John H. Rucker

CSC4820/6820 Interactive Computer Graphics

Fall 2014

Homework 3

Due date: 11:59 pm November 25, 2014 (Saturday)

The aim of this homework is to help improve your understanding the process of lighting a 3D model. This homework is closely related to your project 3 coding.

1. Answer the following questions. Use the sample code attached to the project 3 requirements as an example. If possible, use code segments as examples to answer the questions.

* Briefly describe the steps of lighting a 3D model using OpenGL/GLSL.

The standard lighting model consists of several components: emissive, ambient, diffuse, and specular. The lighting functions will be defined in the fragment shader later. In order to perform proper lighting, all vectors and positions must be in the same “space”. Which space you want to perform lighting in is your choice but it should be consistent. In order to perform the lighting in world-space, you must transform the vertex position and vertex normal into world-space in the vertex shader and pass the world-space attributes to the fragment shader. In addition to this, you must make sure that you provide the eye position (camera’s position) and the light’s position in world-space in the fragment shader.

struct LightSourceProp {

float lightSourcePosition[4];

float diffuseLightIntensity[4];

float specularLightIntensity[4];

float ambientLightIntensity[4];

float constantAttenuation;

float linearAttenuation;

float quadraticAttenuation;

float spotlightDirection[3];

float spotlightCutoffAngle;

float cutOffExponent;

};

LightSourceProp lightSource1 = {

{ -1.0f, -2.0, -1.0f, -1.0f}, // light source position relative to the eye position original {1.0f, 1.0f, 1.0f, 1.0f},

{ 2.0f, 2.0f, 2.0f, 2.0f }, // diffuse light intensity {1.0f, 1.0f, 1.0f, 1.0f},

{4.0f, 4.0f, 4.0f, 4.0f}, // specular light intensity {1.0f, 1.0f, 1.0f, 1.0f},

{0.2f, 0.2f, 0.2f, 1.0f}, // ambient light intensity

1.0f, 0.5f, 0.1f, // constant, linear, and quadratic attenuation factors

{0.0f, 0.0f, -1.0f}, // spotlight direction

{0.0f}, // spotlight cutoff angle (in radian)

{2.0f} // spotexponent

};

Use code segments as examples if possible. How is the job divided between the OpenGL part and the GLSL part? The OpenGL part feeds information to the GLSL (vertex/fragment shader) for processing/calculations.

In a vertex shader you can write code for tasks such as: Vertex position transformation using the modelview and projection matrices, Normal transformation, and if required its normalization, Texture coordinate generation and transformation, Lighting per vertex or computing values for lighting per pixel, and Color computation

The fragment processor is where the fragment shaders run. This unit is responsible for operations like: Computing colors, and texture coordinates per pixel; Texture application; Fog computation; Computing normals if you want lighting per pixel.

GLuint materialBuffer, lightBuffer, spotLightBuffer;

// Create a uniform buffer to store the surface material properties

glGenBuffers(1, &materialBuffer);

glBindBuffer(GL\_UNIFORM\_BUFFER, materialBuffer);

glBufferData(GL\_UNIFORM\_BUFFER, sizeof(surfaceMaterial1),

(void \*)(&surfaceMaterial1), GL\_STATIC\_DRAW);

// Link the uniform buffer for material with

// the material uniform block in the shader program

glUniformBlockBinding(program, materialBlockIndex, 0);

glBindBufferBase(GL\_UNIFORM\_BUFFER, 0, materialBuffer);

/////////////////////////////////////////////////////////////////////////////////////////////

// Create a uniform buffer to store the lightsource1 properties

glGenBuffers(1, &lightBuffer);

glBindBuffer(GL\_UNIFORM\_BUFFER, lightBuffer);

glBufferData(GL\_UNIFORM\_BUFFER, sizeof(lightSource1),

(void \*)(&lightSource1), GL\_STATIC\_DRAW);

// Link the uniform buffer of light source properties with

// the lighting uniform block in the shader program

glUniformBlockBinding(program, lightBlockIndex, 1);

glBindBufferBase(GL\_UNIFORM\_BUFFER, 1, lightBuffer);

* Write down the equations for directional light and spotlight, respectively. Explain each parameter in the equation.
* // directional light source. The light position is actually the light vector.
* lightVector = lightSourcePosition.xyz;
* // For directional lights, there is no light attenuation.
* attenuation = 1.0;
* } else {
* // spotlight source
* lightVector = normalize(lightSourcePosition.xyz - v);
* float distance = length(lightSourcePosition.xyz - v);
* spotEffect = dot(normalize(spotDirection), normalize(-lightVector));
* if (spotEffect > cos(spotCutoff)) {
* // If the vertex is in the spotlight cone
* attenuation = spotEffect / (constantAttenuation + linearAttenuation \* distance +
* quadraticAttenuation \* distance \* distance);
* } else {
* // If the vertex is outside of the spotlight cone, then there is no light.
* attenuation = 0.0;
* }

Which part of the equation is varying from pixel to pixel (or from vertex to vertex)?

attenuation = spotEffect / (constantAttenuation + linearAttenuation \* distance + quadraticAttenuation \* distance \* distance);

How do you calculate the light vector? By using Lambertian Reflection. 'Lambert's cosine law . The vertex shader to implement this formula will use the lights properties, namely its position, and diffuse intensity. It will also use the materials diffuse setting. Hence to use this shader just set the light as usual in OpenGL.

if (lightType == 0) {

// point light source

lightVector = normalize(lightSourcePosition.xyz - v);

How do you calculate the view vector? vec3 E = normalize(-v); // Eye vector. We are in Eye Coordinates, so EyePos is (0,0,0) Where do the normal vectors come from? Vertex attributes and uniform varibles. A normal is a vector that defines how a surface responds to lighting, i.e. how it is lit. The amount of light reflected by a surface is proportional to the angle between the lights direction and the normal.

