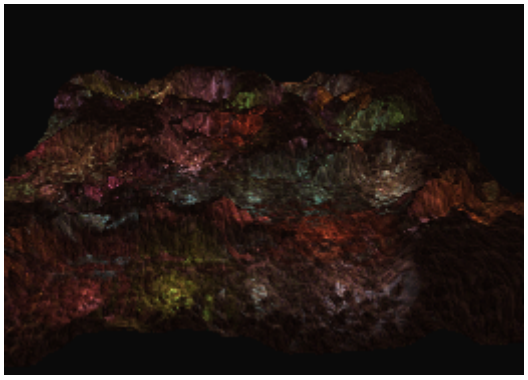
Tutorial 15: Deferred Rendering



Summary

In this tutorial series, you've seen how to create real time lighting effects, and maybe even had a go at creating a few pointlights in a single scene. But how about 10s of lights? Or even 100s? Traditional lighting techniques using shaders are not well suited to handling mass numbers of lights in a single scene, and certainly cannot perform them in a single pass, due to register limitations. This tutorial will show how a large number number of lights can light a scene at once, all of them using the complex lighting calculations you have been introduced, via a modern rendering technique known as deferred rendering.

New Concepts

Deferred rendering, light volumes, G-Buffers, world space recalculation

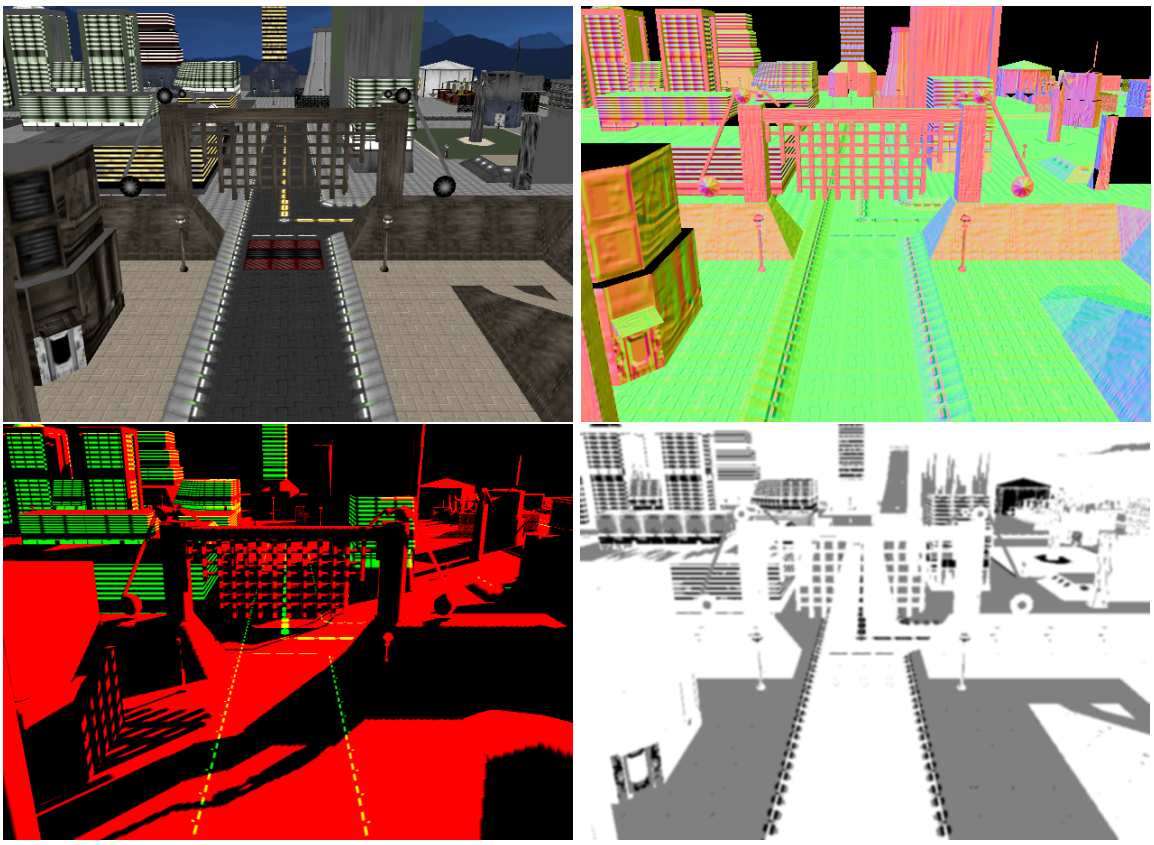
Deferred Rendering

As the number of objects and lights in a scene increases, the problems with performing lighting using a fragment shader similar to the one presented earlier in this tutorial series becomes apparent. As every vertex runs through the pipeline and gets turned into fragments, each fragment gets ran through the lighting equations. But depending on how the scene is rendered, this fragment might not actually be seen on screen, making the calculations performed on it a waste of processing. That's not too bad if there's only one light, but as more and more are added to the scene, this waste of processing time builds up. Objects not within reach of the light may still end up with fragments ran through the lighting shader, wasting yet more time. Ideally, lighting calculations would be performed in *image space* - i.e only ran on those fragments that have ended up as pixels in the final back buffer. Fortunately, there is a way to make real-time lighting a form of post-processing effect, ran in image space - *deferred rendering*. Deferred rendering allows many lights to be drawn on screen at once, with the expensive fragment processing only performed on those pixels that are definitely within reach of the light. Like the blur post-processing you saw earlier, deferred rendering is a *multipass* rendering algorithm, only this time, instead of two passes, three passes are performed - one to draw the world, one to light it, and one to present the results on screen. To pass data between the separate render passes, a large number of texture buffers are used, making use of the ability of a Frame Buffer Object to have multiple colour attachments.

G-Buffers

Deferred rendering works by splitting up the data needed to perform lighting calculations into a number of screen sized buffers, which together are known as the *G-Buffer*. You should be pretty used to multiple buffers by now - classical rasterisation uses at least one colour buffer, a depth buffer, and maybe a stencil buffer. What deferred rendering does is extend this to include a buffer for everything we need to compute our real-time lighting. So, as well as rendering our colour buffer and depth buffer, we might have a buffer containing TBN matrix transformed normals, too. If the vertices to be rendered have a specular power, either as a vertex attribute or generated from a gloss-map, we can write that out to a buffer, too. By using the Frame Buffer Object capability of modern graphics hardware, all of these buffers can be created in a single rendering pass. Just like how we can load in normals from a bump map, we can write them to a frame buffer attachment texture - textures are just data!

