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 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 real-time 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, "position");
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);
```

Loading an .obj model

That was the manual way to get normals into the program. However, usually we will get our vertex data from some modeling package, in some file format. Also, a simple quad is just not going to illustrate the shading very well.

Let's make things more interesting by loading a model from disk. We supply a small class called `OBJModel` to help 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 vertex array objects just as you have just done.

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();
```

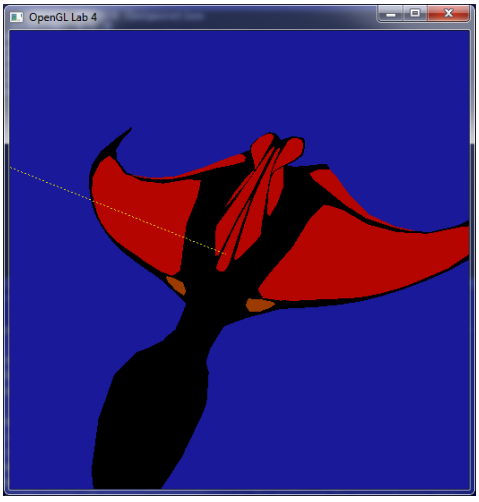
You can run the program now, but the model might look strange. This model has no texture-coordinates defined, so we need to change some things in the fragment shader. The *OBJModel::render()* method will set two uniforms called `has_diffuse_texture` and `material_diffuse_color`. The corresponding uniforms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

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

There's also a helper function that you can use, `sampleDiffuseTexture()`. The helper function will return either the color of the texture, if the model has a texture, or white. Use this function together with the material diffuse color (`material_diffuse_color`) to compute the fragment color:

```
fragmentColor = vec4( sampleDiffuseTexture() *
      material_diffuse_color, 1.0 );
```

Run the program again and enjoy the (somewhat) colorful space fighter (image, right)! While it looks somewhat more interesting than the quad, we might perhaps want some interaction between the surfaces and that light?



Shading

It's time to add some shading to our framework. The lighting model we will use is based on the lecture notes, so now is a good time to go over these to refresh your memory. The

model will contain *ambient*, *diffuse*, *specular*, and *emissive* terms. These are combined (added) to produce a simple (ad-hoc) lighting model that accounts for the most prominent phenomena, while still being very much real-time.

All shading will take place in view space, which is a convenient choice as we easily can acess the transformation to this space. To this end we will add code to transform the positions and normals to view space in the shaders. Therefore 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, declare this matrix as a uniform input to the vertex shader (in simple.vert):

```
uniform mat4 modelViewMatrix;
```

Also, normals can usually 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 * lightPosition;
setUniformSlow(shaderProgram, "viewSpaceLightPosition",
            make_vector3(viewSpaceLightPosition));
```

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

```
uniform vec3 viewSpaceLightPosition;
```

Then calculate the view-space position and normal in the vertex shader and send them on to the fragment shader; add, before *main()* in `simple.vert`:

```
out vec3 viewSpaceNormal;
out vec3 viewSpacePosition;
in  vec3 normalIn;
```

Then in `main()`, to transform the vertex data:

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

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

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

We now have the data needed to compute shading, first a quick overview of the components.

1. **Ambient**
   This term accounts for indirect light, and is the least accurate. The ambient light in the scene is supplied in the uniform `scene_ambient_light`. Materials can have an ambient reflectance term, and/or texture, but we simply make use of the diffuse reflectance, this often works fine.
2. **Diffuse**
   Accounts for direct illumination that is scattered equally in all direction, according to the *lambertian* reflection model. The diffuse light in the scene is supplied in `scene_light`, and the reflectance is the combination of the uniform `material_diffuse_color`, and the texture `diffuse_texture`.
3. **Specular**
   Accounts for more mirror-like behaviour, where light bounces of the surface, we will use the *blinn-phong* model, which is more realistic and better looking than straight phong. The specular light is also represented by `scene_light`, as we do not need to support separate colors for diffuse and specular light (as some models do). Specular reflectance is in the uniform `material_specular_color`.
4. **Emissive**
   Finally, emissive is what light *originates*, at the surface, i.e. the material glows. This is represented in the `material_emissive_color`.

Each of the components are calculated separately and, as they model different phenomena in the lighting equation, can just be summed to produce the total shading. There is a set of empty functions for this in simple.frag. We will start by building up from ambient:

First we need to calculate the ambient reflectance of the material, as mentioned above, this is simply done by multiplying together the diffuse texture and material colors, add at the start of `main` (in `simple.frag`):