//---------------------------

// Lighting related variables

GLint normalID; // vertex attribute: normal

GLint mvMatrixID; // uniform variable: model-view matrix

GLint normalMatrixID; // uniform variable: normal matrix for transforming normal vectors

What happens if there is no normal vector for the object? Then lighting cannot be applied to a surface.

* What’s per-vertex lighting and per-pixel lighting? Per-vertex lighting (Gouraud shading) - computing the surface lighting for each vertex and then interpolating the vertex colors. Per-pixel lighting (Phong shading) normal vectors and positions are interpolated for each fragment and the lighting is computed in the fragment shader.

What are the pros and cons of per-vertex lighting and per-pixel lighting?

Per-vertex lighting

Pros - Per-vertex lighting has some good and some bad qualities. From a programming point of view, the complexity of using an algorithm per vertex rather than per pixel is not increased or decreased, thanks mostly to the current nature of shading technology and languages. If anything, per-vertex lighting could be faster than per-pixel lighting if the algorithm executes fewer times in a frame, because fewer vertices are being processed than pixels in per-pixel lighting, not to mention other issues such as fill rate that can affect performance.

Cons - Depending on the object’s topology, the quality of per-vertex lighting can be less than that of per-pixel lighting. Increasing lighting quality using a per-vertex approach usually requires an increase in polygon count, which can lead to performance side effects such as the need for increased and polished results. Per-pixel lighting has various extensions that allow for the simulation of lots of detail without the actual detail being present, while retaining a positive performance and frame rate.

Per-pixel lighting

Pros – more accurate, better images

Cons – Slow, not very accurate

* A 3D model file provides the position and normal vector for each vertex.

//--------------------------------------------------------------------------------------------

// Traverse the node tree in the aiScene object and draw the meshes associated with each node.

// This function is called recursively to perform a depth-first tree traversal.

void nodeTreeTraversal(const aiNode\* node) {

In per-pixel lighting, how do you get the position and normal of each pixel? Sent to the fragment shader from the C++ code

//-----------------

// Camera related variables

vec3 eyePosition = vec3(1.0f, -1.0f, 3.0f);

//---------------------------

// Lighting related variables

GLint normalID; // vertex attribute: normal

GLint mvMatrixID; // uniform variable: model-view matrix

GLint normalMatrixID; // uniform variable: normal matrix for transforming normal vectors

in vec3 N; // interpolated normal for the pixel

in vec3 v; // interpolated position for the pixel

* What is the model-view matrix used for in the vertex shader?

The model view projection matrix is often used by shaders to map from the vertices you loaded for each model to the screen.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Construct the model matrix

// Create individual translation/rotation/scaling matrices. You transform a 3D object to a

// specific location by a sequence of translations, rotations, and scalings. Each transformation

// is encoded in an elementary transformation matrix.

mat4 scaleMatrix = glm::scale(mat4(1.0f), vec3(1.0f));

mat4 translateMatrix = translate(mat4(1.0f), vec3(0.0f, 0.0f, 0.0f));

mat4 rotationMatrixY = rotate(mat4(1.0f), -45.0f, vec3(0.0f, 1.0f, 0.0f));

What is the normal matrix used for in the vertex shader?

GLint normalMatrixID; // uniform variable: normal matrix for transforming normal vectors

* What is a Uniform Buffer Object (UBO)? Refers to the OpenGL buffer object that is used to provide storage for uniforms. The term "uniform blocks" refer to the GLSL language grouping of uniforms whose storage come from buffer objects.

Why do we use Uniform Buffer Object to store uniform variables such as lighting properties or surface material properties? Uniform buffers have several uses.

Switching between uniform buffer bindings is typically faster than switching dozens of uniforms in a program. Therefore, uniform buffers can be used to quickly change between different sets of uniform data for different objects that share the same program.

Also, uniform buffer objects can typically store more data than non-buffered uniforms. So they can be used to store and access larger blocks of data than unbuffered uniform values.

// Create a uniform buffer to store the surface material properties

glGenBuffers(1, &materialBuffer);

glBindBuffer(GL\_UNIFORM\_BUFFER, materialBuffer);

glBufferData(GL\_UNIFORM\_BUFFER, sizeof(surfaceMaterial1),

(void \*)(&surfaceMaterial1), GL\_STATIC\_DRAW);

// Link the uniform buffer for material with

// the material uniform block in the shader program

glUniformBlockBinding(program, materialBlockIndex, 0);

glBindBufferBase(GL\_UNIFORM\_BUFFER, 0, materialBuffer);

What is the alternative to using UBO? Framebuffer Objects (FBOs)

What are the downsides of using UBO? The main disadvantage is the size limitation How do you connect a UBO with a uniform data block in a shader program?

// Link the uniform buffer for material with

// the material uniform block in the shader program

glUniformBlockBinding(program, materialBlockIndex, 0);

glBindBufferBase(GL\_UNIFORM\_BUFFER, 0, materialBuffer);

* If something is wrong with your lighting implementation in the GLSL shader, what are some of the debugging techniques?

1. What are the mistakes you made in the process of implementing project 2? If there are many, list the main ones.
2. What are the mistakes you made in the process of implementing project 3? If there are many, list the main ones.

Submit the report in text, Word, or PDF files to Desire2Learn under the dropbox “Homework3”. Write your name in the report.