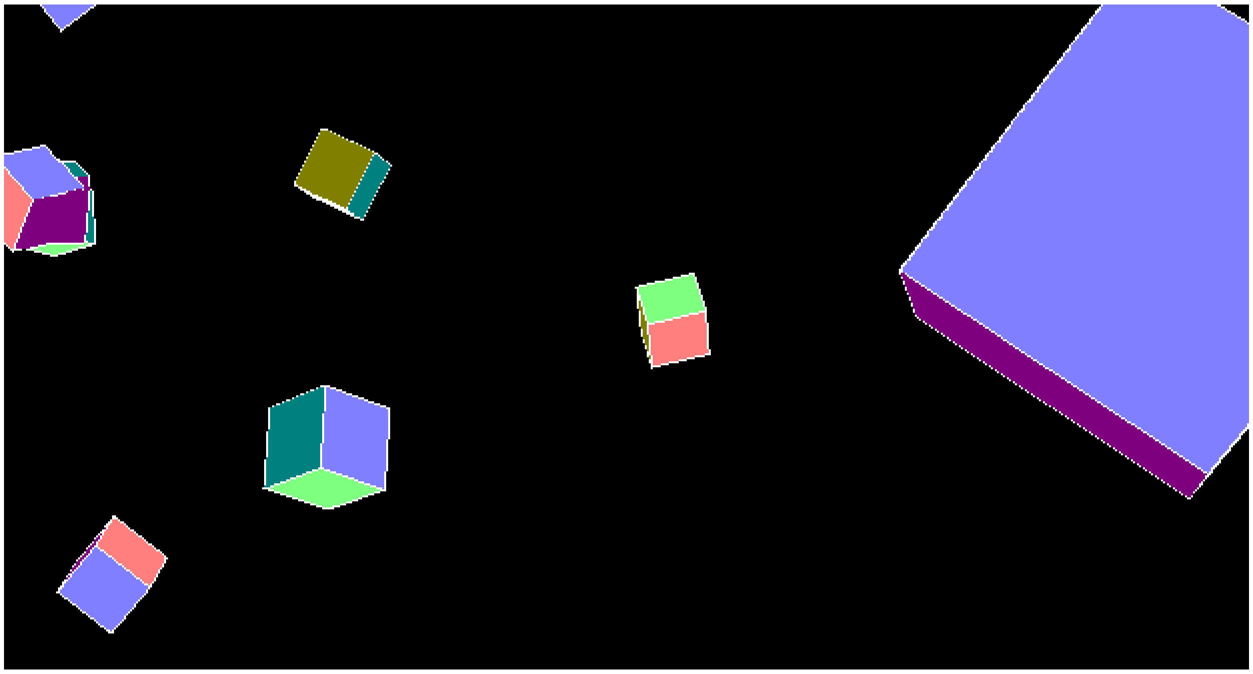
20th December 2014

Urho Shaders. Edge Detection.

Urho3D shading

Trying to create an edge detection shader with one pixel precision.It has taken some time to understand the urho way to write shaders.



The Pipeline

1. RenderPath
2. Technique
3. Material
4. Shader

The Code

RenderPath

Forward.xml

```
<renderpath>
    <command type="clear" color="fog" depth="1.0" stencil="0" />
    <command type="scenepass" pass="base" vertexlights="true" metadata="base" />
    <command type="scenepass" pass="refract">
        <texture unit="environment" name="viewport" />
    </command>
</renderpath>
```

The first command "clear" erases the last frame drawn.
The second command is our first render pass, called "base"
The third and last command is a second render pass, that is called refract. It has a texture element that is using the

viewport. In this case it should be using the first rendered pass. No refraction is going to occur, but the same idea where we will be sampling the rendered buffer to draw the outline.

Technique

simple.xml

```
<technique vs="simple" ps="simple" >
    <pass name="base" vsdefines="BASE" psdefines="BASE"/>
    <pass name="refract" vsdefines="EDGE" psdefines="EDGE" blend="alpha"/>
</technique>
```

This is where things get a little weird. The relationship between the Technique the Material and the Shader can be a little confusing. For the entirety of this technique we will be using the same shader "simple". You can however, define a shader for each pass specifically if desired. To make things more confusing, the vsdefines, and psdefines allows you to create specific code in the shader based on those defines. Notice here, there are "BASE" and "EDGE". That will come up again in the shader file.

Material

simple.xml

```
<material>
    <technique name="Techniques/research/simple.xml" />
</material>
```

There are no textures, or any other variables being sent here. Only defining the technique to use. At a later time, it might help to send along some extra object specific data to assist in overlapping colors.