Normally, each of these buffers will be 32bits wide - the internals of graphics cards are optimised to handle multiples of 32 bits. This raises some important issues. The normals we've been loading in from bump maps are 24 bits - we store the x,y, and z axis as an unsigned byte. It's therefore quite intuitive to think about writing them out to the G-Buffer as 24 bits, too. But this leaves us with 8 bits 'spare' in our G-Buffer, as the alpha channel would not be used. What to do with these 8 bits of extra data is entirely up to the exact implementation of deferred rendering that is to be used. One popular solution is to put some per-pixel data in it , such as how much specularity to apply, or maybe some flags to determine whether lighting should affect the model, or if it should be blurred in a post-process pass. Sometimes, it might be decided to store the normals in a higher precision instead, and maybe have 12-bits for the x and y axis, with the z axis reconstructed in a fragment shader. Whatever per-pixel information required to correctly render the scene is placed into this G-Buffer, in a single rendering pass.



Example G-Buffer output for a cityscape rendered using deferred rendering. Clockwise from top left: Diffuse colours, World space normals, depth, and material information such as fullbright and shadowed areas

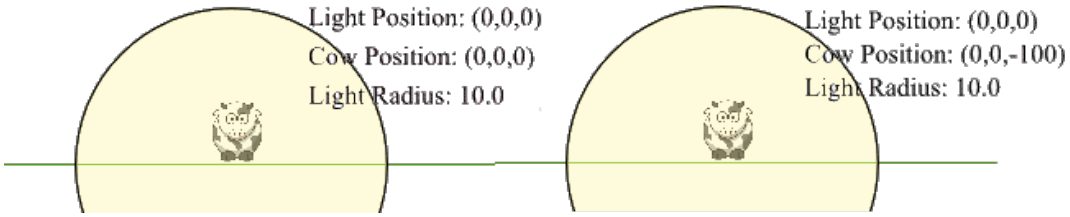
Light Volumes

Unlike the 'traditional' lighting techniques, where lighting is calculated in the same rendering pass as the vertices are transformed in, deferred rendering does it in a second pass, once the G-Buffer has been filled (i.e. lighting has been *deferred* until after the geometry has been rendered). In this pass, we render light *volumes*, to the screen, using the same view and projection matrices as the first render pass - so we are drawing lights in relation to our camera.

But what actually *is* a light volume? If you cast your mind back to the real time lighting tutorials, you were introduced to the concept of light *attenuation*, where each light had a position, and a *radius*. That means that our light was essentially a sphere of light, illuminating everything within it. That's what a light volume is, a shape that encapsulates the area we want the light to illuminate - except in deferred rendering, we actually draw a mesh to the screen, at the position where our light is. This mesh could simply be a sphere for a pointlight, or a cone for a spotlight - any shape we can calculate the attenuation function for can be used for a light volume.

Light volume A encapsulates nothing, while light volume b encapsulates part of the tree - lighting calculations will only be performed on the fragments of tree inside the light volume

However, remember that in 'view space' (i.e in relation to the camera), a light volume could appear to encapsulate an object, but in actuality the object is too far away from the light to be within its volume. So there is a chance that a light volume could 'capture' an object not inside it, which must be checked for in the second render pass. Consider the following example of a light volume appearing to contain an object not within its radius:



The cow on the left is small, and is fully enclosed within the light volume. The cow on the right is far away, so although it looks like it is within the light volume from the camera's perspective, it is actually outside of the light volume's radius

Deferred Lighting Considerations

So, how does splitting our rendering up into two stages help us render lots of lights? By using light volumes, and only performing lighting calculations within that volume, we can drastically reduce the number of fragment lighting calculations that must be performed.

Think about the traditional process: Imagine we are using our bump mapping shaders, expanded to contain an array of 16 lights, spread throughout the game world. So for every fragment written to the back buffer, we must perform 16 sets of lighting calculations.

The vast majority of the time, any particular fragment will only be lit by 1 or 2 lights, so that in itself causes a lot of wasted processing - even if we used the **discard** statement to 'early out' of the lighting process, we've still waste processing, and fragment shaders aren't particularly strong at using loops to start with. It gets worse! Imagine we have a really complex scene, with lots of objects overlapping each other. That means lots of fragments will be overwritten, so the time spent on calculating lighting for them is effectively wasted. Even if we use the early-z functionality by drawing in front-to-back order, and use face culling to remove back facing triangles, we still might get fragments that are written to many times. If we want to draw more lights than we can calculate in a single fragment shader, we must perform multiple rendering passes, drawing *everything* in the world multiple times!

Deferred rendering solves all of these problems. When it comes to performing the lighting calculations in the second pass, we already know exactly which fragments have made their way to the G-Buffer, so we get no wasted lighting calculations - every G-Buffer texel that is 'inside' a light volume *will* be seen in the final image. By using additive blending of output fragments in the second render pass, G-Buffer texels that are inside mulitple light volumes get multiple lighting calculation results added together. Finally, by only running the expensive lighting fragment shader on fragments that are inside light volumes, we save time by not running any calculations on fragments that will never be lit. By using deferred rendering, we go from maybe 8 or so lights on screen at a time before rendering slows down, to being able to quickly process 1000s of small lights, and maybe 100s of larger lights.

As with seemingly everything in graphical rendering, deferred rendering does have its downsides, too. The most obvious problem is space - the G-Buffer consists of multiple screen-sized textures, all of which must in be in graphics memory. It is easy to end up using 64Mb of graphics memory just for the G-Buffer - no problem on a PC thesedays, but tricky on a console with only 256Mb of graphics memory!

The second problem is related to the first - *bandwidth*. All of these G-Buffer textures must be sampled at least once per lit fragment, which can add up to a lot of data transfer every frame in cases where lots of lights overlap each other. This increase in bandwidth is offset to some degree by being able to do everything in a single lighting pass, and by guaranteeing that if data is transferred, then it is because the data almost certainly *will* be used.

Thirdly, deferred rendering doesn't help us with the problem of transparent objects. If we write transparent objects to the G-Buffer, what is behind the transparent fragments won't be shaded correctly, as the normals and depth read in will be for the transparent object! As with 'forward' rendering, it is usual to perform deferred lighting on only opaque objects, and then draw the transparent ones on top, using 'normal' lighting techniques, or maybe not even lit at all.

Having to render data into G-Buffers, rather than always having vertex attributes to hand, does place some limitations on the types of effect used - there's only so much data we can pack into the G-Buffer without requiring even more textures, further increasing space and bandwidth requirements.

Lastly, depending on the API used, we might not be able to perform antialiasing using deferred rendering. Antialiasing is the process of removing 'jaggies' - the jagged edges along the edges of polygons. Older hardware cannot store the information required to perform hardware antialiasing in a G-Buffer texture. Antialising using edge detection and a blur filter has become a popular solution to this.

