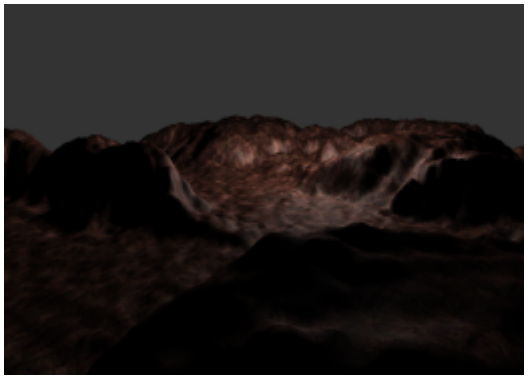
Tutorial 11: Real-Time Lighting A



Summary

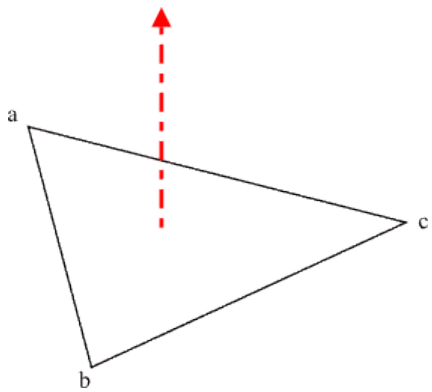
The graphical applications you have been making so far have been very basic, with only texture maps to add realisim. Lighting effects have long been used to enhance the realism in games, and the following two tutorials will show you how to perform such lighting calculations in your shaders.

New Concepts

Vertex Normals, Normal Matrix, Flat Shading, Gourard Shading, Blinn-Phong Shading, Specularity, Lambert's Cosine Law

Normals

In order to be able to simulate the effects of light bouncing off the surfaces of the meshes in our games, we must first be able to determine the direction the surfaces are facing in. We do this using the *normal* of each surface. This is a normalised direction vector, pointing away from the surface, like so:



Like everything else related to geometry rendering, normals are vertex *attributes*, and so are interpolated in the vertex shader, forming a surface normal for each fragment in the scene.

Calculating Normals

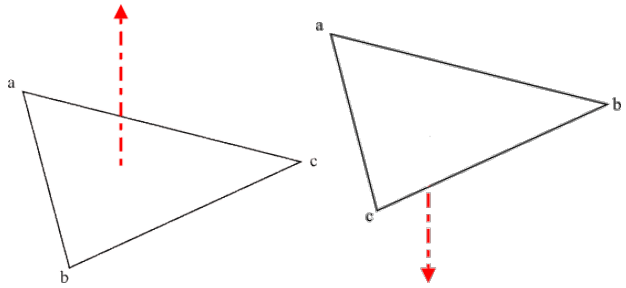
Nearly all of the geometry you'll render on screen is made up of triangles, and it is fortunately very easy to calculate the surface normal of a triangle. The *cross product* of two vectors a and b produces a vector that is orthogonal to both a and b, ideal to use as a surface normal:

$$cross(Vector3a, Vector3b) = Vector3(a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x)$$

But which two vectors will result in the correct normal for a triangle? Simple - we pick one of the triangle's vertices, and the two direction vectors that point towards the other two triangle vertices can be used to generate the normal for a triangle by cross product. Remember though, that surface normals should be normalised!

$$surface\ normal = normalise(b - a \times c - a)$$

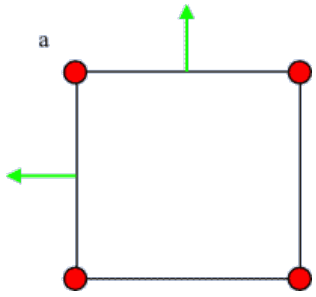
Which triangle vertices get used as a, b and c depends on the vertex *winding*, a concept you should remember from earlier in the tutorial series. The winding will determine which face of the triangle the resulting normal will be facing away from - the cross product of vectors a and b is the **inverse** of the cross product of vectors b and a:



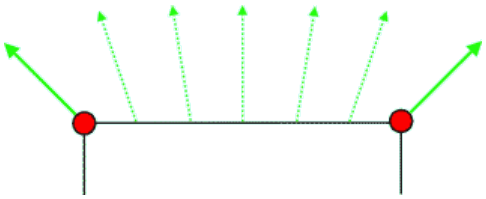
Left: Anti-clockwise vertex winding Right: Clockwise vertex winding

Normal issues

OK, so now you know how to calculate the normal for a surface, but there's a problem - normals are defined at the *vertex* level. You might think that just setting the vertex normal of every vertex in a surface to equal the surface normal would be enough, but that won't always work. Earlier in the tutorial series, you were introduced to index buffers, a rendering technique that allows vertices to be reused across a model to improve performance and save memory. If each vertex stores a single normal, what happens in cases where a single vertex is part of multiple faces, each of which faces in a wildly differing direction. Take for example, vertex a of the cube below:



