Assignment 7 – Lights

Overview

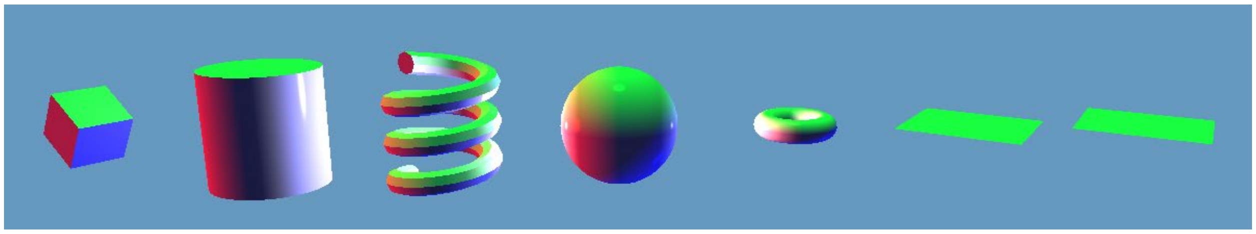
Now that you've got 3D objects in your scene, you've probably realized how bland they look without any sort of shading. For this assignment, you'll be adding lighting calculations to shade your 3D models. You'll need to implement **at least five lights** for this assignment, in addition to an ambient term.

Task Overview

Here is the high-level overview of tasks, which are explained in more detail on the pages that follow:

- ❑ Create a **shader include file** to hold common lighting and other helper functions
- ❑ **Decide which pixel shader** – an existing one or a new one – will be used for lighting
- ❑ Update your material class to hold a **roughness value** for specular calculations
- ❑ Update your **vertex shader** to properly **pass new data** to the pixel shader
- ❑ Update your **C++ code** to properly **pass new data** to the pixel shader
- ❑ Update C++ and shaders to support the following:
    - ❑ An **ambient** term
    - ❑ At least **three directional lights**
    - ❑ At least **two point lights**
- ❑ Add a user interface for **changing your lights** in some way
- ❑ Ensure you have no **warnings**, **memory leaks** or **DX resource leaks**

Completed Example



This scene has 5 lights:

- Red directional light pointing right (1,0,0)
- Green directional light pointing down (0, -1, 0)
- Blue directional light pointing up at an angle (-1, 1, -0.5), normalized of course
- White point light with a range of 10 positioned left of the sphere (between sphere & helix)
- Dim white point light with a range of 10 positioned right of the sphere (between sphere & torus)

The ambient color is (0.1, 0.1, 0.25).
All seven objects are using the same material with a roughness of 0.15 and a white color tint (1,1,1)

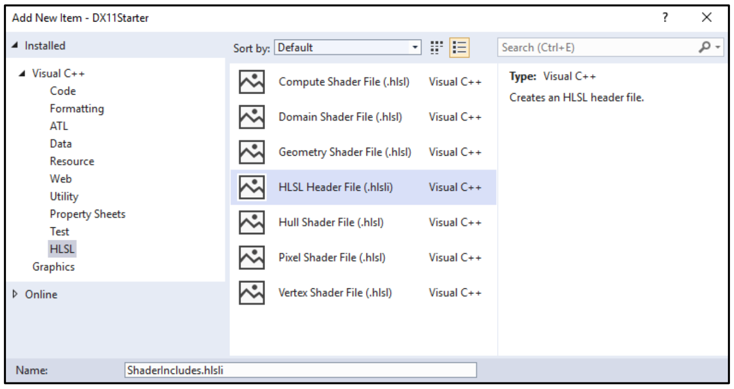
Task 1: Create a Shader Include File

By the end of this assignment, you'll have several helper methods for various lighting calculations. As we add new shaders in future assignments, you'll need to ensure they all have access to these functions; making a small change in one will often necessitate making the exact same change in another. As you might imagine, this can quickly get messy and is certainly prone to error.

To make your life easier, you should place common structs and helper methods in a *shader include file*. This is simply a text file, often (but not always) using the *.hlsli* file extension, which can contain the HLSL code and structures that you might want to share among your various shaders. You can then *#include* this file at the top of each actual shader file.

Creating a Shader Include File

You can create one of these files quickly inside Visual Studio. In the Solution Explorer, right click on your *Shaders* filter, choose Add > New Item…, then in the window (see the screenshot to the right) choose *HLSL* in the left column and *HLSL Header File* in the center.


