UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Lighting, Materials and 3D Models

In this lab, we will look at adding a directional light source to a scene, material properties and loading simple models.

First, have a look at the "utilities.h" file in the Lab 6 project. This file is used for convenience to contain some of the common elements that we will use. Notice that code to include various headers, e.g., C++ headers, OpenGL headers, have been placed in this file. The file also contains various vertex formats, along with structs for light properties and material properties.

Directional Light

Directional light simulates a source at a distance where the scene is illuminated with parallel light rays.

Look at the code in the vertex shader. The intensity of light that reflects off a surface depends on the orientation of the surface with respect to the light. A normal vector provides an indication of this orientation. In the vertex shader, each vertex has position and normal attribute variables.

```
layout(location = 0) in vec3 aPosition;
layout(location = 1) in vec3 aNormal;
```

For Phong shading, i.e. per fragment lighting, the vertex positions and normals must output by the vertex shader

```
out vec3 vPosition;
out vec3 vNormal;
```

these values will be interpolated, then form the input to the fragment shader. Before the vertex shader outputs these values, the position and normal are multiplied by the model matrix and normal matrix (uniform variables set from in the application code), respectively. This means lighting calculations will be performed in world space (some people prefer to do this in view space).

```
vPosition = (uModelMatrix * vec4(aPosition, 1.0f)).xyz;
vNormal = uNormalMatrix * aNormal;
```

Now look at the code in the fragment shader. The position and normal are input to the fragment shader.

```
in vec3 vPosition;
in vec3 vNormal;
```

The code then declares a struct for light properties and another struct for material properties. This is followed by uniform variables for the camera viewpoint, light properties and material properties, which will be set in the main application code.

```
uniform vec3 uViewpoint;
uniform Light uLight;
uniform Material uMaterial;
```

In the shader's **main()** function, the code first computes the unit vectors required to calculate the intensity of reflected light, based on the Blinn-Phong reflection model shown in figure 1.

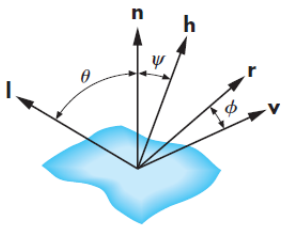

Figure 1: Blinn-Phong reflection model.

Note that all vectors are normalised to unit vectors.

```
vec3 n = normalize(vNormal);
vec3 v = normalize(uViewpoint - vPosition);
vec3 l = normalize(-uLight.dir);
vec3 h = normalize(l + v);
```

With these unit vectors, the code calculates the intensity of reflected light using the Blinn-Phong reflection model and outputs the resulting colour.

```
vec3 Ia = uLight.La * uMaterial.Ka;
vec3 Id = vec3(0.0f);
vec3 Is = vec3(0.0f);
float dotLN = max(dot(l, n), 0.0f);

if(dotLN > 0.0f)
{
	Id = uLight.Ld * uMaterial.Kd * dotLN;
	Is = uLight.Ls * uMaterial.Ks * pow(max(dot(n, h), 0.0f), uMaterial.shininess);
}

fColor = Ia + Id + Is;
```

3D Models

A model's mesh is simply a collection of information like vertex positions, normals, texture coordinates, etc. placed in a file. 3D model files have different file formats. We will look at a basic class for loading and rendering 3D models based on the Wavefront .obj file format.

The .obj file format is one of the simplest 3D model file formats. You can open it using a text editor to view its contents (some .obj files can be found in the Lab 6 models folder). The contents of such files will look something like this:

```
...
v 0.437500 0.164063 0.765625
v -0.437500 0.164063 0.765625
v 0.500000 0.093750 0.687500
...
vn 0.189764 -0.003571 0.981811
vn 0.646809 -0.758202 0.082095
vn 0.999573 -0.014496 -0.024445
...
vt 0.315596 0.792535
vt 0.331462 0.787091
vt 0.331944 0.799704
...
f 294/289/289 296/291/291 310/343/336 362/363/351
f 311/345/338 297/292/292 295/290/290 363/365/353
...
```

where 'v' is for a vertex position, 'vn' is for a vertex normal, 'vt' is for a texture coordinate, and 'f' is for a face, i.e. it stores vertex indices that form a face.

While we can write our own reader to import the contents, this is tedious. This section introduces the SimpleModel class that uses the Open Asset Import Library (Assimp) to load and render simple .obj files.