Vertex a is part of three faces, one facing up, one facing left, and one facing 'out'. The green arrows indicate the intuitive surface normal for the respective faces, but which would be the correct normal to use as an attribute for vertex a? No matter which face is chosen, the others will end up with a vastly incorrect normal. One common solution is to use the normalised *sum* of the normals for each face the vertex is used in. As normals get interpolated by the vertex shader, we get a normal that approximately represents each surface:

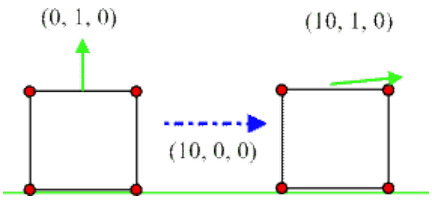


Solid arrows are the normalised sum normals for each vertex, dotted arrows are interpolated values

Cubes are something of a worse-case scenario for per-vertex normals with indices, and quite often, mesh artists will forgo indexing on cubes, so that more accurate normals can be used per-face. Otherwise, this normalised sum method works quite well, as you'll see in the example program for this tutorial, where it'll be used to generate smooth normals for the heightmap landscape we created.

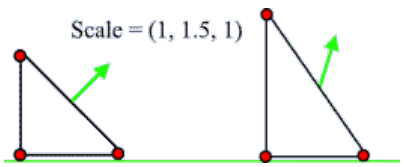
Normal Matrix

If a normal faces out from a vertex or surface, it should be fairly obvious that if the vertex is transformed (i.e by the model matrix), then so too should the vertices normal. But which matrix should be used to transform the normal? You might initially assume you can just use the model matrix to transform the normal, but it turns out we can't. The model matrix might contain translation information, and so the further from the origin this translation is, the more distorted the normal direction would become:



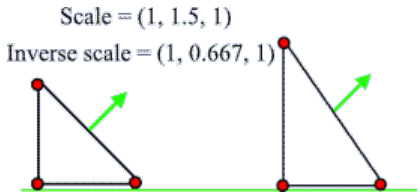
At the origin, the upward facing normal is correct, but as the box is translated, so too is the normal

OK, so that's the model matrix out. But how about if we just ignored the translation component of the model matrix, and used the upper 3x3 section? Well, that *might* work, except in cases where there was a *non-uniform scale* applied to the model matrix:



A non-uniform scale, in this case in the y-axis, distorts the resulting transformed normal

It's a bit of a pain to try and detect non-uniform scales in the model matrix, so we need another solution. To fix the non-uniform scale problem, we need to use the *inverse transpose* of the model matrix. This will preserve the *rotations* in the model matrix, but will *invert* the scales, resulting in normals that still face in the correct direction:



Using the inverse scale will preserve normal directions

As matrix inversion is quite costly, if it is guaranteed that a model matrix has a uniform scale, just using the model matrix is fine, but calculating the inverse transpose in the vertex shader is a good general purpose solution to normal rotation.

Lighting

Game engines have performed lighting calculations since long before hardware accelerated graphics became the norm, stretching back to the per-sector lighting found in games like *Doom*. As graphics hardware has improved, so too have the lighting models used in games, with lighting calculations moving from being per-face, to per-vertex, and finally to being on a per-fragment basis. On whatever level the lighting is calculated, there are various models used to determine the final colour of a lit surface, although they are all generally based around the *Phong reflection model*, which splits light into the notions of ambience, diffusion, and specularity, each of which has its own colour.

Ambience

Ambient light represents light that has bounced off numerous surfaces in a scene, creating a small, even amount of light that lights everything in a scene equally. Ambient lighting has no direction, only a brightness, and so it is trivial to compute the amount of ambient light created by a light source:

$$ambience = ambient\ light\ colour$$

Diffusion

Diffusion represents how much light from a given light source is scattered off a surface in all directions - the light is *diffused* uniformly, and is visible from all directions equally. The closer a surface is to directly facing a light source, the more light it receives, and so the brighter it appears. This can be calculated using the normal of a surface, as introduced earlier. We can work out exactly how much light is diffused by a surface using *Lambert's cosine law*, which states that the brightness of a surface is directly proportionate to the cosine between the surface's normal and the 'incident' vector. This incident vector is the normalised direction vector between the light source and the surface, and can be calculated simply:

