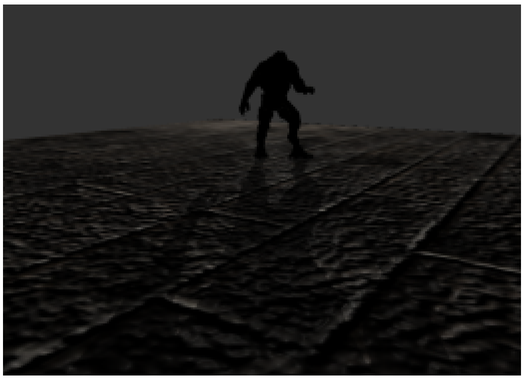
Tutorial 14: Shadow Mapping



Summary

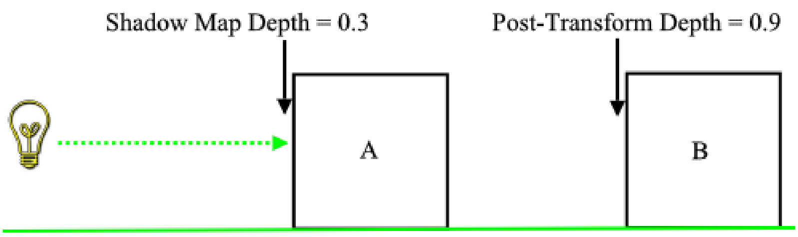
Although it is possible to simulate darkness due to a light surface facing away from a light source, this does not take into account the shadows cast by light being obstructed by an object. In this tutorial, we'll take a look at *shadow mapping*, a method of simulating such shadows using multiple render passes, and how to implement it using a Frame Buffer Object.

New Concepts

Shadow maps, depth comparison, surface acne, shadow samplers

Shadow Mapping

Shadow maps effectively simulate the shadows cast upon a surface when there is an object between a light source and the surface - these objects are known as *occluders*. Although not a new technique, it is only relatively recently that graphics hardware has had the processing and memory capacity to utilise shadow maps effectively. Shadow mapping is a two pass process, meaning that the geometry that makes up the world is rendered twice. In the first pass, the world is rendered from the *light's* point of view, as if it were a camera, with the *depth buffer* of this pass saved. In the second pass, the world geometry is drawn from the *camera's* point of view, using a shader that has the first pass depth texture, and a transformation matrix that will transform vertices into the light's view space as variables - just as if the scene were being rendered from the light's point of view. By comparing the resulting transformed z value, and the depth value in the depth texture, it can be worked out whether a vertex is 'behind' or 'in front' of the depth values - if it is behind, then it is behind whatever object wrote to the depth buffer, and so has been occluded from the light.



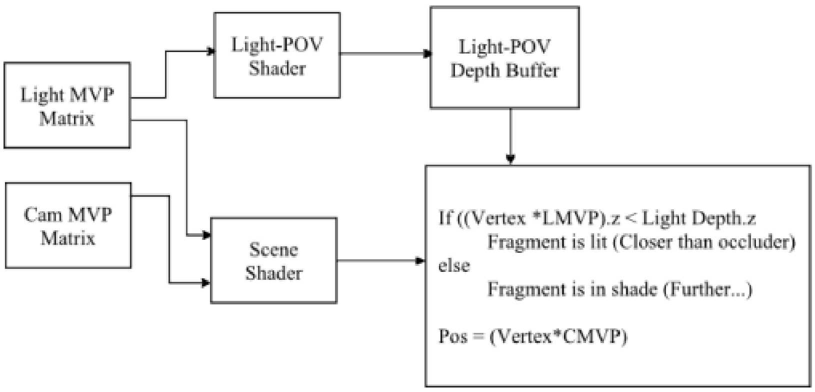
Object B has a greater post-transform depth than the value written to the shadow map by object A, and so object B occludes object A, casting a shadow on to it.