Add New Item - DX11Starter
Installed
Sort by: Default
Search (Ctrl+E)
Visual C++
Code
Formatting
ATL
Data
Resource
Web
Utility
Property Sheets
Test
HLSL
Graphics
Online
Compute Shader File (.hlsl) Visual C++
Domain Shader File (.hlsl) Visual C++
Geometry Shader File (.hlsl) Visual C++
HLSL Header File (.hlsli) Visual C++
Hull Shader File (.hlsl) Visual C++
Pixel Shader File (.hlsl) Visual C++
Vertex Shader File (.hlsl) Visual C++

Type: Visual C++
Creates an HLSL header file.

Name: ShaderIncludes.hlsli

You can simply make one big include file that holds everything you'll ever need to share among shaders. Alternatively, you can make several, each with a different purpose. For instance, you might want one for structs, one for lighting-related functions and (eventually) one for texture-related functions. The exact organization is up to you, but name it (or them) appropriately.

Include Guard

To be on the safe side, it's good practice to start with a preprocessor include guard. Since *#pragma once* doesn't work with the HLSL compiler, you'll need a more traditional C/C++ style include guard:

```
#ifndef __GGP_SHADER_INCLUDES__ // Each .hlsli file needs a unique identifier!
#define __GGP_SHADER_INCLUDES__

// ALL of your code pieces (structs, functions, etc.) go here!

#endif
```

The exact name(s) you choose for your include guard are up to you, but it should be obviously unique (hence the underscores and capitalization). Don't use an identifier that might appear elsewhere!

Note: If you have two (or more) shader include files, each needs its own, **unique** include guard name!

Contents of a Shader Include File

So, what can go into a shader include file? Technically, it can have anything that you'd define in a standard shader file: structs, helper functions, static constant variables, #define preprocessor directives, etc. While you can even include resource declarations like cbuffers, these are often shader-specific, so I **would not** suggest moving them to the include file, at least for this assignment.

More specifically, for this assignment, you'll probably want to start by moving any **structs** defined in VertexShader.hlsl and PixelShader.hlsl over to the include file. Note that VertexToPixel is defined identically in both shaders, so you only need it to be in the header once. You'll also want to move any **helper functions** over to the include file. Order of function definitions matters, since a function can only be called if it already exists.

> *If you don't have many (or any) helper functions yet, that's ok! Throughout this assignment, you'll be creating several new structs and functions. In addition, anything you find yourself doing over and over could be moved to a function. It'll be much cleaner for `main()` to simply be a few function calls instead of 50+ lines of code.*

#include

Once you've moved things over, simply add the following line to the top of each .hlsl shader file:

```
#include "YourIncludeFileHere.hlsli"
```

Testing

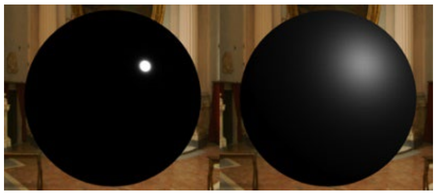
Your code should compile and run at this point, and you shouldn't see any differences.

Task 2: Pick a Shader

The rest of the assignment assumes you'll be working in a **single pixel shader** (plus your shader include file) that will handle all lighting and, eventually, features like textures and other advanced rendering concepts. I suggest altering **PixelShader.hlsl** for this, but you can instead opt to make a new shader. If you do make a new shader, it will need the same basics (VertexToPixel struct, cbuffer with colorTint, main(), etc.) as PixelShader.hlsl to start.

Note that, while it won't explicitly be pointed out later, you'll need to ensure you're loading and using any new shaders you create. (This also means making materials that actually use them!)

Task 3: Material Roughness & Camera Position



Part of this assignment will have you implement a specular calculation, which aims to approximate the reflection of a light source. In other words: the shiny highlights that appear on very smooth surfaces. You'll need a *roughness* value and the *camera's current position* for this calculation, so let's prepare those.

Whether or not a surface has these highlights (and their overall appearance) is a property of the *surface itself*. To implement this, **add an extra float variable** to your Material class. We're going to call this the material's *roughness*. Require this value as part of the material's constructor and save it appropriately.

