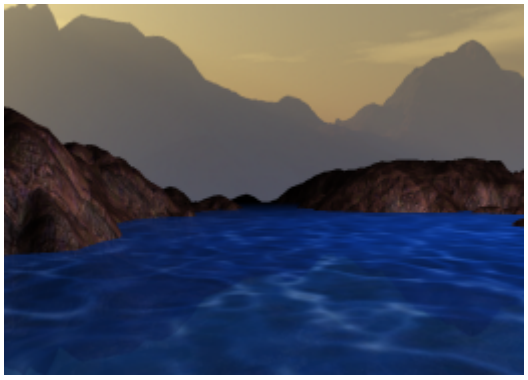
# Tutorial 13: Cube Mapping



## Summary

In real life, shiny objects reflect incoming light - think of a mirror, or the reflection of the setting sun on a pool of water. The lighting model introduced last tutorial can't simulate such reflections, and to do so would require expensive ray racing operations. Instead, games use an *environment map* - a set of static images that represent the surrounding scenery that can be sampled from to emulate 'real' reflections. This tutorial will demonstrate how to use these environment maps to achieve two things - reflections, and sky boxes.

### New Concepts

Cube maps, Cube samplers, environment mapping, reflecting a direction vector, sky boxes

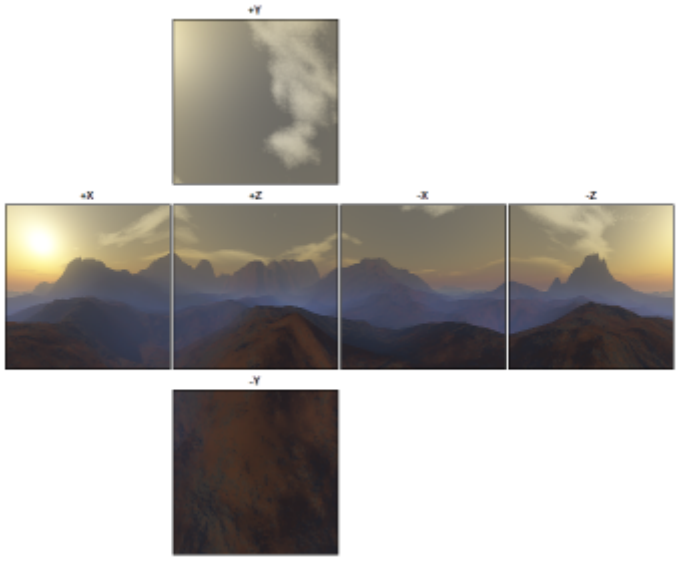
## Environment Maps

Environment maps have long been used to add computationally inexpensive reflections to graphical scenes. It is now common in games to see water puddles and other wet surfaces reflect the sky, and for shiny materials like metal and glass to reflect light from the environment around them.

*An example of environment map reflections on a car windscreen in Deus Ex: Human Revolution*

## Cube Maps

Modern games generally use *cube maps*, which contain enough visual data to create reflections for any given surface angle. As the name suggests, cube maps are made up of 6 separate images, which when formed into a cube, form a fully enclosed scene. Imagine taking a photograph of your surroundings at 90 degree intervals, including the ceiling and floor - if you stuck those photographs together, you'd create a cube containing a seamless image of everything you could see. It might look something like this:

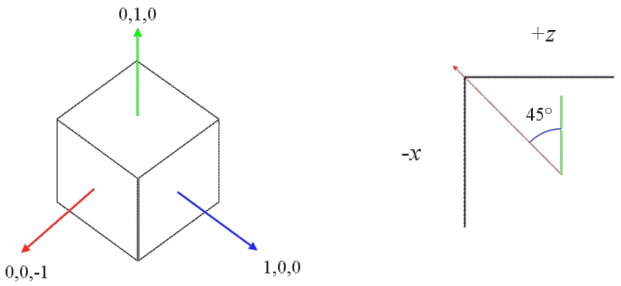


*An example cube map*

Generally these cube map images are an artist created *static* scene of mountains or other such far away features; although it's possible to render a *dynamic* cube map, this is prohibitively computationally expensive, as the scene would have to be processed once for each cube side.