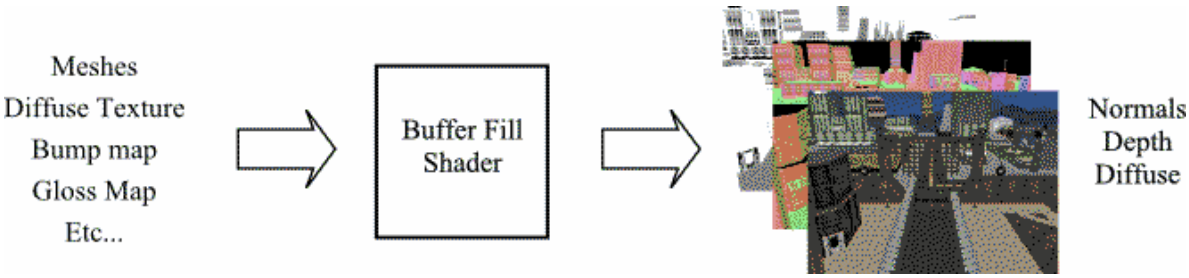
Despite these drawbacks, variations on deferred rendering have become the standard method of drawing highly complex, lit scenes - Starcraft 2, Killzone 2, and games using the CryEngine 3 and Unreal 3 game engines all support deferred lighting in some form.

Rendering Stages

Basic deferred rendering is split into three render passes - the first renders graphical data out into the G-Buffer, the second uses that data to calculate the accumulated light for each visible pixel, and the third combines the basic textures and lighting into a final scene.

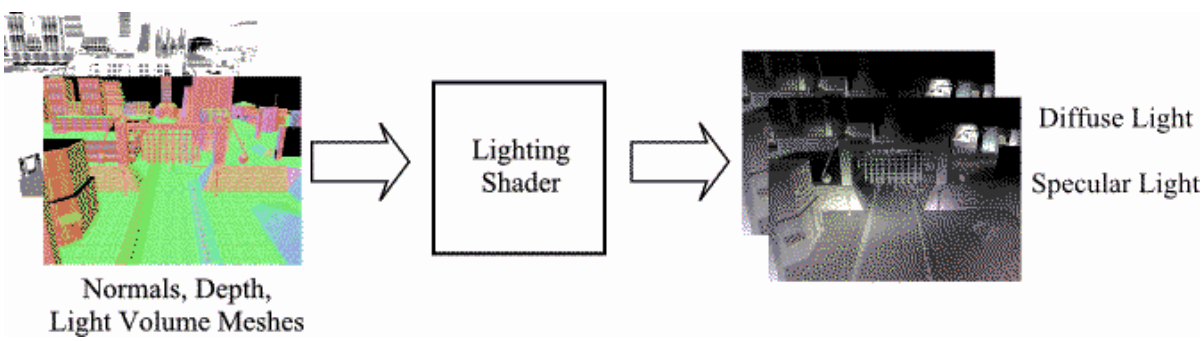
1) Fill the G-Buffer

First, the G-Buffer must be filled. This stage is similar to how you were rendering things before we introduced per-pixel lighting. For every object in the scene, we transform their vertices using a 'ModelViewProjection' matrix, and in a fragment shader sample their textures. However, instead of rendering the textured result to the back buffer, we output the data we need to perform lighting to several colour buffers, via a frame buffer object with multiple attachments.



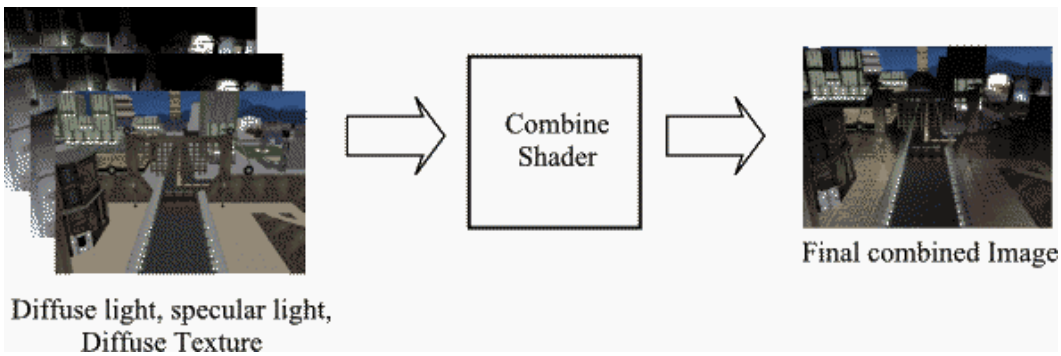
2) Perform Lighting Calculations

With the G-Buffer information rendered into textures, the second pass of calculating the lighting information can be performed. By rendering light volumes such as spheres and cones into the scene, the fragments which *might* be lit can be 'captured', and processed using the lighting fragment shader. In the fragment shader, the world position of the fragment being processed is reconstructed with help from the G-Buffer depth map - using this we can perform the lighting calculations, or *discard* the fragment if the fragment is too far away from the light volume to be rendered. We must still check whether fragments 'captured' by the light volume are outside the light's attenuation, due to situations where the light volume covers objects that are very far away.



3) Combine Into Final Image

In the final draw pass, we draw a single screen-sized quad, just like we do to perform post-processing techniques. In the fragment shader, we simply sample the lighting attachment textures, and the diffuse texture G-Buffer, and blend them together to create the final image.



Example Program

To show off deferred rendering, we're going to use our heightmap again. This time though, instead of rendering a single light in the centre of the heightmap, we're going to use deferred rendering to draw 64 lights, spread throughout the scene. And to make it even more interesting, those lights are going have random colours, and rotate around the centre of the heightmap! We're going to use spherical light volumes for this program, to show off how using a light volume produces just the same results as the traditional real time lighting methods you have been introduced to. We don't need any new classes in the nclgl this time around, just a new *Renderer* class in the *Tutorial 15* project. Also in the *Tutorial 15* project, we need *5* new text files, for the vertex and fragment shaders of deferrred rendering's 3 render passes. For the first pass we're going to reuse the *bumpVertex* shader, but combine it with a new fragment shader, called *bufferFragment.glsl*. For the second pass, we need two files, *pointLightvert.glsl* and *pointLightFrag.glsl*, and for the third pass we need, *combineVert.glsl* and *combineFrag.glsl*. Don't worry, only one of these shaders is a long one!

