Lab 4 – Shading and Environment Mapping

In this lab, we will start with some basic shading of our scene. That is, we will calculate the color of each fragment based on its material and on where it is in relation to a *light source*. Then we will add *environment-mapping* which is a commonly used technique to simulate specular reflection in realtime graphics applications. The idea is that the environment around some object is rendered to some texture, either as a preprocessing step or each frame, and then, when shading the object, this texture is used to look up what will be reflected for each fragment.

Run the program and look over the code as usual. The only things you have not seen in previous labs in the code so far are:

- **Assignment:** The Quad is defined and drawn as a *Triangle Strip* instead of as separate triangles. What does this mean? Why might this be more efficient?

  _____

- **Assignment:** There is now a light position defined right after the camera position (also in spherical coordinates). Make the light rotate by making *light_theta* depend on *currentTime* (you can update it every frame in the display() function). The light is drawn as a yellow sphere.

Normals

To do any sort of interesting shading (and environment mapping is no exception), each vertex of the mesh will need to have a normal associated with it. Normals are sent along with vertexes just like vertex colors or texture coordinates. So, after the vertex position definitions you just changed, add:

```
        // Define the normals for each of the four points of the quad
    float normals[] = {
            0.0f,   0.0f, 1.0f,         // v0 -          v0    v2
            0.0f,   0.0f, 1.0f,         // v1 -          |  /|
            0.0f,   0.0f, 1.0f,         // v2 -          | / |
            0.0f,   0.0f, 1.0f          // v3 -          v1    v3
    };
```

Also create a buffer object for the normals with:

```
    glGenBuffers( 1, &normalBuffer );
    glBindBuffer( GL_ARRAY_BUFFER, normalBuffer );
    glBufferData( GL_ARRAY_BUFFER, sizeof(normals),
                  normals, GL_STATIC_DRAW );
```

Also, we will need to be able to read the normals from the shader so add a line:

```
    glBindAttribLocation(shaderProgram, 0, "vertex");
    glBindAttribLocation(shaderProgram, 2, "texCoordIn");
    glBindAttribLocation(shaderProgram, 1, "normalIn");
```

Set up the pointer to the buffer object:

```
glBindBuffer( GL_ARRAY_BUFFER, normalBuffer );
glVertexAttribPointer(1, 3, GL_FLOAT, false, 0, 0);
```

And enable the new vertex attribute array:

```
glEnableVertexAttribArray(1);
```

Shading

It's time to add some simple shading to our framework. We will start with a simple diffuse component. First however, we will need to be able to access the view-space coordinates of the vertices and normals in the shaders, so we need to send the *modelView* matrix along to the vertex shader. After the line:

```
setUniformSlow(shaderProgram, "modelViewProjectionMatrix",
               projectionMatrix * viewMatrix * modelMatrix);
```

add:

```
setUniformSlow(shaderProgram, "modelViewMatrix", viewMatrix *
                                                 modelMatrix);
```

Note that we are here making use of a convenient helper function, `setUniformSlow`, that we provide in `glutil.h/cpp`, it simply wraps the calls to `glGetUniformLocation` and `glUniform*`. The function is overloaded for a few of the most common types, e.g. `float4x4`, `float` and `float3`.

Next, add this matrix as a uniform input to the vertex shader (in simple.vert):

```
uniform mat4 modelViewMatrix;
```

Also, normals can not be transformed with the ordinary *modelView* matrix (see Course Book chapter 4.1.7). So, in *display()*, after the line you just added, add:

```
setUniformSlow(shaderProgram, "normalMatrix",
               inverse(transpose(viewMatrix * modelMatrix)));
```

And again, add a uniform to the vertex shader:

```
uniform mat4 normalMatrix;
```

Finally, we will need the lights position in view space. Add the lines (in *display(),* after previous setUniformSlow)**:**

```
float4 lightPosition =
       make_vector4(sphericalToCartesian(light_theta, light_phi, light_r), 1.0f);
float4 viewSpaceLightPosition = viewMatrix * modelMatrix * lightPosition;
setUniformSlow(shaderProgram, "viewSpaceLightPosition",
               make_vector3(viewSpaceLightPosition));
```

And add the uniform in the fragment shader (simple.frag):

```
uniform vec3 viewSpaceLightPosition;
```

Allright! We have all we need to shade our fragments. We will calculate the view-space position and normal in the vertex shader and send them on to the fragment shader so add (before *main()* in simple.vert):

```
out vec3 viewSpaceNormal;
out vec3 viewSpacePosition;
```

```
in  vec3 normalIn;
```

Then in *main()*:

```
viewSpaceNormal = (normalMatrix * vec4(normalIn,0.0)).xyz;
viewSpacePosition = (modelViewMatrix * vec4(vertex, 1.0)).xyz;
```