First Pass Detail

In the first pass, the scene is rendered from the light's point of view - shadow mapping works easiest when a light is *unidirectional*, such as a spotlight. To perform shadow mapping on an *omnidirectional* light such as pointlights, 6 shadow maps would have to be generated at 90 degree angles - like cube map generation! In this pass, as much of the rendering pipeline as possible is disabled to speed up rendering, as the only required output from this pass is a depth buffer, containing the depths of all of the geometry closest to the light - objects that may potentially occlude the light. Some graphics hardware can render into a depth buffer twice as fast if writing to colour buffers is disabled, and unless the rendered geometry requires alpha manipulation, it is possible to even skip texture sampling in the first pass, to further speed up rendering. At the end of this pass, we'll have a 'shadow map' containing depths from the light's point of view.

Second Pass Detail

In the second pass, the scene is rendered in the way you should be used to be now - from the camera's point of view, with all of the lighting and cube mapping required in the shader. In this second pass, for every object that is drawn, a *shadow matrix* is sent to the vertex shader - this matrix contains the object's model matrix, multiplied by the view matrix of the light's point of view, multiplied by the projection matrix. From this shadow matrix, the final transformed position of each vertex from the light's point of view can be calculated. This can then be interpolated to provide the post-transform position on a per-fragment basis. Then, in the fragment shader, the shadow-transform z value of each fragment can be compared against the depth value at the x and y axis position in the shadow map - if the vertex depth is *less* than (or equal to) the shadow depth, then the vertex is *in front* of whatever occluding object was in the shadow map (or it *is* the occluding object), but if it is *greater* than the depth sampled from the shadow map, then the object is *behind* the occluding object, and so should be in shadow.



A diagram showing the interaction between the matrices and scene drawing from the light's point of view, and that of the camera

Sampling Shadow Maps

Like Cube Mapping, shadow maps are sampled using a specialised texture sampler. This time, the sampler takes in a 4-component vector - remember, transformed vectors have a w component, too. The sampler *usually* then returns either 0.0 or 1.0, corresponding to whether the depth map has a depth greater than the transformed z value of the vector, at the texel at the position x, y of the vector. Unlike colour textures, depth textures contain non-linear information, so they cannot be interpolated in the same way. Fortunately, modern graphics hardware can perform a process known as *Percentage Closer Filtering*, which performs multiple shadow map samples, and then averages the result, providing some level of filtering.

Bias Matrix

In order to sample the depth buffer texture, however, we must do a further transformation. Clip space runs from -1.0 to 1.0 on each axis, but a texture's axis runs from 0.0 to 1.0. So, we need to transform the vector used to sample the shadow map - the opposite way that we transformed tangent space normal map texture samples into vectors. We can do this using the following **bias matrix**:

$$\begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It is a simple transformation matrix, containing a translation and a scale component - it scales the clip space coordinates from being between -1.0 and 1.0 to being between -0.5 to 0.5, via a scale of 0.5 on each axis. It then translates each coordinate by 0.5, moving the coordinates from a space of -0.5 to 0.5 to a space of 0.0 to 1.0.

Shadow Issues

Surface Acne

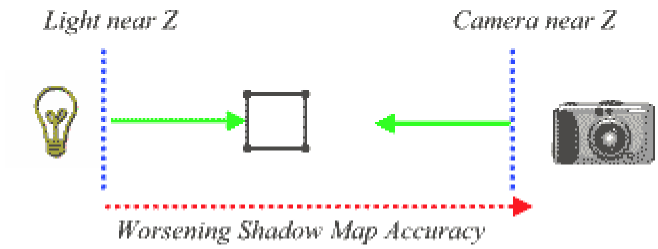
As you'll probably recall, depth buffers aren't entirely accurate, and suffer from rounding issues. You should also remember that depth buffers aren't entirely linear - most of their precision is close to the near plane of the projection matrix. Unfortunately, light occluding objects won't necessarily be close to the light, and therefore won't be in the area of most depth precision. This results in potentially incorrectly shadowed fragments, caused by slightly inaccurate depth comparisons. This is at its worst when calculating the amount of shadow for a visible occluding surface - that is, one visible by both the camera and the shadow casting light. As this scenario results in depth values very similar between the two depth values, it results in what is sometimes known as *surface acne* - where the surface appears to cast a shadow upon itself when the rounding errors result in the shadow map value being less than the fragment depth.