Renderer header file

First off, our new *Renderer* class header file. There's nothing much new in here, just lots of old! We have a **define**, *LIGHTNUM* that will determine how many lights are rendered on screen - you'll see how this is used shortly. We also have 4 new functions, a helper function for each of the three deferred rendering passes, and a function that will create a new screen sized FBO attachment. We stick this into a function as we're creating quite a lot of FBO attachments in this tutorial, and the code is almost identical, so it makes sense to stick it all together to reduce the chances of errors creeping in.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/Camera.h"
5  #include "./nclgl/OBJmesh.h"
6  #include "./nclgl/heightmap.h"
7
8  #define LIGHTNUM 8   //We'll generate LIGHTNUM squared lights...
9
10 class Renderer : public OGLRenderer    {
11 public:
12    Renderer(Window &parent);
13    virtual ~Renderer(void);
14
15    virtual void RenderScene();
16    virtual void UpdateScene(float msec);
17
18 protected:
19    void  FillBuffers();          //G-Buffer Fill Render Pass
20    void  DrawPointLights();      //Lighting Render Pass
21    void  CombineBuffers();       //Combination Render Pass
22 //Make a new texture...
23    void  GenerateScreenTexture(GLuint &into, bool depth = false);
```

renderer.h

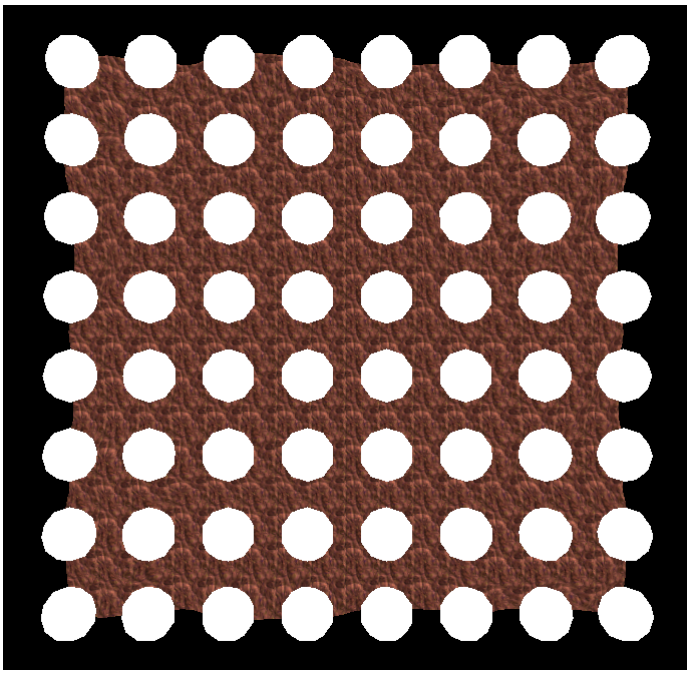
Now for our member variables. We have 3 *Shaders*, a pointer to some *Light* **structs**, a *HeightMap*, a light volume *Mesh*, a quad *Mesh*, and a *Camera*. We're going to use a *rotation* **float** to keep how much to rotate our lights by, two FBOs, and 5 FBO attachment textures - now you can see why we stuck the attachment texture initialisation into a function!

```
24     Shader*  sceneShader;          //Shader to fill our GBuffers
25     Shader*  pointlightShader;     //Shader to calculate lighting
26     Shader*  combineShader;        //shader to stick it all together
27
28     Light*   pointLights;          //Array of lighting data
29     Mesh*    heightMap;            //Terrain!
30     OBJMesh* sphere;               //Light volume
31     Mesh*    quad;                 //To draw a full-screen quad
32     Camera*  camera;               //Our usual camera
33
34     float    rotation;             //How much to increase rotation by
35
36     GLuint   bufferFBO;            //FBO for our G-Buffer pass
37     GLuint   bufferColourTex;      //Albedo goes here
38     GLuint   bufferNormalTex;      //Normals go here
39     GLuint   bufferDepthTex;       //Depth goes here
40
41     GLuint   pointLightFBO;        //FBO for our lighting pass
42     GLuint   lightEmissiveTex;     //Store emissive lighting
43     GLuint   lightSpecularTex;     //Store specular lighting
44 };
```

renderer.h

Renderer Class file

We begin our *Renderer* class with its **constructor**, as ever. We start off by initialising the rotation variable to 0.0f, and creating a new *Camera* and quad *Mesh*. Then, beginning on line 10, we begin the process of creating enough Lights for our deferred scene. What we want to create is a grid of light volumes, each with its own position, colour and radius. The number of lights along each axis of the light grid is controlled by the *LIGHTNUM* define in the header file, so by default, we'll create 8 lights along each axis, creating 64 lights in total:



The HeightMap terrain, with 64 light volumes lighting its area

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     rotation = 0.0f;
5     camera   = new Camera(0.0f,0.0f,
6     Vector3(RAW_WIDTH*HEIGHTMAP_X / 2.0f,500,RAW_WIDTH*HEIGHTMAP_X));
7
8     quad = Mesh::GenerateQuad();
9
10    pointLights = new Light[LIGHTNUM*LIGHTNUM];
11    for(int x = 0; x < LIGHTNUM; ++x) {
12       for(int z = 0; z < LIGHTNUM; ++z) {
13          Light &l   = pointLights[(x*LIGHTNUM)+z];
14
15          float xPos = (RAW_WIDTH*HEIGHTMAP_X / (LIGHTNUM-1)) * x;
16          float zPos = (RAW_HEIGHT*HEIGHTMAP_Z / (LIGHTNUM-1)) * z;
17          l.SetPosition(Vector3(xPos,100.0f,zPos));
18
19          float r = 0.5f + (float)(rand()%129) / 128.0f;
20          float g = 0.5f + (float)(rand()%129) / 128.0f;
21          float b = 0.5f + (float)(rand()%129) / 128.0f;
22          l.SetColour(Vector4(r,g,b,1.0f));
23
24          float radius = (RAW_WIDTH*HEIGHTMAP_X / LIGHTNUM);
25          l.SetRadius(radius);
26       }
27    }
```

renderer.cpp

So, we initialise enough *Lights* to fill the scene (line 10), and then set their positions (line 17) and colours (line 22) using a double **for** loop - one to traverse each axis. We want the colour of each light to be quite bright, so it is set to 0.5, with a random float between 0.0 and 0.5 added to it. The loop is set up so that as the *LIGHTNUM* **define** is changed, the positions and radii are still useful values.

Once the *Lights* are set up, we can create the *HeightMap* (just as we do in the bump mapping tutorial), and create a *Mesh* for the light volume. We're going to use another OBJ mesh, just like how we generated a *Mesh* from an OBJ file in the scene graph tutorial.

```
28    heightMap      = new HeightMap("../Textures/terrain.raw");
29    heightMap->SetTexture(SOIL_load_OGL_texture(
30       "../Textures/Barren Reds.JPG",SOIL_LOAD_AUTO,
31       SOIL_CREATE_NEW_ID,SOIL_FLAG_MIPMAPS));
32
33    heightMap->SetBumpMap(SOIL_load_OGL_texture(
34       "../Textures/Barren RedsDOT3.tga", SOIL_LOAD_AUTO,
35       SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
36
37    SetTextureRepeating(heightMap->GetTexture(),true);
38    SetTextureRepeating(heightMap->GetBumpMap(),true);
39
40    sphere = new OBJMesh();
41    if(!sphere->LoadOBJMesh("../Meshes/ico.obj")) {
42       return;
43    }
```