## Sampling Cube Maps

Just like any other texture, in order to use a cube map, it must be *sampled*. However, instead of sampling a cube map with a 2D texture coordinate, as with usual 2D textures, cube maps are sampled with a specialised texture sampler using a normalised direction vector - just like the vertex normals you've been using. Imagine the direction vector as being a ray that begins in the middle of the cube formed by the cube mapped textures, with the point where the ray intersects the cube being the texel sampled. So for example, a direction vector of ( 0, 1, 0 ) will sample from the middle of the positive y axis cube map texture, which in the example cube map above would be the sky. The cube map sampler will automatically handle which texture will be sampled from a given direction vector, and in modern rendering APIs sampling can also occur *between* the cube map textures. Imagine a 45° reflection vector - the 'ray' of this vector would intersect exactly at the corner between two cubemap textures, so to sample a correct texel, bilinear interpolation between the two textures takes place.



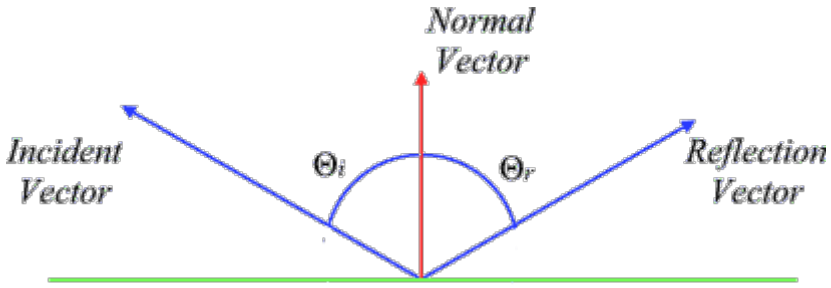

*Left: Sampling direction vectors Right: An example of a vector resulting in cross-texture interpolation*

### Reflections

So, we now know that cube maps are sampled using direction vectors - but which direction vectors? We can't use vertex normals, as they would sample the same point on the cube map regardless of where the viewer was - Imagine a mirror that reflected only from its normal, it'd be a static scene, no matter where you viewed it from. So, to sample a cube map correctly we need a *reflection* vector. Using a reflection vector takes into account how light bounces off a surface from the perspective of the viewer. To calculate the reflection vector for a surface, we need its *normal*, and the *incident* vector that runs from the *view point* to the *surface* - just like the calculations for specularity. Only instead of using them to calculate the Blinn-Phong half-angle, we're going to use them to calculate the angle of *reflection* from the angle of *incidence* like so:

$$R = I - 2N(N \cdot I)$$

Where $R$ is the reflected vector, $I$ is the incident, $N$ is the normal, and $N{\cdot}I$ is their dot product. This will calculate an angle identical to the angle of incidence, but pointing away from the surface being reflected.




*An example reflection vector calculated from a normal and an incident, showing that* $\theta_I = \theta_R$

As the angle of reflection is view-dependent, using it to sample the cube map will result in accurate reflections that can be used in dynamic scenes.

## Skyboxes

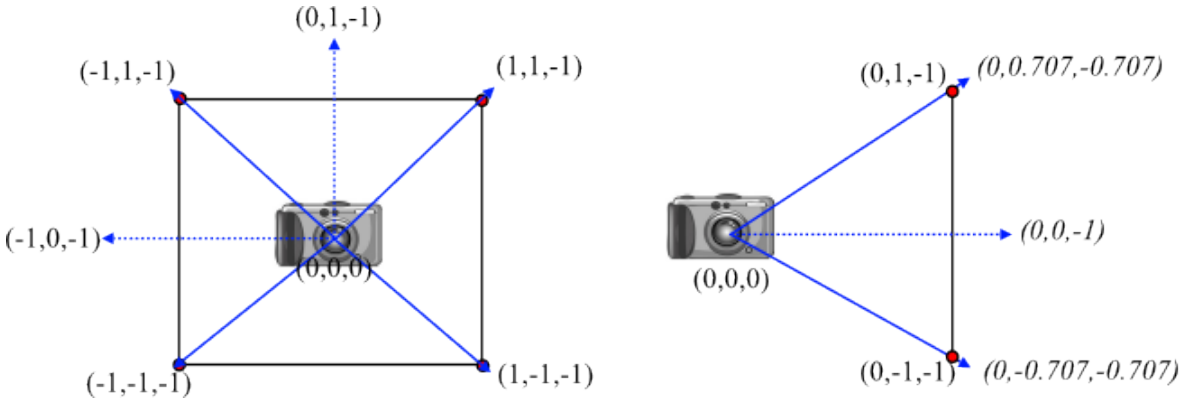
Another popular use for cube maps is that of *sky boxes*. Instead of having a solid background colour, it has long been popular to have an image in the background that moves with the viewpoint, providing a more immersive experience. Even early 3D shooters like id Software's *Doom* had a simple form of viewer-oriented background sky:


*Simple parallax sky texture in Doom II: Hell On Earth*

As cube map textures contain the visual data for an all-encompassing view, they can be used to create a realistic background to a game world. Sometimes, this is created by literally enclosing a game level inside a giant cube, with the separate cube map textures applied as normal 2D textures on the

cubes faces, but it's possible to use the specialised cube map sampler to project the skybox onto the scene, gaining the advantages cube mapping brings, such as the seamless cross-texture interpolation mentioned earlier.