$$incident = normalise(light\ position - surface\ position)$$

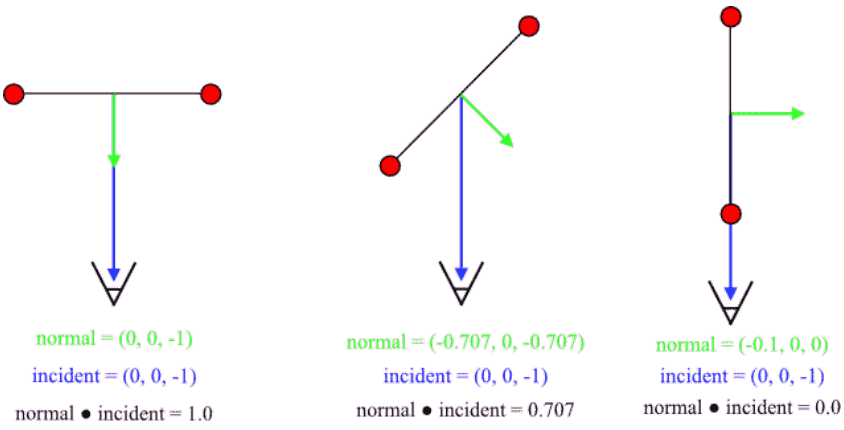
We can determine the cosine of the two normalised direction vectors by calculating their *dot product*, an operation which can be performed as follows:

$$\text{dot(Vector3 a, Vector3 b)} = a_x b_x + a_y b_y + a_z b_z$$

As cosines range from 1.0 and -1.0, the dot product result should be *clamped* to be between 0.0 and 1.0 to correctly work out the final amount of diffused light reflected from a surface:

$$diffusion = diffuse\ colour \cdot clamp(dot(normal,incident),\ 0.0,\ 1.0)$$

This works fine as a cosine of less than 0.0 would mean a surface was facing away from the light and so should receive no light from the light source.

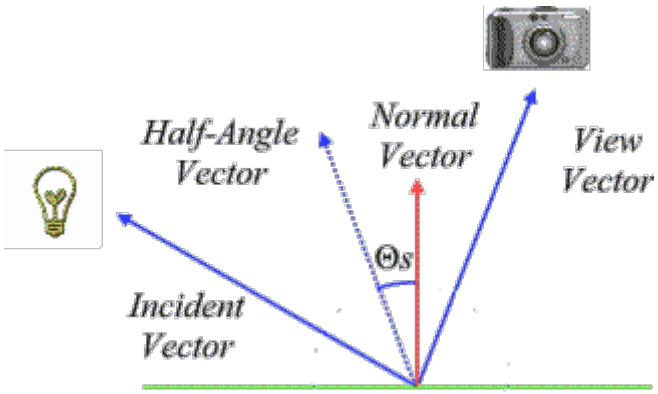


Example of the amount of Lambertian diffusion reducing as the surface faces further away from the view point

Specularity

While diffusion represents light that has scattered equally in all directions, *specularity* models light that has been reflected directly towards the view source. This specular reflection causes blurry 'spotlights' to appear on the surface of an object, with the sharpness of these spotlights being proportional to the smoothness of the reflecting surface. This can be explained by *microfacets* - flat surfaces are assumed to be made up of millions of tiny surfaces, each pointing in a slightly different direction; with rougher surfaces having microfacets that diverge further from the surface normal, and so reflecting less light directly towards the viewer. The final specularity of a surface is worked out as follows:

$$Specularity\ Value = specular\ colour \cdot clamp(dot(normal,half\text{-}angle),\ 0.0,\ 1.0)^n$$



Where *normal* is the surface normal, *half-angle* is the angle half way between the incident vector and the normalised direction vector between the view and surface, and n is the*specular power* - which models how smooth a surface is.

An example of the differing specular powers. Left to right: 1.0, 10.0, 100.0

Attenuation

We now know how to calculate the ambient, diffuse and specular lighting terms of the Phong reflection model, but there's still one thing missing. In our lighting model so far, a candle will light a scene just as much as a huge spotlight! What we really want, is for the light emitted by our light sources to *attenuate* with distance. Attenuation isn't really physically accurate, but is cheap to compute, and an effective way of creating varying light sources. We can calculate how far away a surface is from a light source easily, by calculating the *length* of the vector formed by subtracting the surface position from the light source position:

$$distance = length(light\ position - surface\ position)$$

This can be extended out to calculate the distance of each fragment being processed by interpolation - just like vertex attributes. There are various methods by which to work out how attenuated a

light is by distance, the easiest of which is simply *linearly*. If our light source has maximum distance within which it will light a surface (for a spotlight this is essentially a radius, forming a sphere of light around the light position), we can work out linear attenuation as follows:

$$attenuation = 1.0 - clamp(distance\ from\ light\ /\ maximum\ distance,\ 0.0,\ 1.0)$$

Again, we want our attenuation to be clamped between 0.0 and 1.0, as otherwise we'd get negative results, and the maximum distance would not work correctly.

