***Introduction to Graphics and Mobile Gaming***

**LAB 6**

**Lighting**

**Issue 1.0**

1. Contents

[1 Introduction 1](#_Toc36043802)

[2 Code setup 1](#_Toc36043803)

[3 Normals 3](#_Toc36043804)

[4 Types of Lighting 4](#_Toc36043805)

[4.1 Diffuse light 4](#_Toc36043806)

[4.2 Ambient light 5](#_Toc36043807)

[4.3 Specular light 5](#_Toc36043808)

# Introduction

In order to make our graphics look more realistic, we need to introduce the concept of lighting. At the moment, all of our examples are fully lit. We wish to include variation and realism through the use of lighting in this lab. This entails the use of vertex normals and diffuse, specular, and ambient light. OpenGL ES 2.0 does not have an implementation of lighting itself, so it is up to the developer to create this.

In this tutorial, we will implement a directional light. This will involve doing calculations per vertex, and this approach will bring up certain limitations. The Phong reflection model will be used as a basis for this tutorial; this can be read about online (lectures). Ensure that you rename parts that reference the name of the previous project such as “SimpleTriangle”. or rename the project itself.

# Code setup

This section is mainly concerned with providing some initial code that we need to set up the rest of the tutorial. Please note that we are not focused on using textures for now, so you can go ahead and remove all the code related to that.

All the new changes occur in ***“native-lib.cpp”***. We will begin by adding an extra vertex per face. This vertex will be placed in the middle of each face of the cube, but sticking out. This turns the cube more into an approximation of a sphere. This will help in creating the lighting effect. The vertices array will look like this when the changes are made:

**GLfloat verticies**[] = { 1.0f, 1.0f, -1.0f, /\* Back. \*/

 -1.0f, 1.0f, -1.0f,

 1.0f, -1.0f, -1.0f,

 -1.0f, -1.0f, -1.0f,

 0.0f, 0.0f, -2.0f,

 -1.0f, 1.0f, 1.0f, /\* Front. \*/

 1.0f, 1.0f, 1.0f,

 -1.0f, -1.0f, 1.0f,

 1.0f, -1.0f, 1.0f,

 0.0f, 0.0f, 2.0f,

 -1.0f, 1.0f, -1.0f, /\* Left. \*/

 -1.0f, 1.0f, 1.0f,

 -1.0f, -1.0f, -1.0f,

 -1.0f, -1.0f, 1.0f,

 -2.0f, 0.0f, 0.0f,

 1.0f, 1.0f, 1.0f, /\* Right. \*/

 1.0f, 1.0f, -1.0f,

 1.0f, -1.0f, 1.0f,

 1.0f, -1.0f, -1.0f,

 2.0f, 0.0f, 0.0f,

 -1.0f, -1.0f, 1.0f, /\* Bottom. \*/

 1.0f, -1.0f, 1.0f,

 -1.0f, -1.0f, -1.0f,

 1.0f, -1.0f, -1.0f,

 0.0f, -2.0f, 0.0f,

 -1.0f, 1.0f, -1.0f, /\* Top. \*/

 1.0f, 1.0f, -1.0f,

 -1.0f, 1.0f, 1.0f,

 1.0f, 1.0f, 1.0f,

 0.0f, 2.0f, 0.0f };

Due to the addition of the extra vertex per face in this array, we also need to update our colour array. Make sure you remember to enable the colour attribute.

GLfloat colour[] = {1.0f, 0.0f, 0.0f, /\* Back. \*/

 1.0f, 0.0f, 0.0f,

 1.0f, 0.0f, 0.0f,

 1.0f, 0.0f, 0.0f,

 1.0f, 0.0f, 0.0f,

 0.0f, 1.0f, 0.0f, /\* Front. \*/

 0.0f, 1.0f, 0.0f,

 0.0f, 1.0f, 0.0f,

 0.0f, 1.0f, 0.0f,

 0.0f, 1.0f, 0.0f,

 0.0f, 0.0f, 1.0f, /\* Left. \*/

 0.0f, 0.0f, 1.0f,

 0.0f, 0.0f, 1.0f,

 0.0f, 0.0f, 1.0f,

 0.0f, 0.0f, 1.0f,

 1.0f, 1.0f, 0.0f, /\* Right. \*/

 1.0f, 1.0f, 0.0f,

 1.0f, 1.0f, 0.0f,

 1.0f, 1.0f, 0.0f,

 1.0f, 1.0f, 0.0f,

 0.0f, 1.0f, 1.0f, /\* Bottom. \*/

 0.0f, 1.0f, 1.0f,

 0.0f, 1.0f, 1.0f,

 0.0f, 1.0f, 1.0f,

 0.0f, 1.0f, 1.0f,

 1.0f, 0.0f, 1.0f, /\* Top. \*/

 1.0f, 0.0f, 1.0f,

 1.0f, 0.0f, 1.0f,

 1.0f, 0.0f, 1.0f,

 1.0f, 0.0f, 1.0f

 };

