Assignment 5: Texturing

CS 148 Autumn 2016-2017

Due Date: Monday, October 31th, 2016 by 7pm

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to CA’s on Piazza or come to office hours. Start early!

Introduction

In this assignment we will briefly explore texture mapping and what you need to do to be able to use textures in your scene. This assignment is lighter than last week’s in terms of coding, but we recommend you spend time finding/creating nice textures for your scene! Note that for this assignment, you are free to use the Blinn-Phong shader or the Epic Shader you wrote last week. If you decide to use the Epic Shader, you will have to adjust it to support textures.

UV Mapping

UV mapping is the process of assigning a UV coordinate to each vertex of the mesh. As mentioned in class, this is what is used to sample the texture. UV coordinates do not usually magically appear on meshes though, many times, you will have to create your own UV coordinates. If you look in your OBJ file and see lines that begin with “vt”, you can ignore this as your mesh comes with texture coordinates already. For other meshes, you will have to generate your UV coordinates using a 3d modeling package like Maya or Blender.

This tutorial for Maya may be useful: [here](#) in addition to this tutorial: [here](#). Alternatively, if you use Blender: [here](#).

Colors

Now assuming that your model has UV coordinates, you will want to create a texture for it. In Maya, you can paint a texture onto your mesh after creating a UV map for it. This tutorial does a nice job explaining how you can go about doing that. This tutorial covers the equivalent for Blender (it also covers UV Mapping again).

Alternatively, you can save your UV map as an image in Maya or Blender. Afterwards, you can insert that UV map as a layer in Photoshop or GIMP (or whatever your image editing tool of choice is) and create your own texture that way.
Getting Started

Setup

Go to main.cpp and change the line that says:

```cpp
#define APPLICATION Assignment4
```

to

```cpp
#define APPLICATION Assignment5
```

The scene will be constructed inside “Assignment5.cpp” instead of “Assignment4.cpp.”

Coding

Blinn-Phong

There should be very minimal coding in this assignment for applying a texture to your objects. If you look in Assignment5.cpp, the relevant lines are:

```cpp
shader->SetTexture(BlinnPhongShader::TextureSlots::DIFFUSE,
    TextureLoader::LoadTexture("brick/bricktexture.jpg");
shader->SetTexture(BlinnPhongShader::TextureSlots::SPECULAR,
    TextureLoader::LoadTexture("brick/bricktexture.jpg");
```

The call to SetTexture does two things:

1. Loads the texture using the TextureLoader namespace. Just like the mesh loader, the path specified is relative to the “assets” folder.
2. Tells the shader program which texture it is (i.e. TextureSlots::DIFFUSE). The integer value of this enum doubles as the texture unit to use.

If you want more information about the OpenGL behind the scenes, feel free to ask on Piazza/ask in office hours. If you want more information about what the assignment framework is doing, read the Epic BRDF section. The framework, by default, allows you to use textures for the diffuse/specular color for use in Blinn-Phong shading. Additionally, it also supports normal mapping and displacement mapping in the Blinn-Phong shader. Finally, it supports cube map textures by loading up the images for the 6 faces of the cube. Examples of all of the above can be found in the default scene in Assignment 5.

Cool Things to Do

There are many cool things that you can do to further enhance your image. Just to name a few and to give you a few pointers for getting started:

Epic BRDF

To implement textures for the Epic BRDF, you will have to modify your Epic BRDF to have the texture functionality that the Blinn-Phong shader already has. This section provides a rough idea on how to do so. For the rest of this section, I will assume that you named your C++ class “EpicShader” and have files named “EpicShader.h” and “EpicShader.cpp.” For all the steps below you can view the code in the file’s
Blinn-Phong shader equivalent to get started. After you finish these steps, you will be able to use instructions for Blinn-Phong for the Epic BRDF.

**C++ Modifications**

Chances are that if you copy and pasted the BlinnPhongShader class, you will already have many of these modifications.

1. In BlinnPhongShader.h, there exists a TextureSlots structure with an enum in it. Copy this to your EpicShader.h. Remove specular.
2. In BlinnPhongShader.h, copy over the SetTexture function declaration to EpicShader.h.
3. In BlinnPhongShader.cpp, copy over the SetTexture function definition to EpicShader.cpp (rename BlinnPhongShader to EpicShader).
4. In BlinnPhongShader.cpp, copy the contents of SetupShaderMaterials to the SetupShaderMaterials in EpicShader.cpp. Remove everything that has to do with the specular texture.
5. In BlinnPhongShader.cpp, copy the SetShaderUniform line relating to the diffuseTexture in the UpdateMaterialBlock function to the UpdateMaterialBlock function in EpicShader.cpp.

