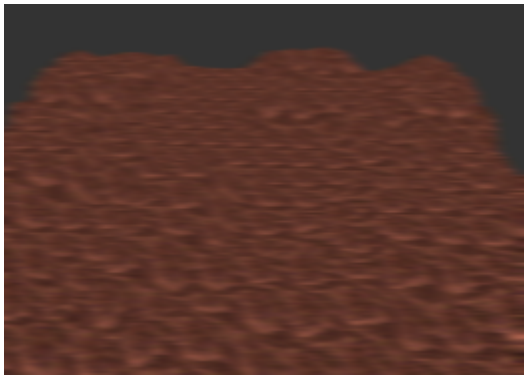
Tutorial 10: Post Processing



Summary

In modern games, the actual rendering of geometry to a buffer is not the end of the rendering process. Additional post processing techniques are applied to the buffer before it is displayed, such as tone mapping, bloom, and blurring. This tutorial will outline the basic techniques required to perform such post processing in your graphical applications.

New Concepts

Frame Buffer Objects, multiple render targets, Render-To-Texture, Post processing, ping-pong texturing

Frame Buffer Objects

As well as the front and back buffers you are used to rendering to, OpenGL supports rendering to multiple destinations, known as *Frame Buffer Objects*. These FBOs contain a number of individual textures known as *attachments*, that contain colour, depth, and stencil information. The geometry you are rendering can be redirected to be drawn either to one of these FBOs, or the 'normal' rendering target. You can think of the standard back, depth, and stencil buffer combination as being an inbuilt 'default' FBO that is always available.

Each FBO may have *optional* depth, stencil, and colour components, but *must* have *at least* one attachment - what would it render to otherwise! OpenGL FBOs support *multiple rendering targets* - that is, a single fragment shader can write to multiple colour attachments. The number of colour attachments supported is dependent on your graphics hardware, but may be as many as 8 separate attachments. Why would you want to render to multiple colour textures? Well, you might want to store the displayed scene's normals on a per pixel basis in order to do some extra processing on them, or write fragments above a certain luminance to a separate render target in order to blur them at a later stage. That's the beauty of FBO multiple render targets - whatever rendering information you think you might need, you can get on a per pixel basis, and store into a texture.

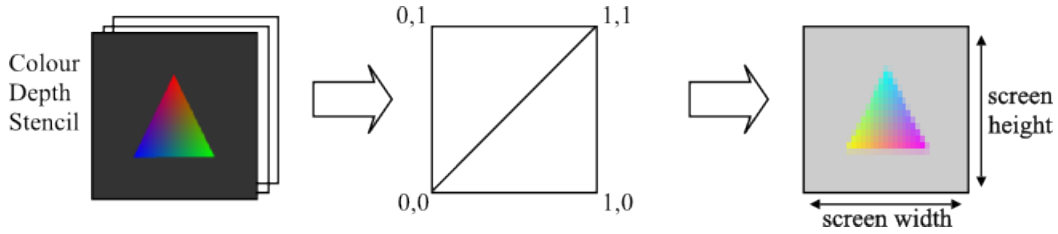
These attachments are literally just normal OpenGL 2D textures, and so may be accessed as such. So you could, for example, render one frame into an FBO, and then use that FBO's colour attachment as a texture in the next frame's rendering! There are a few limitations to the textures you attach to your FBOs, however. The textures you use as attachments cannot be of a compressed type, which would be undesirable anyway, due to the overhead in compressing and recompressing pixel data as

geometry is rendered. Also, depending on the exact extensions supported by the graphics hardware, all of the attachments may need to be of the same dimensions.

While FBOs do not support *compressed* textures, they *do* support *packed* textures. When emulating the usual graphics pipeline with FBOs, it is usual to have a 32bit colour attachment, to provide 8 bits for r,g,b and a channels, and to have a 24bit z-buffer. Some graphics hardware though, requires all attachments to be of the same bit-depth. One easy solution is to just have a 32bit z-buffer and utilise the extra precision that brings, but what if you also wanted a stencil attachment? 32bits of stencil data per pixel is probably overkill! This is where texture packing comes in handy. Instead of separate stencil and depth textures, it is possible to have a combined packed texture, with 24 bits specified for depth information, and 8 bits for stencil information.