If you recall from class, the value we use to control the "shininess" of our specular calculations was referred to as the *specular exponent*: a somewhat large number that had no specific bounds or real-world units. However, it's rather useful to be able to define our material properties within an explicit range, with useful (and predictable) minimum and maximum values.

To this end, roughness will range from 0 - 1, which gives us a very controlled range of possible values. When roughness is 1.0, the surface should be as rough (matte) as possible and have no highlights. When roughness is 0.0, the surface should be as smooth (mirror-like) as possible and have very precise highlights. We'll convert this roughness value to a useful specular exponent in the shader itself.

Roughing It: C++

Update the materials you create to include a roughness value between 0 – 1. The exact values you choose at this point are up to you. Once you have specular lighting implemented later on, it should be easy to test new values.

Send both the *roughness* and the *camera's current position* to your pixel shader before drawing each entity. Use the SetFloat() and SetFloat3() functions of SimpleShader, respectively, for this. It could happen in the Material class (if you have a PrepareMaterial() helper), or directly in Game::Draw().

Roughing It: HLSL

Update your pixel shader's cbuffer to have a float variable for *roughness* and a float3 *cameraPosition*.

You won't be using these until later in the assignment, but if you'd like to test whether or not the values are making to the pixel shader, you can return one as a color. For roughness, it would look like so:

```
        return float4(roughness.rrr, 1); // Replicates value x3 (this is temporary)
```

A roughness of 0 will result in the color (0,0,0), so objects will look black. A roughness of 1 is (1,1,1), so the objects will look white. Anything in between will be greyscale. Once you're satisfied that the data is making it to the pixel shader, remove that line and move on to the next task.

Task 4: Ambient

An ambient "light" is meant to be a dirt-simple approximation of *global illumination*: all the light that bounces around a scene. Since it's not an explicit light source itself, we often call it an *ambient term*. This is simply a single color that represents the minimum light level in a scene.

In other words, if you had to pick a single color to represent the overall color of the environment, what would it be? Since this color will be applied evenly to every single pixel you render, it should be *subtle* (values in the 0.0 – 0.25 range, at most). Some examples:

- Outside, it might be grey (0.2, 0.2, 0.2) or sky blue (0.0, 0.1, 0.25) depending on clouds.
- In a forest, it might be green (0.0, 0.1, 0.0) or brown (0.1, 0.1, 0.05).
- In a very dark environment, it could be extremely low (0.05, 0.05, 0.05) or even black (0, 0, 0).

Since you don't really have much of an environment yet, try to match this to the color that you use to clear the screen (but darker). By default, this is cornflower blue, though you may have changed it.

Implementing Ambient: C++

Your program should define an ambient color as an XMFLOAT3 in C++ (probably in the Game class) and pass it to any shader that will handle lighting.

Since this isn't *Entity* data or *Material* data, you won't (easily) be able to set the data within either of those classes. Instead, as part of your entity rendering loop in Game::Draw(), you can get the pixel shader of the current entity's material and set it that way. Something like this:

```
currentEntity->GetMaterial()->GetPixelShader()->SetFloat3("ambient", ambientColor);
```

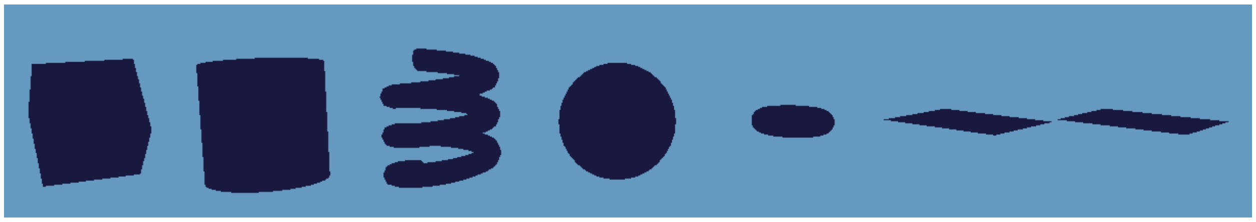
For this assignment, most (if not all) materials might all use the same shader, but that won't necessarily be true in future assignments.

Implementing Ambient: HLSL

Ensure your pixel shader's cbuffer has a float3 for the ambient term. Since lights tint surface colors, multiply the ambient term and the color tint. For now, that's all you need to return from the shader.

Example

Ambient color of (0.1, 0.1, 0.25). All objects have a color tint of (1,1,1). Result is dark, but not black.