For example, here's a scene from the example program for this tutorial: The skeletal mesh we used earlier in the tutorial series, visible to both a shadow casting spot light, and the camera. With nothing to combat it, surface acne is clearly visible on the HellKnight's face, causing 'strips' of shadow where the slight errors accumulate enough. Thankfully, there are a few solutions to this issue. One is to draw the *back faces* of each object to the shadow map, so the rough shapes of the objects are written to the shadow map, but at a greater depth. Another is to offset the vertices used by the depth comparison by some amount - either offset them towards the camera, or slightly in the direction of their normal. By offsetting the vertices slightly, the depth comparison is more likely to be consistent across a surface, and reduce surface acne.

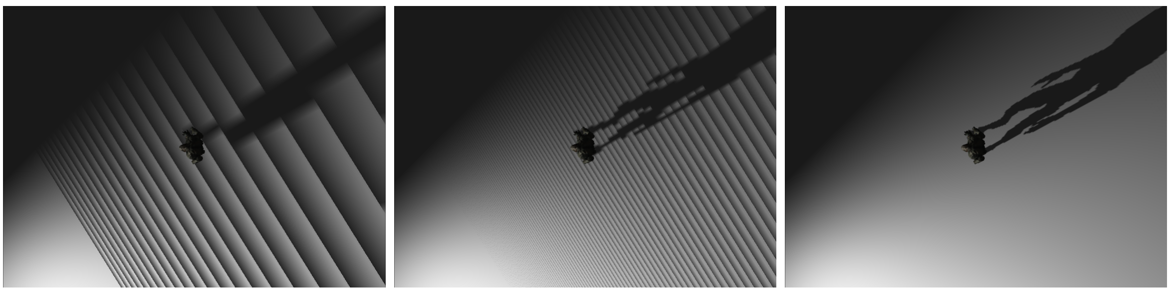
Left: An example of 'surface acne' and Right: Vertex offsetting reduces the acne effect

Aliasing

Depth buffers only store discrete samples of the depth of the scene. This means that one depth value has to cover an area of geometry - and the further away from the light's view point that geometry is, the fewer depth values cover it. This results in *aliasing*, noticeable blockiness around the edges of the shadows. What's worse is that something that's in the distance from the light's point of view, and therefore receiving few depth values, may be close from the *camera's* point of view - this results in a case of *duelling frusta*, where the closer to being parallel the light and camera frusta are, the fewer depth values there are where it counts, up close:



The easiest solution to this is to increase the resolution of the shadow map, so that even far away surfaces have more shadow map resolution covering them. This results in 'sharper' shadows that more accurately represent the object that is casting the shadows, at the expense of taking up more space.



A scene shadowed using shadowmaps of a resolution of 32, 128, and 512 texels square, respectively

There are many potential solutions to the problems of aliasing and surface acne - increasing the shadow map resolution, and adjusting the near and far planes are just two. *Perspective* shadow maps, *exponential* shadow maps, *variance*, and *cascaded* shadow maps are all attempts at creating the 'perfect' solution to the problems of the shadow mapping algorithm. Depending on how large the scenes you wish to create, and the amount of graphics memory you want to dedicate for use as a shadow map, then the basic shadow map technique may suit you fine.

