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. ALSO, usually a day or two after the assignment goes out, we create an assignment update + FAQ thread, PLEASE READ IT PERIODICALLY.

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

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/BRDF.usf as well as Engine/Shaders/ShadingModels.usf (the function StandardShading). You can trace this function back farther if desired.

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:

\[
\begin{align*}
N & \text{ - the normal vector of a vertex} \\
V & \text{ - the normalized vector from the vertex to the camera} \\
L & \text{ - the normalized vector from the vertex to the light} \\
H & = \frac{L + V}{\|L + V\|} \\
c_{diff} & \text{ - the diffuse color of the material} \\
c_{spec} & \text{ - the specular color of the material} \\
c_{light} & \text{ - the color of the light} \\
x_{vert} & \text{ - vertex position}
\end{align*}
\]

Material

Epic Games uses three tunable parameters:

\[
\begin{align*}
r & \text{ - roughness} \\
s_{c} & \text{ - specular} \\
m & \text{ - metallic}
\end{align*}
\]

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:

\[
\begin{align*}
c_{diff} & = (1 - m)c_{base} \\
c_{spec} & = \text{lerp}(0.08s_c, c_{base}, m)
\end{align*}
\]

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).
**BRDF**

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

**Diffuse**

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

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

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

**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:

\[
D = \frac{\alpha^2}{\pi((N \cdot H)^2(\alpha^2 - 1) + 1)^2}
\]

\[
k = \frac{(r + 1)^2}{8}
\]

\[
G_1(v) = \frac{N \cdot v}{(N \cdot v)(1 - k) + k}
\]

\[
G = G_1(L)G_1(V)
\]

\[
F = c_{spec} + (1 - c_{spec})2^{(-5.55473(V \cdot H) - 6.98316)(V \cdot H)}
\]

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.

**Final Color**

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

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

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.

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.

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:

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

\[
L = N
\]

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.
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:

\[
c'_{brdf} = a_{total}c_{brdf}
\]

Getting Started

Terminology

On Piazza and in office hours, we have noticed that there is some confusion over the terminology used in the framework and/or assignments so hopefully this clears up some of the confusion.

- “image” - This is what gets displayed by OpenGL in the window every frame.
- “model” or “mesh” - This is the 3D object that you made in Maya/Blender/etc. or downloaded from the internet. This is commonly stored in an OBJ file.
- “scene” - Is the 3D representation of your world. At first, this is empty. But as you keep adding scene objects and lights, your world begins to be populated.
- “scene object” - This is an object that lives within the scene. It may be made up of one or more “models”.

Setup

In the assignment zip, there are three folders “common”, “assignment3”, and “blinnphong”. As before, copy the “assignment3” folder to be next to assignment1 and assignment2. Next, replace “source/shaders/blinnphong” with the new “blinnphong” folder. Finally, replace your current “common” folder with the new “common” folder. However, if you have already made changes to your “common” folder, there are the relevant changes:

- common/MediaLayer.cpp - Fixed an bug with SDL. This was sent out during Assignment 1. Also added multi-sample anti-aliasing for nicer images (new, lines 47 and 48).
- common/Rendering/ForwardRenderer.h/cpp - Fixed a bug with the renderer which resulted in Z-fighting. This was sent out with Assignment 2.
- common/Rendering/Shaders/BlinnPhongShader.cpp - Fixed a bug with specular textures for the Blinn-Phong Shader.
- common/Scene/SceneObject.cpp - Fixed a bug with improper object transformations. This was sent out during Assignment 2.
- common/Utility/Mesh/Simple/PrimitiveCreator.cpp - Added a plane creation function.
- common/Utility/Mesh/Loading/MeshLoader.cpp - Fixed an issue where points and lines were not being stripped properly from the mesh.
- common/Scene/Light/Light.h/cpp - Added more light types as well as functions to change the light attenuation.

Coding

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 2, 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.

