ASSIGNMENT 4: SHADING AND LIGHTING

CS 148 Autumn 2017-2018

Due Date: Monday, October 23rd, 2017 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! This assignment is more coding intensive than the others. FINALLY, please SEARCH Piazza before asking questions. Chances are someone already asked the same question. READ THIS DOCUMENT THOROUGHLY.

![Figure 1: Screenshot from the Unreal Engine 4 Infiltrator Demo (from Epic’s SIGGRAPH 2013 presentation).](image)

1 Introduction

This week, we will explore more advanced BRDFs, materials, and lighting types. As you might imagine, Blinn-Phong and simple point lights simply do not look realistic enough. As a result, many companies (i.e. Epic, Crytek, EA DICE, Disney, etc.) have been moving towards physically based shading and lighting. We will not fully explore everything they do in this assignment, but we will give you a brief overview! Should you be interested in this topic, you are more than welcome to explore it more yourself. There are three parts towards achieving more realistic shading:

- BRDF
- Material
- Lighting

In this assignment, we will explore what Epic Games does for their BRDF and Material in their Unreal Engine 4 as well as some new types of lights. Additionally, you may find it very useful to actually look at
Unreal Engine’s source code. Follow the instructions here to get access to it. Namely the file: Engine/Shaders/Private/BRDF.ush as well as Engine/Shaders/Private/ShadingModels.ush. The most relevant functions to look at are D_GGX, Vis_Schlick, F_Schlick in BRDF.ush. You can trace this function back farther if desired. Note that these files do not implement the exact equations we provide in the homework, so please don’t let these files confuse you! However, the files are similar enough that if you understand what’s going on in this homework, you should feel confident in understanding how this important component of a real-world game engine works! (Note: it is not required to look at these files; it’s just if you’re curious, and provides more context for this assignment.)

2 Background

The bulk of the information here can be found in Epic Games’s 2013 SIGGRAPH presentation which you can see here. You may find it to be a useful resource. The way that the Unreal Engine performs its rendering causes it to be very inefficient to use more than one material type/shading type (they used a deferred renderer). As a result, it is extremely beneficial for them to have a “one-size-fits-all” BRDF which can represent all types of materials well.

There are a couple variables that I will be using throughout this section:

- $N$ - the normal vector of a vertex
- $V$ - the normalized vector from the vertex to the camera
- $L$ - the normalized vector from the vertex to the light
- $H = \frac{L + V}{\|L + V\|}$
- $c_{diff}$ - the diffuse color of the material
- $c_{spec}$ - the specular color of the material
- $c_{light}$ - the color of the light
- $x_{vert}$ - vertex position

2.1 Material

Epic Games uses three tunable parameters:

- $r$ - roughness
- $s_c$ - specular
- $m$ - metallic

Additionally, throughout their equations, they use $\alpha = r^2$. You may wonder, then, how $c_{diff}$ and $c_{spec}$ are calculated. Their material has another parameter, the base color which is just some color passed to the material (in our case, for this week, that will just be some vertex color). Let’s call it $c_{base}$. Then:

$$c_{diff} = (1 - m)c_{base}$$
$$c_{spec} = \text{lerp}(0.08s_c, c_{base}, m)$$
where lerp is a linear interpolation from the first term to the second term using the third term as the interpolation value (in GLSL, this function is called mix).

2.2 BRDF

As in Blinn-Phong, there are two parts to a BRDF: the diffuse component and the specular component.

2.2.1 Diffuse

In Epic’s BRDF, they use Lambertian diffuse. In other words, given the diffuse color of the material, the final diffuse color is:

\[ d = \frac{c_{\text{diff}}}{\pi} \]

Note that the dot product between the normal and the light direction will be accounted for later.