Shader

simple.glsl

```
#include "Uniforms.glsl"
#include "Samplers.glsl"
#include "Transform.glsl"
#include "ScreenPos.glsl"

#ifdef BASE
    varying vec4 vColor;
#endif
#ifdef EDGE
    varying vec4 vScreenPos;
#endif

#ifdef COMPILEPS
float color_difference(in vec4 sc, in vec4 nc){
  return abs(sc.r-nc.r)+abs(sc.g-nc.g)+abs(sc.b-nc.b);
}

vec4 get_pixel(in sampler2D tex, in vec2 coords, in float dx, in float dy) {
 return texture2D(tex,coords + vec2(dx, dy));
}

// returns pixel color
float IsEdge(in sampler2D tex, in vec2 coords){
  float dxtex = 1.0 / 1920.0; //image width;
  float dytex = 1.0 / 1080.0; //image height;
  float cd[8];

  vec4 sc = get_pixel(tex,coords,float(0)*dxtex,float(0)*dytex);
  cd[0] = color_difference( sc, get_pixel(tex,coords,float(-1)*dxtex,float(-1)*dy
  cd[1] = color_difference( sc, get_pixel(tex,coords,float(-1)*dxtex,float(0)*dyt
  cd[2] = color_difference( sc, get_pixel(tex,coords,float(-1)*dxtex,float(1)*dyt
  cd[3] = color_difference( sc, get_pixel(tex,coords,float(0)*dxtex,float(1)*dyte

  vec4 alt1 = get_pixel(tex,coords,float(1)*dxtex,float(1)*dytex);
  vec4 alt2 = get_pixel(tex,coords,float(1)*dxtex,float(0)*dytex);
  vec4 alt3 = get_pixel(tex,coords,float(1)*dxtex,float(-1)*dytex);
  vec4 alt4 = get_pixel(tex,coords,float(0)*dxtex,float(-1)*dytex);

  if( length(alt1.rgb) < 0.1 ){ cd[4] = color_difference( sc, alt1 ); }else{ cd[4
  if( length(alt2.rgb) < 0.1 ){ cd[5] = color_difference( sc, alt2 ); }else{ cd[5
  if( length(alt3.rgb) < 0.1 ){ cd[6] = color_difference( sc, alt3 ); }else{ cd[6
  if( length(alt4.rgb) < 0.1 ){ cd[7] = color_difference( sc, alt4 ); }else{ cd[7

  return cd[0]+cd[1]+cd[2]+cd[3]+cd[4]+cd[5]+cd[6]+cd[7];
}
#endif

void VS(){
    mat4 modelMatrix = iModelMatrix;
    vec3 worldPos = GetWorldPos(modelMatrix);
    gl_Position = GetClipPos(worldPos);

    #ifdef EDGE
      vScreenPos = GetScreenPos(gl_Position);
    #endif

    #ifdef BASE
        vec3 n = iNormal+vec3(1.0);
        n*=0.5;
        vColor = vec4(n,1.0);
    #endif
}

void PS(){
    #ifdef BASE
        vec4 diffColor = vColor;
        gl_FragColor = diffColor;
    #endif

    #ifdef EDGE
      vec4 color = vec4(0.0,0.0,0.0,1.0);
      if(IsEdge(sEnvMap,vScreenPos.xy / vScreenPos.w)>=2.0){
        color.rgba = vec4(1.0);
        //color = get_pixel(sEnvMap,vScreenPos.xy / vScreenPos.w,float(0)*(1.0/19
      }
      gl_FragColor = color;
    #endif
}
```

This is where all the heavy lifting is done. There is a little to go over, and a lot of this is very Urho3D specific. It is glsl, but modified in a way that Urho3D expects it. Both the vertex and the fragment/pixel shader are all together. The vertex specific code in VS() get converted behind to scenes to the vertex program's main() function. And the same applies to PS() to the pixel program's main(). Notice the "ifdef COMPILEPS". When Urho3D does what it does to the shader code, this tells it to compile that block to the pixel shader code specifically. The same goes for "ifdef COMPILEVS", although you will not see that block in this shader code. You will also notice the return of the "BASE" and the "EDGE" defines from the technique inside the PS() and VS() functions. Those defines from the technique, allow you to run only parts of each shader, the parts in the ifdef block.

To break this down further. At the very top are includes that are part of the Urho3D core shader files. It does some of the work for you as far as easy access to textures and screenspace, the latter we use for the actual edge drawing.

When the "Base" pass is rendered, we only care about varying vColor. Which in the vertex program we are setting to the normal. Later we might want to also have a randomized color setn in per object to mix with it to deal with same color over lap. Because in the pixel program, we are renderig that color as a solid. And if we have green over green, we can not determine a line there later on. But just a small deviation will allow the edge portion of the shader to find that line.

After the "Base" pass, the "Edge" pass is using the "Base" buffer to do a much simplified edge detection. The idea started with this blog post:http://coding-experiments.blogspot.com/2010/06/edge-detection.html [http://coding-experiments.blogspot.com/2010/06/edge-detection.html] but landed in a similar but different place. Since we are not operating on an image, but a very specifically rendered 3D scene where we expect very flat colors to differentiate between. The code is not optimized, just quickly prototyped to get a basic working shader together.

It's not a perfect solution yet. And you can see some simple issues that still need to be resolved. Over lapping colors, some lines are not drawn based on the angle. Some lines, based on the angle done seem to fill in enough. As the shader improves, I will try to make updates here. Also going over more specifically what the shader is doing.

Posted 20th December 2014 by jimmy gass

Labels: glsl, shaders, urho

1 View comments

Anonymous February 27, 2017 at 8:47 PM

any links to the code?

Reply

Enter your comment...

Comment as: bashkirov.serge ▼ Sign out

Publish Preview ☐ Notify me