Task 5: Normals & World Position

Before you can tackle actual light sources, you need to ensure that your pixel shader has all of the requisite data. Namely, it needs a *surface normal* for the current pixel, as well as the *world position* of that pixel. These are interpolated by the rasterizer from vertex data, so you'll need to pass them through the pipeline, which requires adding additional variables to the output structure (VertexToPixel).

Add a float3 called "normal" to the VertexToPixel struct, using the semantic "NORMAL". Add another float3 called "worldPosition", using the semantic "POSITION". (We don't have to worry about padding and boundaries here, as that is only for cbuffers.)

You should have a normal coming *into* the vertex shader from last assignment. If the vertex has some kind of transformation being applied, you need to apply a similar transformation to the normal. There are a few issues, however: Translation isn't useful when transforming a normal, the world matrix is 4x4 not 3x3, and non-uniform scales are problematic. To solve the first two issues, cast the matrix to 3x3:

```
output.normal = mul((float3x3)world, input.normal); // Not quite there yet!
```

To properly transform a normal using a potentially non-uniform scale, however, you'll need to use the *inverse transpose* of the world matrix. You're already calculating that matrix in your Transform in C++; pass it into the vertex shader as an extra matrix variable through the constant buffer, then apply it:

```
output.normal = mul((float3x3)worldInvTranspose, input.normal); // Perfect!
```

For the world position of the pixel, you'll need to convert localPosition to a float4, multiply by the world matrix and then grab the first 3 components of the resulting vector:

```
output.worldPosition = mul(world, float4(input.localPosition, 1)).xyz;
```

Normals in the Pixel Shader

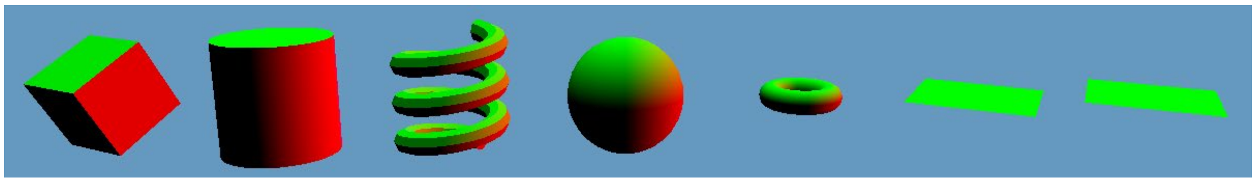
Ensure your pixel shader properly expects this data in its input struct. If you're storing VertexToPixel in a shader include file, this is already done. Your first step in the pixel shader itself is to normalize the incoming normal, as interpolation of normals across the face of a triangle results in non-unit vectors:

```
input.normal = normalize(input.normal);
```

For some quick testing, return the normal or world position as a color on the next line:

```
return float4(input.normal, 1); // This is temporary
```

Run your program. You should see a mix of red, green, yellow and black colors (and potentially blue if you move the camera to the backside of the objects). These are the normals of the vertices being visualized as colors. Once you're confident all the data is correct, remove that return statement.



Task 6: Your First Light

This task will guide you through the basics of adding one simple directional light to your engine. You'll add several more yourself in the next task.

Lighting: C++

C++ will be responsible for defining the actual *data* of the light(s) in your scene: color, position, direction, etc. This data will need to be sent to the pixel shader before drawing each entity (in our simple, un-optimized engines anyway).

> *An advanced engine might have a Light Manager or Renderer in charge of determining a subset of lights to use when drawing various entities depending on distance and attenuation.*
>
> *Lights in an advanced engine might even have their own Transform objects for position and orientation, allowing the lights to move or rotate over time. They could even inherit from a GameEntity class, or be a component added to GameEntities. The transform's data would be used when preparing the light's position and direction data for the pixel shader. That's not a requirement for this assignment, just some food for thought.*

Lights.h

Create a new header file (Lights.h) to hold C++ side light structs and constants. Start by using #define to define the three major light types as simple integer values.

```
#define LIGHT_TYPE_DIRECTIONAL     0
#define LIGHT_TYPE_POINT           1
#define LIGHT_TYPE_SPOT            2
```

Define a struct called "Light". This struct will be robust enough to handle any data we need for any of our three major light types (directional, point and spot). It needs the following variables, *in this order*:

- int Type // Which kind of light? 0, 1 or 2 (see above)
- XMFLOAT3 Direction // Directional and Spot lights need a direction
- float Range // Point and Spot lights have a max range for attenuation
- XMFLOAT3 Position // Point and Spot lights have a position in space
- float Intensity // All lights need an intensity
- XMFLOAT3 Color // All lights need a color
- float SpotFalloff // Spot lights need a value to define their "cone" size
- XMFLOAT3 Padding // Purposefully padding to hit the 16-byte boundary

Order matters here because we're going to duplicate this struct in the Pixel Shader soon, which will then be used within a constant buffer. Recall that vectors in constant buffers cannot cross 16-byte boundaries, so it makes sense to interlace the single float values with the float3's to avoid data packing issues. When using SimpleShader, it allows us to essentially disregard padding *between individual variables* defined directly in a constant buffer, but multiple variables *inside a single struct in HLSL* still require extra care on our part.

Creating a Directional Light in C++

Create a variable of type `Light` in Game.h (no pointer necessary), and initialize it during Game::Init(). Any easy way to initialize an entire struct with zeros is to use { }'s, like so: `directionalLight1 = {};`

Now that the struct's variables all hold zeros, you should set only those necessary for a directional light:

- **Type** should be LIGHT_TYPE_DIRECTIONAL
- **Direction** can be any valid 3D direction. You'll be normalizing it in the shader. Example: (1, -1, 0)
- **Color** should ideally be bright so it's obvious. Choose any color. Example: (0.2, 0.2, 1.0)
- **Intensity** can be anything, though 1.0 makes the result predictable for now.

Sending The Data

You'll need to send a few more things to the pixel shader before drawing your entities. Start by passing the entire light struct using SimpleShader's SetData() method during your entity render loop like so:

```
pixelShader->SetData(
    "directionalLight1",      // The name of the (eventual) variable in the shader
    &directionalLight1,       // The address of the data to set
    sizeof(Light));           // The size of the data (the whole struct!) to set
```

Even though you haven't added this variable to the shader yet, the code should still run without errors at this point. The *Set* methods of SimpleShader are designed to fail gracefully (and return false) rather than breaking if the variable isn't found.

Lighting: Pixel Shader

There are several additions you'll need to make in the pixel shader: matching your C++ light struct, adding a Light variable to your cbuffer and actually performing various lighting calculations.

Defining a Light

HLSL needs the same light definitions as C++. Go to your shader include file and mimic the three *LIGHT_TYPE_* #defines and the Light struct itself from C++. Remember to match the struct layout exactly. Since you'll need it later, also create a #define for the maximum specular exponent value:

```
#define MAX_SPECULAR_EXPONENT 256.0f
```

Add a variable of type `Light` to your pixel shader's cbuffer. The name must match the string you used in SetData() in C++ (see above). If you followed this document exactly, it's probably "directionalLight1".

Test whether or not this data is making it to the pixel shader by temporarily returning the color of the light in the shader, as shown below. This should make your objects solidly *that color* for testing.

```
return float4(directionalLight1.Color, 1); // Temporary
```

Once you're sure the data is getting there, remove that line and move on to the next step.

Implementing a Basic Diffuse Lighting Calculation

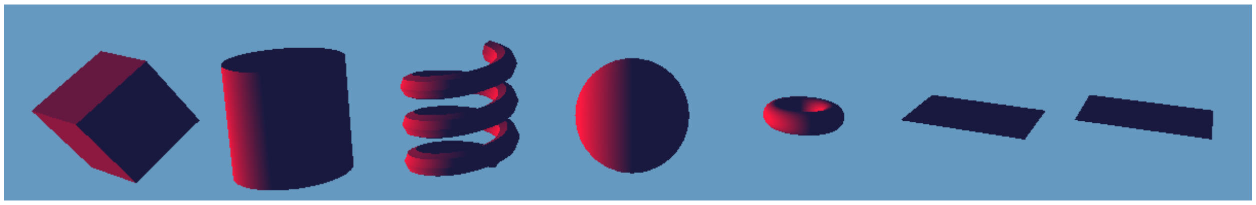
You've got all the data necessary to calculate basic diffuse lighting for this particular pixel. Here are the calculations to get this done. Create any **helper functions** you deem necessary, remembering that you'll need to perform some of these steps *for each light* you eventually want to approximate.