Example Program

To demonstrate shadow mapping, we're going to create a simple graphical demo consisting of the HellKnight skeletal mesh that was introduced earlier in the tutorial series standing on a large 'ground' quad, and a single light that will cause the HellKnight to cast shadows both over itself, and over the ground. We don't need any new nclgl classes, or to even modify any existing ones - all we need is our usual new *Renderer* class, and 4 shader programs. So, in the Tutorial 15 project file, add a *Renderer* class, and 4 text files - *shadowVert.glsl*, *shadowFrag.glsl*, *shadowSceneVert.glsl* and *shadowSceneFrag.glsl*.

Renderer header file

Our Renderer class has a few new functions - 4 **protected** functions to draw the floor and HellKnight, in the two separate render passes required for shadow mapping to work. Then, we have a shader for each render pass, a FBO to handle the shadow map generation pass, and pointers to the required class instances.

On line 7, we also have a **define**, which defines how large the shadow map will be - remember, the higher the shadow map resolution, the less pixellated the resulting shadows will appear. Larger shadow maps take up more memory, though, so by default we'll use a dimension of 2048 - resulting in a 16MB texture, which is still quite large!

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/Camera.h"
5  #include "./nclgl/MD5Mesh.h"
6
7  #define SHADOWSIZE 2048       //New!
8
9  class Renderer : public OGLRenderer {
10 public:
11    Renderer(Window &parent);
12    virtual ~Renderer(void);
13
14    virtual void UpdateScene(float msec);
15    virtual void RenderScene();
16
17 protected:
18    void  DrawMesh();           //New!
19    void  DrawFloor();          //New!
20    void  DrawShadowScene();    //New!
21    void  DrawCombinedScene();  //New!
22
23    Shader*  sceneShader;
24    Shader*  shadowShader;
25
26    GLuint   shadowTex;
27    GLuint   shadowFBO;
28
29    MD5FileData*   hellData;
30    MD5Node*    hellNode;
31    Mesh*    floor;
32    Camera*  camera;
33    Light*   light;
34 };
```

renderer.h

Renderer Class file

We start our *Renderer* class **constructor** by creating a new *Camera* and *Light*, and loading in the skeletal animation mesh and its animation file. Then, we create the two shaders we need - one to create the shadow map, and one to draw the scene, creating shadows from the shadow map as necessary.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     camera = new Camera(-8.0f, 40.0f, Vector3(-200.0f,50.0f,250.0f));
5     light  = new Light(Vector3(-450.0f,200.0f,280.0f),
6                        Vector4(1,1,1,1), 5500.0f);
7
8     hellData     = new MD5FileData("../Meshes/hellknight.md5mesh");
9     hellNode     = new MD5Node(*hellData);
10
11    hellData->AddAnim("../Meshes/idle2.md5anim");
12    hellNode->PlayAnim("../Meshes/idle2.md5anim");
13
14    sceneShader   = new Shader("shadowscenevert.glsl",
15                               "shadowscenefrag.glsl");
16    shadowShader = new Shader("shadowVert.glsl","shadowFrag.glsl");
17
18    if(!sceneShader->LinkProgram() || !shadowShader->LinkProgram()) {
19       return;
20    }
```

renderer.cpp

In the post processing tutorial, we created a depth texture, suitable for using as an FBO's depth attachment. We need to do the same thing here, too, but as we don't need to 'borrow' any bits to store a stencil buffer in this tutorial, we're going to use the OpenGL symbolic constant **GL_DEPTH_COMPONENT** for the format and type, giving us the full 32bits of depth precision. As we're going to be performing a comparison function between this depth texture and the depth buffer, we also need to tell OpenGL how this comparison will be performed. We do so on line 34, using the symbolic constant **GL_TEXTURE_COMPARE_MODE** to define which parameter we're setting, and the symbolic constant **GL_COMPARE_R_TO_TEXTURE** to define that we're testing against the depth value of the texture (R is the axis going 'into' the texture).

```
21    glGenTextures(1, &shadowTex);
22    glBindTexture(GL_TEXTURE_2D, shadowTex);
23    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
24    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
25    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
26
27    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT,
28       SHADOWSIZE, SHADOWSIZE, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
29
30    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
31                    GL_COMPARE_R_TO_TEXTURE);
32
33    glBindTexture(GL_TEXTURE_2D, 0);
```

