15-462 Project 2: Texture Mapping and Shaders

Release Date: Tuesday, February 10, 2009

Due Date, Tuesday, February 24, 2009 at 23:59:59

1 Overview

In this project, you will learn to do texture mapping in OpenGL and implement bump mapping, environment mapping and Fresnel effect in OpenGL Shading Language (GLSL).

Before you start, you will need to download the staff code from the course website and update your project 1 code according to section 4 of this document.

Starter Code: `http://www.cs.cmu.edu/~15462/proj/02/p2.tar.gz`

You may also find the following documents very useful for GLSL programming:

GLSL Quick Reference Guide:
`http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf`
GLSL Specification:
`http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf`
OpenGL Reference Pages:
`http://www.opengl.org/sdk/docs/man/`

2 Submission Process

Your handin directory may be found at
`/afs/cs.cmu.edu/academic/class/15462-s09-users/`*andrewid*`/p2/`.
You should submit the entire contents of the handout, your modified files, and any screenshots in this directory.

Regardless of what machine you use to code your assignment, it **must** compile and run correctly on the machines in the Wean 5336 cluster. There are a lot of students in the course and only 25 machines, so you probably don't want to wait until the last minute to test it.

To test your code, we will simply `cd` into the directory, then run `make clean; make`. Therefore, your code should should be placed in the root directory uncompressed (i.e. the makefile's absolute path should be `.../15462-s09-users/`*andrewid*`/p2/Makefile`.).

In addition, you must fill out the `p2.txt` file in the handout, describing which features you have implemented for project 2, and which you have not. In particular, if you have missing features, be sure to explain how much you accomplished and why, and if you did extra work, be sure to explain what that was and how you did it. Furthermore, you should detail anything you think we need to know to understand your code.

3 Tasks

3.1 Required Tasks

- Texture map spheres and triangles the same as the staff scene (30 points)
- Implement bump mapping shader that utilizes normal map (30 points)
- Implement Fresnel shader with Sphere Mapping (30 points)
- Have good coding style and well-commented code (10 percent)

Please see the relevant sections for specific requirements for each shader. Also, note that commenting your shader code is extremely important.

3.2 Extra Credit

Extra credit ideas:

- Replace sphere mapping with cube mapping.
- Add a background to the scene.
- Implement your own crazy shaders.

4 Files to Update/Edit

The staff code for project 2 includes the following files:

- `effect.cpp`
- `effect.h`
- `staffldr.cpp`
- `shaders/bump_vert.glsl`
- `shaders/bump_frag.glsl`
- `shaders/fresnel_vert.glsl`
- `shaders/fresnel_frag.glsl`

To get started, you need to copy them to your project 1 folder and overwrite existing files of the same name. Also, if you are on Windows, don't forget to change your `.vcproj` to include `effect.cpp`, otherwise your compiler will prompt linking errors.

You're free to modify any file as long as there are no warnings in that file that tells you not to do it. For this project, most of your work will probably happen in the following files, depending on how modularized your code is:

- `effect.h/.cpp`: define shaders' OpenGL interface
- `geom/*.h/.cpp`: modify to use shaders and textures
- `project.h/.cpp`: initialize shaders/textures
- `shaders/*.glsl`: shader implementations

The names of the shader files correspond to the effects. So, for example, `bump_vert.glsl` is the vertex shader for bump mapping.

5 Texture Mapping

OpenGL texture mapping is covered in lecture 7 (http://www.cs.cmu.edu/~15462/lec/07/lec07.pdf). Your task is to add texture mapping to both the `Triangle` and `Sphere` classes. The `Material` class defines which (if any) texture to use. A material has a texture if the `Material::texture_name` member is a non-empty string.

5.1 Loading OpenGL Texture Objects

Both the `Material` and `SphereMap` classes have existing functionality that loads textures into memory. However, to use such textures, you will need to edit the `load_texture` functions to create an OpenGL texture object from said loaded texture. Consult the Red Book chapter on textures for more information. Textures are loaded as 32-bit RGBA arrays (each color is 4 bytes: the red, blue, green, and alpha components, in order. `color_array_to_vector` shows how to convert from the array to a `Vec4`).

5.2 Sphere Texture Mapping

While triangle texture mapping is straightforward (the `Triangle` class gives you the texture coordinates of each vertex), spheres are slightly more complex. If for sphere drawing you used `glutSolidSphere`, in project 1, then you will have rewrite the drawing code to accomplish texture mapping since `glutSolidSphere` does not define texture coordinates. Scene 0 is designed specifically to test sphere texture mapping.