Lighting Calculation

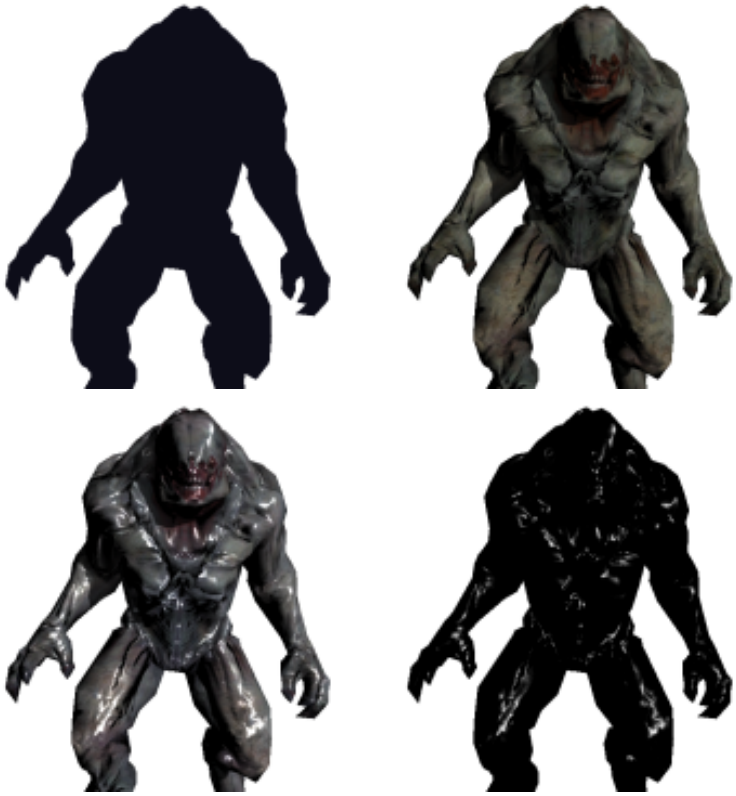
The final result of the Phong reflection model of ambience, diffusion, specularity, and attenuation is as follows:

$$Final\ Diffuse = surface\ diffuse\ colour \cdot light\ diffuse\ colour \cdot brightness \cdot attenuation$$

$$Final\ Specularity = Surface\ specular\ colour \cdot light\ specular\ colour \cdot Specularity\ Value \cdot attenuation$$

$$Final\ Colour = Ambience + Final\ Diffuse + Final\ Specularity$$

In practice, some games will omit separate surface and light specular colours, in order to save processing and space. Also, note how ambient light does not get attenuated.



An example of the lighting terms of the Phong reflection Model. Clockwise from top left: Ambient, diffuse, specular, final image

Lighting Models

The Phong reflection model can be applied to your geometry in various ways, each of which has a differing level of computational complexity.

Flat Shading

Flat shading is the simplest method of performing real-time lighting. Every surface has a single normal, from which the lighting is derived. No interpolation takes place - every pixel that makes up any given surface is lit identically.

Gouraud Shading

In Gouraud shading, the diffuse, specularity, and attenuation is calculated on a *per-vertex* basis, and then interpolated between vertices to calculate the final colour per pixel - just as vertex colours were interpolated to create the triangle colours way back in Tutorial 1! This is more computationally expensive than Flat shading, but cheaper than working out the value per-fragment, as it is a simple interpolation operation. How good Gouraud shading looks is highly dependent on how many vertices make up a model - the more vertices, the more accurate the interpolation will be.

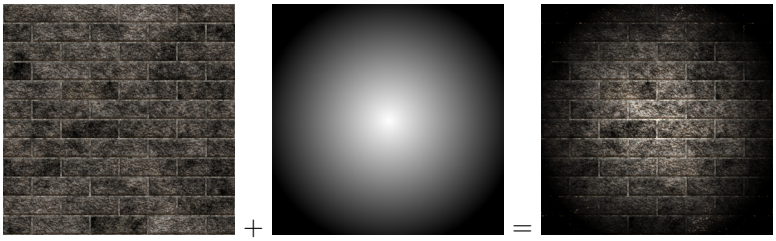
Phong Shading

Finally, in Phong shading everything is calculated on a per-fragment basis, although normals are still obtained by interpolating between vertices. Obviously, this is the most expensive of the lighting models, but modern hardware is capable of performing phong shading for multiple lights in real-time.

An identical mesh rendered using the Flat, Phong, and Gourard lighting models, respectively.

Static Lighting

Another popular lighting method is to use a *lightmap*. Instead of performing costly lighting calculations in real-time, the levels in some games undergo an extensive pre-processing step, in which the lighting for every polygonal face in the world is processed offline, and stored in a unique lightmap texture per face. This lightmap can then be blended with the desired texture map using multitexturing at runtime, creating a realistic, albeit static representation of light and shadow in a game world. The individual lightmaps for each face are generally of a far lower resolution than the textures they will be blended with to save space, relying on filtering to create a smooth lighting gradient. Lightmaps also do not have specularity information - remember, specularity is determined by the camera position, which can only be determined at run-time! Even modern games like id Software's *Rage* use such pre-calculated lighting to efficiently add light and shadow to their levels.