```
vec3 ambient = material_diffuse_color * sampleDiffuseTexture();
```

Check out `sampleDiffuseTexture()` in the code, which is a helper function to support materials with and without textures. Next we use this to calculate the reflected ambient light, replace :

```
fragmentColor = texture(diffuse_texture, texCoord.xy);
```

With:

```
vec3 shading = calculateAmbient(scene_ambient_light, ambient);
fragmentColor = vec4(shading, 1.0);
```

Now you must also implement the function `calculateAmbient`, to do the correct thing with its arguments. Try to do this yourself, locate `TODO #1`, in the code (in a comment). The solution is at the end of this PM, under **Solutions**, soluton #2.

Ambient light does not really add much realism, so we will proceed to add diffuse light as well. Again we start by working out how much diffuse light is reflected by the material. Add, right the declaration of `ambient`:

```
vec3 diffuse = sampleDiffuseTexture() * material_diffuse_color;
```

Note that this is in fact identical to the ambient reflectance, which is what we want in this simplified model. Now, we add the call to calculate the diffuse shading to the mix:

```
vec3 shading = calculateAmbient(scene_ambient_light, ambient)
  + calculateDiffuse(scene_light, diffuse, normal, directionToLight);
```

This function takes quite a few arguments that we have not defined yet, so lets get to that first. The `normal` is just a re-normalized copy of the input `viewSpaceNormal`, add:

```
vec3 normal = normalize(viewSpaceNormal);
```

Then we also need the direction to the light, which we can calculate from the light position (in view space), and the *sample* position (i.e. the position that we wish to compute shading for, also in view space).

```
vec3 directionToLight =
          normalize(viewSpaceLightPosition - viewSpacePosition);
```

Now you should attempt to implement the diffuse shading in the function `calculateDiffuse`. Refer back to the lecture notes for the lambertian shading model. Locate `TODO #2`, in the fragment shader. The solution is #2.

The shading is now a bit more interesting, but we still have two more components to go. Next we add the specular component, which approximates glossy reflection of light. Glossy reflections do not usually pick up the material color, the way diffuse and ambient do, and is therefore modeled with separate colors and textures (we do not support using a specular texture). Again we compute the reflectance near the oters:

```
vec3 specular = material_specular_color;
```

And now we add to the shading again:

```
vec3 shading = calculateAmbient(scene_ambient_light, ambient)
  + calculateDiffuse(scene_light, diffuse, normal, directionToLight)
  + calculateSpecular(scene_light, specular, material_shininess,
                      normal, directionToLight, directionFromEye);
```

And again we have a new parameter that we must supply: `directionFromEye`. The direction from the eye to the sample position, which is of course also in view space. In view space, conveniently, the eye is *defined* as being at the origin, and we therefore get that: `directionFromEye` = `viewSpacePosition` – {0, 0, 0}, which is the same as (with normalization thrown in):

```
vec3 directionFromEye = normalize(viewSpacePosition);
```

Perhaps unsurprisingly, we now exhort you to try to implement `calculateSpecular`,on your own. Refresh your memory from the lecture notes, and the book. Look for `TODO #3`, and the solution is, again, at the end: Solution #3.

If you check out the specular high-lights, notice that the highlights are a bit washed out looking. This is because the traditional specular computations we have made use of does not conserve energy, we will make use of a normalization term to correct for this. For more details see Real Time Rendering 3rd Edition, Section 7.6, speciellt Figurer 7.36 och 7.37. Add to `calculateSpecular`:

```
float normalizationFactor = ((materialShininess + 2.0) / 8.0);
```

Then multiply the specular comtribution by this factor before returning (Solution #4). The specular highlights should now be **noticably** more intense and vibrant.

The final term in our model is emissive, material emittance is computed thus:

```
vec3 emissive = sampleDiffuseTexture() * material_emissive_color;
```

This is simply added to the shading, implement this. Notice how the cockpit, exhaust and underside of the space ship are no longer black. They are in fact giving off light (this will be much more noticable in tutorial #5).

This concludes the basic implementation of the lighting model, we will now move on to cube map reflections and then integrate this into the lighting model.

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 cube map 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.png, …). Now, to load these images into a texture, add the lines:

```
cubeMapTexture = 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
setUniformSlow(shaderProgram, "diffuse_texture", 0 );;
glUniform1i( texLoc, 0 );
setUniformSlow(shaderProgram, "environmentMap", 1);
```

Before drawing our fighter, we want to bind the cube map to texture unit 1 (the 2D texture is on unit 0) so, in *drawScene(),* add:

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

Using the Cube Map

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

```
uniform samplerCube environmentMap;
```

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(directionFromEye, normal);
```

Unfortunately, we need to look up the environment map in world space, as the data in it is supposed to represent the entire environment. Therefore we will transform the reflection vector to world space, using an inverse normal transform from view to world space.

```
vec3 reflectionVector = (inverseViewNormalMatrix *
                  vec4(reflect(directionFromEye, normal), 0.0)).xyz;
```

Now we also must supply the uniform `inverseViewNormalMatrix`. In simple.frag add:

```
uniform mat4 inverseViewNormalMatrix;
```

And to set this uniform add:

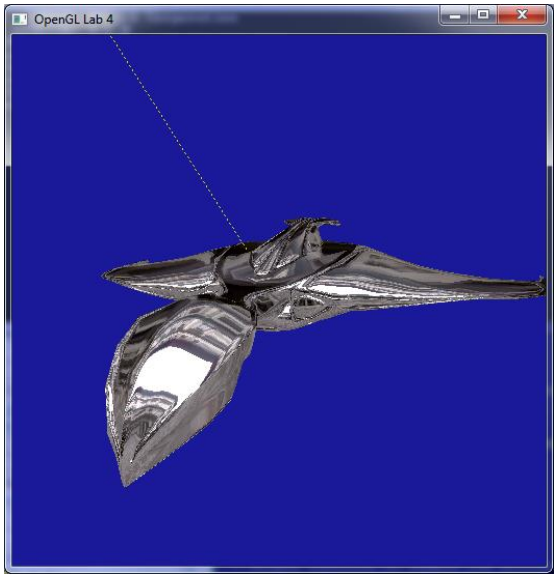
```
setUniformSlow(shaderProgram, "inverseViewNormalMatrix",
      transpose(viewMatrix));
```

Then use this vector to lookup the reflected color in the Cube Map, in simple.frag:

```
vec3 envMapSample = texture(environmentMap, reflectionVector).rgb;
```

To see the result, temporarily set the output color to the environment map sample:

```
//fragmentColor = vec4(shading, 1.0);
fragmentColor = vec4(envMapSample, 1.0);
```



Run the program again. Shiny! A bit too shiny perhaps. We now want to integrate this into the light model somehow. We do this by simply adding the environment map sample multiplied with the specular reflectance to the final shading. The rationale behind this is that the mirror reflection is also a form of specular reflection. Adding this is fine as the environment map does not contain the *direct* light from the light source, which is added earlier in `calculateSpecular`. Attempt to do this. The solution is #5.