2.2.2 Specular

Where the BRDF starts to differentiate itself, however, is in the specular term. Epic Games uses the Cook-Torrance Microface model:

\[ s = \frac{DFG}{4(N \cdot L)(N \cdot V)} \]

You can read more about these terms in the presentation, but all you have to know is the math behind the calculations:

\[
D = \frac{\alpha^2}{\pi((N \cdot H)^2(\alpha^2 - 1) + 1)^2} \\
k = \frac{(r + 1)^2}{8} \\
G_1(v) = \frac{N \cdot v}{(N \cdot v)(1 - k) + k} \\
G = G_1(L)G_1(V) \\
F = c_{\text{spec}} + (1 - c_{\text{spec}})2(-5.55473(V \cdot H) - 6.98316)(V \cdot H)
\]

A couple things to be wary of:

- \( F \) is a vector! Not a scalar like the others.
- In \( G_1(v) \), \( v \) is the input to the function and is not the same as \( V \) which is the vector from the vertex to the camera.
2.2.3 Final Color

Then to put it all together, the final color is:

\[ c_{\text{final}} = c_{\text{light}}(N \cdot L)(d + s) \]

2.3 Lighting

So in this assignment, we will be implementing three types of lights: point lights, directional lights, and hemisphere lights. These lights are fairly simple. If you decide to go a step further and implement area lights or image-based lighting, you would have to implement sampling.

2.4 Point Light

A point light a source of light that we say exists at only one point in space. Point lights are fairly simple and you can already see an example of them in action in the Blinn Phong shader in “source/shaders/brdf/blinnphong/frag.” For a point light, the only thing you need to do is pass the light position: \( l_{\text{pos}} \) and the input light color \( c_{\text{light}} \) to the light shader. Then, you can calculate:

\[ L = \frac{l_{\text{pos}} - x_{\text{vert}}}{||l_{\text{pos}} - x_{\text{vert}}||} \]

And then shade using the BRDF equations as usual.

2.5 Directional Light

Directional lights can be thought of like a point light that is infinitely far away from you, as a result, all the lights are pointing in the same direction. Directional lights are commonly used to emulate the sun. Instead of passing in the light position, you would pass in the direction the light is facing in so you would pass in the light direction \( l_{\text{dir}} \) and the light color \( c_{\text{light}} \). Then, you would calculate \( L \) as:

\[ L = -l_{\text{dir}} \]

It is negated because the light direction is in the opposite direction of the vector from the vertex to the light.

2.6 Hemisphere Light

Without the use of image based lighting, it is hard to get the “lighting” that comes from the scene (aka global illumination). One way to fake this is to use hemisphere lights. You can imagine that there is an imaginary sphere around every object and when the normal points upwards, it would see one color and when it points downwards, it sees another. And when it points towards the side, it takes a little bit of both. See figure 2 for a nice visualization.

This may seem a little complex, but the math behind it is fairly simple. Instead of passing one color to the shader, you must instead pass two colors \( c_{\text{sky}} \) and \( c_{\text{ground}} \). Then, the resultant light color is:
Figure 2: Sourced from Sun and Black Cat

\[ c_{light} = \text{lerp}(c_{ground}, c_{sky}, \text{clamp}(N \cdot (0, 1, 0) \ast 0.5 + 0.5, 0, 1)) \]
\[ L = N \]

The only problem with hemisphere lighting is that it produces inaccurate specular reflections since you would not expect the sky to give off a specular highlight. In other words, we use hemisphere lighting to very simply approximate diffuse interreflection. As a result, you want to disable specular highlights for hemisphere lighting only.

### 2.7 Light Attenuation

Usually, you want to attenuate the light the further it is from you. This obviously does not apply to directional or hemisphere lights, but it does apply to point lights. Let’s say that a vertex is some distance \( t \) from the point light and let’s assume that the light has a “radius” of \( l_{radius} \). Then, one can calculate the attenuation as:

\[
a_{total} = \frac{\text{clamp}(1 - (\frac{t}{l_{radius}})^4, 0, 1)^2}{a_{constant} + a_{linear}t + a_{quad}t^2}
\]

where \( a_{constant} \), \( a_{linear} \) and \( a_{quad} \) are variables you pass to the shader. Then the final color is:

\[ c'_{final} = a_{total}c_{final} \]
3 Getting Started

3.1 Setup

Assuming you followed the instructions to download the code from Assignment 1, to get the assets for assignment 4:

```shell
git fetch source
git merge source/master
```

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

```c
#define APPLICATION Assignment3
```

to

```c
#define APPLICATION Assignment4
```

The scene will be constructed inside “Assignment4.cpp” instead of “Assignment3.cpp.”