An example of a simple multitextured lightmap

Example Program

The example program for this tutorial will build upon our indexed heightmap, and add a single point light to the centre of our landscape. We're going to need a new class, to represent a simple light, so add a class called *Light* to the *nclgl* Visual Studio project. We're also going to need a new shader to calculate our lighting, so add two files called *PerPixelVertex.glsl* and *PerPixelFragment.glsl* to the *Shaders* folder. The main file for this tutorial doesn't contain any new functionality, so the contents of this tutorial's main file is identical to that of Tutorial 9.

Mesh Class

Lighting requires normals, so we need to add some functionality to the *Mesh* class. We need a new *MeshBuffer* **enum** to represent the normal vertex attribute. We also need a new **protected** function to generate normal data, and a new member variable - a **pointer** to some normal data.

Header File

```
1  enum MeshBuffer {
2     VERTEX_BUFFER, COLOUR_BUFFER, TEXTURE_BUFFER,
3     NORMAL_BUFFER, INDEX_BUFFER,  MAX_BUFFER
4  };
```

Mesh.h

```
5  class Mesh  {
6  ...
7  protected:
8  void        GenerateNormals();
9
10 Vector3*    normals;
11 ...
12 };
```

Mesh.h

Class File

In our **constructor**, we should make our new *normals* pointer have a **NULL** value, and in the **destructor**, we should **delete** it.

```
1  Mesh::Mesh(void)  {
2  ...
3  normals    = NULL;
4  ...
5  }
```

Mesh.cpp

```
6  Mesh::~Mesh(void) {
7  ...
8  delete[]normals;
9
10 ...
11 }
```

Mesh.cpp

To create the normals to use in our applications, we are going to use the *GenerateNormals* function. This needs to be able to handle the two different vertex storage cases - with and without indices. In either case, we check to see whether we already have allocated memory to store normal data, and do so if not. If there is already normal data, we need to set them all to be zero. If this seems slightly unintuitive, remember that the skeletal animation classes we created earlier in the series modify their vertex information, and will require normals to be regenerated.

If the indices pointer is not **null**, we can reasonably assume that the *Mesh* has indexed vertices, so on line 19 we have an if statement to decide whether to calculate the normalised sum of all contributing faces or not. In either case, we do much the same thing - we grab the next triangle (lines 21-23, 35-37), calculate the cross product from the inter-vertex direction vectors, in anti clockwise order (lines 25, 39), and then set the appropriate location in the *normals* array. Note that the resulting normal is added to the existing value if the vertices are indexed (lines 28-30). Finally, on lines 47-49, we normalise the resulting normals to unit length.

```
12 void  Mesh::GenerateNormals() {
13    if(!normals) {
14       normals = new Vector3[numVertices];
15    }
16    for(GLuint i = 0; i < numVertices; ++i){
17       normals[i] = Vector3();
18    }
19    if(indices) { //Generate per-vertex normals
20       for(GLuint i = 0; i < numIndices; i+=3){
21          unsigned int a = indices[i];
22          unsigned int b = indices[i+1];
23          unsigned int c = indices[i+2];
24
25          Vector3 normal = Vector3::Cross(
26                (vertices[b]-vertices[a]),(vertices[c]-vertices[a]));
27
28          normals[a]     += normal;
29          normals[b]     += normal;
30          normals[c]     += normal;
31       }
32    }
33    else{ //It's just a list of triangles, so generate face normals
34       for(GLuint i = 0; i < numVertices; i+=3){
35          Vector3 &a     = vertices[i];
36          Vector3 &b     = vertices[i+1];
37          Vector3 &c     = vertices[i+2];
38
39          Vector3 normal = Vector3::Cross(b-a,c-a);
40
41          normals[i]     = normal;
42          normals[i+1]   = normal;
43          normals[i+2]   = normal;
44       }
45    }
46
47    for(GLuint i = 0; i < numVertices; ++i){
48       normals[i].Normalise();
49    }
50 }
```

Mesh.cpp

In order to buffer our mesh vertex normals, we must modify the *BufferData* function, just as we did for texture coordinates a while back. The code to buffer a number of **Vector3s** should be pretty familiar to you by now, just remember to use the new *NORMAL_BUFFER* **enum**:

```
51 void  Mesh::BufferData()   {
52 ...
53    if(normals) {
54       glGenBuffers(1, &bufferObject[NORMAL_BUFFER]);
55       glBindBuffer(GL_ARRAY_BUFFER, bufferObject[NORMAL_BUFFER]);
56       glBufferData(GL_ARRAY_BUFFER, numVertices*sizeof(Vector3),
57                    normals, GL_STATIC_DRAW);
58       glVertexAttribPointer(NORMAL_BUFFER, 3, GL_FLOAT, GL_FALSE,0,0);
59       glEnableVertexAttribArray(NORMAL_BUFFER);
60    }
61 ...
```