Skyboxing with cube maps is usually performed, oddly enough, using a single quad. A quad is drawn to fill the screen, similar to a post-processing stage, but often as the first thing renderer into a scene. Then, instead of applying the cube map textures as standard 2D textures to the quad, sampled via texture coordinates, the direction vectors formed by normalising the vertex positions are used, instead. These vertex direction vectors are then *interpolated* to provide the correct direction vector for each fragment.

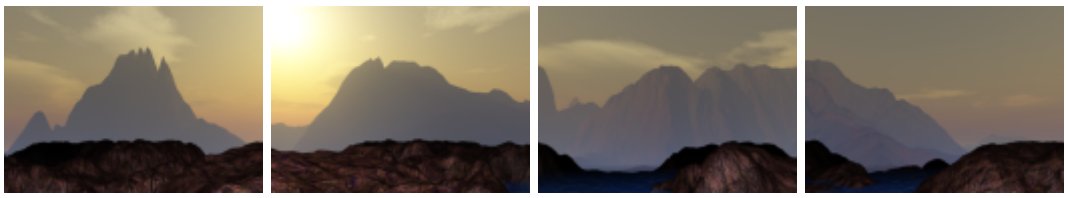



*Left: Head on view of the vertex positions between the camera and the full screen quad. Right: Side view showing interpolated direction vectors formed from normalised vertex positions (shown in italics)*

For the skybox to rotate with the viewpoint moves, the direction vectors obtained should be rotated by the *camera* matrix - the inverse of the view matrix used in vertex shaders. For example, consider a normalised direction vector pointing at (0,0,-1). This would sample directly from the centre of the negative $z$ cube map texture - if our camera has an identity matrix for it's view, this'll be correct. Now, consider if our camera looks straight up. As we're drawing a full screen quad, the direction vector would still be (0,0,-1). If we rotate it by the camera's inverted transform, the direction vector will end up at (0,1,0), sampling the positive $y$ cubemap.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

By rendering the cube using a skybox cube map shader at the start of a frame, the virtual worlds you create for your games will have the appearance of having mountainous regions or huge skyscrapers, all *without* having to render massive amounts of polygons.



*Panoramic view of a skybox rendered using cube maps*

## Example Program

To demonstrate how to perform cube mapping, we're going to create an example program based off last tutorial's lit heightmap scene - only this time around, we'll have a skybox instead of a dull grey background, and the heightmap will have some simple water, that correctly reflects the skybox. As we've not used the texture matrix in a while, to finish the scene we'll make the water move using texture matrix manipulation. To do all this, we'll need three shaders in total - the per fragment lighting shader we created last tutorial, and two new ones; one each to handle the skybox and water reflections. So, in your *Shaders* solution folder, you should create 3 new text files - *skyboxVertex.glsl*, *skyboxFragment.glsl*, and *reflectFragment.glsl*, as well as a new *Renderer* class that inherits from *OGLRenderer*, and a main file, *tutorial12.cpp*. Our main file is just the same as the previous few tutorials, so we won't go over it again.

## Renderer header file

Our example program **overrides** the *RenderScene* and *UpdateScene* **public virtual** functions, as with the other tutorials in this series. It also has 3 new **protected** functions, pointers to 3 *Shader* instances, as well as a *HeightMap*, a *Mesh*, a *Light*, and a *Camera*. Finally, we have an OpenGL texture name for our cube map, and a **float** member variable to control our water movement.

```
1  #pragma once
2
3  #include "./nclgl/OGLRenderer.h"
4  #include "./nclgl/Camera.h"
5  #include "./nclgl/HeightMap.h"
6
7  class Renderer : public OGLRenderer    {
8  public:
9     Renderer(Window &parent);
10    virtual ~Renderer(void);
11
12    virtual void RenderScene();
13    virtual void UpdateScene(float msec);
14
15 protected:
16    void        DrawHeightmap();
17    void        DrawWater();
18    void        DrawSkybox();
19
20    Shader*     lightShader;
21    Shader*     reflectShader;
22    Shader*     skyboxShader;
23
24    HeightMap*  heightMap;
25    Mesh*       quad;
26
27    Light*      light;
28    Camera*     camera;
29
30    GLuint      cubeMap;
31
32    float       waterRotate;
33 };
```