Post Processing

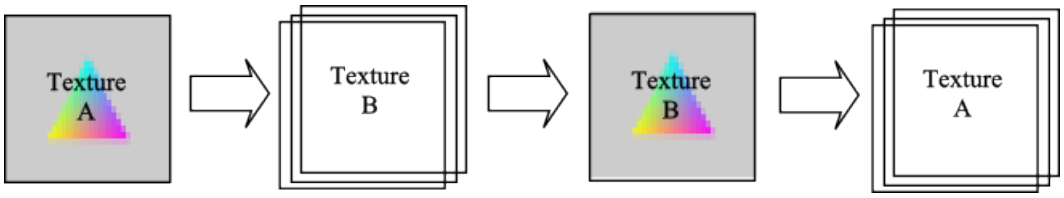
Being able to render an entire scene into an accessible texture opens up a wide range of post processing techniques, such as blurs, image sharpening, cell shading, motion blur, and so on. Any operation you can do on an image in *Photoshop*, you could write as a fragment shader to act upon your FBO attachments. The usual method of performing a post processing operation in OpenGL is by rendering a single textured quad that fills the screen, with the FBO colour attachment bound as its texture, and some form of special post processing shader applied. The results of the post process shader will fill the screen, and look just as if the scene geometry had been rendered straight to the back buffer - except of course, with the post process applied to it. Of course, when rendering one of these 'post process' quads, the results could be rendered into *another* FBO, which can then be used as an input into *another* post processing shader!



By rendering the scene into an FBO, and binding the resulting colour texture to a quad, the quad can be rendered to fill the screen, with a post processing shader applied

Ping-pong texturing

A variation on the above process for rendering multiple post processing effects is *ping-pong texturing*. This is achieved by using two colour textures, but alternating between using them as inputs and outputs in post processing stages - this can be seen as an extension of how the front and back buffers of the rendering process are used. This can be used to successively apply the same post processing effect, such as a blur, to the scene, or to apply numerous additional post processing shaders - no matter how many post processes are required, only an input and an output texture are required, flipping between them.



A post-process quad with texture A is rendered into texture B via an FBO. Texture B is then rendered via another post-process quad into texture A, ready for another repetition of the process

Example Program

To show how to achieve post processing using FBOs, we're going to write a simple example program. This program will render last tutorial's heightmap into an FBO, and then use that FBOs colour attachment as input into a simple blur post processing effect. Then, we'll do some 'ping pong' texturing passes to effectively increase the strength of the blur. Such blur filters are sometimes combined with another type of post-processing, edge detection filters, to blur the jagged edges of polygons, as a form of antialiasing. We'll be writing another new fragment shader in this tutorial, so create a file called *processfrag.glsl* in your *Shaders* Visual Studio project folder.

Renderer header file

In our header file, we're going to define three new **protected** functions to split our rendering up - one to draw the heightmap, one to perform the post processing, and one to draw the result of the post processing on screen. As for variables, we have a new **define** - *POST_PASSES*, which will determine how many times the blur stage is repeated. We also have a *Camera*, and two pointers to *Shaders* this time - one for normal rendering using the texturing shader we wrote earlier in the series, and one for post processing, using a new fragment shader. We also have two *Meshes* - a heightmap, and a quad to draw our post processed scene onto the screen. Finally, we have two Frame Buffer Objects and their attachment textures - one to render the heightmap into, and one to perform the post process blur stage.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/HeightMap.h"
5  #include "./nclgl/Camera.h"
6
7  #define POST_PASSES 10
8
9  class Renderer : public OGLRenderer    {
10 public:
11    Renderer(Window &parent);
12    virtual ~Renderer(void);
13
14    virtual void RenderScene();
15    virtual void UpdateScene(float msec);
16
17 protected:
18    void        PresentScene();
19    void        DrawPostProcess();
20    void        DrawScene();
21
22    Shader*     sceneShader;
23    Shader*     processShader;
24
25    Camera*     camera;
26
27    Mesh*       quad;
28    HeightMap*  heightMap;
29
30    GLuint      bufferFBO;
31    GLuint      processFBO;
32    GLuint      bufferColourTex[2];
33    GLuint      bufferDepthTex;
34 };
```

renderer.h

Renderer Class file

Our **constructor** begins in a fairly standard way - we create a new *Camera*, and quad, and create the *HeightMap* just as we did in tutorial 6. This time we have *two* shaders, so we create them both here, then *Link* them both, as usual. We also set the heightmap's texture to be repeating, just as before.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     camera    = new Camera(0.0f,135.0f,Vector3(0,500,0));
5     quad      = Mesh::GenerateQuad();
6
7     heightMap = new HeightMap(TEXTUREDIR"terrain.raw");
8     heightMap->SetTexture(
9     SOIL_load_OGL_texture(TEXTUREDIR"Barren Reds.JPG",
10    SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
11
12    sceneShader    = new Shader(SHADERDIR"TexturedVertex.glsl",
13                                SHADERDIR"TexturedFragment.glsl");
14    processShader  = new Shader(SHADERDIR"TexturedVertex.glsl",
15                                SHADERDIR"processfrag.glsl");
16
17    if(!processShader->LinkProgram() || !sceneShader->LinkProgram() ||
18       !heightMap->GetTexture()) {
19       return;
20    }
21
22    SetTextureRepeating(heightMap->GetTexture(),true);
```

