UOW AUSTRALIA

SCIT

School of Computing & Information Technology

CSCI336 – Interactive Computer Graphics

Point Light Source

In this tutorial, we will look at implementing per-vertex/per-fragment point light sources.

Vertex Normals

Lighting computation requires surface normals to determine the orientation of a surface with respect to the light source. We can add this information for each vertex. Open the Tutorial5a project. The code in Tutorial5a.cpp defines the vertices of a cube with normal vectors:

```
Vertex g_vertices[] = {
	// Front: triangle 1
	// vertex 1
	-0.5f, 0.5f, 0.5f,   // position
	0.0f, 0.0f, 1.0f,    // normal
	// vertex 2
	-0.5f, -0.5f, 0.5f,  // position
	0.0f, 0.0f, 1.0f,    // normal
	// vertex 3
	0.5f, 0.5f, 0.5f,    // position
	0.0f, 0.0f, 1.0f,    // normal
...
...
};
```

These vertex normals will be passed to the vertex shader as attribute variables. Have a look at the code in the vertex shader: PerVertexLightingVS.vert

The normals will be passed to:

```
in vec3 aNormal;
```

Per Vertex Lighting

Two structs are declared to contain point light and material properties:

```
struct Light
{
	vec3 position;
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
};

struct Material
{
```

```
        vec3 ambient;
        vec3 diffuse;
        vec3 specular;
        float shininess;
};
```

Next, several of uniform variables are declared for the model-view-projection and model-view matrices:

```
uniform mat4 uModelViewProjectionMatrix;
uniform mat4 uModelViewMatrix;
uniform mat4 uViewMatrix;
```

and for point light and material properties:

```
uniform Light uLight;
uniform Material uMaterial;
```

These uniform variables must be passed to the shader from the application program.

In the shader's main function, the vertex position and normal are calculated in eye coordinates. To do this, the vertex position and normal are multiplied by the model and view matrices:

```
        vec3 EC_VertexPosition = (uModelViewMatrix * vec4(aPosition, 1.0)).xyz;
        vec3 EC_Normal = (uModelViewMatrix * vec4(aNormal, 0.0)).xyz;
```

Once we have these values, the four required lighting vectors for calculating Phong lighting can be calculated. Figure 1 gives a visual depiction of the vectors (note that E is used in the code instead of **v**):

```
vec3 N = normalize(EC_Normal);
vec3 L = normalize((uViewMatrix * vec4(uLight.position, 1.0)).xyz - EC_VertexPosition);
vec3 E = normalize(-EC_VertexPosition);
vec3 R = reflect(-L, N);
```

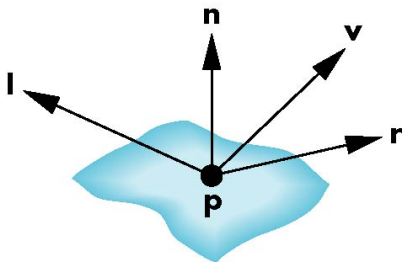

Figure 1: Vectors for Phong lighting.

The Phong lighting components for diffuse, specular and ambient are then computed:

```
        vec3 ambient  = uLight.ambient * uMaterial.ambient;
        vec3 diffuse  = uLight.diffuse * uMaterial.diffuse * max(dot(L, N), 0.0);
        vec3 specular = vec3(0.0f, 0.0f, 0.0f);
```

The specular contribution should only be calculated if the light source is not behind the surface:

```
        if(dot(L, N) > 0.0f)
```

```
        specular = uLight.specular * uMaterial.specular
               * pow(max(dot(E, R), 0.0), uMaterial.shininess);
```

Finally, the diffuse, specular and ambient are added together and passed as the output of the vertex shader that will be interpolated for each fragment:

```
vColor = diffuse + specular + ambient;
```