3.2 Overview

The task to create a new BRDF shader as well as implementing various lights may seem like a giant task at first, but take it one step at a time and it won’t be so bad (and start early, hint)! The thing that you must first understand is how the assignment framework is connected. As you might already know, each “SceneObject” can have one or more “RenderingObject”s and each “RenderingObject” has a “ShaderProgram” associated with it. While in the previous week you played with the relationship between the “SceneObject” and the “RenderingObject”s, this week you will play with the relationship between a “RenderingObject” and a “ShaderProgram.”

In Assignment 3, you may have noticed that for the sphere and for the outlander model, I create a “BlinnPhongShader” object from an “std::unordered_map.” This map contains the path to the GLSL vertex shader as well as the GLSL fragment shader. The “BlinnPhongShader” object is ONLY able to interface with those specific vertex and fragment shaders because it has been coded to specifically deal with the shader uniforms (parameters) found within the vertex and fragment shaders. This object also stores the “material” for the object. The “SetDiffuse” and “SetAmbient” calls that you see are functions that set the material properties.

Now you may wonder how the light is supposed to know what properties it is supposed to have and this is via the “LightProperties” system. You can see that a “BlinnPhongLightProperties” structure is created which holds a “diffuseColor” and “specularColor” as these are the properties the light needs. This structure is created by a static method in “BlinnPhongShader” called “CreateLightProperties.” Note that in hindsight, this was a particularly poor design decision as it limits the number of shaders we can use per scene to one. This is not too big a deal since for the assignment, since you probably only want one main shader with some minor variations. In reality, you would probably want one giant “LightProperties” structure that will hold every single light property for every single shader type.

4 Instructions

Note that no “fill in the code here” sections are given for this assignment. You will create will be creating many classes from scratch and making sure they fit the interface that is necessary for use with the rest of the framework. Read and follow this section in order. These instructions are more detailed in the beginning to
get you started. Later, it assumes that you will look at the surrounding code and the sample code to be able to figure things out. If you cannot, feel free to ask on Piazza or in office hours.

4.1 Bootstrapping

Start with the Blinn-Phong shader as a basis for your Epic shader. Do the following:

1. Make duplicates of the following GLSL shaders:
   - “shaders/brdf/blinnphong/frag/noSubroutine/blinnphong.vert” and rename to “epic.vert.”
   - “shaders/brdf/blinnphong/frag/noSubroutine/blinnphong.frag” and rename to “epic.frag.”

2. Make duplicates of the following C++ files:
   - “common/Scene/Light/LightProperties.h” and rename to “EpicLightProperties.h”
   - “common/Rendering/Shaders/BlinnPhongShader.h” and rename to “EpicShader.h”
   - “common/Rendering/Shaders/BlinnPhongShader.cpp” and rename to “EpicShader.cpp”

   Inside the header (.h) files, be sure to:
   - Change the header guard (#ifndef... #define...) so that it properly reflects the new file name. In other words, it is says BLINN_PHONG_SHADER and the filename is now “EpicShader.h”, change BLINN_PHONG_SHADER to EPIC_SHADER for both the #ifndef and the #define.
   - Change the class/struct name to match the filename.

   For the EpicLightProperties struct, make sure that it publically inherits from the LightProperties. Note that you will have to include the LightProperties header file “common/Scene/Light/LightProperties.h.” Also make sure that it is empty (note that diffuseColor and specularColor will be inherited).