renderer.cpp

Now for some new stuff! We need to generate three textures - a depth texture, and two colour textures, to use as attachments for our two FBOs. Refer back to appendix A of tutorial 3 if you can't remember how to create textures. Note how each texture has dimensions equal to the screen size! We *could* make it less if we wanted, but that'd result in a lower quality final image, as we'd have less than one FBO texel per screen pixel, resulting in interpolation. As we don't want any filtering to take place, we turn off min and mag filtering, and additionally clamp the texture - this is just to make sure no sampling takes place that might distort the screen edges. Note how on line 40, we use *GL_DEPTH_STENCIL* as the texture format, and *GL_UNSIGNED_INT_24_8* as the texture size - this is how to define the *packed* texture format described earlier, giving us a total of 32 bits per pixel on both the colour and depth attachment textures. Wo won't actually be needing the stencil buffer in this tutorial, but it's worthwhile knowing how to set up packed formats, and stencil buffers are often used to selectively apply post processing effects to only part of the screen.

```
23    //Generate our scene depth texture...
24    glGenTextures(1, &bufferDepthTex);
25    glBindTexture(GL_TEXTURE_2D, bufferDepthTex);
26    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
27    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
28    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
29    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
30    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH24_STENCIL8,  width, height,
31                 0, GL_DEPTH_STENCIL, GL_UNSIGNED_INT_24_8, NULL);
```

renderer.cpp

```
32    //And our colour texture...
33    for(int i = 0; i < 2; ++i) {
34       glGenTextures(1, &bufferColourTex[i]);
35       glBindTexture(GL_TEXTURE_2D, bufferColourTex[i]);
36       glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
37       glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
38       glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
39       glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
40       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8,  width, height, 0,
41                    GL_RGBA, GL_UNSIGNED_BYTE, NULL);
42    }
```

renderer.cpp

Just like with textures and shader programs, we *generate* new Frame Buffer Objects, using the OpenGL command **glGenFramebuffers**. We then *bind* an FBO, and attach textures to it to render into - we'll just attach textures to the *bufferFBO* for now, as our post process FBO 'ping pongs' its attachments. Finally we *unbind* the FBO, enable depth testing, and set *init* to true.

```
43    glGenFramebuffers(1,&bufferFBO); //We'll render the scene into this
44    glGenFramebuffers(1,&processFBO);//And do post processing in this
45
46    glBindFramebuffer(GL_FRAMEBUFFER, bufferFBO);
47    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
48                           GL_TEXTURE_2D,bufferDepthTex, 0);
49    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_STENCIL_ATTACHMENT,
50                           GL_TEXTURE_2D, bufferDepthTex, 0);
51    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
52                           GL_TEXTURE_2D, bufferColourTex[0], 0);
53    //We can check FBO attachment success using this command!
54    if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
55    GL_FRAMEBUFFER_COMPLETE || !sceneDepthTex || !sceneColourTex[0]) {
56       return;
57    }
58    glBindFramebuffer(GL_FRAMEBUFFER, 0);
59    glEnable(GL_DEPTH_TEST);
60    init = true;
```