renderer.cpp

Next up, we create the three shader programs required for the three render passes. We're reusing one shader file from an earilier tutorial, but the other 5 are all new!

```
44    sceneShader = new Shader("../Shaders/BumpVertex.glsl",
45                             "bufferFragment.glsl");
46    if(!sceneShader->LinkProgram()) {
47       return;
48    }
49
50    combineShader = new Shader("combinevert.glsl","combinefrag.glsl");
51    if(!combineShader->LinkProgram()) {
52       return;
53    }
54
55    pointlightShader = new Shader("pointlightvert.glsl",
56                                  "pointlightfrag.glsl");
57    if(!pointlightShader->LinkProgram()) {
58       return;
59    }
```

renderer.cpp

Now the *Shaders* are done with, its time to move on to the FBOs and its attachments. To perform deferred rendering we need two FBOs - one for the first rendering pass, and one for the second; the third render pass outputs to the *back buffer*. SO, on lines 60 and 61, we generate two FBOs. Each of these FBOs is going to have two colour attachments, in the first pass to keep the texture samples and normals, and in the second pass to keep diffuse and specular lighting. To tell OpenGL which colour channels to render to, we must pass it a pair of named constants - in this case we'll be drawing into the first and second colour attachments, so on line 63, we create a pair of GLenums, equating to the relevent named constants. Then, on lines 68 to 72, we use the function *GenerateScreenTexture*, which will be described shortly, to generate the FBO attachment textures. For now, just note that we send a value of **true** to the first function call, which creates the depth texture.

```
60    glGenFramebuffers(1,&bufferFBO);
61    glGenFramebuffers(1,&pointLightFBO);
62
63    GLenum buffers[2];
64    buffers[0] = GL_COLOR_ATTACHMENT0;
65    buffers[1] = GL_COLOR_ATTACHMENT1;
66
67    //Generate our scene depth texture...
68    GenerateScreenTexture(bufferDepthTex, true);
69    GenerateScreenTexture(bufferColourTex);
70    GenerateScreenTexture(bufferNormalTex);
71    GenerateScreenTexture(lightEmissiveTex);
72    GenerateScreenTexture(lightSpecularTex);
```

renderer.cpp

With the FBOs and their attachment textures created, we can start binding attachments to their respective FBOs. First, we'll set up the FBO for the first deferred rendering pass. It's not too different to the FBO creation process in the post-processing tutorial, only this time we're creating *two* colour attachments. Once all of the textures are attached, a call to **glDrawBuffers**, using the buffers variable made on line 63, lets the FBO know that it is going to render into both of its colour attachments.

```
73    //And now attach them to our FBOs
74    glBindFramebuffer(GL_FRAMEBUFFER, bufferFBO);
75    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
76          GL_TEXTURE_2D, bufferColourTex, 0);
77    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
78          GL_TEXTURE_2D, bufferNormalTex, 0);
79    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
80          GL_TEXTURE_2D, bufferDepthTex, 0);
81    glDrawBuffers(2, buffers);
82
83    if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
84        GL_FRAMEBUFFER_COMPLETE) {
85       return;
86    }
```

renderer.cpp

Next we simply do the same for the second rendering pass. Note that in this pass, we don't have a depth attachment at all - we're going to reconstruct world positions in the shader, using the depth buffer of the first rendering pass, passed in as a normal texture. As we're done with FBOs for now, on line 99 we unbind the FBO, then enable the OpenGL states we need - depth testing, culling, and blending.

```
87    glBindFramebuffer(GL_FRAMEBUFFER, pointLightFBO);
88    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
89       GL_TEXTURE_2D, lightEmissiveTex, 0);
90    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
91       GL_TEXTURE_2D, lightSpecularTex, 0);
92    glDrawBuffers(2, buffers);
93
94    if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
95        GL_FRAMEBUFFER_COMPLETE) {
96       return;
97    }
98
99    glBindFramebuffer(GL_FRAMEBUFFER, 0);
100   glEnable(GL_DEPTH_TEST);
101   glEnable(GL_CULL_FACE);
102   glEnable(GL_BLEND);
103
104   init      = true;
105}
```

renderer.cpp

Finally our **constructor** is finished! Now to tear it all down in the **destructor**. Remember that the *pointLights* pointer is an array, and so should be **deleted** using **delete**[].

```
106Renderer::~Renderer(void)  {
107   delete      sceneShader;
108   delete      combineShader;
109   delete      pointlightShader;
110
111   delete      heightMap;
112   delete      camera;
113   delete      sphere;
114   delete      quad;
115   delete[]    pointLights;
```

```
116     glDeleteTextures(1,&bufferColourTex);
117     glDeleteTextures(1,&bufferNormalTex);
118     glDeleteTextures(1,&bufferDepthTex);
119     glDeleteTextures(1,&lightEmissiveTex);
120     glDeleteTextures(1,&lightSpecularTex);
121 
122     glDeleteFramebuffers(1,&bufferFBO);
123     glDeleteFramebuffers(1,&pointLightFBO);
124     currentShader = 0;
125 }
```

renderer.cpp

In the constructor, we used a helper function, called *GenerateScreenTexture*. Here's how it works. It takes in two parameters - the first is a reference to a **GLuint** texture name, and the second is a **boolean** which controls whether we are generating a depth texture or a colour texture. The second parameter has a default value of **false** - that's why one one call had a second parameter in the **constructor**, as we only need to generate a depth texture once. The process for generating either type of texture is pretty similar, and should be familiar from the post-processing tutorial, and the appendix of tutorial 3. We generate a screen-sized texture, and use the *depth* parameter to decide on the type and format of the texture (on lines 136 and 138).

```
126 void Renderer::GenerateScreenTexture( GLuint &into , bool depth)  {
127     glGenTextures(1, &into);
128     glBindTexture(GL_TEXTURE_2D, into);
129 
130     glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
131     glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
132     glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_NEAREST);
133     glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_NEAREST);
134 
135     glTexImage2D(GL_TEXTURE_2D, 0,
136         depth ? GL_DEPTH_COMPONENT24 : GL_RGBA8,
137         width, height, 0,
138         depth ? GL_DEPTH_COMPONENT : GL_RGBA,
139         GL_UNSIGNED_BYTE, NULL);
140 
141     glBindTexture(GL_TEXTURE_2D, 0);
142 }
```

renderer.cpp

In the *UpdateScene* function for this tutorial, as well as our usual camera stuff, we're going to set the *rotation* variable, to a value derived from *msec*, so we get a nice and consistent rotation for our 64 lights.

```
143 void Renderer::UpdateScene(float msec) {
144     camera->UpdateCamera(msec);
145     viewMatrix = camera->BuildViewMatrix();
146     rotation = msec * 0.01f;
147 }
```

renderer.cpp

We have another small RenderScene function this time around, as the actual rendering is handled by three helper functions - one for each rendering pass. We make sure the back buffer is cleared, and the buffers are swapped once everything is drawn, though.

```
148 void Renderer::RenderScene()  {
149    glBindFramebuffer(GL_FRAMEBUFFER, 0);
150    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
151
152    FillBuffers();
153    DrawPointLights();
154    CombineBuffers();
155    SwapBuffers();
156 }
```

