Shaders Introduction

COMP3015 Lab 2

Diffuse Model

Let's start by setting up the project (use the previous downloaded template, or modify your last week project). Please examine the code carefully in order to understand it. We try to abstract all of these elements that you covered previously, so your focus will be on shader implementation.

Setup

1. Please download & unpack the folder called "Additional files" needed for this lab from DLE – under Shaders introduction tab.
2. Examine the folder content: We have a file called torus which defines a set of vertices for a torus model. We also have a trianglemesh class that handles all the elements needed for drawing a mesh (buffers, drawing, deleting, etc). Lastly, we have a "drawable.h" file, which is a base class that allows the trianglemesh to be drawable.
3. Copy the files from the unpacked folder to your project (in the helper folder). After that add all these files to your project.

OpenGL (Scenebasic Uniform)

Header

1. Add 'torus.h' & 'glm.hpp' as includes in the 'scenebasic_uniform.h' file
2. The variable vaoHandle is going to be handled by the trianglemesh class, so remove it from the scenebasic_uniform.h file.
3. Remove the float variable named angle as it will become redundant
4. In the same section add a variable type Torus & a method of type void called setMatrices().

CPP

Setup

4. At the top of the ‘scenebasic_uniform.cpp’ file, make sure you have the following includes and usings:

```
#include "scenebasic_uniform.h"

#include <iostream>

using std::cerr;

using std::endl;

#include <glm/gtc/matrix_transform.hpp>

using glm::vec3;

using glm::mat4;
```

5. Modify the constructor to initialise a torus instead of the angle(0). The torus should be initialised as torus(0.7f, 0.3f, 30, 30). If you don’t understand what the torus class does, look at its source code.

Init Scene Function

The initScene() method requires modification. Delete everything except the call to the compile() method, enable the depth test using ‘glEnable(GL_DEPTH_TEST);’ & initialise the model matrix as shown in the following code:

```
model = mat4(1.0f);
```

Initialise the view matrix using the following line of code:

```
view = glm::lookAt(vec3(0.0f,0.0f,2.0f), vec3(0.0f,0.0f,0.0f), vec3(0.0f,1.0f,0.0f));
```

Initialise the projection matrix with the same format used for initialising the model matrix. Next, we need to set three uniforms for the shader named ‘Kd’, ‘Ld’ & ‘LightPosition’. See the shader format below. The ‘Kd’ & ‘Ld’ variables can be setup by referencing last week’s shader. For LightPosition however, use this code:

```
prog.setUniform("LightPosition", view * glm::vec4(5.0f,5.0f,2.0f,1.0f) );
```

The compile() method stays unchanged. If you created new shaders from scratch, change the name of the shaders. Additionally, the update() method can be left empty.

Render Function

The render() method needs a few additions. Remove everything except the code to clear the colour buffer, clear the colour & depth buffers, call the setMatrices() method & call render() on the torus.

Set Matrices Function

We need a new method called setMatrices() for the purpose of setting the values of the matrices needed in our shaders. Create our model view matrix variable with the name ‘mv’ of type mat4. Next, set the ‘ModelViewMatrix’ uniform to ‘mv’ & set the ‘NormalMatrix’ uniform as displayed in the following code:

```
prog.setUniform("NormalMatrix", glm::mat3( vec3(mv[0]), vec3(mv[1]), vec3(mv[2])));
```

Set the uniform for the model view projection matrix called ‘MVP’ and pass in the projection matrix multiplied by the model view matrix.

Resize Function

6. In the resize() function, we need to update the projection matrix with the following code:

```
glViewport(0,0,w,h);

width = w;

height = h;

projection = glm::perspective(glm::radians(70.0f), (float)w/h, 0.3f, 100.0f);
```

If there are any errors, try following the code logic and solving the errors one by one. Let’s move on to the shaders.

Shaders

Vertex

For diffuse shading, please check Lecture 2 for the formula and for a further understanding of the process. Let's analyse the structure of the vertex shader and understand what we need:

1. Our vertex shader requires two vec3 input variables called VertexPosition & VertexNormal indexed at locations 0 & 1 respectively.
2. We also require one vec3 output variable named LightIntensity.
3. Lastly, we need seven uniform variables: One vec4 named LightPosition, two vec3 variables named Kd & Ld, two mat4 variables named ModelViewMatrix & MVP & one mat3 variable named NormalMatrix.