renderer.h

## Renderer Class file

We start our *Renderer* class constructor by initialising the *Camera*, *HeightMap*, *Light*, and *Mesh* class instances. We then initialise our three *Shaders*, and link them, returning if the linking process fails. Nothing new so far - the camera position and light **constructor** values are just as they were in the last tutorial.

```
1  #include "Renderer.h"
2
3  Renderer::Renderer(Window &parent) : OGLRenderer(parent) {
4     camera          = new Camera();
5     heightMap       = new HeightMap(TEXTUREDIR"/terrain.raw");
6     quad            = Mesh::GenerateQuad();
7
8     camera->SetPosition(Vector3(RAW_WIDTH*HEIGHTMAP_X / 2.0f,
9                                 500.0f,RAW_WIDTH*HEIGHTMAP_X));
10
11    light = new Light(Vector3((RAW_HEIGHT*HEIGHTMAP_X / 2.0f),500.0f,
12                      (RAW_HEIGHT*HEIGHTMAP_Z / 2.0f)),
13                      Vector4(0.9f,0.9f,1.0f,1),
14                      (RAW_WIDTH*HEIGHTMAP_X) / 2.0f);
15
16    reflectShader   = new Shader(SHADERDIR"PerPixelVertex.glsl",
17                                 SHADERDIR "reflectFragment.glsl");
18    skyboxShader    = new Shader(SHADERDIR"skyboxVertex.glsl",
19                                 SHADERDIR "skyboxFragment.glsl");
20    lightShader     = new Shader(SHADERDIR"PerPixelVertex.glsl",
21                                 SHADERDIR"PerPixelFragment.glsl");
22
23    if(!reflectShader->LinkProgram() || !lightShader->LinkProgram() ||
24       !skyboxShader->LinkProgram()) {
25       return;
26    }
```

renderer.cpp

Next up, we apply the water.tga texture to the quad, and the rock texture and bump maps to the heightMap. On line 37 is the first real new piece of code - the initialisation of the cube map. We're going to use a handy function of the SOIL library to generate our cubemap, which takes 6 filenames, corresponding to the positive and negative $x$, $y$, and $z$ axis textures required to form the full cube - the order is important! See how even though we have 6 cube map textures, only one OpenGL texture name is required. We then check that all of the textures have been correctly generated, and if so, enable repeating on the water and rock textures.

```
27    quad->SetTexture(SOIL_load_OGL_texture(TEXTUREDIR"water.TGA",
28       SOIL_LOAD_AUTO, SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
29
30    heightMap->SetTexture(SOIL_load_OGL_texture(
31       TEXTUREDIR"Barren Reds.JPG", SOIL_LOAD_AUTO,
32       SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
33
34    heightMap->SetBumpMap(SOIL_load_OGL_texture(
35       TEXTUREDIR"Barren RedsDOT3.JPG", SOIL_LOAD_AUTO,
36       SOIL_CREATE_NEW_ID, SOIL_FLAG_MIPMAPS));
```

renderer.cpp

```
37    cubeMap = SOIL_load_OGL_cubemap(
38       TEXTUREDIR"rusted_west.jpg",  TEXTUREDIR"rusted_east.jpg",
39       TEXTUREDIR"rusted_up.jpg",    TEXTUREDIR"rusted_down.jpg",
40       TEXTUREDIR"rusted_south.jpg", TEXTUREDIR"rusted_north.jpg",
41       SOIL_LOAD_RGB,
42       SOIL_CREATE_NEW_ID, 0
43    );
44
45    if(!cubeMap || !quad->GetTexture() || !heightMap->GetTexture() ||
46       !heightMap->GetBumpMap()) {
47       return;
48    }
49
50    SetTextureRepeating(quad->GetTexture(),true);
51    SetTextureRepeating(heightMap->GetTexture(),true);
52    SetTextureRepeating(heightMap->GetBumpMap(),true);
```