renderer.cpp

Here's how we perform the first deferred rendering pass, which outputs the texture colours and normals of every pixel on screen. As we're rendering into an FBO, the first function called is to bind that FBO, and clear its contents. Calling **glClear** using the colour buffer bit on an FBO with multiple colour targets, will clear every bound target. Beyond that, rendering the scene is much as you've seen before - we bind the texture and bumpmap texture units to the shader, set up the matrices, and draw the heightmap. It's the shader that will perform all of the interesting stuff to put the values into the correct FBO attachment textures. Although we don't calculate any lighting in this pass, we *do* calculate the TBN matrix, so we need the bump maps of every object we draw.

```
157 void  Renderer::FillBuffers()  {
158    glBindFramebuffer(GL_FRAMEBUFFER, bufferFBO);
159    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
160
161    SetCurrentShader(sceneShader);
162    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
163                "diffuseTex") , 0);
164    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
165                "bumpTex")  , 1);
166
167    projMatrix   = Matrix4::Perspective(1.0f,10000.0f,
168                  (float)width / (float)height, 45.0f);
169    modelMatrix.ToIdentity();
170    UpdateShaderMatrices();
171
172    heightMap->Draw();
173
174    glUseProgram(0);
175    glBindFramebuffer(GL_FRAMEBUFFER, 0);
176 }
```

renderer.cpp

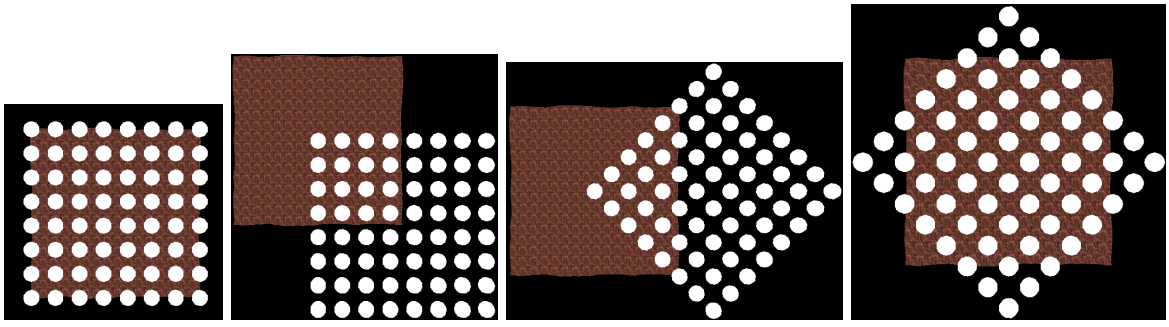
Now for the second pass, which we perform in the function *DrawPointLights*. Like the first pass (and the rendering you are used to), we begin by binding the appropriate shader, binding the appropriate FBO, and clearing its colour buffer (we have no depth buffer in this pass). In this pass, though, we don't clear the colour to dark grey like normal, but to black. That's because we're going to be blending the colour of the results of this pass in the third pass, so we want unlit pixels to be black. If the clear colour was grey, it would appear that the entire scene had been lit by a grey light, and appear washed out. We also want fragments that are 'captured' by multiple lights to have the sum of those light colours as its colour, so we enable additive blending on line 185.

Then on lines 187 to 196, we bind the depth and normal FBO attachments from the first pass as **uniform** textures in this pass, giving our fragment shader access to the roughness and position of the fragments captured by our light volumes. Then, we send our camera's position to the shader, so we can calculate specularity, and also the *pixelSize* value we first saw in the post-processing tutorial.

```
177 void Renderer::DrawPointLights() {
178    SetCurrentShader(pointlightShader);
179
180    glBindFramebuffer(GL_FRAMEBUFFER, pointLightFBO);
181
182    glClearColor(0,0,0,1);
183    glClear(GL_COLOR_BUFFER_BIT);
184
185    glBlendFunc(GL_ONE,GL_ONE);
186
187    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
188                "depthTex"), 3);
189    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
190                "normTex") , 4);
191
192    glActiveTexture(GL_TEXTURE3);
193    glBindTexture(GL_TEXTURE_2D, bufferDepthTex);
194
195    glActiveTexture(GL_TEXTURE4);
196    glBindTexture(GL_TEXTURE_2D, bufferNormalTex);
197
198    glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
199                "cameraPos"), 1,(float*)&camera->GetPosition());
200
201    glUniform2f(glGetUniformLocation(currentShader->GetProgram() ,
202                "pixelSize"), 1.0f / width, 1.0f / height);
```

renderer.cpp

Now to draw the lights! we can do the light volume drawing using a double **for** loop, but what about rotating them? If we just have a rotation matrix for each light's model matrix, they'll simply spin around their origin, which as they're spheres, will not look very interesting! Instead, we want to rotate them in relation to the centre of the *HeightMap*. So for each light, we're going to go through the following process. Firstly, we're going to translate by half way along the heightmaps width and length, moving the 'origin' to the centre of the heightmap. Then, we're going to rotate by a rotation matrix, rotating around this new origin. Then, we're going to translate in the opposite direction to the first translation, moving the light back to the origin being in the corner of the heightmap. You should recognise this from the texture mapping tutorial, where we did the same thing to the texture matrix, so that the texture rotated about the middle of the triangle. By doing this every frame, the entire light grid will slowly rotate around the centre of the height map.



From left to right: Lights at their local origins, with the world origin in the top left. Translating light origins to be from the centre of the heightmap. Rotating the lights by 45 degrees. Translating back to an origin at the top left

So, on line 206 we calculate the translation matrix to 'push' the light to be in relation to the middle of the screen, and on 207 we calculate the matrix to 'pop' it back. Then, on line 209, we begin the double for loop to set up the model matrix for each light (line 214), update the shader light variables (line 223), and update the shader matrix variables (line 225).