- Calculate the **normalized direction *to this light***:
    - Negate the light's direction, normalize that and store it in another float3 variable
    - (You can't store it back in the *light* variable because it's from a *constant buffer)*
- Calculate the **diffuse amount** *for this light* using the N dot L diffuse lighting equation:
    - Could be a helper function: `float3 Diffuse(float3 normal, float3 dirToLight)`
    - Use the `dot(v1, v2)` function with the surface's normal and the direction *to the light*
    - The surface normal should already be normalized from a previous step
    - The dot product can be negative, which will be problematic if we have multiple lights, so use the `saturate()` function to clamp the result between 0 and 1 before returning
- Calculate the **final pixel color**
    - This is based on the surface's color (the *colorTint* variable), the light's diffusion amount (N dot L result), the light's color and the overall ambient color
    - The approximations of light – ambient and diffuse in this case – are calculated separately and added together, like so (adjust for your variable names):
    - `(diffuse * light.Color * surfaceColor) + (ambientColor * surfaceColor)`
- Return that result from the pixel shader, using 1 for alpha: `float4(finalColor, 1)`

Your program should now be applying ambient and one directional light to all of the objects being drawn with this shader. If the lighting seems off, be sure you're properly negating the light's direction. You can always temporarily return numbers as color values for quick and dirty visual debugging.

Example

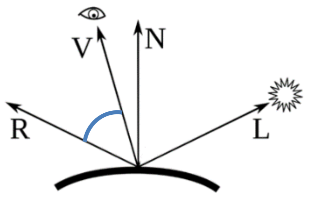
Ambient of (0.1, 0.1, 0.25) and a red (1,0,0) directional light pointing right (1,0,0)



Task 7: Specular Reflections

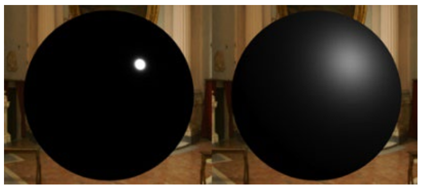
At this point, you have diffuse light being calculated for a single light source. What about shiny objects? Your next step is to handle specular reflections.

Specular Exponent



Recall from class that part of the Phong specular calculation is the dot product between a perfect *reflection* vector (R) and the surface-to-camera *view* vector (V), as shown in the diagram to the right, then raising that result to a power. The first part – the dot product – allows us to determine how close the camera is to that perfect reflection. Raising the result to a power causes that value to fall off (get closer to zero) very quickly, resulting in a bright highlight that appears to follow us around as we move around the object. The overall calculation in HLSL will look similar to the following, where R is the *reflection* vector and V is the *view* (direction from surface to camera) vector. Saturate() ensures the result doesn't go negative.

```
float spec = pow( saturate( dot( R, V ) ), specExponent );
```



Exactly which value we use as the exponent depends on the roughness of the surface we're trying to approximate. As shown in the example to the right, reflections on a shinier surface appear smaller and tighter (having a quick falloff), whereas reflections on a rougher surface appear larger and fall off more gradually.

As mentioned earlier in the assignment, we're defining roughness from 0 – 1. However, the specular exponent has no real-world unit or explicit upper bound; generally, you'd just pick values that *look good*. Larger specular exponent values (triple-digit numbers) result in a "shinier" looking surface while smaller numbers (low, single-digit numbers) result in a duller look.

You'll need to convert from roughness to a sensible specular exponent value. This ends up being quite easy, as you've already defined a MAX_SPECULAR_EXPONENT value in your shader include:

```
float specExponent = (1.0f - roughness) * MAX_SPECULAR_EXPONENT;
```

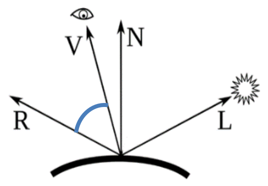
When roughness is 0, the surface is super shiny, so you get the max value. When roughness is 1, the surface is not shiny at all, so you get a value of zero. In between, you get varying degrees of shininess.

Easy, right? Unfortunately, this is where the approximation breaks down. Raising to the zero power (when a surface is not reflective at all) will return 1.0 and cause the specularity to be pure white everywhere, which isn't useful. So how do we handle this? The easiest solution is to simply use a conditional statement in your shader whereby you only perform the specular calculation if the exponent value is greater than some very small value, like 0.05. Otherwise, the result should simply be zero.