renderer.cpp

Finally in this constructor, we initialise the *waterRotate* **float**, enable depth testing, and set up a perspective projection matrix. We also enable alpha blending, as we want the water to be slightly transparent. We also use **glEnable** with a new OpenGL symbolic constant - **GL_TEXTURE_CUBE_MAP_SEAMLESS**. This enables bilinear filtering to be applied across the edges of the 6 cube map textures, so we get a seamless skybox. If you tried to do this tutorial without bilinear filtering, you'd get pixellated skies, and if you used bilinear without the seamless extension enabled, you'd probably see lines at the corners of the individual cube map textures, breaking the illusion.

```
53    init        = true;
54    waterRotate = 0.0f;
55
56    projMatrix  = Matrix4::Perspective(1.0f,15000.0f,
57                  (float)width / (float)height, 45.0f);
58
59    glEnable(GL_DEPTH_TEST);
60    glEnable(GL_BLEND);
61    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
62    glEnable(GL_TEXTURE_CUBE_MAP_SEAMLESS);
```

renderer.cpp

As ever, in the **destructor** we **delete** everything we created in the **constructor** - and make *currentShader* equal 0 to prevent double deletion problems with the destructor of *OGLRenderer*.

```
63 Renderer::~Renderer(void)  {
64    delete camera;
65    delete heightMap;
66    delete quad;
67    delete reflectShader;
68    delete skyboxShader;
69    delete lightShader;
70    delete light;
71    currentShader = 0;
72 }
```

renderer.cpp

Our *UpdateScene* function has something new in it this time - we want our water texture to slowly rotate and give the impression of flowing water, so on line 80 we increase the *waterRotate* member variable that we're going to use to control this by a small amount, governed by the frame time. Other than that, it's just the usual - update our camera, and generate a new view matrix from it.

```
73  void Renderer::UpdateScene(float msec)   {
74     camera->UpdateCamera(msec);
75     viewMatrix   = camera->BuildViewMatrix();
76     waterRotate += msec / 1000.0f;
77  }
```

renderer.cpp

Our scene rendering is split up again, this time into *DrawSkybox*, *DrawHeightmap*, and *DrawWater*, so all *RenderScene* does is clear the screen, call these functions, and swap the buffers.The order is important - the water transparency effect won't work if *DrawWater* is called first, as the heightmap colour data won't be there to blend with, and *DrawSkybox* must be before the others, as otherwise our skybox would be drawn over the top of the heightmap and water!

```
78  void Renderer::RenderScene()  {
79     glClear(GL_DEPTH_BUFFER_BIT | GL_COLOR_BUFFER_BIT);
80
81     DrawSkybox();
82     DrawHeightmap();
83     DrawWater();
84
85     SwapBuffers();
86  }
```

renderer.cpp

Our first rendering sub-function is *DrawSkybox*. This will draw a full-screen quad similar to the one used in the post-processing tutorial, which we will use to project our cube map as a background skybox. First off, the function is bookended by calls to disable and enable writes to the depth buffer - we don't want the quad to fill the depth buffer, and cause our heightmap and water to be discarded. Next, we enable our skybox shader, and update the shader matrices. we don't actually use the model matrix in the vertex shader, so there's no need to change it. After that, we simply draw our quad, and unbind the skybox shader.

```
87  void  Renderer::DrawSkybox()  {
88     glDepthMask(GL_FALSE);
89     SetCurrentShader(skyboxShader);
90
91     UpdateShaderMatrices();
92     quad->Draw();
93
94     glUseProgram(0);
95     glDepthMask(GL_TRUE);
96  }
```

renderer.cpp

Next up is *DrawHeightmap*. Everything in this function should all be familiar to you from last tutorial. We enable the per fragment lighting shader, set the shader light and shader uniform variables, before finally drawing the height map. Easy!

```
97  void  Renderer::DrawHeightmap()  {
98     SetCurrentShader(lightShader);
99     SetShaderLight(*light);
```

```
100     glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
101                  "cameraPos") ,1,(float*)&camera->GetPosition());
102
103     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
104                 "diffuseTex"), 0);
105     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
106                 "bumpTex"), 1);
107
108     modelMatrix.ToIdentity();
109     textureMatrix.ToIdentity();
110
111     UpdateShaderMatrices();
112
113     heightMap->Draw();
114
115     glUseProgram(0);
116 }
```