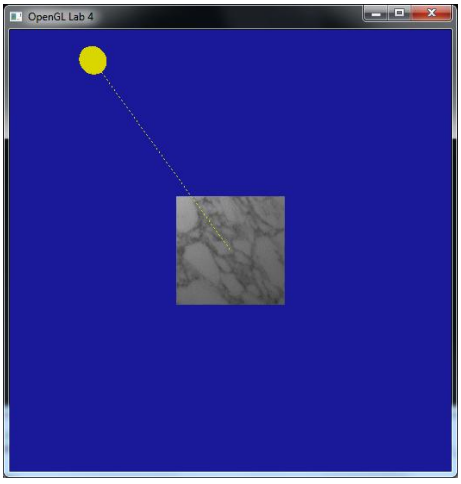
Now we add these as inputs to the fragment shader (before *main()* in simple.frag):

```
in vec3 viewSpaceNormal;
in vec3 viewSpacePosition;
```

Then, in main() in the fragment shader, calculate the diffuse reflectance and multiply the texture color by that value (replace the code in *main()* with):

```
vec3 posToLight = normalize(viewSpaceLightPosition -
                            viewSpacePosition);
float diffuseReflectance = max(0, dot(posToLight,
                                  normalize(viewSpaceNormal)));
vec4 diffuseColor = texture(diffuse_texture, texCoord);
fragmentColor = diffuseReflectance * diffuseColor;
```

Run the program and see that the quad is now shaded based on where the light is. It should look something like this:



Loading an .obj model

Shaded or not, a quad is still just a quad and quite boring to look at. Let's make things more interesting by loading a model from disk. We supply a small OBJModel class that helps you do this, and you do not have to study it in detail unless you are interested. There is no magic being introduced here though, the class simply loads the vertex data from disk and creates a vertex array object just as you have done before.

Declare the global variable in the beginning of lab4_main.cpp:

```
OBJModel *fighterModel;
```

Then, in *initGL()* read the model from disk:

```
fighterModel = new OBJModel;
fighterModel->load("../scenes/fighter.obj");
```

Finally in display, replace the glDrawArrays call with:

```
fighterModel->render();
```

You can run the program now, but the model might look strange. This model has no texturecoordinates defined, so we need to change some things in the fragment shader. The *OBJModel::render()* method will set two uniforms called `has_diffuse_texture` and `material_diffuse_color`. Add these uniforms to your fragment shader:

```
uniform int has_diffuse_texture;
uniform vec3 material_diffuse_color;
```

And then change the code that sets the diffuse color to:

```
vec4 diffuseColor = (has_diffuse_texture == 1) ?
                        texture(diffuse_texture, texCoord):
                        vec4(material_diffuse_color, 1.0);
```

Run the program again and enjoy the colorful space fighter!

Loading the Cube Map

There are various ways of storing the whole environment in one texture (such as *spherical texture mapping* and *paraboloid texture-mapping*). In this lab we will use a technique called *Cube Mapping* which is simple in that it simply stores the environment as six images corresponding to the six faces of a cube surrounding the object. For more info on environment mapping, see chapter 8.4 of the course book.

A Cubemap in OpenGL is just a special sort of texture. It is loaded and configured much like a normal (GL_TEXTURE_2D) texture, except it has six images that must be loaded separately. First of all, take a look at the six images we will be using in the project directory (cube0.png, cube1.pn, …). Now, to load these images into a texture add the lines:

```
texcubemap = loadCubeMap("cube0.png", "cube1.png",
                         "cube2.png", "cube3.png",
                         "cube4.png", "cube5.png");
```

…after the loading of the 2D texture. *loadCubeMap()* is a little helper function we made for you (in glutil.cpp). Take a look at it. It works just like loading a 2D texture as you did in Lab 2.

Now we need to associate a sampler uniform with the second texture unit, so we can look at the texture from the shaders. Find the place where this is done for the 2D texture and add:

```
// Get the location in the shader for uniform tex0
int texLoc = glGetUniformLocation( shaderProgram, "diffuse_texture");
glUniform1i( texLoc, 0 );
// Get the location in the shader for uniform envtexture
texLoc = glGetUniformLocation( shaderProgram, "envtexture" );
// Set tex1 to 0, to associate it with texture unit 1
glUniform1i( texLoc, 1 );
```

Before drawing our fighter, we want to bind the cubemap to texture unit 1 (the 2D texture is on unit 0) so add:

```
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_CUBE_MAP, texcubemap);
```

in *drawScene()*.

Using the Cube Map

Now we have all we need to add reflection of the environment to our quad. Let's just modify the shaders. Add to the fragment shader:

```
uniform samplerCube envtexture;
```

before the *main()* function. Finally, we will sample the environment map, to get a reflection per fragment. The first thing we need to do is figure out the reflection vector (the vector that goes towards the eye, reflected around the normal vector, in view-space):

```
vec3 reflectionVector = reflect(normalize(viewSpacePosition),
                                normalize(viewSpaceNormal));
```

Then use this vector to lookup the reflected color in the Cube Map. Add the following line at the end of *main()*.

```
vec4 reflectedColor = texture(envtexture, reflectionVector);
```

And add the reflected color to the calculated fragmentColor:

```
fragmentColor += reflectedColor;
```

Run the program again. Shiny! A bit too shiny perhaps. **Change the fragment shader so that only 30% of the reflected color is added to the final color.**

When done, show your result to one of the assistants. Have the finished program running and be prepared to answer some questions about what you have done.