In the code in Tutorial5a.cpp, the values for the uniform variables must be passed to the shaders. In the init() function, their locations are determined using:

```
g_MVP_Index = glGetUniformLocation(g_shaderProgramID, "uModelViewProjectionMatrix");
g_MV_Index = glGetUniformLocation(g_shaderProgramID, "uModelViewMatrix");
g_V_Index = glGetUniformLocation(g_shaderProgramID, "uViewMatrix");

g_lightPositionIndex = glGetUniformLocation(g_shaderProgramID, "uLight.position");
g_lightAmbientIndex = glGetUniformLocation(g_shaderProgramID, "uLight.ambient");
g_lightDiffuseIndex = glGetUniformLocation(g_shaderProgramID, "uLight.diffuse");
g_lightSpecularIndex = glGetUniformLocation(g_shaderProgramID, "uLight.specular");

g_materialAmbientIndex = glGetUniformLocation(g_shaderProgramID, "uMaterial.ambient");
g_materialDiffuseIndex = glGetUniformLocation(g_shaderProgramID, "uMaterial.diffuse");
g_materialSpecularIndex = glGetUniformLocation(g_shaderProgramID, "uMaterial.specular");
g_materialShininessIndex = glGetUniformLocation(g_shaderProgramID, "uMaterial.shininess");
```

The point light and material properties are initialised as follows:

```
g_light.position = glm::vec3(10.0f, 10.0f, 10.0f);
g_light.ambient = glm::vec3(0.2f, 0.2f, 0.2f);
g_light.diffuse = glm::vec3(1.0f, 1.0f, 1.0f);
g_light.specular = glm::vec3(1.0f, 1.0f, 1.0f);

g_material.ambient = glm::vec3(1.0f, 1.0f, 1.0f);
g_material.diffuse = glm::vec3(0.2f, 0.7f, 1.0f);
g_material.specular = glm::vec3(0.2f, 0.7f, 1.0f);
g_material.shininess = 10.0f;
```

Before the model is rendered, the values for the uniform variables are passed using:

```
glUniformMatrix4fv(g_MVP_Index, 1, GL_FALSE, &MVP[0][0]);
glUniformMatrix4fv(g_MV_Index, 1, GL_FALSE, &MV[0][0]);
glUniformMatrix4fv(g_V_Index, 1, GL_FALSE, &g_viewMatrix[0][0]);

glUniform3fv(g_lightPositionIndex, 1, &g_light.position[0]);
glUniform3fv(g_lightAmbientIndex, 1, &g_light.ambient[0]);
glUniform3fv(g_lightDiffuseIndex, 1, &g_light.diffuse[0]);
glUniform3fv(g_lightSpecularIndex, 1, &g_light.specular[0]);

glUniform3fv(g_materialAmbientIndex, 1, &g_material.ambient[0]);
glUniform3fv(g_materialDiffuseIndex, 1, &g_material.diffuse[0]);
glUniform3fv(g_materialSpecularIndex, 1, &g_material.specular[0]);
glUniform1fv(g_materialShininessIndex, 1, &g_material.shininess);
```

Compile and run the program. You should see a lit rotating cube.

In the vertex shader, try commenting out individual components (i.e. ambient, diffuse and specular) and you should see the contribution of each component.

Also, try changing the light and material properties.

Loading Models with Normals

You have seen how to load a 3D mesh's vertex positions from an .obj file using the Open Asset Import Library (Assimp) from the previous tutorial. Here we will look at using the library to load vertex normals and triangle faces. Open the Tutorial5b project. The following struct is declared to store the mesh vertices and faces, along with the number of vertices and faces for a mesh.

```
typedef struct Mesh
{
	Vertex* pMeshVertices;			// pointer to mesh vertices
	GLint numberOfVertices;			// number of vertices in the mesh
	GLint* pMeshIndices;		// pointer to mesh indices
	GLint numberOfFaces;		// number of faces in the mesh
} Mesh;
```

In the load_mesh() function, when the scene is imported, the smooth normal and join identical vertices flags are set:

```
	const aiScene* pScene = aiImportFile(fileName, aiProcess_Triangulate
		| aiProcess_GenSmoothNormals | aiProcess_JoinIdenticalVertices);
```

Since we will be using the triangle faces to render the model rather than from the vertices, it makes sense to avoid having storing identical vertices and averaging the values of the normals per vertex (in the event that there are multiple normals defined for a vertex).