**GLSL Modifications**

Here, I will be referencing the GLSL shaders found in source/shaders/brdf/blinnphong and I will assume that your Epic Shader code is found in epic.vert and epic.frag.

1. In epic.vert, remove the line that says “fragmentColor = vertexColor;” as well as the ”out vec4 fragmentColor;” line.
2. In blinnphong.vert, copy the “out vec2 fragmentUV;” and “fragmentUV = vertexUV;” lines to epic.vert.
3. In epic.frag, remove the line that says “in vec4 fragmentColor;” and replace it with a line that says “in vec2 fragmentUV;”
4. In blinnphong.frag, copy the “uniform sampler2D diffuseTexture;” to epic.frag.
5. In epic.frag, replace all places where you otherwise used fragmentColor to be “texture(diffuseTexture, fragmentUV)”

Note in all of the instructions above, feel free to rename the misnomer of diffuseTexture to something else.

**Shadow Mapping**

The simplest version of shadow mapping is doing shadow mapping for directional lights. Essentially, what you do is:

1. Render the scene from the viewpoint of the camera and write depth information to a texture (using a custom shader).
2. Pass this depth information to the actual shader (Blinn-Phong, Epic, etc.) and calculate the distance you are from the light.
3. If you are farther than the depth stored in the texture, do not contribute any color. If you are as close (i.e you are the object that the light hit), proceed to color yourself as per normal.

This is obviously a simplification. You will have to do some research to get this working properly! You can see tutorials for how to do this [here](#) and [here](#).

Point lights are a little trickier as light can emanate towards every direction. Essentially, the crucial difference is that instead of rendering depth information to one texture, you will have to render depth information to 6 textures (a cube texture). You can see a tutorial for this [here](#). The above link to opengl-tutorial.org also touches on point lights near the end.

Finally, you would not perform shadow mapping for hemisphere lights since that is meant to simulate indirect diffuse lighting. One last thing to note, shadow mapping can become expensive very quickly. As a result,
you probably only want to enable a few lights to cast shadows. If you want to see more advanced shadow mapping techniques, look at this Crytek Presentation.

**Image Based Lighting**

Image based lighting takes the concept of environment mapping and uses it to perform lighting calculations! An environment map is a 6 sided texture (cube) and you can sample from that texture to get the color you want to use. In the previous assignment, you may have noticed that if you set metallic to 1.0 then the only thing you will see is a specular dot. But this is not right. For example, gold in the real world would be considered as having a metallic property of 1.0, but you would see more than just a specular dot! The reason for this is that there is lighting coming from the environment and the way we emulate this is usually with an environment map. Image base lighting takes this concept a step further and says that we want to use this environment to also light the scene! This way you can use real-world light data to light your virtual scenes. There are two parts to this: specular and diffuse. I would suggest implementing specular first before diffuse as diffuse becomes a lot more complicated.

You can learn how to use a cube textures here. You can also read more about implementing image based lighting here.

You can see the math on to properly do image based lighting with Epic's BRDF and material in their SIGGRAPH 2013 course here. This is the same presentation that I linked last week.

**Resources**

Now the questions is where to find environment maps. In particular, you want HDR environment maps so that you can get accurate lighting data. There are many free databases out there that contain HDR panoramas:

- Open Footage
- USC
- HDR Labs

Then you will want to convert these panoramas instead cube textures. One way you can do this is to load up your favorite modeling package, create a sphere, use the panorama image as a texture, put a camera inside the sphere and render the scene 6 times: once in the up direction, then to the right, left, down, forward, and backwards. At that point you’ll have textures for all six sides of the cube! You can Google around for some automated techniques.

**Grading**

This assignment will be graded on the following requirements:

- At least two objects in YOUR scene (not the example scene) are textured.
- You use at least two unique textures in your scene (not the example scene).
- At least one your own models (one that you made) is textured.

according to the following rubric:

- 4 – At least two models (including one that you made) textured.
- 2 – One model textured.
- 1 – Overly simplistic textures.
- 0 – Nothing handed in.
Your textures should not be a single color—that would be an overly simplistic texture.