renderer.cpp

The last rendering sub-function is *DrawWater*. It's pretty similar to *DrawHeightmap*, but note that we use the shader *reflectShader*, and instead of binding texture unit 1 to a uniform called *bumpTex*, we are binding texture unit 2 to a uniform called *cubeTex*. The water in our scene is simply going to be a single quad, which needs to be translated to the middle of the heightmap, scaled out across the heightmap, and rotated so that it intersects the heightmap horizontally instead of vertically. The *heightX* and *heightZ* local variables hold $x$ and $y$ axis positions of the centre of the heightmap, while *heightY* is the water level - you can modify this to your liking. We also set the texture matrix in this function, to make the texture coordinates repeat, and be rotated about the $z$-axis by the value *waterRotate*.

```
117 void  Renderer::DrawWater()    {
118     SetCurrentShader(reflectShader);
119     SetShaderLight(*light);
120     glUniform3fv(glGetUniformLocation(currentShader->GetProgram(),
121                  "cameraPos") ,1,(float*)&camera->GetPosition());
122
123     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
124                                      "diffuseTex"), 0);
125
126     glUniform1i(glGetUniformLocation(currentShader->GetProgram(),
127                                      "cubeTex"), 2);
128
129     glActiveTexture(GL_TEXTURE2);
130     glBindTexture(GL_TEXTURE_CUBE_MAP, cubeMap);
131
132     float heightX = (RAW_WIDTH*HEIGHTMAP_X / 2.0f);
133
134     float heightY = 256 * HEIGHTMAP_Y / 3.0f;
135
136     float heightZ = (RAW_HEIGHT*HEIGHTMAP_Z / 2.0f);
137
138     modelMatrix =
139        Matrix4::Translation(Vector3(heightX, heightY, heightZ)) *
140        Matrix4::Scale(Vector3(heightX,1, heightZ)) *
141        Matrix4::Rotation(90,Vector3(1.0f,0.0f,0.0f));
142
143     textureMatrix = Matrix4::Scale(Vector3(10.0f,10.0f,10.0f)) *
144              Matrix4::Rotation(waterRotate,Vector3(0.0f,0.0f,1.0f));
145
```

```
146     UpdateShaderMatrices();
147
148     quad->Draw();
149
150     glUseProgram(0);
151 }
```

renderer.cpp

## Mesh Class

In this tutorial, and the next tutorial, we're going to be using the *GenerateQuad* function, to create geometry to perform lighting calculations on. That means our quad must have normals and tangents, just like the *HeightMap*, and anything else we draw. Fortunately, we can do this easily, and we don't even have to use the *GenerateNormals* and *GenerateTangents* functions! The quad's normal simply points down negative $z$, and its tangent points along positive $x$, so we can explicitly set these values, in the same way we set the colour per vertex. So, our *GenerateQuad* function now looks like this:

```
1  Mesh* Mesh::GenerateQuad() {
2     Mesh* m = new Mesh();
3
4     m->numVertices = 4;
5     m->type = GL_TRIANGLE_STRIP;
6
7     m->vertices       = new Vector3[m->numVertices];
8     m->textureCoords  = new Vector2[m->numVertices];
9     m->colours        = new Vector4[m->numVertices];
10    m->normals        = new Vector3[m->numVertices];
11    m->tangents       = new Vector3[m->numVertices];
12
13    m->vertices[0] =  Vector3(-1.0f, -1.0f, 0.0f);
14    m->vertices[1] =  Vector3(-1.0f, 1.0f, 0.0f);
15    m->vertices[2] =  Vector3(1.0f, -1.0f, 0.0f);
16    m->vertices[3] =  Vector3(1.0f,  1.0f, 0.0f);
17
18    m->textureCoords[0] = Vector2(0.0f, 1.0f);
19    m->textureCoords[1] = Vector2(0.0f, 0.0f);
20    m->textureCoords[2] = Vector2(1.0f, 1.0f);
21    m->textureCoords[3] = Vector2(1.0f, 0.0f);
22
23    for(int i = 0; i < 4; ++i) {
24       m->colours[i]  = Vector4(1.0f, 1.0f,1.0f,1.0f);
25       m->normals[i]  = Vector3(0.0f, 0.0f,-1.0f);
26       m->tangents[i] = Vector3(1.0f, 0.0f,0.0f);
27    }
28
29    m->BufferData();
30
31    return m;
32 }
```

Mesh.cpp

## Cubemap Reflection Shader

The first new shader we are going to write in this tutorial will use our new cube map to create reflections on the water in the scene. We can reuse the per-fragment lighting vertex shader, but we'll need a new fragment shader, so the following program will go in the *reflectFragment.glsl* file.