> *A shader conditional? Yup! It's quick and easy to implement, and all pixels here will end up taking the same branch. Another common solution: two shaders – one for "reflective" materials that performs the phong calculation and one for "matte" materials that skips it. Really, the best solution is to just use a better BRDF, which is coming soon!*

Phong it Up

Now that you have the requisite data, you can perform the Phong specular calculation in the pixel shader. Much like the diffuse calculation, it must be repeated for *each light* that your shader handles. I **highly** recommend writing helper methods to simplify your overall shader, rather than copying and pasting the same sets of calculations over and over (a habit which itself is prone to errors).



Before performing the calculation, you'll need two vectors you probably don't have yet: the perfect **reflection** of the light source and the **view** (surface-to-camera) vector, both of which should be normalized.

The **view** vector (the vector from this pixel to the camera) requires the world position of the pixel and the camera’s position. Simple vector subtraction does the rest. This vector can be calculated once per shader, as it doesn't rely on any per-light data. (Adjust variables accordingly.)

```
float3 V = normalize(cameraPosition - pixelWorldPosition);
```

The **reflection** is easy enough to calculate using HLSL's reflect() function as seen below. "Incoming light direction" is the *direction the light is traveling*, **not** the direction back towards the light. The normal is simply the surface normal. Both of these should already be normalized, so the result should be as well. This vector needs to be recalculated *for each light*, as each light usually has a different direction!

```
float3 R = reflect(incomingLightDirection, normal);
```

Here is the specular calculation in HLSL again for your reference:

```
float spec = pow( saturate(dot(R, V)), specExponent );
```

This result is then added to your diffuse calculation for this light. However, there are some considerations about specular reflections and surface colors.

Specular Reflections & Surface Color

The question of whether the specular reflection should be tinted by the surface color often comes up. Some quick internet searches usually result in conflicting examples: some say to tint the specular by the surface color and some that don't. In other words, which of the following is "correct"?

```
float3 light = surfaceColor * (diffuse + specular); // Tint specular?
float3 light = surfaceColor * diffuse + specular;   // Don’t tint specular?
```

The real answer is that it depends on a number of factors. For instance, metals reflect light differently than non-metals (like plastic, wood or cloth). Diffusion of light works a bit differently with metals and ends up being almost imperceptible, while their reflections are very obvious. Metals also tint everything they reflect; think of a very smooth gold bar or shiny copper pipes. Non-metals, on the other hand, have very perceptible diffusion as well as reflection (when the surface is smooth enough, anyway).

To compound this, some objects have a gloss or clear coat on them, which is technically transparent but very reflective; think of the paint job on a brand-new car. Some light reflects off the gloss coat, never making it to the surface underneath.

What does this mean for us? Unfortunately, none of our lighting calculations (our BRDFs) take these kinds of parameters into account quite yet. In a few weeks, when we get to Physically Based Rendering, we'll look at some that do. For now: experiment and use whichever one you think looks best!

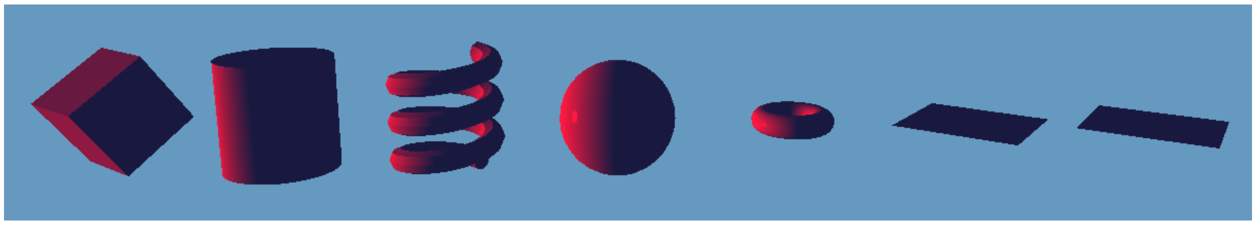
The Reality of Approximation

There are certainly some faults with these classic, but simple, real-time lighting approximations. The reason that they were used for so long is that they're relatively cheap in terms of performance, and the results look pretty good. We're not necessarily going for physically accurate, as much as physically plausible. That being said, there's room for improvement! We'll be getting there soon.