We texture map spheres by latitude and longitude. The first texture coordinate, u is based on the longitude, and the second, v, is based on the latitude.

You can also compute the vertices and normals based on latitude/longitude, and so you can generate all the values for a sphere by iterating over the longitude and latitude.

Given a longitude $\theta \in [0, 2\pi]$, latitude $\phi \in [0, \pi]$, and radius r, we can compute the vertex $\vec{v}$, normal $\vec{n}$, and texture coordinates $\vec{c} = [u, v]^T$ by

$$\vec{n} \quad = \quad [\sin(\phi)\cos(\theta), -\cos(\phi), \sin(\phi)\sin(\theta)]^T$$

$$\begin{aligned}\vec{v} &= r\vec{n} \\ \vec{c} &= \left[\frac{\theta}{2\pi}, \frac{\phi}{\pi}\right]^T\end{aligned}$$

6 Bump Mapping

Bump mapping is a commonly used hack in computer graphics. It creates the illusion of bumping effect on a surface that originally lacks details. It is essentially a mutli-texturing technique. We need two textures to do bump mapping, a diffuse map, and a normal map. The diffuse map is just a regular texture, which defines color, while the normal map is usually generated from the diffuse map by software and defines the normals for each pixel in the diffuse map. Each pixel(stored as RGBA value) in the normal map defines a normal in tangent space, with R as the tangent coordinate , G the binormal coordinate, and B the normal coordinate. Normal maps tend to have a blue hue, because most of the normals will be pointing out from the surface.

The general idea behind bump mapping is that, instead of using the interpolated normals, we use the normals stored in the normal map to do lighting computation. For example, we want to map the picture of a wall onto a triangle. Without any extra processing, the texture mapped triangle will appear very flat, but if we have a normal map that contains the normals of a real, uneven wall for every point in the wall texture, and compute colors according to these normals, the surface will look much better. We can use an nVidia Photoshop plugin to construct normal maps from any texture. But as we mentioned earlier, normals are represented in tangent space. So the trick is to transform vectors from eye/world space to tangent space or back.

We know that any vector in 3D space can be written as a linear combination of three independent vectors (basis vectors). Often they are the identity vectors $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$. But we can use other basis vectors as well, as long as they are all independent vectors.

Tangent space is the coordinate system that resides on top of a surface. Its basis vectors are the tangent T, the binormal B, and the normal N. T and B both lie on the surface, so it's easy to see that any point on the surface is a linear combination of T and B. If we know T and B in world space, we can find the world coordinates of any vector, given its T B representation. For example, if we want to find the point p which has uv coordinates $s = (0.3, 0.5)$, and we know that T and B are $(\frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$ and $(\frac{-1}{\sqrt{3}}, \frac{1}{\sqrt{3}}, \frac{1}{\sqrt{3}})$, then we can express this point in world coordinates as

$$p = 0.3 \times T + (-0.5) \times B$$

we use -0.5 instead of 0.5 here because uv coordinates are left-handed while T, B coordinates are right-handed. So we want to reverse the v-axis (this is actually a scaling transformation).

Similarly, assume we have an arbitrary surface that contain three vertices v_1, v_2, and v_3, and their corresponding uv coordinates c_1, c_2 and c_3, all in counterclockwise order. We refer to the vector from v_1 to v_2 as $\vec{v}_{12}$, and the vector from c_1 to c_2 as $\vec{c}_{12}$. We refer to the two components of $\vec{c}_{12}$ as $\vec{c}_{12}^{u}$ and $\vec{c}_{12}^{v}$. Just like the example above, any vector on this surface can be rewritten as a linear combination of T and B. So we have

$$\begin{aligned}\vec{v}_{12} &= \vec{c}_{12_u} \times T + (-\vec{c}_{12_v}) \times B \\ \vec{v}_{13} &= \vec{c}_{13_u} \times T + (-\vec{c}_{13_v}) \times B\end{aligned}$$

Rewrite this in matrix form, we have

$$M = \begin{bmatrix} \vec{c}_{12_u} & -\vec{c}_{12_v} \\ \vec{c}_{13_u} & -\vec{c}_{13_v} \end{bmatrix}$$

$$\begin{bmatrix} \vec{v}_{12} \\ \vec{v}_{13} \end{bmatrix} = M \begin{bmatrix} T \\ B \end{bmatrix}$$

To solve T and B, we have

$$\begin{bmatrix} T \\ B \end{bmatrix} = M^{-1} \begin{bmatrix} \vec{v}_{12} \\ \vec{v}_{13} \end{bmatrix}$$

If we expand M^{-1}, it's simply

$$\begin{bmatrix} T \\ B \end{bmatrix} = \frac{1}{|M|} \begin{bmatrix} -\vec{c}_{13_v} & \vec{c}_{12_v} \\ -\vec{c}_{13_u} & \vec{c}_{12_u} \end{bmatrix} \begin{bmatrix} \vec{v}_{12} \\ \vec{v}_{13} \end{bmatrix}$$

To transform a vector from tangent space to eye space, you need to multiply it by the matrix $A = [TBN]$ (i.e. T, B, and N are the columns of the matrix), and to go from eye space to tangent space, you need A^{-1}, but since T, B and N are orthonormal vectors, $A^{-1} = A^T$.