**Bootstrap**

Now that you have a general idea of how everything works, here are some pointers on how to get started. Feel free to ignore everything here, these are nothing but tips. But if you have no idea how to get started, read this. Also make sure to read the general tips section at the bottom of this assignment.

1. Copy over all the Blinn-Phong related items and rename them to something new (i.e. EpicShader, EpicBRDF, whatever). After making a copy of everything (GLSL shaders, BlinnPhongShader, BlinnPhongLightProperties), the next thing you will want to do modify the assignment to use your new shader. Note: be careful of which GLSL shaders you use. If you are on Mac OSX or using OpenGL 3.3 (or earlier), you will want to use the shaders without subroutines (noSubroutine folder).
2. Next, make the point light subroutine/function and the global light subroutine/function return black aka vec4(0,0,0,0) as you prepare the shader for major modifications.
3. The first thing you want to do is to write the DIFFUSE functionality for a POINT LIGHT. While doing so, you will want to change the shader materials found in “InputMaterial” to match the Epic BRDF (metallic, roughness, specular). After doing that, you will need to modify the C++ shader equivalent (i.e whatever you copied BlinnPhongShader to). Namely, you will want to change MATERIALPROPERTY_NAMES, the call to SetupUniformBlock in the constructor, and the implementation of UpdateMaterialBlock). Make sure that you change the material setter function! The existing code should give you a general idea of what to change. Verify that diffuse works.
4. Implement the specular color. Verify that specular works.
5. Now generalize your shader code to handle directional lights. You can do this either via subroutines or via if/else if/else statements. You will then want to go into the SetupShaderLighting function in C++ and change it as necessary to make sure your shader knows which path to go down. You see all the calls in that function relating to “pointLighSubroutine” and “globalLightSubroutine”? You will want to do something similar for a directional light (note if you are not using subroutines, you will want to see where it says Light::LightType::POINT and Light::LightType::GLOBAL). Additionally, you will need to add a new light property in the shader for the light direction. See the general tips section for more guidance on what to do on the GLSL side. Add a directional light to your scene and make sure it works! Finally, don’t forget to create a new type of light on the C++ side!
6. Next generalize your shader code again to include hemisphere lights. Again, you will need to add new light properties and make sure it gets chosen. See the general tips section for more guidance on this. Add a hemisphere light to your scene and make sure it works! Don’t forget to disable specular lighting for hemisphere lights. Don’t forget to create a new type of light on the C++ side.
7. Finally, modify the AttenuateLight function to include the light radius. In the light attenuation section of this writeup, you will want to implement what is in the numerator. This is only relevant for point lights.

And then you are done! Hurray!

**General Tips**

- In BlinnPhongShader::SetupShaderLighting, I initially added a call to Light class’s “SetupShaderUniform” function. In hind-sight this is a bad idea and you should handle setting the shader uniforms within the shader program object.
Another poor design choice I made was in the Blinn-Phong fragment shader where I separate out the point light’s position from the properties. In hind-sight, all these light properties should be together in one structure as there’s no real reason to keep them separate.

It would be wise to work with the provided scene first before moving on to your (most likely) more complex scene.

In your shading calculations, you may want to use “vec3” instead of “vec4” (the Blinn-Phong shader uses vec4). However, the final color must still be a “vec4.” This is more accurate since the alpha does not matter in the BRDF calculations. This is only relevant if you plan on having transparent objects.

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.

- Disney SIGGRAPH 2012 Presentation: here.
- The Order 1886 SIGGRAPH 2013 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.

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/rman/tree/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.

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.

- + – Exceeds the requirements via one or more artistic/technical contributions.
- ✓ – Meets all of the requirements
• − − Does not meet the requirements but still produces a drawing.
• 0 − The submitted solution does not produce a drawing.

Note that although we require your scene to contain a point light, directional light, and a hemisphere light for the demo, there is no need for you to keep all three types of lights in your scene for your final image. Should you decide that one or more light types do not fit your image well, then feel free to remove them after the demo!

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