The following code for reading and storing the normals and faces:

```
	// if mesh contains normals
	if (pMesh->HasNormals())
	{
		// read normals and store in the array
		for (int i = 0; i < pMesh->mNumVertices; i++)
		{
			const aiVector3D* pVertexNormal = &(pMesh->mNormals[i]);

			mesh->pMeshVertices[i].normal[0] = (GLfloat)pVertexNormal->x;
			mesh->pMeshVertices[i].normal[1] = (GLfloat)pVertexNormal->y;
			mesh->pMeshVertices[i].normal[2] = (GLfloat)pVertexNormal->z;
		}
	}

	// if mesh contains faces
	if (pMesh->HasFaces())
	{
		// store number of mesh faces
		mesh->numberOfFaces = pMesh->mNumFaces;

		// allocate memory for vertices
		mesh->pMeshIndices = new GLint[pMesh->mNumFaces*3];

		// read normals and store in the array
		for (int i = 0; i < pMesh->mNumFaces; i++)
```

```
        {
                const aiFace* pFace = &(pMesh->mFaces[i]);

                mesh->pMeshIndices[i * 3] = (GLint)pFace->mIndices[0];
                mesh->pMeshIndices[i * 3 + 1] = (GLint)pFace->mIndices[1];
                mesh->pMeshIndices[i * 3 + 2] = (GLint)pFace->mIndices[2];
        }
}
```

Since we are using faces to render the model, we need to use an index buffer:

```
glGenBuffers(1, &g_IBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_IBO);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GLint)* 3 * g_mesh.numberOfFaces,
        g_mesh.pMeshIndices, GL_STATIC_DRAW);
```

bind it to the vertex array object:

```
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_IBO);
```

and to render elements using:

```
glDrawElements(GL_TRIANGLES, g_mesh.numberOfFaces*3, GL_UNSIGNED_INT, 0);
```

Compile and run the program.

The program also provides a simple GUI for rendering the model in wireframe and controlling its orientation. Rotate the sphere model, notice that the lighting does not look smooth. This is because lighting was calculated for each vertex, then the colour for each vertex was interpolated for each rendered triangle. The next section shows how to modify the shaders to perform per fragment lighting.

Per Fragment Lighting

Open the Tutorial5c project. The only difference between the code in Tutorial5b.cpp and Tutorial5c.cpp is the file names used when loading the shaders:

```
g_shaderProgramID = loadShaders("PerFragLightingVS.vert", "PerFragLightingFS.frag");
```

The difference between the programs is in the shaders.

As an overview, the Phong lighting calculation is the same. What we want to do is to perform Phong shading. To do this, we need to interpolate the normals across the triangles (before we perform the Phong lighting calculation), which is achieved by passing the four vectors to from the vertex shader to the fragment shader.

Have a look at the code in PerFragLightingVS.vert:

Instead of outputting the vertex colour, we output the four lighting vectors (which will be interpolated and passed as input to the fragment shader):

```
out vec3 vN;
out vec3 vL;
out vec3 vE;
```

```
out vec3 vR;
```

Unlike in Tutorial5b, in the vertex shader the Phong lighting calculation has been removed. We only compute the values of the four lighting vectors.

Now, look at the code in PerFragLightingFS.frag, these vectors are the input to the fragment shader after having been interpolated by the graphics pipeline:

```
// interpolated values from the vertex shaders
in vec3 vN;
in vec3 vL;
in vec3 vE;
in vec3 vR;
```

Note that we are also using the following struct and uniform variables in the vertex shader:

```
struct Light
{
	vec3 position;
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
};

// uniform input data
uniform mat4 uModelViewProjectionMatrix;
uniform mat4 uModelViewMatrix;
uniform mat4 uViewMatrix;
uniform Light uLight;
```

Uniform variables have global scope in the shaders of a shader program object. However, they have to be declared in the respective shader code where they will be used. Hence, the following is declared in the fragment shader. Note that while some of these have been removed from the vertex shader, the program will still work even if you leave the declarations in the vertex shader:

```
struct Light
{
	vec3 position;
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
};

struct Material
{
	vec3 ambient;
	vec3 diffuse;
	vec3 specular;
	float shininess;
};

// uniform input data
uniform Light uLight;
uniform Material uMaterial;
```

In the fragment shader's main() function, the interpolated vectors are first normalized:

```
    vec3 N = normalize(vN);
    vec3 L = normalize(vL);
    vec3 E = normalize(vE);
    vec3 R = normalize(vR);
```

Then Phong lighting calculations are performed using the interpolated vectors:

```
vec3 ambient  = uLight.ambient * uMaterial.ambient;
vec3 diffuse  = uLight.diffuse * uMaterial.diffuse * max(dot(L, N), 0.0);
vec3 specular = vec3(0.0f, 0.0f, 0.0f);

if(dot(L, N) > 0.0f)
    specular = uLight.specular * uMaterial.specular
       * pow(max(dot(E, R), 0.0), uMaterial.shininess);
```

Finally, the computed colour is passed as the output:

```
       fColor = diffuse + specular + ambient;
```

Compile and run the program.