   Inside the source (.cpp) file, be sure to:
   - Change the #include statement path from the BlinnPhongShader to the EpicShader header file.
   - Change everything that references “BlinnPhongShader” to reference “EpicShader”.

   Inside, the CPP file, there are many checks to see whether DISABLE_OPENGL_SUBROUTINES is defined or not defined (note #ifdef means if defined and #ifndef means if not defined). For this assignment, we will pretend that OpenGL subroutines are not supported. At the top of the “EpicShader.cpp” file, add the line #define DISABLE_OPENGL_SUBROUTINES.

At this point, you should be able to go into Assignment4.cpp and change the paths to the “blinnphong.vert/frag” file to the “epic.vert/frag” files you just made, BlinnPhongShader to EpicShader, and LightProperties to EpicLightProperties. Your code should compile, run, and look exactly the same. Don’t forget to include your new header files. Note if you’re running from the command line, you must re-run the CMake commands for the cs148opengl project from Assignment 1. For Visual Studio/XCode, I believe that the CMake command will be automatically re-run for you. If not, try re-running CMake as in Assignment 1 manually.

4.2 Implementing the BRDF

All the math implementation will be inside the “epic.frag” file and you will be modifying “EpicShader.h/cpp” and “EpicLightProperties.h” as necessary to get the proper data to your shader. Here’s what you will need to do to get your Epic BRDF implemented with a point light (unless otherwise noted, instructions are for the epic.frag file):

1. There is no “ambient” term for the Epic BRDF, let’s remove it. There’s a number of steps you need to do to accomplish this:
   - (a) Change globalLightSubroutine to return vec4(0) (aka have no effect on the final color).
   - (b) Remove vec4 matAmbient from the InputMaterial struct.
   - (c) You will need to change the MATERIAL_PROPERTY_NAMES array to match your changes to the InputMaterial struct in the “EpicShader.h/cpp” files. Change the length of the array from
4 to 3 and remove the “InputMaterial.matAmbient” item.

(d) Inside the `UpdateMaterialBlock` function, remove the `memcpy` dealing with ambient.

2. Notice that the Epic BRDF has three tunable parameters. Change the `InputMaterial` struct inside the frag file to reflect these parameters (three floats). Modify the `MATERIAL_PROPERTY_NAMES` array with the proper names and the `UpdateMaterialBlock` function to also reflect that change (note that you want to be copying floats not vec4s). It will be helpful to add these new parameters into your EpicShader class and have public functions to be able to modify them.

3. Notice that the equations have a singular light term instead of a diffuse and specular color for each light. Change the `LightProperties` struct to have a singular light color and change the `SetupShaderLighting` in “EpicShader.cpp” to only set a single color instead of a diffuse color and a specular color.

4. Inside the `pointLightSubroutine`, you should see computations for `diffuseColor` and `specularColor`. Change those equations to match the equations explained above. Note that `c_base` is passed into the fragment shader via the variable `fragmentColor`. You can see where the vertex colors are made inside “Assignment4.cpp” inside the “GenericSetupExample” function.

After completing this section, you should able to see the effect of each of the parameters (roughness, specular, metallic) on the shading of the spheres. Note that there’s a lot of code inside the CPP files that deal with textures, you can safely ignore that for now. **Note: Don’t forget to do the light attenuation!**

### 4.3 Extending to Directional and Hemisphere Lights

You might have realized that the shader only handles one light at a time—you may then wonder how the framework handles multiple lights. What is implemented is a multi-pass forward renderer. This means that each object is rendered \( n \) times where \( n \) is the number of lights (obviously, optimizations can be made to handle more than one light in each pass, but that is besides the point). For each pass, you compute the color of a pixel for a given light \( c_i \) and in the end the final color is \( c_{\text{final}} = \sum_{i=0}^{n} c_i \) (aka, sum up the contribution to the color for each light). So what we do is to tell the shader via a uniform int on what type of light we are currently using. The shader then takes that int (`lightingType`) to be able to call the right function to do the right computations. Making sure the right lighting uniforms are set is the job of the `SetupShaderLighting` function which takes in a pointer to the light being rendered. Currently, you’ll note that the light properties inside the frag file are split between the `LightProperties` struct and the `PointLight` struct. It may be easier (though not necessary) to combine this into the single `LightProperties` struct. Note that you will want to create a DirectionLight class and a HemisphereLight class that inherit from the Light class (“common/Scene/Light/Light.h”) to be able to distinguish between the lights inside the `SetupShaderLighting` function. It will be easier to debug any problems here by having only one light in your scene at a given time.