renderer.cpp

In our **destructor**, we **delete** anything we created, including the FBO attachment textures, and the FBO itself.

```
61 Renderer::~Renderer(void)  {
62    delete sceneShader;
63    delete processShader;
64    currentShader = NULL;
65
66    delete heightMap;
67    delete quad;
68    delete camera;
69
70    glDeleteTextures(2,bufferColourTex);
71    glDeleteTextures(1,&bufferDepthTex);
72    glDeleteFramebuffers(1,&bufferFBO);
73    glDeleteFramebuffers(1,&processFBO);
74 }
```

renderer.cpp

The *UpdateScene* function is the same as tutorial 6 onwards, while the *RenderScene* function is much shorter, as the rendering will be done by the three functions *DrawScene*, *DrawPostProcess*, and *PresentScene*. We do call *SwapBuffers* here, though.

```
75 void Renderer::UpdateScene(float msec) {
76    camera->UpdateCamera(msec);
77    viewMatrix = camera->BuildViewMatrix();
78 }
79
80 void Renderer::RenderScene()  {
81    DrawScene();
82    DrawPostProcess();
83    PresentScene();
84    SwapBuffers();
85 }
```

renderer.cpp

To perform post processing using FBOs, we must first render the scene into a texture. The *DrawScene* function will do just that, by binding the *bufferFBO* Frame Buffer Object, and rendering the heightmap into its first colour attachment - *sceneColourTex[0]*. The call to **glBindFramebuffer** on line 87 changes the active rendering target to our new FBO, followed by a call to **glClear** - this function will work just as it does for the normal rendering output, and will clear the currently attached FBO textures. We then set up the matrices, *bind* the texturing shader and draw the heightmap, before finally clearing up by *unbinding* both the shader and the FBO.

```
86 void Renderer::DrawScene() {
87    glBindFramebuffer(GL_FRAMEBUFFER, bufferFBO);
88    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT |
89            GL_STENCIL_BUFFER_BIT);
90
91    SetCurrentShader(sceneShader);
92    projMatrix     = Matrix4::Perspective(1.0f,10000.0f,
93                                (float)width / (float)height, 45.0f);
94    UpdateShaderMatrices();
95
96    heightMap->Draw();
97
98    glUseProgram(0);
99    glBindFramebuffer(GL_FRAMEBUFFER, 0);
100 }
```

renderer.cpp

DrawScene rendered our heightmap scene into the FBO attachments, now to perform the post processing operation on it! To do this we're going to use *another* FBO, but, instead of rendering the heightmap, we're going to render a single screen-sized quad, using a special post processing fragment shader. If you remember back to earlier in the tutorial series when we created the *Mesh* class function *GenerateQuad*, you'll recall that the quad had vertex positions from -1.0 to 1.0 on the x and y axis. So, the easiest way to draw a quad that will fill the screen is to make the *view* matrix an identity matrix, and the *projection* matrix an orthographic matrix that goes from -1.0 to 1.0 on each axis.

Like *DrawScene*, we begin by binding the post-processing FBO, and then attaching *bufferColourTex[1]* to it, and clearing it. Then we switch to the post processing shader, and set up the orthographic projection. We're going to be drawing a quad on screen multiple times, so we *disable* depth testing temporarily, so the quad will always be drawn, overwriting the previous contents of the FBO colour attachment.

Before we render the quad using our *processShader* program, we must set a **uniform** variable in the fragment shader. It's quite common for post processing fragment shaders to sample a bound texture across multiple adjacent texels, and to do this it needs to know how large each texel is. As texture coordinates are measured from 0.0 to 1.0, to get the texel size, we divide 1 by the width and height of the texture - which in this case, will be the dimensions of the screen, as we're going to bind a screen-sized texture to the quad we're going to draw.

```
101 void Renderer::DrawPostProcess() {
102    glBindFramebuffer(GL_FRAMEBUFFER, processFBO);
103    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
104                           GL_TEXTURE_2D, bufferColourTex[1], 0);
105    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
106
107    SetCurrentShader(processShader);
108    projMatrix = Matrix4::Orthographic(-1,1,1,-1,-1,1);
109    viewMatrix.ToIdentity();
110    UpdateShaderMatrices();
111
112    glDisable(GL_DEPTH_TEST);
113
114    glUniform2f(glGetUniformLocation(currentShader->GetProgram(),
115                "pixelSize"), 1.0f / width,1.0f / height );
```

renderer.cpp

Next, is the actual post processing effect stage. The fragment shader is going to perform a 2-pass gaussian blur - 1 pass to blur vertically, and 1 pass to blur horizontally. Both passes are performed by the same shader, switched between by the **uniform int** *isVertical*. We're also going to 'ping pong' between colour buffers, so that vertical the horizontal blurring stage is performed first, and then the result of that stage is then vertically blurred. Finally, this is placed inside a simple **for** loop, to show how the blur effect can be progressively built up over multiple passes.