Mesh.cpp

HeightMap Class

Our *HeightMap* class needs one *tiny* change - before we we call *BufferData* in its **constructor**, we need to call our new *GenerateNormals* function.

```
1 HeightMap::HeightMap(std::string name) {
2 ...
3    GenerateNormals();
4    BufferData();  //From previous tutorial...
5 ...
```

HeightMap.cpp

Shader Class

A new vertex attribute means we have to change the *Shader* function *SetDefaultAttributes* slightly, so we can automatically set the normal attribute when necessary:

```
1 void  Shader::SetDefaultAttributes()   {
2    glBindAttribLocation(program, VERTEX_BUFFER,  "position");
3    glBindAttribLocation(program, COLOUR_BUFFER,  "colour");
4    glBindAttribLocation(program, NORMAL_BUFFER,  "normal");//New!
5    glBindAttribLocation(program, TEXTURE_BUFFER, "texCoord");
6 }
```

Shader.cpp

Light Class

The lights we're going to be simulating in our example programs are simple point lights - they illuminate the area around them, creating a sphere of light. So, our *Light* class needs a position, a radius (for attenuation), and a colour - we're keeping things really simple, and using the same colour for ambience, diffusion, and specularity.

```
#pragma once

#include "Vector4.h"
#include "Vector3.h"

class Light {
public:
   Light(Vector3 position, Vector4 colour, float radius) {
      this->position = position;
      this->colour   = colour;
      this->radius   = radius;
   }

   ~Light(void){};

   Vector3  GetPosition() const         { return position; }
   void     SetPosition(Vector3 val)    { position = val; }

   float    GetRadius() const           { return radius; }
   void     SetRadius(float val)        { radius = val; }

   Vector4  GetColour() const           { return colour; }
   void     SetColour(Vector4 val)      { colour = val; }

protected:
   Vector3  position;
   Vector4  colour;
   float    radius;
};
```

Light.h

OGLRenderer Class

For the remaining tutorials in this series, we're going to be dealing with lights, in one way or another. This means we're going to have to send the data from our new Light class to shaders as uniform variables. To make this easier, we're going to add a new **protected** function to our *OGLRenderer* base class - *SetShaderLight*.

Header file

```
class OGLRenderer {
...
protected:
void        SetShaderLight(const Light &l);
...
}
```

OGLRenderer.h

Class file

The function itself is pretty trivial, simply setting the **uniform** variables *lightPos*, *lightColour*, and *lightRadius* of the currently bound shader to the values of the *Light* parameter.

```
void OGLRenderer::SetShaderLight(const Light &l) {
   glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
         "lightPos")    ,1,(float*)&l.GetPosition());

   glUniform4fv(glGetUniformLocation(currentShader->GetProgram(),
         "lightColour"),1,(float*)&l.GetColour());

   glUniform1f(glGetUniformLocation(currentShader->GetProgram() ,
         "lightRadius"),l.GetRadius());
}
```

OGLRenderer.cpp

Renderer Class

That's all of the inner workings of our rendering system updated, now to apply them in our example program's *Renderer* class.

Header file

The class definition for the Renderer class itself is pretty standard. All that has changed from Tutorial 6 is we have a new **protected** pointer to a *Light* class instance.

```
#pragma once

#include "./nclgl/OGLRenderer.h"
#include "./nclgl/Camera.h"
#include "./nclgl/HeightMap.h"

class Renderer : public OGLRenderer    {
public:
   Renderer(Window &parent);
   virtual ~Renderer(void);

   virtual void RenderScene();
   virtual void UpdateScene(float msec);

protected:
   Mesh*       heightMap;
   Camera*     camera;
   Light*      light;
};
```

renderer.h

Renderer Class file

As ever, our *Renderer* class file begins by including the *Renderer* class header file, and defining the class **constructor**. We first initialise our *camera*, *heightMap*, and *currentShader* member variables, with the shader using the two new source files we created earlier. You should recognise the rest of the code from the height map tutorial:

```
1  #include "Renderer.h"
2  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
3     camera      = new Camera(0.0f,0.0f,Vector3(
4     RAW_WIDTH*HEIGHTMAP_X / 2.0f,500,RAW_HEIGHT*HEIGHTMAP_Z));
5
6     heightMap       = new HeightMap("../Textures/terrain.raw");
7     currentShader   = new Shader("../Shaders/PerPixelVertex.glsl",
8                        "../Shaders/PerPixelFragment.glsl");
9
10    heightMap->SetTexture(SOIL_load_OGL_texture(
11                "../Textures/Barren Reds.JPG", SOIL_LOAD_AUTO,
12                SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
13
14    if(!currentShader->LinkProgram() || !heightMap->GetTexture() ) {
15       return;
16    }
17
18    SetTextureRepeating(heightMap->GetTexture(),true);
```