Open the SimpleModel.h file and have a look at the code. It starts by including some Assimp headers:

```
#include <assimp/Importer.hpp>      // C++ importer interface
#include <assimp/scene.h>           // output data structure
#include <assimp/postprocess.h>     // post processing flags
```

It then declares a struct to store information for a Mesh, which contains a vertex buffer object, index buffer object and vertex array object. This is followed by the declaration of the SimpleModel class. The class simply contains a **loadModel()** function that accepts a file name and whether to load texture coordinates. This function will either call the **loadMesh()** or **loadMeshWithTexture()** private member functions to load a mesh. The **drawModel()** function will render a valid model.

Now, open the SimpleModel.cpp file and look at the code in the **loadModel()** function. The code creates an Assimp importer instance and uses it to read contents from a model file into an Assimp scene object. The flags request the importer to split all faces into triangles, to generate smooth normals for all vertices and to identify and joins identical vertex data sets within all imported meshes.

```
Assimp::Importer importer;

const aiScene *scene = importer.ReadFile(filename,
       aiProcess_Triangulate |
       aiProcess_GenSmoothNormals |
       aiProcess_JoinIdenticalVertices);
```

If the scene loads successfully, another function will be called to load the first mesh. Note that models can have more than one mesh. However, this simple class will only load the first mesh of a model. The mesh will be loaded with or without textures.

```
if(!texture)
       loadMesh(scene->mMeshes[0]);
else
       loadMeshWithTexture(scene->mMeshes[0]);
```

In the **loadMesh()** function, vectors are created to store vertices and indices of the mesh. The function checks whether the mesh contains vertex coordinates, normals and faces. If not, the function will return without loading the mesh. Otherwise, the code will process the mesh by loading all vertex coordinates and normals, then by loading vertex indices of the mesh faces.

Once loaded, the code will create a vertex buffer object, a vertex index object and a vertex array object for the mesh. Finally, the model will be marked as valid.

The **loadMeshWithTexture()** function works similarly, but also loads the first set of texture coordinates. Meshes can have more than one set of texture coordinates. However, this simple class only loads the first set.

The **drawModel()** function simply checks whether the model is valid. If it is, the code binds its vertex array object and renders the vertices.

Setting Up and Rendering the Scene

Look at the code in the directionLight.cpp file. The code declares variables for light properties, material properties and for a model.

```
Light gLight;
Material gMaterial;
SimpleModel gModel;
```

The contents of these are initialised in the **init()** function. The light direction and the intensity of the ambient, diffuse and specular components are set as follows:

```
gLight.dir = glm::vec3(0.3f, -0.7f, -0.5f);
gLight.La = glm::vec3(0.8f);
gLight.Ld = glm::vec3(0.8f);
gLight.Ls = glm::vec3(0.8f);
```

Material properties:

```
gMaterial.Ka = glm::vec3(0.33f, 0.22f, 0.03f);
gMaterial.Kd = glm::vec3(0.78f, 0.57f, 0.11f);
gMaterial.Ks = glm::vec3(0.99f, 0.94f, 0.8f);
gMaterial.shininess = 27.9f;
```

Loading a model:

```
gModel.loadModel("./models/sphere.obj");
```

Before rendering the model, various shader uniform variables must be set. This is done in the **render_scene()** function.

First, the shader program is loaded:

```
gShader.use();
```

Pass the light properties to the shader:

```
gShader.setUniform("uLight.dir", gLight.dir);
gShader.setUniform("uLight.La", gLight.La);
gShader.setUniform("uLight.Ld", gLight.Ld);
gShader.setUniform("uLight.Ls", gLight.Ls);
```

Note that the lighting conditions typically apply to the entire scene (unlike material properties). As such, it may only need to be set once at the start of the rendering function.

Next, pass the material properties to the shader:

```
gShader.setUniform("uMaterial.Ka", gMaterial.Ka);
gShader.setUniform("uMaterial.Kd", gMaterial.Kd);
gShader.setUniform("uMaterial.Ks", gMaterial.Ks);
gShader.setUniform("uMaterial.shininess", gMaterial.shininess);
```

Note that for scenes with multiple objects, where each object has different material properties, the material will have to be set for each object before the object is rendered.

Pass the camera viewpoint to the shader:

```
gShader.setUniform("uViewpoint", glm::vec3(0.0f, 2.0f, 4.0f));
```

Note that this is the same as the viewpoint used to create gViewMatrix in the init() function.