renderer.cpp

The code for creating a FBO, and a depth attachment for it, should be familiar to you from the post processing tutorial. Note that this time, we have a value of **GL_NONE** as the parameter to **glDrawBuffer** - we don't actually need any colour information for our shadow map pass, so we don't have a colour attachment at all.

```
    glGenFramebuffers(1, &shadowFBO);

    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT ,
                           GL_TEXTURE_2D, shadowTex, 0);
    glDrawBuffer(GL_NONE);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

renderer.cpp

Finally, we create the floor using *GenerateQuad*, and set its texture and bump map. We then enable depth testing, and create a perspective projection appropriate for our scene.

```
    floor = Mesh::GenerateQuad();
    floor->SetTexture(SOIL_load_OGL_texture("../Textures/brick.tga"
        , SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
    floor->SetBumpMap(SOIL_load_OGL_texture("../Textures/brickDOT3.tga"
        , SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));

    glEnable(GL_DEPTH_TEST);

    projMatrix = Matrix4::Perspective(1.0f,15000.0f,
                 (float)width / (float)height, 45.0f);

    init = true;
}
```

renderer.cpp

Nothing too surprising in the **destructor** - we **delete** the FBO and depth texture we created, as well as the shaders (remembering to set *currentShader* to **NULL**, too), and the meshes, camera, and light. *UpdateScene* updates the camera, and the animation for our HellKnight skeletal mesh, but we don't update the view matrix yet - we render from two different viewpoints.

```
Renderer::~Renderer(void)  {
    glDeleteTextures(1,&shadowTex);
    glDeleteFramebuffers(1,&shadowFBO);
    delete camera;
    delete light;
    delete hellData;
    delete hellNode;
    delete floor;

    delete sceneShader;
    delete shadowShader;
    currentShader = NULL;
}

void Renderer::UpdateScene(float msec) {
    camera->UpdateCamera(msec);
    hellNode->Update(msec);
}
```

renderer.cpp

Our *RenderScene* function relies on two subfunctions, *DrawShadowScene*, and *DrawCombinedScene*, so it doesn't contain too much this time around:

```
72 void Renderer::RenderScene()  {
73    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
74
75    DrawShadowScene();   //First render pass...
76    DrawCombinedScene(); //Second render pass...
77
78    SwapBuffers();
79 }
```

renderer.cpp

To add real-time shadows to our scene, we must first generate a depth buffer image from the pointlight's point of view, using a frame buffer object. So, like the post processing tutorial Frame Buffer Objects were introduced in, to render into an FBO we must first *bind* it (line 82) and *clear* whatever was previously in its buffers (line 84). Next up is a function you may not have used much before, **glViewport**. Unlike last time we used FBOs, our shadow map depth buffer is much bigger than the screen resolution - 2048 by 2048. So, to render into all of it, we must temporarily increase OpenGL's virtual window size. Then, we disable all colour writes using **glColorMask** - we only have a depth buffer attached to the shadowmap FBO, so there's no point processing colours, and in fact it may improve performance to do so. Then, we enable the simple shader program we'll define later.