renderer.cpp

Next up, we initialise our Light member variable, with a position in the middle of our heightmap, a colour of white, and a radius of half the width of the heightmap. This should give us a nicely attenuated scene, bright in the middle, with the corners descending into darkness. Finally, we create a perspective projection matrix with a *farZ* value large enough to view the entire heightMap at once, enable depth testing, and set *init* to **true**.

```
19    light = new Light(Vector3((RAW_HEIGHT*HEIGHTMAP_X / 2.0f),
20                500.0f,(RAW_HEIGHT*HEIGHTMAP_Z / 2.0f)),
21                Vector4(1,1,1,1), (RAW_WIDTH*HEIGHTMAP_X) / 2.0f);
22
23    projMatrix = Matrix4::Perspective(1.0f,15000.0f,
24                 (float)width / (float)height, 45.0f);
25
26    glEnable(GL_DEPTH_TEST);
27    init = true;
28 }
```

renderer.cpp

The *Renderer* class **destructor** simply **deletes** the camera, light, and heightmap mesh, while *UpdateScene* updates the camera and the view matrix.

```
29 Renderer::~Renderer(void)  {
30    delete camera;
31    delete heightMap;
32    delete light;
33 }
34
35 void Renderer::UpdateScene(float msec) {
36     camera->UpdateCamera(msec);
37     viewMatrix    = camera->BuildViewMatrix();
38 }
```

renderer.cpp

Finally, the *RenderScene* function is pretty similar to that of index buffers tutorial, but this time we need to let our shader know where the camera is in world space (line 57), and call our new function *SetShaderLight* before calling the heightmap's *Draw* function.

```
39 void Renderer::RenderScene()  {
40    glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
41
42    glUseProgram(currentShader->GetProgram());
43    glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
44                                     "diffuseTex"), 0);
45
46    glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
47                   "cameraPos"),1,(float*)&camera->GetPosition());
48
49    UpdateShaderMatrices();
50    SetShaderLight(*light);
51
52    heightMap->Draw();
53
54    glUseProgram(0);
55
56    SwapBuffers();
57 }
```

renderer.cpp

Vertex Shader

The vertex shader this time around has to calculate the direction in world space the normal is pointing, using the *inverse transpose* model matrix - note how we also **cast** it to a **mat3**, removing the translation information as we do so. We also need the world space position of the processed vertex, easily worked out using the model matrix. Both the normal and world space position will be interpolated across the vertices of each triangle, giving us the correct values per-fragment.

```
1  #version 150 core
2  uniform mat4 modelMatrix;
3  uniform mat4 viewMatrix;
4  uniform mat4 projMatrix;
5  uniform mat4 textureMatrix;
6
7  in  vec3 position;
8  in  vec4 colour;
9  in  vec3 normal;     //New Attribute!
10 in  vec2 texCoord;
11
12 out Vertex {
13    vec4 colour;
14    vec2 texCoord;
15    vec3 normal;
16    vec3 worldPos;
17 } OUT;
18
19 void main(void)      {
20    OUT.colour        = colour;
21    OUT.texCoord      = (textureMatrix * vec4(texCoord, 0.0, 1.0)).xy;
22
23    mat3 normalMatrix = transpose(inverse(mat3(modelMatrix)));
24
25    OUT.normal        = normalize(normalMatrix * normalize(normal));
26
```

```
27    OUT.worldPos       = (modelMatrix * vec4(position,1)).xyz;
28    gl_Position        = (projMatrix * viewMatrix * modelMatrix) *
29                          vec4(position, 1.0);
30 }
```

PerPixelVertex.glsl

Fragment Shader

The fragment shader for this tutorial is a bit of a long one! We start off with a uniform texture sampler so we can sample the incoming mesh's texture, and a further 4 uniform values - 3 to represent the point light's attributes, and one to keep the camera's world space position. After that, we have our vertex input block, complete with our new *normal* and *worldPos* variables.

```
1  #version 150 core
2
3  uniform sampler2D diffuseTex;
4
5  uniform vec3   cameraPos;
6  uniform vec4   lightColour;
7  uniform vec3   lightPos;
8  uniform float  lightRadius;
9
10 in Vertex {
11    vec3 colour;
12    vec2 texCoord;
13    vec3 normal;
14    vec3 worldPos;
15 } IN;
16
17 out vec4 gl_FragColor;
```

PerPixelFragment.glsl

In our **main** function, we sample the texture to obtain the base *diffuse* colour of the fragment. Then, we work out the incident vector between the current interpolated fragment world position and the current light's world position, and from that work out the lambertian reflectance for the fragment, by performing a dot product operation on the incident and world space normal. Note that the incident vector is normalised, and dot product values below 0.0 get clamped to 0.0.

```
18 void main(void)   {
19    vec4 diffuse       = texture(diffuseTex, IN.texCoord);
20
21    vec3  incident     = normalize(lightPos - IN.worldPos);
22    float lambert      = max(0.0, dot(incident, IN.normal));
```