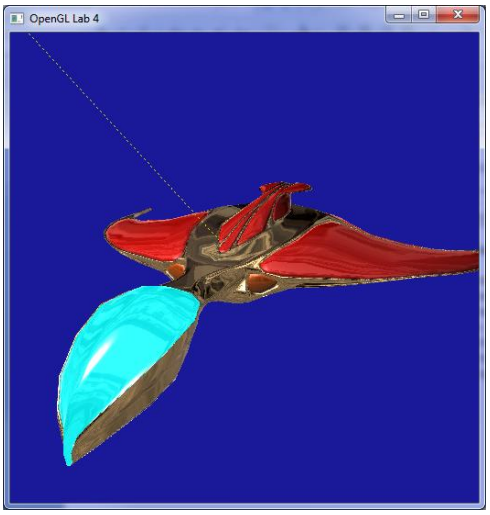
Fresnel Reflections

The final improvement that we w ill add to the lighting model, is a fresnel term, affecting the specular reflections. In reality materials reflect more at glancing angles, which is easily observed in puddles of water (for example); they look like mirrors from a distance, but when looking straight down you can see the road surface beneeth.

The actual phenomenon is quite complex, and is therefore usually approximated. We will use the schlick approximation, which is common in real-time renering, see the course book (Real Time Rendering 3rd edition), Avsnitt 7.5.3. To include it in our shading, add to `simple.frag` as below:

```
vec3 fresnelSpecular = calculateFresnel(specular, normal,
                                        directionFromEye);
```

Then replace the uses of `specular` with `fresnelSpecular`, in the lighting calculations. Now try to implement `calculateFresnel`, locate `TODO #4`. Note that we use the specular reflectance as the 'r0' value, and that this is of type `vec3`. The solution is given in Solution #6. Notice the suble, effect in the below pair of images, to the right, the fresnel term gives a brighter edge.



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.

Solutions

Solution #1:

```
vec3 calculateAmbient(vec3 ambientLight, vec3 materialAmbient)
{
  return ambientLight * materialAmbient;
}
```

Solution #2:

```
vec3 calculateDiffuse(vec3 diffuseLight, vec3 materialDiffuse,
                      vec3 normal, vec3 directionToLight)
{
  return diffuseLight * materialDiffuse
           * max(0, dot(normal, directionToLight));
}
```

Solution #3:

```
vec3 calculateSpecular(vec3 specularLight, vec3 materialSpecular,
                       float materialShininess, vec3 normal,
                       vec3 directionToLight, vec3 directionFromEye)
{
  vec3 h = normalize(directionToLight - directionFromEye);
  return specularLight * materialSpecular
     * pow(max(0, dot(h, normal)), materialShininess);
}
```

Solution #4:

```
vec3 calculateSpecular(vec3 specularLight, vec3 materialSpecular,
                       float materialShininess, vec3 normal,
                       vec3 directionToLight, vec3 directionFromEye)
{
  vec3 h = normalize(directionToLight - directionFromEye);
  float normalizationFactor = ((materialShininess + 2.0) / 8.0);
  return specularLight * materialSpecular
     * pow(max(0, dot(h, normal)), materialShininess)
     * normalizationFactor;
}
```

Solution #5:

```
vec3 shading = calculateAmbient(scene_ambient_light, ambient)
  + calculateDiffuse(scene_light, diffuse, normal, directionToLight)
  + calculateSpecular(scene_light, specular, material_shininess,
                      normal, directionToLight, directionFromEye)
  + emissive
  + envMapSample * specular;
```

Solution #6:

```
vec3 calculateFresnel(vec3 materialSpecular, vec3 normal,
                      vec3 directionFromEye)
{
  return materialSpecular + (vec3(1.0) - materialSpecular)
    * pow(clamp(1.0 + dot(directionFromEye, normal), 0.0, 1.0), 5.0);
}
```