```
80 void Renderer::DrawShadowScene() {
81    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);
82
83    glClear(GL_DEPTH_BUFFER_BIT);
84
85    glViewport( 0, 0, SHADOWSIZE, SHADOWSIZE);
86
87    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
88
89    SetCurrentShader(shadowShader);
```

renderer.cpp

Next, we create a view matrix, and a texture matrix, and and update the current shader's **uniforms**. The view matrix is created using a helper function of the *Matrix4* nclgl class - *BuildViewMatrix*. This takes two vectors - a viewpoint position, and a point in space you want to look at. We want our spotlight to look at the origin, where the HellKnight is standing, so we use *BuildViewMatrix* with parameters of the spotlights position, and the origin.

In the texture matrix, we're going to keep the 'shadow' *view projection* matrix of the spotlight, multiplied by the *bias matrix* we introduced earlier to keep coordinates transformed by this matrix to a 0.0 to 1.0 range suitable for sampling a texture map with. The *biasMap* variable defined for you in the *OGLRenderer* class header file, as a **static const** variable. Once we've update the current shader's matrices, we can draw the floor and the HellKnight, filling the FBO depth buffer with depth information from the spotlight's point of view. Finally, we clear up by unbinding the shader and FBO, enabling writing to colour buffers, and returning the OpenGL viewport to the window size.

```
90    viewMatrix     = Matrix4::BuildViewMatrix(
91                     light->GetPosition(),Vector3(0,0,0));
92    textureMatrix  = biasMatrix*(projMatrix*viewMatrix);
93
94    UpdateShaderMatrices();
95
96    DrawFloor();
```

```
97     DrawMesh();
98
99     glUseProgram(0);
100    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
101    glViewport( 0, 0, width, height );
102
103    glBindFramebuffer(GL_FRAMEBUFFER, 0);
104 }
```

renderer.cpp

Once we have a shadow map depth buffer, we can use it in our main draw pass. *DrawCombinedScene* is quite similar to how we've set up per pixel lighting in the previous tutorials. The only major changes are on line 112 we set a **uniform** shader sampler, *shadowTex*, to sample from texture unit 2, to which we **bind** the shadow FBO depth buffer. Unlike other tutorials, we also wait 'til rendering to update the view matrix (line 123), due to the way we render the scene from multiple view points.

```
105 void Renderer::DrawCombinedScene() {
106    SetCurrentShader(sceneShader);
107    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
108                "diffuseTex"), 0);
109    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
110                "bumpTex"), 1);
111    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
112                "shadowTex"), 2);
113
114    glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
115                 "cameraPos")   ,1,(float*)&camera->GetPosition());
116
117    SetShaderLight(*light);
118
119    glActiveTexture(GL_TEXTURE2);
120    glBindTexture(GL_TEXTURE_2D, shadowTex);
121
122    viewMatrix = camera->BuildViewMatrix();
123    UpdateShaderMatrices();
124
125    DrawFloor();
126    DrawMesh();
127
128    glUseProgram(0);
129 }
```

renderer.cpp

Both *DrawShadowScene* and *DrawCombinedScene* rely on a further two functions to draw the HellKnight and floor, respectively. They both set the model and texture matrices, and call the appropriate *Draw* function. Remember, in this tutorial, we're borrowing the texture matrix to hold the shadow matrix of the spotlight we're casting shadow maps from. We then multiply that by the model matrix of the current object, giving us a model view projection matrix to send to the vertex shader to perform shadow map comparisons with.

```
130 void Renderer::DrawMesh()  {
131    modelMatrix.ToIdentity();
132
133    Matrix4 tempMatrix  = textureMatrix * modelMatrix;
134
135    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
136          , "textureMatrix"),1,false, *&tempMatrix.values);
```

```
    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
            , "modelMatrix")  ,1,false, *&modelMatrix.values);

    hellNode->Draw(*this);
}

void Renderer::DrawFloor() {
    modelMatrix = Matrix4::Rotation(90, Vector3(1,0,0)) *
                  Matrix4::Scale(Vector3(450,450,1));
    Matrix4 tempMatrix  = textureMatrix * modelMatrix;

    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
            , "textureMatrix"),1,false, *&tempMatrix.values);
    glUniformMatrix4fv(glGetUniformLocation(currentShader->GetProgram()
            , "modelMatrix")  ,1,false, *&modelMatrix.values);

    floor->Draw();
}
```

renderer.cpp

Shadow Map Generation Shader

To render the scene from the spotlight's point of view, we're going to use the most basic shader imaginable - in the vertex program we transform the incoming vertex position, and in the fragment program we output a simple colour. We'll never even see that colour, as we disabled colour writing, a fragment shader just must output a colour. That's it!

Vertex Shader