PerPixelFragment.glsl

With the Lambertian reflectance calculated, we can move on to calculating the *attenuation* of the light source at the current fragment. The distance between the current fragment and the light position is easily worked out on line 23 - remember, *worldPos* will be an *interpolated* value calculated on a per-fragment basis from the vertices that make up the geometry, and so should accurately match the actual world space position of the fragment. We can then calculate the light attenuation on line 24, using the linear attenuation method discussed earlier.

```
23    float dist         = length(lightPos - IN.worldPos);
24    float atten        = 1.0 - clamp(dist / lightRadius, 0.0, 1.0);
```

PerPixelFragment.glsl

Next up is calculating the *specularity* lighting component of the fragment. The *Blinn-Phong* lighting model we are using for specularity requires the 'half angle' vector, which we can approximate by normalising the sum of the incident and the view direction vector. We calculate the view direction on line 25, using the interpolated world space position of the fragment, and the uniform world space position of the camera. Then, on line 26, we calculate the approximated half angle vector. Using this half angle vector, we can work out how much specular reflection there is by performing a dot product operation, which we do on line 28 - including clamping the lower bound to 0.0 using the **max** function. We can then work out the final specularity value by raising the specular reflection value to a power representing how 'shiny' the surface is - the higher this power, the more perfectly shiny the surface is, and the tighter our specualrity will be. In the shader we're using a value of 50.0, but this could easily be converted to be a **uniform** variable, or even sampled from a texture - a process known as *gloss mapping*.

```
25    vec3 viewDir      = normalize(cameraPos - IN.worldPos);
26    vec3 halfDir      = normalize(incident + viewDir);
27
28    float rFactor     = max(0.0, dot(halfDir, normal));
29    float sFactor     = pow(rFactor, 50.0 );
```

PerPixelFragment.glsl

We now have enough information to calculate the final colour of our fragment. On line 30, we work out the diffuse component, by multiplying the sampled texture diffuse by the light colour. Then, on line 31, we add specularity by multiplying the light colour by the specular factor. We also multiply the specularity by 0.33 - this is optional, and slightly lowers the brightness of the specular component. Finally, we set our fragment colour to have an rgb value to the attenuated, Lambertian reflected colour, and copy the alpha value straight from the sampled texture - we save a few operations by only doing the attenuation and lambertian multiplies only once. Finally, on line 33 we add a small amount of ambient lighting.

You may have noticed by now, that both the attenuation and the lambertian reflectance stages could result in a value of 0.0, so why bother multiplying the values? The simple answer is, it's probably just as quick to just multiply the colour by 0.0 than it is to put a branching statement in our fragment shader to discard fragments with no lighting - remember, we want to avoid branches in our fragment code!

```
30    vec3 colour       =  (diffuse.rgb * lightColour.rgb);
31    colour            += (lightColour.rgb * sFactor) * 0.33;
32    gl_FragColor      =  vec4(colour * atten * lambert, diffuse.a);
33    gl_FragColor.rgb  += (diffuse.rgb * lightColour.rgb) * 0.1;
34 }
```

PerPixelFragment.glsl

Tutorial Summary

If everything goes to plan, you should see a fully per-pixel light heightmap when running this program. All you've done is add a single point light to the scene, but in the process, you've learned pretty much everything you need to know about lighting. You've learnt about normals, ambient, diffuse, and specular lighting, and how to correctly apply them to the surfaces in your scene. If you fly around the scene a bit, you whould see the specular highlights change as the camera moves. You'll probably notice that this specularity makes the rocky surface of the heightmap look a bit 'plasticy'. Next tutorial you'll see how to change that, and make your lighting look even better, using a process called *bump mapping*.

Further Work

1) The *Light* class in this tutorial has only one colour variable, used for both emissive and specular calculations by the fragment shader. Try adding a separate colour variable for specularity.

2) This tutorial creates only a single light. Try adding an array of 4 *Lights* to the *Renderer* class. What functions need to be changed in the *Renderer* class? What needs to change in the shader? Remember, you can have loops in fragment shaders, and **uniform** shader variables can be arrays...

3) You now know how to implement *point* lights; try modifying it to perform a *direction* light calculation, instead. Direction lights have no radius or position, but do have a normalised direction vector.

4) If you're feeling particularly brave, you might want to try having a spotlight that casts light in a cone, rather than a sphere. You'll still need a radius value to calculate distance, but you'll also need a function to set the angle of the cone, and a direction vector, too. How would you work out whether the fragment is inside the spotlight cone? Dot products may come in useful, here...