We also have to alter the indices array in order to take this change into account. We now draw four triangles per face, all of which use the vertex in the middle of the face. The array will now look like this:

GLushort indices[] = {0, 2, 4, 0, 4, 1, 1, 4, 3, 2, 3, 4, /\* Back. \*/

 5, 7, 9, 5, 9, 6, 6, 9, 8, 7, 8, 9, /\* Front. \*/

 10, 12, 14, 10, 14, 11, 11, 14, 13, 12, 13, 14, /\* Left. \*/

 15, 17, 19, 15, 19, 16, 16, 19, 18, 17, 18, 19, /\* Right. \*/

 20, 22, 24, 20, 24, 21, 21, 24, 23, 22, 23, 24, /\* Bottom. \*/

 25, 27, 29, 25, 29, 26, 26, 29, 28, 27, 28, 29 /\* Top. \*/

 };

The final alteration we need to make before proceeding with the tutorial is to the *“glDrawElements”* call in our render.

**glDrawElements**(GL\_TRIANGLES, 72, GL\_UNSIGNED\_SHORT, indices);

We increase the number of vertices drawn as we are drawing twice the number of triangles.

# Normals

In most lighting models, it is necessary to know which way the polygons are facing so that we can take them into account when generating lighting. We need to know whether the polygon is facing the light or not, and if it isn’t, it shouldn’t be affected by the light.

You might think that because we specified the vertices with the correct winding in order to show which way we wanted them to face, that the system should know. But this is not the case for two reasons. First, there is no tool in OpenGL ES shaders to get this information, and second, specifying the information separately allows us to do more things and makes the possibilities more powerful.

To specify it ourselves, we use “normals” to encode the information. Every vertex in the scene has a vector pointing perpendicularly from the surface it is attached to. In order to send this data to the GPU, we do the same thing we did for colour and position data. We define an array containing per vertex normal; we get the location of the normal variable in the vertex shader and then upload the data to the GPU using *“glVertexAttribPointer*” and *“glEnableVertexAttribArray”.*

We will now go through this step by step in code. We start by adding a new attribute in the vertex shader. This will represent the vertex normals we have just discussed:

 "attribute vec3 vertexNormal;\n"

The next thing we do is create a global variable in the native code (not within the shader). This will hold the vertex normal attribute location:

 GLuint vertexNormalLocation;

We then need to obtain the location of this attribute from the host side in “setupGraphics”:

 vertexNormalLocation = glGetAttribLocation(lightingProgram, "vertexNormal");

Now we create an array that contains all of the vertex normal. Make sure to enable the array.

GLfloat normals[] = { 1.0f, 1.0f, -1.0f, /\* Back. \*/

 -1.0f, 1.0f, -1.0f,

 1.0f, -1.0f, -1.0f,

 -1.0f, -1.0f, -1.0f,

 0.0f, 0.0f, -1.0f,

 -1.0f, 1.0f, 1.0f, /\* Front. \*/

 1.0f, 1.0f, 1.0f,

 -1.0f, -1.0f, 1.0f,

 1.0f, -1.0f, 1.0f,

 0.0f, 0.0f, 1.0f,

 -1.0f, 1.0f, -1.0f, /\* Left. \*/

 -1.0f, 1.0f, 1.0f,

 -1.0f, -1.0f, -1.0f,

 -1.0f, -1.0f, 1.0f,

 -1.0f, 0.0f, 0.0f,

 1.0f, 1.0f, 1.0f, /\* Right. \*/

 1.0f, 1.0f, -1.0f,

 1.0f, -1.0f, 1.0f,

 1.0f, -1.0f, -1.0f,

 1.0f, 0.0f, 0.0f,

 -1.0f, -1.0f, 1.0f, /\* Bottom. \*/

 1.0f, -1.0f, 1.0f,

 -1.0f, -1.0f, -1.0f,

 1.0f, -1.0f, -1.0f,

 0.0f, -1.0f, 0.0f,

 -1.0f, 1.0f, -1.0f, /\* Top. \*/

 1.0f, 1.0f, -1.0f,

 -1.0f, 1.0f, 1.0f,

 1.0f, 1.0f, 1.0f,

 0.0f, 1.0f, 0.0f

 };

The final thing we need to do is alter the code to upload the data to the GPU every frame in *“renderFrame”:*

 **glVertexAttribPointer**(vertexNormalLocation, 3, GL\_FLOAT, GL\_FALSE, 0, normals);

 **glEnableVertexAttribArray**(vertexNormalLocation);

The vertex normal array would typically be generated by the modeling tool; however, for this tutorial, we are just manually writing it to fit our model.