```
#version 150 core

uniform mat4 modelMatrix;
uniform mat4 viewMatrix;
uniform mat4 projMatrix;

in  vec3 position;
void main(void)   {
    gl_Position = (projMatrix * viewMatrix * modelMatrix) *
                  vec4(position, 1.0);
}
```

shadowVert.glsl

Fragment Shader

```
#version 150 core

out vec4 gl_FragColor;

void main(void)   {
    gl_FragColor   = vec4(1.0);
}
```

shadowFrag.glsl

Shadow Scene Shader

Vertex Shader

For the shader program we're going to use to render from the camera's viewpoint, we're going to make just a few changes from the shader program's we had by the end of Tutoral 13. We need another vertex output block variable, *shadowProj*, that will hold the final projected position for the current vertex *from the spotlight's point of view*. We assign this variable on line 37, by multiplying the vertex position by the textureMatrix - we're borrowing the texture matrix and storing the modelviewprojection matrix of the spotlight in it. You'll also see how we add the normal to the vertex - this reduces the possibility of the self-shadowing 'surface acne' effect occurring, by *slightly* expanding outward the position coordinates we're going to use to perform shadow mapping with.

```
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 in  vec3 colour;
10 in  vec3 normal;
11 in  vec3 tangent;
12 in  vec2 texCoord;
13
14 out Vertex {
15    vec3 colour;
16    vec2 texCoord;
17    vec3 normal;
18    vec3 tangent;
19    vec3 binormal;
20    vec3 worldPos;
21    vec4 shadowProj;  //New!
22 } OUT;
23
24 void main(void)      {
25    mat3 normalMatrix = transpose(inverse(mat3(modelMatrix)));
26
27    OUT.colour     = colour;
28    OUT.texCoord   = texCoord;
29
30    OUT.normal     = normalize(normalMatrix * normalize(normal));
31    OUT.tangent    = normalize(normalMatrix * normalize(tangent));
32    OUT.binormal   = normalize(normalMatrix *
33                     normalize(cross(normal,tangent)));
34
35    OUT.worldPos   = (modelMatrix * vec4(position,1)).xyz;
36 //New!
37    OUT.shadowProj  = (textureMatrix * vec4(position+(normal*1.5),1));
38
39    gl_Position    = (projMatrix * viewMatrix * modelMatrix) *
40                     vec4(position, 1.0);
41 }
```

shadowscenevert.glsl

Fragment Shader

Like the vertex program, the fragment program is a modification of the *SceneFragment* shader program from Tutorial 13. We need a new uniform texture sampler, shadowTex, defined on line 5. It's of a special type, **sampler2DShadow**, enabling us to perform special types of texture sampling on it, like the **samplerCube** last tutorial. On line 21, we have our new *shadowProj* vertex attribute.

```
1  #version 150 core
2
3  uniform sampler2D    diffuseTex;
4  uniform sampler2D    bumpTex;
5  uniform sampler2DShadow shadowTex;     //NEW!
6
7  uniform vec4   lightColour;
8  uniform vec3   lightPos;
9  uniform vec3   cameraPos;
10 uniform float  lightRadius;
11
12 in Vertex {
13    vec3 colour;
14    vec2 texCoord;
15    vec3 normal;
16    vec3 tangent;
17    vec3 binormal;
18    vec3 worldPos;
19    vec4 shadowProj;  //New!
20 } IN;
21
22 out vec4 gl_FragColor;
```

shadowscenefrag.glsl

The start of our **main** function is identical to tutorial 13, creating the TBN matrix and direction vectors required to calculate each of the lighting components.