On line 227, we calculate the disance of the camera from the current light, then on line 228 we test against it. Why do we do this? If we have back face culling enabled, and the distance between the light volume and the camera is *less* than the light's radius, then it follows that the camera is *inside* the light volume. That means every face of the light is facing away from the camera, and so will not be drawn! We can't capture any fragments if our light volume isn't drawn. So we flip face culling around to cull *front* faces temporarily when the camera is inside the light's radius. But why don't we just disable face culling entirely for lights? If we did, most pixels would then be captured by a light twice - once by one 'side' of the light volume, facing away, and once by the front 'side, facing towards. So the lighting calculations will be incorrect, as a light would be influencing its captured fragments twice - unless the camera was inside. Swapping culled faces removes the chance of accidentally performing lighting calculations twice for a single light.

```
203    Vector3 translate = Vector3((RAW_HEIGHT*HEIGHTMAP_X / 2.0f),500,
204                                (RAW_HEIGHT*HEIGHTMAP_Z / 2.0f));
205
206    Matrix4 pushMatrix = Matrix4::Translation(translate);
207    Matrix4 popMatrix  = Matrix4::Translation(-translate);
208
209    for(int x = 0; x < LIGHTNUM; ++x) {
210       for(int z = 0; z < LIGHTNUM; ++z) {
211          Light &l          = pointLights[(x*LIGHTNUM)+z];
212          float radius      = l.GetRadius();
213
214          modelMatrix =
215             pushMatrix*
216             Matrix4::Rotation(rotation,Vector3(0,1,0)) *
217             popMatrix *
218             Matrix4::Translation(l.GetPosition()) *
219             Matrix4::Scale(Vector3(radius,radius,radius));
220
221          l.SetPosition(modelMatrix.GetPositionVector());
222
223          SetShaderLight(l);
224
225          UpdateShaderMatrices();
226
227          float dist =(l.GetPosition()-camera->GetPosition()).Length();
228          if(dist < radius) {//camera is inside the light volume!
229             glCullFace(GL_FRONT);
230          }
231          else{
232             glCullFace(GL_BACK);
233          }
234
235          sphere->Draw();
236       }
237    }
```

renderer.cpp

Once we're done sending each light to the shader, we can set everything back to 'default', which in our case is to have back face culling enabled, and the clear colour to be dark grey. To keep things neat, we also unbind the second pass shader and its FBO.

```
238     glCullFace(GL_BACK);
239     glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
240
241     glClearColor(0.2f,0.2f,0.2f,1);
242
243     glBindFramebuffer(GL_FRAMEBUFFER, 0);
244     glUseProgram(0);
245 }
```

renderer.cpp

Finally, the last function! *CombineBuffers* performs the third deferred rendering pass, which takes in the diffuse texture colours from the first render pass, the lighting components from the second, and combines them all together into a final image, using the *combineShader* shader. It does this as a post processing pass, so we just render a single screen sized quad via an orthographic projection, set three textures up, and render the quad. We use the third, fourth, and fifth texture units to do this, so that the *Draw* function of the *Mesh* class does not unbind any of our textures (remember, we've set up the *Draw* function to its diffuse texture in the first texture unit, and its bumpmap in the second).

```
246 void Renderer::CombineBuffers()  {
247     SetCurrentShader(combineShader);
248
249     projMatrix = Matrix4::Orthographic(-1,1,1,-1,-1,1);
250     UpdateShaderMatrices();
251
252     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
253                 "diffuseTex")   , 2);
254     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
255                 "emissiveTex") , 3);
256     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
257                 "specularTex") , 4);
258
259     glActiveTexture(GL_TEXTURE2);
260     glBindTexture(GL_TEXTURE_2D, bufferColourTex);
261
262     glActiveTexture(GL_TEXTURE3);
263     glBindTexture(GL_TEXTURE_2D, lightEmissiveTex);
264
265     glActiveTexture(GL_TEXTURE4);
266     glBindTexture(GL_TEXTURE_2D, lightSpecularTex);
267
268     quad->Draw();
269
270     glUseProgram(0);
271 }
```

renderer.cpp

G-Buffer Shader

The first render pass fills the G-Buffer with the values we need to calculate the lighting for our scene. We use the *bumpVertex.glsl* shader for the vertex program, but we need a new fragment program. Like the *bumpFragment* shader, we have two texture samplers - one to take the diffuse texture, and one for the bump map texture. We work out the TBN and the fragment's world space normal, but then instead of calculating lighting straight away, we output the normal to the second FBO colour attachment.

New to this shader is writing to multiple outputs. Earlier, we bound *two* colour attachments to the G-Buffer FBO - so we can save our the diffuse texture samples, and the per-pixel normals. To fill each of these, we have an output *array*, which our shader can access using **gl_FragColor[0]** and **gl_FragColor[1]** - we put our sampled texture colours in the first, and our TBN transformed normals in the second. These output **vec4s** get automatically bound to the appropriate colour attachment of our FBO, so we don't need to do any binding of values like we have to for **uniforms**.

See how on line 23 we multiply the local normal by 0.5 and then add 0.5 to it? This performs the opposite of the expansion of bump map normals to a range of -1 to 1; this time, we change the range from -0.5 to 0.5 using the multiply, and to a range of 0.0 to 1.0 using the add. We do this as we need the normal written to the attachment to be in a range suitable for a texture.

Fragment Shader

```
1  #version 150 core
2
3  uniform sampler2D diffuseTex;    //Diffuse texture map
4  uniform sampler2D bumpTex;       //Bump map
5
6  in Vertex {
7     vec4 colour;
8     vec2 texCoord;
9     vec3 normal;
10    vec3 tangent;
11    vec3 binormal;
12    vec3 worldPos;
13 } IN;
14
15 out vec4 gl_FragColor[2];        //Our final outputted colours!
16
17 void main(void)   {
18    mat3 TBN       = mat3(IN.tangent, IN.binormal, IN.normal);
19    vec3 normal    = normalize(TBN *
20                     (texture2D(bumpTex, IN.texCoord).rgb)* 2.0 - 1.0);
21
22    gl_FragColor[0]   = texture2D(diffuseTex, IN.texCoord);
23    gl_FragColor[1]   = vec4(normal.xyz * 0.5 + 0.5,1.0);
24 }
```

bufferFragment.glsl

Point Light Shader

Now for the shader used to draw the light volumes on screen. For the light volume vertices, we just do the usual - make an MVP matrix, and transform the vertices by it. We don't care about any other vertex attributes this time around, everything will be generated in the fragment shader. However, to do that we need one extra piece of information, the *inverse* of the matrix formed by multiplying the view and projection matrices together. Then in the fragment shader, we output the lighting to two colour attachments - one for the diffuse light, and one for the specular light.

Vertex Shader

```
1  #version 150 core
2
3  uniform mat4 modelMatrix;
4  uniform mat4 viewMatrix;
5  uniform mat4 projMatrix;
6  uniform mat4 textureMatrix;
7
8  in  vec3 position;
9  out mat4 inverseProjView;
10
11 void main(void)   {
12    gl_Position     = (projMatrix * viewMatrix * modelMatrix)
13                      * vec4(position, 1.0);
14
15    inverseProjView = inverse(projMatrix*viewMatrix);
16 }
```

pointlightvertex.glsl

Recalculating the world space position

For every fragment of our first render pass that is captured by a light volume, we need to work out its world position - both to calculate correct attenuation, and to determine whether the fragment actually is inside the volume, or is merely behind it in screen space.