6.1 Requirements

Your bump mapping shader should

- Support per-pixel Phong shading based on the first light (`GL_LIGHT0`) in the scene.
- Bump map `Triangle` objects in the pool scene (scene 1).

6.2 Recommended Steps

Here are the recommended steps for you to implement the bump mapping shader:

1. Write a shader that can access two textures.
2. In the vertex shader, transform useful vectors to tangent space.
3. Implement per-pixel Phong shading using OpenGL `GL_LIGHT0` as the light source and `gl_FrontMaterial` as the material.
4. Implement bump mapping.

6.3 Mathematics

You should use the following equation to compute the final fragment color, c:

$$c = (v_a b + v_d b + v_s)a$$

where

v_d/v_s are the regular Phong shading diffuse and specular terms except that the coefficients you use should be the product of the corresponding material values and light values. For example, the diffuse coefficient `v_d = material.diffuse * light0.diffuse`

v_a is the ambient term. This is provided by the scene.

b is the texture color.

a is an optional attenuation factor of your light. The `Light` class does not have an attenuation member, so you can use anything for this (1 for no attenuation).

Note: *Do not* forget to transform your vectors to tangent space before doing the lighting computation.

7 Fresnel Effect and Environment Mapping

7.1 Environment Mapping

Environment mapping is a commonly used technique in computer graphics to provide global lighting. Basically you provide a texture that describes the "sky" for a scene, and query this texture based on surface normals. This is very useful when creating relatively static light conditions, like traveling in space, when primary light sources are in far distance. The tricky part is to define the function that maps vectors from 3D space to 2D uv coordinates.

Usually we use two types of environment maps: sphere maps and cube maps. In this assignment we will only have to deal with sphere mapping.

Sphere mapping, as its name indicates, uses a spherical texture as the environment map. Here we provide a common scheme that maps 3d vectors to uv coordinates:

$$\begin{aligned} p &= 2\sqrt{x^2 + y^2 + (z+1)^2} \\ u &= x/p + 0.5 \\ v &= y/p + 0.5 \end{aligned}$$

where x, y, z are the components of the reflected eye vector, and u, v are the coordinates to sample on the texture. To see how this makes sense, try a few corner cases, or consult a TA. The software implementation of sphere mapping is provided for you in `effect.cpp` via the `SphereMap::get_texture_color` function. For project 2, you need to use OpenGL to provide the implementation.

Because one sphere map describes only half of the lighting in a scene, ideally you should use two sphere maps. But this is not required in this project, so you can get away with using only one sphere map, and in the scene we set up, the difference is barely visible. There are a lot of royalty-free sphere maps online, for example, you can get plenty here:
http://www.codemonsters.de/home/content.php?show=spheremaps

7.2 Fresnel Effect

Fresnel effect is the observation that the amount of reflectance you see on a surface depends on the viewing angle (like the specular term in Phong shading). In this part, you will implement Fresnel effect for the water surface, using Schlick approximation. This is described on page 214 of the Shirley textbook.

7.3 Putting It All Together

In general, the value (Fresnel coefficient) you get from Schlick approximation is used as a specular term. So it only affects specular highlights. This makes sense if we're rendering the surface of deep water, because the water appears to have a relatively uniform blue, and sun lights only adds to highlights on the waves. Here is the equation that approximates the final color, c:

$$c = c_e R + c_d(1 - R)$$

where

R is the Fresnel coefficient.
c_e is the environment map color, which you get from the environment map.
c_d is the diffuse value of the material for the water surface.

7.4 Requirements

Your Fresnel shader should

- Use the sphere map given in the constructor as the environment map.
- Use the environment map as the lighting condition, ignore lights in the scene
- Use the Fresnel equations described in page 214 of the Shirley textbook, though you may tune the parameters to your own need.
- Expose a uniform variable as the diffuse color of the water surface. This should be set to the diffuse value of the water surface's material.

