Umeå University
Department of Computing Science
Computer Graphics

Stefan Johansson, stefanj@cs.umu.se

Workshop 2

Lighting and shading

Introduction

The purpose of this workshop is to implement *Gouraud shading* of a sphere using *(Blinn-)Phong lighting model*. On this workshop you can choose to either use the provided code or build upon your own code from the project (if it is functional). Under 'Resources/Workshop 2' you find a zip-file with source code which are slightly modified from Workshop 1. Some of the changes from Workshop 1 are:

`Application:`

- A new class Sphere which approximate a sphere using subdivision of a tetrahedron. The class compute the vertices, normals, and indices for the sphere. You only need to include this class (sphere.cpp and sphere.h) if you like to use your own code from the project and not have a functional object reader with vertex normals.
- The data containing the vertices are in 3D instead of 2D. Hence the second and third parameters to `glVertexAttribPointer` are changed into "`3, GL_FLOAT`".
- We use `glDrawElements`, instead of `glDrawArrays` to draw our faces. Hence, we create an `IndexBuffer` (`ELEMENT_ARRAY_BUFFER`) that contains the indices in the vertex buffer for each triangle. We allocate the buffer with `iBuffer.create()` and `iBuffer.bind()`, and copy the index data using `iBuffer.write`.
- We have put the loading of each matrix into a separate function for future reuse.
- The object (sphere) can be rotated using the space key.

`Vertex shader:`

- We now have a 3D vertex as input parameter and a uniform view matrix *V* and a uniform projection matrix *P*.

Adding Lighting information

Notice that the vertex shader is called once for each vertex, while the fragment shader is called once for each fragment, i.e., each potential pixel. If an object do not have a texture they only have color information on its vertices. For a color to be set on a single fragment we must decide from where to get that color. We have basically two approaches. We can let one vertex decide (called the provoking vertex) and that vertex's color will then be applied on the entire face, called *flat shading,* or we can let the vertex colors interpolate across the face. The latter is called *smooth* or *Gouraud shading* and will be used in this workshop.

We will also add illumination to our object. To do this we must compute the color for each vertex using the ambient, diffuse, and specular lighting information.

Adding normals

To compute the diffuse and specular lighting, the normals of the vertices are required. Lets add an in parameter to our vertex shader:

```
in vec4 vNormal;
```

Now both `vPosition` and `vNormal` should be read each time the vertex shader is called. But from where should the normal information be read? The answer is – the same buffer. This means we need to specify from where the vertices should be read and from where the corresponding normals should be read in the same buffer. The trick is quite simple. We have to add the following.

```
/* In initialize */
locNormals = program->attributeLocation("vNormal");

/* In loadGeometry */
glVertexAttribPointer(locNormals, 3, GL_FLOAT, GL_FALSE, 0,
    BUFFER_OFFSET(vSize));

glEnableVertexAttribArray(locNormals);
```

The procedure is exactly the same as with `vPosition` with one exception. We give a different starting memory address. In this example we have, for simplicity, said that the address is the first free position after the vertex data.

The other thing to do, is to actually copy the normals to the buffer. Since we now have divided the buffer into two segments of data, in "pure" OpenGL we should have been using `glBuffer`**`Sub`**`Data`, however, in Qt we can still use `write`.

```
/* In loadGeometry */
size_t vSize = sphere.sizeVertices();
size_t nSize = sphere.sizeNormals();
vBuffer.allocate(vSize+nSize);
vBuffer.write(0, sphere.verticesData(), vSize);
vBuffer.write(vSize, color.normalsData(), nSize);
```

So starting at memory address `0`, we copy `vSize` bytes of data, and starting at address `vSize` we copy `nSize` bytes of data. The methods `verticesData()` and `normalsdata()` in the Sphere class returns a pointer to the vertices and normals, respectively.

Computing the illumination

The vertex shader now have the vertex normal. In the shader, we now like to change the color of the vertex depending on light condition (Lecture 5). Check Lectures 5 (and 6) how to compute and implement the illumination in the vertex shader. You can chose to implement the Phong lighting model or the Blinn-Phong lighting model.

The last step is to pass the color information on to the fragment shader. So, we add an output parameter `color` in the vertex shader and assign the illumination information to that.

```
...
out vec4 color;

void
main()
{
    ...
    color = ambient + diffuse + specular;
}
```

In the fragment shader, we now accept the input parameter and assign an output parameter to use that color for a fragment (by default the value is interpolated). The qualifier `flat` can also be used, then the fragment color will be the one of the provoking vertex (test it). The fragment shader now looks like this.

```
in vec4 color;
out vec4 fcolor;

void main()
{
    fcolor = color;
}
```

Exercise

Implement Gouraud shading as described above, where the ambient, diffuse, and specular lights are computed in the vertex shader. If you like, you can start with a simple implementation of the illumination where you set the light parameters statically in the vertex shader and do not consider the projection or view matrix. A good set of initial values can be:

$I_a = [0.1, 0.1, 0.1]$
$I_l = [0.4, 0.4, 0.2]$
$k_a = k_d = k_s = 1.0$
$\mathbf{l} = [2.0, 0.5, 5.0, 0.0]$
$\mathbf{v} = [0.0, 0.0, 2.0, 0.0]$

Experiment with the values and do not forget to normalize.

Discuss

Which points and vectors in the vertex shader are depending on the model, view, or projection matrix?
How can the lighting parameters be passed to the shaders?