So, we bind *bufferColourTex[1]* as a colour attachment (line 117), bind *bufferColourTex[0]* as a texture (line 122), and run the post-process shader on a full-screen quad (line 123). Then, we swap around the *bufferColourTex* textures (lines 127 and 130), so that the first stage output is used as the second stage input. This is then used as the input texture of the next iteration of the **for** loop, building up the blur effect.

```
116    for(int i = 0; i < POST_PASSES; ++i)   {
117       glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
118                              GL_TEXTURE_2D, bufferColourTex[1], 0);
119       glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
120                                        "isVertical"), 0);
121
122       quad->SetTexture(bufferColourTex[0]);
123       quad->Draw();
124 //Now to swap the colour buffers, and do the second blur pass
125       glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
126                                        "isVertical"), 1);
127       glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
128                              GL_TEXTURE_2D, bufferColourTex[0], 0);
129
130       quad->SetTexture(bufferColourTex[1]);
131       quad->Draw();
132    }
```

renderer.cpp

With the post processing over, we simply unbind the FBO and shader, and re-enable depth testing.

```
133     glBindFramebuffer(GL_FRAMEBUFFER, 0);
134     glUseProgram(0);
135 
136     glEnable(GL_DEPTH_TEST);
137 }
```

renderer.cpp

At the end of post processing, the final blurred image is held in the *bufferColourTex[0]* texture. So, to draw that texture on screen, we draw another screen sized quad, but this time not to a FBO colour target, but instead straight to the screen. As this is the only rendering stage that draws to the screen, we wait until now to *glClear* the back buffer, then bind the basic texturing shader, set the screen-sized quad orthographic projection, bind the correct texture, and draw the full screen quad. With that, the blurred image of the heightmapped terrain should be on screen.

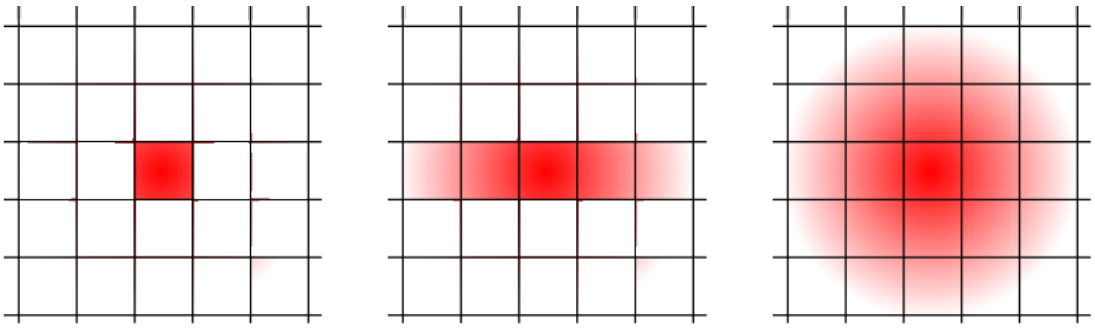
```
138 void  Renderer::PresentScene()  {
139     glBindFramebuffer(GL_FRAMEBUFFER, 0);
140     glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
141     SetCurrentShader(sceneShader);
142     projMatrix = Matrix4::Orthographic(-1,1,1,-1,-1,1);
143     viewMatrix.ToIdentity();
144     UpdateShaderMatrices();
145     quad->SetTexture(bufferColourTex[0]);
146     quad->Draw();
147     glUseProgram(0);
148 }
```

renderer.cpp

Fragment Shader

Our post processing fragment shader is going to perform a type of filtering known as a *Gaussian Blur*. This is performed quite simply - each pixel of a blurred image is formed from the *weighted sum* of surrounding source image pixels, with pixels further away contributing less to the final colour.

