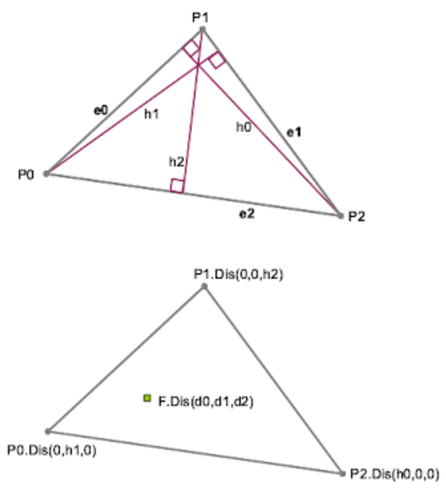
As I said before, the basic idea is that we want each fragment to know about how far it is from any given edge so we can color it appropriately. First, we have to remember that OpenGL is an interpolation engine and it is very good at that. When some attribute is assigned to a vertex and the triangle is rasterized, fragments created inside that triangle get some interpolated value of that attribute depending on how far it is from the surrounding vertices.

Shown below, a triangle has a three-channel attribute assigned to each vertex. In the fragment shader, those values will be some mixture of the three channels. You'll notice, for example that the triangle in the image below gets less red from left to right. The first RGB channel starts at 1, but as one moves towards that channel goes to zero.



That's the basic gist of the attribute interpolation. The nice thing about modern GPUs is it lets us put anything we want in these attributes. They may be used in the fragment shader as colors, normals, texture coordinates, etc. but OpenGL doesn't really care about how you plan on using these attributes as they all get interpolated the same way. In the fragment shader, we can decide how to make sense of the interpolated values.

Finding Edges Programmatically
Notice the pixel values along the top-left area of the triangle above have a low green value because the left and top vertices have no green in them, so pixels moving towards that edge have less and less green. Similarly, the right side of the triangle has pretty much no red in it, because the values in the vertices above and below it have no red. The same holds true for the bottom edge of the triangle having no blue. The insight to be gained here and which is used in "Single-Pass Wireframe Rendering" is that values along the edges of the triangle will have a very low value in at least one of the three channels. If ANY of the channels is close to zero, that fragment is sitting on or very near an edge.



images taken from nVidia's Solid Wireframe paper

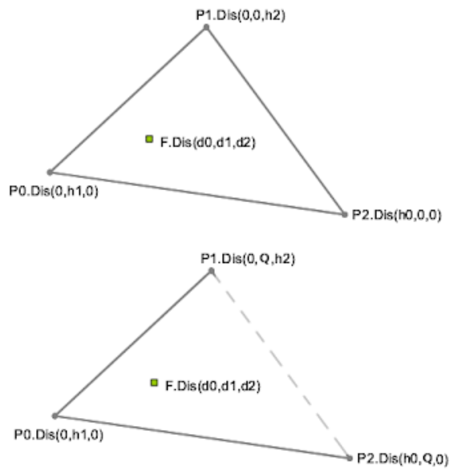
We could just assign similar values as here and just render edges if the value is below some threshold. The problem, though, is that these values aren't in viewport space and we probably want to measure our line thickness in terms of pixels. Otherwise the edge thickness on our screen would change depending on the size of the triangle (maybe you want that, whatever).

As shown in the picture above, we calculate the altitudes of each vertex in screen space and store them in some vertex attribute. In the fragment shader, the value (d0,d1,d2) will be somewhere between the three vertex attributes. As described above, if any of these channels d0, d1 or d2 is close to zero, that means we're sitting on an edge.

nVidia has an excellent paper called Solid Wireframe, which goes into a bit more detail how this works and provides some really great illustrations.

Excluding Edges on polygons
While rendering edges is nice, I may not want every edge of a given triangle to be rendered. For example, if I have some five-sided polygon concave polygon that I break into triangles using some technique like ear clipping (pdf), I may not want the interior edges inside the polygon to be rendered.

A simple way to exclude an edge of a given polygon is to make sure that that value never goes to zero by setting the channels of the other vertices to some high amount *Q*. This *Q* can be any value higher than your maximum edge-width. In my program, I set it to 100 since I'll probably never be drawing edges thicker than that.



If *Q* is relatively high, fragments along that edge will not have low values in any channel

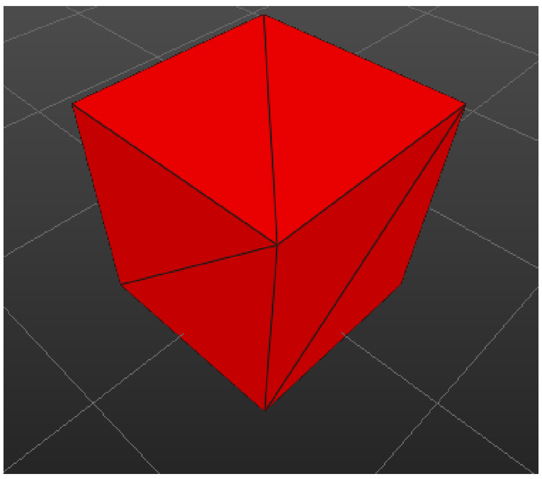
Designating which edges to exclude requires an additional vertex attribute sent down from the program. I attach a float to each vertex with a 0 or 1 whether or not I want to exclude that edge from being rendered. I then update my geometry shader accordingly.