# Types of Lighting

## Diffuse light

In order to create the diffused lighting effect described in the lectures, we need to make the following changes to the vertex shader main function:

 " vec3 transformedVertexNormal = normalize((modelView \* vec4(vertexNormal, 0.0)).xyz);"

 " vec3 inverseLightDirection = normalize(vec3(0.0, 1.0, 1.0));\n"

 " fragColour = vec3(0.0);\n"

Like we did with the vertex positions, we must modify the normals using the “modelViewMatrix” so that they match up. As a result of normals being vectors as opposed to positions, we set the w component of the normals to 0; this means that any translations in the model view matrix are ignored. The next thing we do is set up the direction that we wish our light to come from. As mentioned in the lecture, the light direction is reversed so we have to code the opposite direction. Finally, we set the fragment colour to zero.

In order to perform the calculation required for the diffuse lighting, we need the following code:

 " vec3 diffuseLightIntensity = vec3(1.0, 1.0, 1.0);\n"

 " vec3 vertexDiffuseReflectionConstant = vertexColour;\n"

 " float normalDotLight = max(0.0, dot(transformedVertexNormal, inverseLightDirection));\n"

 " fragColour += normalDotLight\*vertexDiffuseReflectionConstant \* diffuseLightIntensity;\n"

The first line of code is related to the RGB intensity of the diffuse light; this allows us to set the light to a certain colour. In this example, it is a white light. The next line sets an RGB constant to outline what ratio of the diffuse light this surface should reflect (between 1 and 0). This provides the ability to set the colour of the surface. Given a ratio of (1.0, 0.0, 0.0), 100% of the red light will be reflected whilst 0% of the green and blue light is reflected. This will lead to the objects appearing red. In this case, we just set it to the colours we have stored in *“vertexColour”*.

After this, we do the dot product of the normal and the light direction in order to get the cosine of the angle between the two. The “max” function calls that the negative values are ignored as we do not wish for light to affect these. The final thing we do is combine them together and add it to the fragment colour.

## Ambient light

Diffuse lighting is good for displaying light that falls on the object directly from the source, but we also need to think about light that is being reflected from the objects. Objects in the real world that aren’t directly in light still aren’t pitch-black; modeling this is known as “global illumination”. To recreate this accurately in OpenGL ES is very consuming and technical, so we will simulate a similar visual effect called “Ambient light”. Essentially, we define everything in the scene as being hit by a small uniform amount of light.

We will need to alter our vertex shader again to include this lighting:

 " vec3 ambientLightIntensity = vec3(0.1, 0.1, 0.1);\n"

 " vec3 vertexAmbientReflectionConstant = vertexColour;\n"

 " fragColour += vertexAmbientReflectionConstant \* ambientLightIntensity;\n"

The first line sets the ambient light intensity; this essentially sets how much reflected light there is in the scene. The next line sets a constant outlining the ratios for the ambient lighting and what the surface should reflect (same as the previous diffuse example). The final line multiplies these values together and adds the result to the *“fragColour”* like before. This will give everything in the scene some light.

## Specular light

In order to introduce specular light into our scene, there are a few things we need to do. In the vertex shader, we add:

 " vec3 inverseEyeDirection = normalize(vec3(0.0, 0.0, 1.0));\n"

 " vec3 specularLightIntensity = vec3(1.0, 1.0, 1.0);\n"

 " vec3 vertexSpecularReflectionConstant = vec3(1.0, 1.0, 1.0);\n"

 " float shininess = 2.0;\n"

 " vec3 lightReflectionDirection = reflect(vec3(0) - inverseLightDirection,

 transformedVertexNormal);\n"

 " float normalDotReflection = max(0.0, dot(inverseEyeDirection,

 lightReflectionDirection));\n"

 " fragColour += pow(normalDotReflection, shininess) \*

 vertexSpecularReflectionConstant \* specularLightIntensity;\n"

The direction of the eye is crucial when calculating specular lighting. Like before, we require the inverse eye direction. Then, like ambient and diffuse lighting, we have the RGB light intensity constant. Previously, we used the vertex colour, but here, we set it to (1.0, 1.0, 1.0). This means that when the reflected direction is equal to the eye vector, the specular colour will be exactly (1.0, 1.0, 1.0). This provides us with a white highlight.

Then, we define the “shininess” float that determines how “tight” the highlight is. We use this as the exponent for the dot product of the two vectors later on. Due to the dot product only being between 0 and 1, the higher the exponent, the smaller the result of that pow calculation will be. This means that the specular value will decrease more rapidly as the reflected direction moves away from the eye vector. We use the reflect function in order to find the value of the reflected direction. Finally, we do the dot product of the reflected direction and the eye vector, multiplying everything together in order to calculate the specular contribution.