In the main() function, we need to follow a set of steps:

4. Convert the vertex normal to eye coordinates. The result should be a vec3. Use the NormalMatrix & make sure the result is normalised

```
vec3 n = normalize( NormalMatrix * VertexNormal);
```

5. Convert the vertex position to eye coordinates. The result should be a vec4, so use the ModelViewMatrix
6. Calculate the direction from light position to the vertex point. The result should be a vec3. Make sure the resulting vector is normalised (using the key word 'normalise'). The calculation may need to be type casted
7. Implement the diffuse shading calculation. The result should be "LightIntensity". In order to do this, use the keyword 'dot' and make sure you use 'max'. Please look up 'max' in GLSL and try to understand why we need to use it. Additionally, experiment with not using max & see what happens

Fragment

1. We need one vec3 input variable named LightIntensity
2. We need one vec4 output variable named FragColor indexed to location 0
3. In the Main() function, pass LightIntensity to FragColor

Note: Make sure all the naming of the uniforms is consistent with the names used in setting the uniforms in your code, otherwise it would not work.

Optional

If you want to rotate your torus towards the camera, please use this line in initScene() under model = mat4(1.0f);

```
model = glm::rotate(model, glm::radians(-35.0f), vec3(1.0f,0.0f,0.0f));
```

This rotates the model -35 radians around the z axis… I think.. what do you think? What if I wanted to do a rotation around the y axis as well?

Phong Model

The Phong model is a combination of three different forms of lighting: Ambient, diffuse & specular lighting. Ambient lighting is a constant & omnipresent light source, diffuse lighting is directional lighting & specular lighting is meant to represent glare upon an object's surface.

You can re-use the template project and create the Phong model. Just reapply the same steps as for the diffuse model for the Phong model.

Since the Phong model is an extension of what has already been implemented, we only need to implement the diffuse & specular forms of lighting. Revisit the lecture slides and the lecture video for a refresher on the Phong model implementation.

In the OpenGL code, we need to set some extra uniforms. Think about what extra uniforms you need for the Phong model. I would recommend increasing the number of sides and number of rings in the torus to at least 50 in order to get a nice smooth look with vertex calculations.

Notably, you can use GLSL's dedicated function called "reflect" for calculating the specular lighting's reflection vector. Look it up in the documentation and use it.

Shader Functions

In GLSL functions, the parameter evaluation strategy is called by value-return (also called call by copy-restore or call by value-result). Parameter variables can be qualified with in, out, or inout. Arguments corresponding to input parameters (those qualified with in or inout) are copied into the parameter variable at call time, and output parameters (those qualified with out or inout) are copied back to the corresponding argument before the function returns. If a parameter variable does not have any of the three qualifiers, the default qualifier is in.

In your vertex shader, where you transform your normal to camera space and transform the vertex position to camera space, you can replace them with this function:

```
//get position and normal to camera space
void getCamSpaceValues ( out vec3 norm, out vec3 position )
{
        norm = normalize( NormalMatrix * VertexNormal);
        position = (ModelViewMatrix * vec4(VertexPosition,1.0)).xyz;
}
```

In your main() function, you call getCamSpaceValues() this way:

```
void main()
{
        vec3 camNorm, camPosition;
        getCamSpaceValues(camNorm, camPosition);
}
```

Flat Shading

Once you have implemented the Phong model, you can try flat shading in the shader.
Flat shading is enabled by qualifying the vertex output variable, as well as its corresponding fragment input variable, with the flat qualifier. This qualifier indicates that no interpolation of the value is to be done before it reaches the fragment shader. The value presented to the fragment shader will be the one corresponding to the result of the invocation of the vertex shader for either the first or last vertex of the polygon.

1. In your vertex shader change your LightIntensity declaration to this:

```
flat out vec3 LightIntensity;
```

2. Do the same for the declaration in the fragment shader:

```
flat in vec3 LightIntensity;
```

Extras

Further topics to explore:

1. Uniform blocks and uniform buffer objects.
    a. Look into OpenGL Superbible how to use uniform block
    b. Check out OpenGL4 Shading Language Cookbook - Working GLSL Shaders – Using uniform blocks and uniform buffer objects
    c. Check out this link to LearnOpenGL
2. Two-sided shading
    a. In depth explanation with complex shader
    b. Look into OpenGL4 Shading Language Cookbook – The Basics of GLSL shaders – Implementing two-sided shading
3. Discarding fragments in a shader
    a. OpenGL software dev kit
    b. OpenGL4 Shading Language Cookbook – The Basics of GLSL shaders – Discarding fragments to create a perforated look