### Fragment Shader

We start off in familiar fashion to the per-fragment lighting fragment shader, with two texture samplers, **uniform** variables for the lighting and camera, and the Vertex interface block. Note, however, that the *cubeTex* sampler is of a new type - *samplerCube*. This new sampler type allows us to sample our cubemap using a direction vector, and will automatically handle which of the 6 separate textures the resulting sample will be taken from.

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

reflectFragment.glsl

We then calculate the diffuse colour, incident vector, light distance, and light attenuation, just like we do in the per-fragment lighting fragment shader. On line 25, though, we have some new code. We sample the *cubeTex* sampler with a direction vector, which we calculate using the glsl **reflect** function, which as its name implies, calculates the angle of reflection that was introduced earlier. On line 28 we combine the attenuated light colour and the sampled cubemap reflection colour. That's everything! The *samplerCube* variable and **reflect** function do most of the hard work - all we have to do is decide how to blend in the result.

```
19 void main(void)   {
20    vec4 diffuse    = texture(diffuseTex, IN.texCoord) * IN.colour;
21    vec3 incident   = normalize(IN.worldPos - cameraPos);
22    float dist      = length(lightPos - IN.worldPos);
23    float atten     = 1.0 - clamp(dist / lightRadius, 0.2, 1.0);
24    vec4 reflection = texture(cubeTex,
25                      reflect(incident,normalize(IN.normal)));
26
27    fragColour   = (lightColour*diffuse*atten)*(diffuse+reflection);
28 }
```

reflectFragment.glsl

## Cubemap Skybox Shader

To create our skybox on screen, we're going to sample from a *samplerCube* again. This time however, we'll be generating the direction vector ourselves. Earlier, in the *DrawSkybox* function, we rendered a full-screen quad. To draw the quad we've cheated a bit - in this shader we're actually not going to use the model and view matrices to translate the quad anywhere or draw it in relation to the camera. This means it'll be drawn with its origin at the clip space origin, so we move it back 1 unit, so the quad is drawn right up against our near plane, covering the screen entirely! We will use the projection matrix though - so changes in the field of vision will be reflected in the shape of the skybox cube, by stretching the sides of the quad out beyond the sides of the screen.

We *are* going to use the viewMatrix elsewhere, to rotate a direction vector, which will then be interpolated like any other vertex data passed to the fragment shader, so there will be a unique direction vector for each fragment. As we don't rotate the quad's vertices, the normalised direction vector generated from the vertex positions for the 'middle' of the screen will always be (0,0,-1), so we rotate it by the view matrix to point in the direction our camera is looking.

### Vertex Shader

We want the direction vector used for cubemap sampling to rotate with the camera view so that the correct cubemap texel is sampled, so on line 17, we rotate the direction vector by multiplying it by the view matrix...sort of. We don't want to apply the translation component to the rotation, as this would warp our direction vector as the camera moved away from the origin, so we *downcast* the view matrix to a **mat3** - remember, the translation component is in the fourth column of the view matrix. We also actually use the *transpose* of the view matrix. Why is this? As we're rotating an object rather than the camera, we need the *inverse* of the view matrix to rotate the world objects correctly - remember how we 'inverted' the camera matrix by negating all of the camera class variables in the *BuildViewMatrix* function? We're going to use another trick to avoid the *inverse* operation here, too. If the view matrix has no scaling or shearing, then the inverse of a square matrix is equal to its transpose - a much quicker operation to perform! Try it - both *inverse* and *transpose* will return the same results.

Finally, we must also take into consideration the aspect ratio of the screen, and the desired field of vision - the sky should distort and zoom in/out consistently with the rest of the world. To do so, we must therefore use the *projection* matrix. We can scale the $x$ and $y$ axes of the normal vector by getting the *reciprocal* of the associated diagonal values of the projection matrix - for these values, the reciprocal is the same as what the *inverse* of the projection matrix would provide.

```
1  #version 150 core
2  uniform mat4 viewMatrix;
3  uniform mat4 projMatrix;
4
5  in  vec3 position;
6
7  out Vertex {
8     vec3 normal;
9  } OUT;
10
11 void main(void)      {
12    vec3 multiply = vec3(0,0,0);
13    multiply.x    = 1.0f / projMatrix[0][0];
14    multiply.y    = 1.0f / projMatrix[1][1];
15
16    vec3 tempPos   = (position * multiply) - vec3(0,0,1);
17    OUT.normal     = transpose(mat3(viewMatrix)) * normalize(tempPos);
18    gl_Position    = vec4(position, 1.0);
19 }
```