```
23 void main(void)   {
24    mat3 TBN        = mat3(IN.tangent, IN.binormal, IN.normal);
25    vec3 normal        = normalize(TBN *
26             (texture2D(bumpTex, IN.texCoord).rgb * 2.0 - 1.0));
27
28    vec4 diffuse       = texture2D(diffuseTex, IN.texCoord);
29
30    vec3  incident     = normalize(lightPos - IN.worldPos);
31    float lambert      = max(0.0, dot(incident, normal));  //Different!
32
33    float dist         = length(lightPos - IN.worldPos);
34    float atten        = 1.0 - clamp(dist / lightRadius, 0.0, 1.0);
35
36    vec3 viewDir       = normalize(cameraPos - IN.worldPos);
37    vec3 halfDir       = normalize(incident + viewDir);
38
39    float rFactor      = max(0.0, dot(halfDir, normal));   //Different!
40    float sFactor      = pow(rFactor, 33.0 );
```

shadowscenefrag.glsl

On line 41, we have a new **float** , simply called shadow. You can think of this as the inverse amount of shadowing for each fragment, so the starting value of 1.0 equates to their being no shadowing at all. Then, on line 43, we have an **if** statement - if the vertex, transformed to be in relation to the spotlight, has a positive w component, and is therefore in front of the spotlight, we sample the shadowmap, using another new glsl function *textureProj*, which takes in a 4 component vector - our vertices clip space position from the spotlight's point of view, and a sampler, in this case our shadow map.

Internally, the **textureProj** function performs the perspective divide on the input vector, multiplying x, y, and z by w. It then samples the texture sampler parameter at the x and y positions, and, as we've passed in a depth map with a *comparison* flag set on it, returns the comparison value between the depth map, and the z value of the vector. So, if the vector's post-divide z value is greater than that of the equivalent texel of the depth texture, it returns 0.0, otherwise returning 1.0. If the rendering hardware supports shadow map filtering, you might get a filtered value between 0.0 and 1.0, providing slightly softer edges to your shadows.

Once we have the amount of shadow for the current fragment, we need to use it! We're going to cheat a little bit, and multiply the *lambert* variable by the *shadow* variable, so fragments in shadow end up with a *lambert* of zero, and thus receive no diffuse or specular lighting. They'll still receive a little bit of ambient lighting, though. That's everything - it's pretty easy to add in shadow mapping, as the graphics hardware does all of the difficult comparison work for us!

```
    float shadow        = 1.0;    //New!

    if(IN.shadowProj.w > 0.0) { //New!
        shadow     = textureProj(shadowTex,IN.shadowProj);
    }

    lambert *= shadow;    //New!

    vec3 colour         =  (diffuse.rgb * lightColour.rgb);
    colour              += (lightColour.rgb * sFactor) * 0.33;
    gl_FragColor        =  vec4(colour * atten * lambert, diffuse.a);
    gl_FragColor.rgb   += (diffuse.rgb * lightColour.rgb) * 0.1;
}
```

shadowscenefrag.glsl

Tutorial Summary

Upon running the example program, you should be able to see the HellKnight, standing in the middle of a brickwork floor. As well as the usual diffuse and specular highlights on the surfaces, you should also be able to see that the HellKnight casts a shadow on itself, and a shadow extends behind it across the floor. That's the result of the shadow map comparison performed in the fragment shader. Although there are many more ways of implementing shadow mapping, many of which result in higher quality shadows, the basic shadow mapping technique shown here is a good starting point, and should help add further realism to your scenes. In the next tutorial, you'll take a look at a lighting technique called *deferred rendering*, which allows 10s or even 100s of lights to be visible on screen at once, at minimal processing cost.

Further Work

1) Try adding movement controls to the spotlight, so you can see how shadows react to the spotlight's position.

2) Maybe you'd like to try adding in another spotlight? You'll need another depth texture...

3) How about a *pointlight*? You'll need 6 depth textures, and 6 more render passes, but it should be possible! What type of texture could be used to store all 6 viewpoints?