Normally, this is performed in one large pass, whereby all of the pixels in a square area contribute to some amount to the final pixel colour - 5 by 5 and 7 by 7 are common sizes for this *gaussian matrix*. To reduce the number of texture samples required to perform a blur effect suitable for use in games, it is common to reduce this to a two stage algorithm, first doing a 1D blur horizontally, and then a 1D vertical blur.



Starting from a single red pixel, the horizontal and vertical gaussian blur stages result in the right image

Our gaussian blur shader is going to perform a 5 by 5 matrix blur on the FBO colour target containing the heightmap scene, split into a horizonal and vertical pass. The gaussian blur shader relies on being able to calculate the texture coordinate of the surrounding texels of the source image - the *pixelSize* **uniform** variable should contain the correct size of each texel. We also have the *isVertical* **uniform int**, to determine which blur pass the shader should calculate.

On line 14, we have an array of **const floats** - these are the weights by which each texel will be multiplied to determine how much each texture sample will contribute to the final fragment colour. If you add these weights up, you should be able to see that the total of these weight values is 1.0. This is necessary to maintain the overall brightness of the image after blurring - if it were greater than 1, the image would get brighter after every blur pass, or darker if it were less than 1.0.

In the **main** function, we begin with a declaration of an array of **vec2s**. This will hold the texture sample coordinates for the surrounding texels, in either the horizontal or vertical axis. We determine which values to put in the *values* array via the **if** statement on line 19 - if we're in the vertical pass, we modify the y axis of the *values* array, otherwise the x axis. The values added are determined by multiplying the *pixelSize* **uniform** variable - note how it starts off *negating* the values.

On line 30, we perform the sampling of the FBO input texture. In each iteration of the **for** loop, we sample a surrounding texel of the FBO texture, by adding the corresponding *values* **vec2** (which may be *negative*) to the interpolated *texCoord* value, and then multiplying the sample by the current weight.

```
1  #version 150 core
2
3  uniform sampler2D diffuseTex;
4  uniform vec2      pixelSize;
5  uniform int       isVertical;
6
7  in Vertex   {
8     vec2 texCoord;
9     vec4 colour;
10 } IN;
11
12 out vec4 fragColour;
13 //Technically, this isn't quite a 'gaussian' distribution...
14 const float weights[5] = float[](0.12, 0.22, 0.32, 0.22, 0.12);
15
16 void main(void)   {
17    vec2 values[5];
18
19    if(isVertical == 1) {
20       values = vec2[](vec2(0.0,-pixelSize.y*2),
21          vec2(0.0,-pixelSize.y*1), vec2(0.0,0.0),
22          vec2(0.0,pixelSize.y*1) , vec2(0.0,pixelSize.y*2) );
23    }
24    else{
25       values = vec2[](vec2(-pixelSize.x*2, 0.0),
26          vec2(-pixelSize.x*1, 0.0),vec2(0, 0.0),
27          vec2(pixelSize.x*1,0.0)  ,vec2(pixelSize.x*2,0.0) );
28    }
29
30    for(int i = 0; i < 5; i++ )  {
31       vec4 tmp = texture2D(diffuseTex, IN.texCoord.xy + values[i]);
32       fragColour += tmp * weights[i];
33    }
34 }
```

processfrag.glsl

Tutorial Summary

If you run the example program, you should see the heightmap, but with the gaussian blur applied to it. By modifying the *POST_PASSES* **define**, you can increase or decrease the strength of the blur, as progressively more passes will be applied to the scene. It's not too impressive, but in completing this tutorial, you've learnt how to apply post processing effects to your scenes, via the use of a fragment shader a screen-sized quad. You've also learnt how to render into a Frame Buffer Object, and swap its attachments as necessary. FBOs are useful for more than just post processing - you could have a 4 player split screen game by rendering multiple viewpoints into FBOs, and drawing the outputs as quads on screen.

Further Work

1) Investigate *sobel filters* - a type of edge detection algorithm. You should be able to modify *processFrag* to perform edge detection rather than image blurring.

2) Games sometimes simulate a 'double vision' effect when the player character is hit. You should be able to replicate such an effect as a post process filter, by taking two samples - one at the normal position, and one from somewhere nearby.

3) Try adding multiple *Cameras* to the scene, and rendering each camera's viewpoint to an FBO attachment. You should be able to draw each viewpoint on screen as a quad, to create a multiple viewpoint effect. Or perhaps draw from a top down viewpoint to use as an on-screen map?