skyboxVertex.glsl

### Fragment Shader

The fragment shader is really easy - the output colour is simply the result of sampling the *cubeTex* sampler using the interpolated direction vector we kept in the *normal* variable. We normalise it to unit length to account for minor errors that can occur with interpolation.

```
1  #version 150 core
2
3  uniform samplerCube  cubeTex;
4  uniform vec3         cameraPos;
5
6  in Vertex {
7     vec3 normal;
8  } IN;
9
10 out vec4 fragColour;
11
12 void main(void)   {
13    fragColour = texture(cubeTex,normalize(IN.normal));
14 }
```

skyboxFragment.glsl

## Tutorial Summary

If everything works properly, you should see the now-familiar heightmap , with per-fragment lighting. However, you should now have rocky mountains in the distance, courtesy of the cube map and the skybox shader. Try looking around with the mouse - you should be able to see mountains all around, and be able to look straight up and see clouds. What's more, you should see water has filled your terrain - a closer inspection should show that the cubemap is also reflected correctly in the water. If you look straight down into the slowly moving water, you should see the clouds reflected back at you. Not bad for being a cube and a quad with a shader applied to it! Another pretty simple effect, but you should now know a little bit about environment mapping using cube maps, and how to sample cube maps using direction vectors. In the next tutorial we're going to add something else to our graphical scenes - shadows!

## Further Work

1) Last tutorial you were introduced to *bump maps*. The water texture used in this tutorial has a bump map in the *Textures* folder, called *waterDOT3.tga*. How would you modify the cubemap reflection shader to make use of this bump map?

2) By using the quad mesh and modifying the skybox shader, you should be able to add a perfectly reflecting mirror in your scene. Or, perhaps it's a portal to somewhere else? The quad doesn't necessarily have to use the same cubemap as the skybox...

3) In the example program, the heightmap is drawn on top of the skybox. That can result in quite a lot of *overdraw*, where pixels are drawn multiple times in a frame. Can you think of any way of reducing this overdraw? Investigate how the stencil buffer introduced in tutorial 5 could be used to do this.

4) Take a look at http://brainwagon.org/2002/12/05/fun-with-environment-maps/

## Appendix A: Generating Cube Maps the *hard way*

In tutorial 3, you saw how to create OpenGL textures from texture data, using the OpenGL function **glTexImage2D**. Turns out, generating cube maps uses the same function - remember, each of the individual textures is still 2D, even if we can sample the end result using a 3D direction vector. The main change over 2D textures is the first parameter of the **glTexImage2D** function. Instead of using **GL_TEXTURE_2D**, one of 6 symbolic constants is used - one for each axis direction. In the semi-pseudocode below, the array *axis* on line 7 contains the required symbolic constants. The only other change is to **glBindTexture** and **glTexParameteri** - like with glTexImage2D, instead of using the symbolic constant **GL_TEXTURE_2D**, **GL_TEXTURE_CUBE_MAP** is used instead. Not much to it, really!

```
1   string names[6] = {
2      "west.tga"  ,  "east.TGA"  ,
3      "up.TGA"    ,  "down.TGA"  ,
4      "south.TGA" ,  "north.TGA"
5   };
6
7   GLuint axis[6] = {
8      GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X,
9      GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y,
10     GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
11  };
12
13  GLuint skyBoxCubeMap;
14  glGenTextures(1, &skyBoxCubeMap);
15  glBindTexture(GL_TEXTURE_CUBE_MAP, skyBoxCubeMap);
16
17  for(int i = 0; i < 6; ++i) {
18     Texture*t = <Texture data from filename names[i]>
19
20     glTexImage2D(axis[i], t->mipmaplevel, t->internalFormat,
21                  t->width,  t->height, t->border,
22                  t->format, t->datatype,  t->data);
23
24     glTexParameteri(GL_TEXTURE_CUBE_MAP,
25                     GL_TEXTURE_MIN_FILTER,GL_LINEAR);
26     glTexParameteri(GL_TEXTURE_CUBE_MAP,
27                     GL_TEXTURE_MAG_FILTER,GL_NEAREST);
28     glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_S,GL_CLAMP);
29     glTexParameteri(GL_TEXTURE_CUBE_MAP,GL_TEXTURE_WRAP_T,GL_CLAMP);
30  }
31
32  glBindTexture(GL_TEXTURE_CUBE_MAP, 0);
```

OpenGL Cube Map Loading