The matrices are calculated and passed to the shader:

```
glm::mat4 MVP = gProjectionMatrix * gViewMatrix * gModelMatrix;
glm::mat3 normalMatrix = glm::mat3(glm::transpose(glm::inverse(gModelMatrix)));

gShader.setUniform("uModelViewProjectionMatrix", MVP);
gShader.setUniform("uModelMatrix", gModelMatrix);
gShader.setUniform("uNormalMatrix", normalMatrix);
```

Finally, the model is rendered.

```
gModel.drawModel();
```

A user interface element for displaying the direction of the directional light is defined in the **create_UI()** function:

```
TwAddVarRW(twBar, "Direction", TW_TYPE_DIR3F, &gLight.dir, " group='Light' opened=true ");
TwAddVarRW(twBar, "La", TW_TYPE_COLOR3F, &gLight.La, " group='Light' ");
TwAddVarRW(twBar, "Ld", TW_TYPE_COLOR3F, &gLight.Ld, " group='Light' ");
TwAddVarRW(twBar, "Ls", TW_TYPE_COLOR3F, &gLight.Ls, " group='Light' ");
```

Compile and run the program.

Materials (Add the code in this section yourself)

Only a single material is defined in the code. More than one material is typically required in a complex scene. In this section, we will look at creating different material and providing a user interface element to allow the user to select from the different materials.

We will use a C++ map data structure to store different material properties, declare the following as a global variable:

```
std::map<std::string, Material> gMaterials;
```

This will allow us to create a number of materials and use the map's key to identify elements.

Next, create an enum class to represent two different material options:

```
enum class MaterialType { PEARL, JADE };
```

and a variable to record which material was selected:

```
MaterialType gMaterialSelected = MaterialType::PEARL;
```

Define the two materials in the **init()** function as follows:

```
gMaterials["Pearl"].Ka = glm::vec3(0.25f, 0.21f, 0.21f);
gMaterials["Pearl"].Kd = glm::vec3(1.0f, 0.83f, 0.83f);
gMaterials["Pearl"].Ks = glm::vec3(0.3f, 0.3f, 0.3f);
gMaterials["Pearl"].shininess = 11.3f;

gMaterials["Jade"].Ka = glm::vec3(0.14f, 0.22f, 0.16f);
gMaterials["Jade"].Kd = glm::vec3(0.54f, 0.89f, 0.63f);
gMaterials["Jade"].Ks = glm::vec3(0.32f);
gMaterials["Jade"].shininess = 12.8f;
```

You can find settings for different material properties from various websites, for example: https://www.opengl.org/archives/resources/code/samples/sig99/advanced99/notes/node153.html

Before rendering the model, set the shader uniform based on the selected material:

```
std::string material;

if (gMaterialSelected == MaterialType::PEARL)
	material = "Pearl";
else if (gMaterialSelected == MaterialType::JADE)
	material = "Jade";

gShader.setUniform("uMaterial.Ka", gMaterials[material].Ka);
gShader.setUniform("uMaterial.Kd", gMaterials[material].Kd);
gShader.setUniform("uMaterial.Ks", gMaterials[material].Ks);
gShader.setUniform("uMaterial.shininess", gMaterials[material].shininess);
```

Next, create a user interface entry (in the **create_UI()** function) for the user to select between the different materials. To do this, define the text that will be displayed for the various options:

```
TwEnumVal materialValue[] = {
```

```
        {static_cast<int>(MaterialType::PEARL), "Pearl"},
        {static_cast<int>(MaterialType::JADE), "Jade"},
};
```

Then, define the options (ignore warning regarding C++11 enum class):

```
TwType materialOptions = TwDefineEnum("materialType", materialValue, 2);
```

The last argument is the number of options.

Finally, create an entry for the options:

```
TwAddVarRW(twBar, "Material", materialOptions, &gMaterialSelected, " group='Selection' ");
```

Compile and run the program.

Exercises

You should now understand directional light, material properties and loading and rendering simple models. Try modifying the program to do the following:

- Define other material properties
- Load different models and create a user interface element for the user to select a different model
- Create a scene with a number of models, where each model is rendered with a different material

References

Among others, much of the material in this tutorial was sourced from:

- Angel & Shreiner, “Interactive Computer Graphics: A Top-Down Approach with OpenGL”, Addison Wesley
- https://www.opengl.org/archives/resources/code/samples/sig99/advanced99/notes/node153.html
- http://anttweakbar.sourceforge.net/doc/
- https://learnopengl.com/