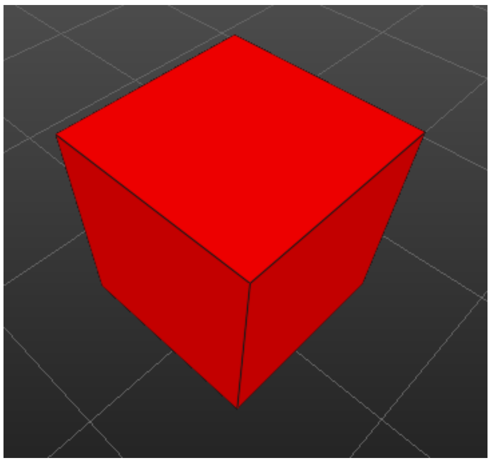
my updated vertex shader...

```
#version 120
#extension GL_EXT_gpu_shader4 : enable
in vec3 vertex;
in vec4 color;
in vec3 normal;
in float excludeEdge;
varying vec3 vertWorldPos;
varying vec3 vertWorldNormal;
varying float vertExcludeEdge;
uniform mat4 objToWorld;
uniform mat4 cameraPV;
uniform mat4 normalToWorld;
void main() {
vertWorldPos = (objToWorld * vec4(vertex,1.0)).xyz;
vertWorldNormal = (normalToWorld * vec4(normal,1.0)).xyz;
gl_Position = cameraPV * objToWorld * vec4(vertex,1.0);
vertExcludeEdge = excludeEdge;
gl_FrontColor = color;
}
```

and my updated geometry shader...

```
#version 120
#extension GL_EXT_gpu_shader4 : enable
#extension GL_EXT_geometry_shader4 : enable
varying in vec3 vertWorldPos[3];
varying in vec3 vertWorldNormal[3];
varying in float vertExcludeEdge[3];
varying out vec3 worldNormal;
varying out vec3 worldPos;
uniform vec2 WIN_SCALE;
noperspective varying vec3 dist;
void main(void)
{
float MEW = 100.0; // max edge width
// adapted from 'Single-Pass Wireframe Rendering'
vec2 p0 = WIN_SCALE * gl_PositionIn[0].xy/gl_PositionIn[0].w;
vec2 p1 = WIN_SCALE * gl_PositionIn[1].xy/gl_PositionIn[1].w;
vec2 p2 = WIN_SCALE * gl_PositionIn[2].xy/gl_PositionIn[2].w;
vec2 v0 = p2-p1;
vec2 v1 = p2-p0;
vec2 v2 = p1-p0;
float area = abs(v1.x*v2.y - v1.y * v2.x);
dist = vec3(area/length(v0),vertExcludeEdge[1]*MEW,vertExcludeEdge[2]*MEW);
worldPos = vertWorldPos[0];
worldNormal = vertWorldNormal[0];
gl_Position = gl_PositionIn[0];
EmitVertex();
dist = vec3(vertExcludeEdge[0]*MEW,area/length(v1),vertExcludeEdge[2]*MEW);
worldPos = vertWorldPos[1];
worldNormal = vertWorldNormal[1];
gl_Position = gl_PositionIn[1];
EmitVertex();
dist = vec3(vertExcludeEdge[0]*MEW,vertExcludeEdge[1]*MEW,area/length(v2));
worldPos = vertWorldPos[2];
worldNormal = vertWorldNormal[2];
gl_Position = gl_PositionIn[2];
EmitVertex();
EndPrimitive();
}
```


Without edge removal, each triangle has its own edge


Triangle mesh rendered excluding certain edges

Posted by strattonbrazil at 8:10 AM

5 comments:

Draco18s said...

Hmm, old old blog post.
Anyway, how is it that you go about setting the per-vertex information to exclude the edges? Did you embed it in the mesh data?

2/04/2016 7:50 PM

Unknown said...

This comment has been removed by the author.

2/05/2016 4:58 AM

Unknown said...

I used the concept <a href="https://dl.dropboxusercontent.com/u/70097519/UNITY/bezier_on_gpu/bezier_on_gpu.html>here</a> you can see the demo in browser and/or download the source package.

2/05/2016 5:28 AM

Anonymous said...

I would also like to know Draco18s answer :)

2/21/2016 8:20 PM

Evan said...

What other shapes can it do?

Regards,

Shalin @ Creately

9/28/2016 3:04 AM

Post a Comment

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)