### 4.4 Swap your Scene In

At this point, you should be able to take out the sphere scene and do a similar setup to what you did for the scene from Assignment 3. Except that this time you will want to load the shaders you just made, create lights, etc. to get your scene shaded with the Epic BRDF using a point light, directional light, and hemisphere light. Note that the camera is different from Assignment 3 (look at “Assignment4::CreateCamera” and “Assignment4::SetupCamera”). I highly suggest you read the pinned “Camera, Light, and Geometry Position/Rotation/Scale using Autodesk Maya” post on Piazza to help you in placing your lights, objects, and cameras.
5 Further Reading

These are just a bunch of links to SIGGRAPH presentations by various companies about their move towards physically based rendering. You may find it interesting.

• Unreal Engine SIGGRAPH 2013 Presentation: here.
• Disney SIGGRAPH 2012 Presentation: here.
• The Order 1886 SIGGRAPH 2013 Presentation: here.
• EA Frostbite SIGGRAPH 2014 Presentation: here.

Additionally, there is a ton of freely available source code for you to read online:

• Unreal Engine 4: Instructions for access can be found here.
• Pixar’s Renderman: Get the free and non-commercial version here.

6 Cool Things to Do

There are many cool things that you can do to further enhance your image. Just to list off a few:

1. Additional Material Parameters. The Unreal Engine material is a highly simplified version of the Disney material (look in Further Reading for links to the presentations). You can find a complete implementation of the Disney shader in RenderMan. On my Mac, this is located at “/Applications/Pixar/RenderManStudio-20.3-maya2016/rmantree/lib/examples/RIS/plugins/bxdf/PxrDisney.cpp”.

2. Areas Lights. If you look in the Unreal Engine presentation, you can see their use of area lights. Area lights are pretty important for getting proper lighting! However, this is a pretty complex topic. The book “GPU Pro 5” has a more thorough discussion on area lights. Stanford has this book available online here.

7 Grading

This assignment will be graded on the following requirements:

• Your scene is shaded using the Unreal Engine BRDF and material.
• Your scene contains at least one point light.
• Your scene contains at least one directional light.
• Your scene contains at least one hemisphere light.

according to the following rubric.

• 4 – All three light types (point/directional/hemisphere) using the Epic BRDF are implemented and in the scene at the same time.
• 3 – Two functional light types (point/directional/hemisphere) using the Epic BRDF are implemented.
• 2 – One functional light type (point/directional/hemisphere) using the Epic BRDF is implemented.
• 1 – Attempted to code up the Epic BRDF but nothing works.
• 0 – Fail to show up for grading.

The contribution to the shading of each light must be clearly demonstrated to receive credit for that light type. If you want, you can toggle each light on/off individually to show the effect of each individual light.

Note: In the end, all three lights must be in the scene at the same time to receive full points.
7.1 Clarifying Points

The term *Your Scene* refers to the scene you made from Assignment 3 (improvements and changes are fine). In other words, the default scene with the spheres is not valid for grading. Note that if you did not do so last week, you will have to make sure your camera, lights, and meshes are placed properly or else your shading will not be displayed clearly.

![Image of a scene lit with a directional light and a hemisphere light using the Unreal Engine BRDF and material using the assignment framework.](image)

Figure 3: A scene lit with a directional light and a hemisphere light using the Unreal Engine BRDF and material using the assignment framework.