The first stage of this is to calculate the fragment's *screen space* position - where it is on screen. We can do this using the GLSL command **gl_FragCoord**. This function returns the screen coordinates of the currently processed fragment, so if the screen was 800 pixels wide, the furthest right fragment processed would have a gl_FragCoord.x value of 800. However, before we work out the world space position, we need this value to first be in the 0.0 to 1.0 range, so we can sample the G-Buffer attachment textures using it. So, we divide it by the *pixelSize* **uniform** variable, which you should remember from the post processing tutorial. That gives us x and y axis from the 0-1 range, but what about z? We can use thse x and y coordinates to sample a z coordinate from the depth buffer, giving us all 3 axis' in the 0-1 range.

Once we've used these x and y coordinates to sample any other colour attachment textures (such as the sample to the normal attachment on line 21 of the fragment shader), we can move this fragment position from screen space (0 - 1 range) to *clip space* (-1.0 - 1.0 range), by multiplying it by 2.0 and subtracting 1.0. Then, this value gets multiplied by the inverse projection-view matrix we created in the vertex shader. Finally, we divide the resulting x, y, and z values by the clip space w value, moving us from post-divide clip space to *world space*.

Fragment Shader

Here's how we're going to perform the lighting per fragment. Our **uniform** values are the same as the bump mapping tutorial, only this time we're taking in the inverse matrix we created in the vertex shader, and outputting *two* colours - in one texture we're going to save the diffuse colour, and in a second we're going to save the specular colour. Between lines 17 and 24 we reconstruct the world position of the fragment we are working on, and then between lines 26 and 31, we see how far the fragment is from the light volume, and if it's too far away, **discard** the fragment. In the previous lighting tutorials, we multiplied our specularity by the lambertian reflectance amount to save some processing and local variables, but *really* it should be separate. So, this time we write out the diffuse lighting, multiplied by the lambertian reflectance value, to the first colour attachment, and the specular value multiplied by the specular factor to the second attachment.

```
1  #version 150 core
2
3  uniform sampler2D    depthTex;
4  uniform sampler2D    normTex;
5
6  uniform vec2         pixelSize;
7  uniform vec3         cameraPos;
8
9  uniform float        lightRadius;
10 uniform vec3         lightPos;
11 uniform vec4         lightColour;
12
13 in  mat4    inverseProjView;
14 out vec4    gl_FragColor[2];
15
16 void main(void)   {
17    vec3 pos       = vec3((gl_FragCoord.x * pixelSize.x),
18                          (gl_FragCoord.y * pixelSize.y), 0.0);
19    pos.z          = texture(depthTex, pos.xy).r;
20
21    vec3  normal   = normalize(texture(normTex, pos.xy).xyz*2.0 - 1.0);
22
23    vec4 clip      = inverseProjView * vec4(pos * 2.0 - 1.0, 1.0);
24    pos            = clip.xyz / clip.w;
25
26    float dist     = length(lightPos - pos);
27    float atten    = 1.0 - clamp(dist / lightRadius, 0.0, 1.0);
28
29    if(atten == 0.0) {
30       discard;
31    }
32
33    vec3 incident  = normalize(lightPos  - pos);
34    vec3 viewDir   = normalize(cameraPos - pos);
35    vec3 halfDir   = normalize(incident + viewDir);
36
37    float lambert   = clamp(dot(incident, normal),0.0,1.0);
38    float rFactor   = clamp(dot(halfDir, normal) ,0.0,1.0);
39    float sFactor   = pow(rFactor, 33.0 );
40
41    gl_FragColor[0] = vec4(lightColour.xyz * lambert * atten, 1.0);
42    gl_FragColor[1] = vec4(lightColour.xyz * sFactor * atten*0.33,1.0);
43 }
```

pointlightfragment.glsl

Buffer Combine Shader

In the last pass, we're going to draw a screen sized quad, and combine the textured colour attachment from the first pass, with the two lighting textures from the second pass. We'll also do ambient lighting in the fragment shader, to lighten up any areas that weren't lit up by our light volume pass.

Vertex Shader

All we're doing is rendering a single quad, so the vertex shader is very simple - we transform the quad's vertices, and output its texture coordinates.

```
1  #version 150 core
2  uniform mat4 projMatrix;
3
4  in  vec3 position;
5  in  vec2 texCoord;
6
7  out Vertex {
8     vec2 texCoord;
9  } OUT;
10
11 void main(void)   {
12    gl_Position     = projMatrix * vec4(position, 1.0);
13    OUT.texCoord    = texCoord;
14 }
```

combinevert.glsl

Fragment Shader

The fragment shader isn't much more complicated. We have three textures - one containing our textured scene, and two for the light components. We simple sample each of those textures, and blend them together, including a small amount of 'ambient' colouring (line 17).

```
1  #version 150 core
2  uniform sampler2D diffuseTex;
3  uniform sampler2D emissiveTex;
4  uniform sampler2D specularTex;
5
6  in Vertex {
7     vec2 texCoord;
8  } IN;
9
10 out vec4 gl_FragColor;
11
12 void main(void)   {
13    vec3 diffuse        = texture(diffuseTex , IN.texCoord).xyz;
14    vec3 light          = texture(emissiveTex, IN.texCoord).xyz;
15    vec3 specular       = texture(specularTex, IN.texCoord).xyz;
16
17    gl_FragColor.xyz    = diffuse * 0.2;   //ambient
18    gl_FragColor.xyz   += diffuse * light; //lambert
19    gl_FragColor.xyz   += specular;        //Specular
20    gl_FragColor.a      = 1.0;
21 }
```

combinefrag.glsl

Tutorial Summary

Upon running the program, you should see the heightmapped terrain, lit by 64 light volumes. The light volumes slowly rotate around the centre of the heightmap, illuminating the scene using a variety of colours. The attenuation of light, and the movement of specular highlights should look like 'traditional' real time lighting techniques, but on a much grander scale. Deferred lighting makes the handling of such a large number of lights possible. These lights can be of any shape and size, and the process of lighting a scene is completely decoupled from its rendering, allowing incredibly rich graphical scenes to be rendered quickly. Deferred rendering is bandwidth and memory intensive, but the tradeoff in terms of visual quality is such that deferred rendering has become a popular method of creating realistic lit scenes in modern games, widely adopted in both console and PC titles.

Further Work

1) In the example program of this tutorial, we used a light volume loaded in as an OBJ mesh. What shape was the mesh? How does this shape still produce spherical light volumes? Why do we use a simplified shape for our light volumes? Hint: how many lights are we drawing, and how many *could* we draw?

2) Spheres aren't the only shape of light volume we can use in deferred rendering. If you've attempted spot lights before, try adding a 'deferred' spotlight to the example program. What shape of simplified geometry could we use for a spotlight?

3) Some variations of deferred lighting don't actually use world space light volumes to capture lit pixels! How could you capture pixels for a point light using a single quad? In which space would this be drawn in?

4) Investigate these presentations on recent titles that have used some form of deferred lighting:

http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

http://www.pcgameshardware.com/aid,674502/Starcraft-2-Technology-and-engine-dissected/News/

http://www.spuify.co.uk/?p=323