8 New Classes

8.1 The Effect Class

Each shader should have a corresponding class defined in `effect.h`. All shader classes must inherit the Effect class. There are stubs in `effect.h/.cpp` that already have the infrastructure in place. Do not change the signatures of already defined functions because the staff scene need them to compile correctly.

You should finish the implementations by adding class members and class functions that create and use GLSL programs (defined in `shaders/*.glsl`) to achieve the given effects.

8.2 The EnvironmentMap and SphereMap Classes

Also in `effect.h`, the `EnvironmentMap` class is defined as an abstraction of an environment map. We provide one concrete partial implementation, `SphereMap`, which uses the mathematics described in section 7.1. You must finish the implementation by creating an OpenGL texture object to store the sphere map.

9 Accessing Textures in GLSL

Here is a simple tutorial to help you get started on the bump mapping shader. It outlines the basic steps you need to take to set uniform variables(attributes are similar) and access textures. This is not meant to be comprehensive. You could refer to online tutorials, OpenGL(R) Shading Language (2nd Edition), GLSL Language Specification or the lecture slides for more information. The language spec has a lot of useful information. At least you should read the section on keywords like uniform, attribute, and varying

So the first step to bump mapping is to provide textures to your shaders. To access multiple textures in GLSL, you need to bind your textures to different texture units, and let the shaders know the bindings. You need to do a couple things for this to happen.

First, you need to bind your textures. There are at least two texture units in OpenGL, and usually a lot more, depending the graphics card you're using. In the default OpenGL environment, you can only operate on one texture unit. This will limit your texture mapping to only one texture. But modern graphics card usually support eight or more, which allows techniques like bump mapping. Binding a texture to a specific texture unit is easy. All you need to do is call `glActiveTexture(GL_TEXTUREi)` before the binding. The `i` here could any number from 0 to a predefined constant(max number of texture units - 1). Here is what it should look like in your program.

```
draw()
{
    ...
    // use texture unit 0 as the current texture unit
    glActiveTexture(GL_TEXTURE0);
    // assign gl_tex0 to the current texture unit
    glBindTexture(GL_TEXTURE_2D, gl_tex0);

    // use texture unit 1 as the current texture unit
    glActiveTexture(GL_TEXTURE1);
    // assign gl_tex1 to the current texture unit
    glBindTexture(GL_TEXTURE_2D, gl_tex1):
    ...
}
```

After binding the textures, you need to inform the shaders where to look for them. This is done through a special type of GLSL primitive called `sampler`. To access 2D textures, you need to declare variables as `uniform sampler2D`. Putting this in code, this is what your bump mapping shader might look like:

```
// fragment shader
uniform sampler2D diffusemap;
uniform sampler2D normalmap;

void main(void)
{
    blah...
    // sample the texture unit given by diffusemap at point uv_coords
    vec4 color = texture2D(diffusemap, uv_coords);
    blah...
}
```

Each sampler variable should store the texture unit index you want it to sample from. For example, if you bind a diffuse texutre to `GL_TEXTURE0`, and you want the sampler `diffusemap` to use this texture, you need to set `diffusemap` to 0. To do this, you need to get the handle(or location) of the sampler variable, before you start drawing and after you load the shaders. Here is some code that does that:

```
load
{
    ...
    // switch to the appropriate shader first
    glUseProgramObjectARB(program);
    // get the handle of diffusemap
    diffusemap_handle = glGetUniform1i(program, "diffusemap");
    // set diffusemap to access texture unit 0
    glUniform1iARB(diffusemap_handle, 0);
    // swtich back to the default OpenGL shader
    glUseProgramObjectARB(0);
    ...
}
```

Some functions have variations that work for other GLSL primitives, like `glUniform*`, `glGetUniform*`, ext. You should always look up the functions you want to use in the official manuals(GLSL language spec and OpenGL reference pages), and make sure you understand the parameters you're passing them.

10 Before You Start

Writing shaders is a very experimental and iterative process, and very much fun, too. Invent your own shaders and try experimenting with different schemes. You'll discover that changing a couple lines of code may result in a totally new effect (turning water into mercury maybe?). You may not get exactly the effect as in the reference shot. That is okay, as long as your shaders meet all the requirements listed above. We won't take off points just because you use a different constant somewhere or take a higher power of a cosine term.

However, it should be entirely obvious that your shaders are doing the right thing.

Document your code very well, especially when you're writing GLSL. It must be made clear that you really understand the concepts involved. Furthermore, reading GLSL is often painful because it has very little abstraction and people tend you use short variable names. So try to avoid that, and, really, document your code. Make sure you start early.

Good Luck and Have Fun.