The shading will appear a lot smoother than per vertex lighting, at the cost of more computation.

Note that the lighting vectors had to be normalised in the fragment shader, because the interpolation process changes the vectors. You can try uncommenting the following lines, which use the vectors without normalising them:

```
//diffuse  = uLight.diffuse * uMaterial.diffuse * max(dot(vL, vN), 0.0);
//specular = uLight.specular * uMaterial.specular
       * pow(max(dot(vE, vR), 0.0), uMaterial.shininess);
```

If you compile and run the program with the above lines uncommented, you may be able to see that the lighting looks different.

Multiple Lights

This section shows an example of how to combine the contributions of multiple lights. For each light source, we can simply add the contribution of that light to the colour of the fragment. Open the Tutorial6d project.

This program will implement two point light sources. In the fragment shader:

```
#define MAX_LIGHTS 2
```

and

```
uniform Light uLight[MAX_LIGHTS];
```

The lighting calculation is placed in a function. The light's index and surface normal are passed to this function:

```
vec3 calculateLighting(int lightIndex, vec3 N)
{
	...
}
```
This function calculates lighting using the Blinn-Phong model and returns the colour contribution of one light source. We simply have to accumulate the contributions of each light source. In the main function:

```
	vec3 colour = vec3(0.0, 0.0, 0.0);


	for(int i = 0; i < MAX_LIGHTS; i++)
	{
		colour += calculateLighting(i, N);
	}

	fColor = colour;
```

The code in Tutorial6d.cpp has to change to mirror the changes in the shader code. The following are snippets of the changes:

```
#define MAX_LIGHTS 2
...
Light g_light[MAX_LIGHTS];		// light properties
...
GLuint g_lightPositionIndex[MAX_LIGHTS];
GLuint g_lightDirectionIndex[MAX_LIGHTS];
GLuint g_lightAmbientIndex[MAX_LIGHTS];
GLuint g_lightDiffuseIndex[MAX_LIGHTS];
GLuint g_lightSpecularIndex[MAX_LIGHTS];
GLuint g_lightTypeIndex[MAX_LIGHTS];
```

In the init() function, the variable locations are determined for each light source:

```
g_lightPositionIndex[0] = glGetUniformLocation(g_shaderProgramID, "uLight[0].position");
...
g_lightPositionIndex[1] = glGetUniformLocation(g_shaderProgramID, "uLight[1].position");
...
```

The properties of each light source are also initialised:

```
g_light[0].position = glm::vec3(5.0f, 5.0f, 5.0f);
...
g_light[1].position = glm::vec3(-5.0f, -5.0f, 5.0f);
...
```

Before rendering, each light source's properties are set:

```
	for (int i = 0; i < MAX_LIGHTS; i++)
	{
		glUniform3fv(g_lightPositionIndex[i], 1, &g_light[i].position[0]);
		glUniform3fv(g_lightAmbientIndex[i], 1, &g_light[i].ambient[0]);
		glUniform3fv(g_lightDiffuseIndex[i], 1, &g_light[i].diffuse[0]);
		glUniform3fv(g_lightSpecularIndex[i], 1, &g_light[i].specular[0]);
		glUniform1i(g_lightTypeIndex[i], g_light[i].type);
	}
```

Compile and run the code. The tweakbar allows you to move the position of the lights.

You can also examine each individual light's contribution, by commenting and uncommenting the respective lines in the fragment shader:

```
//for(int i = 0; i < 1; i++)
//for(int i = 1; i < MAX_LIGHTS; i++)
```

Things to try

You should now understand how to add a point light source to a scene. Try modifying the program to do the following:

- Flat shading
- Implement attenuation
- Allow the user to dynamically adjust light and material properties while the program is running
- Construct a room with lighting

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
- http://www.opengl-tutorial.org/
- http://ogldev.atspace.co.uk/index.html