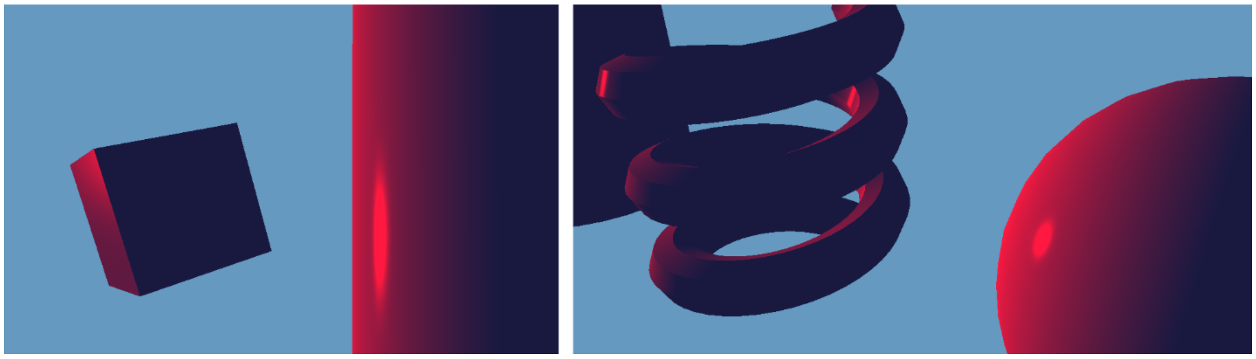
It turns out we're even breaking the laws of physics a bit here, too: a surface shouldn't reflect more light than it receives, but we're never balancing our outgoing light. If light is reflecting, then it's not diffusing. Conservation of energy is another big part of Physically-Based Rendering.

Examples

Same scene setup as before (red light pointing right), now with specular reflections.



Reflections visible on various objects as the camera moves around the scene.



Task 8: More Directional Lights

Add at least **two more directional lights** to your scene. This will require you to:

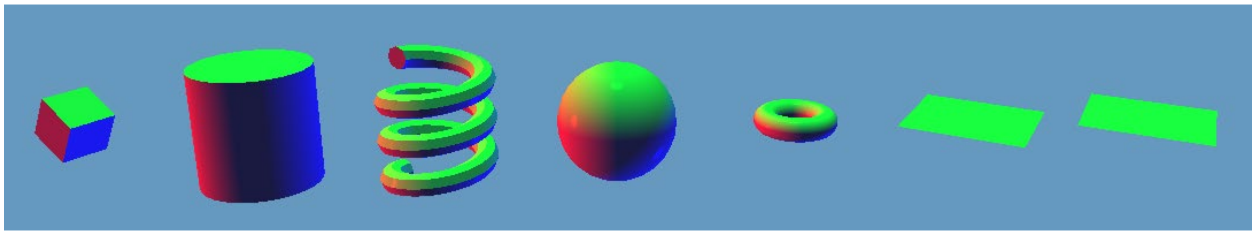
- Create a second and third light in C++ (ideally with obviously different colors and directions)
- Pass those lights to the pixel shader
- Define a second and third light variable in your pixel shader's constant buffer
- Redo the same lighting calculations as before (both diffuse and specular) with each light
- Add the results of all three lights together, along with ambient, for the final pixel color

If you haven't done so already, you could (and should) create a helper function to handle a single directional light. Since light is additive, you can simply call that method three times, adding the results. Don't forget to include ambient at the end.

See the last page of this document for a way to simplify dealing with multiple lights.

Example

Three directional lights (red, green and blue) pointing in different directions.



Task 9: Point Lights

Following many of the same steps as described for handling directional lights, now add support for point lights. You need **at least two point lights** for this assignment. The lighting calculations (diffuse and specular) are *identical* for point lights. The only differences are attenuation and the light's direction.

Point Light Direction

Point lights are defined by a *position* in world space, but you need a *direction* to perform lighting calculations. To remedy this, you'll need to calculate the direction from the surface (the current pixel) to the light. You have both of these positions available in the shader, so calculate the normalized vector between the two points. Pass this into your existing diffuse and specular calculations.

Attenuation

Light in the real world doesn't travel at full intensity infinitely. It attenuates (lessens) the farther it gets from its source. We need a way to account for this, so our point lights properly have zero impact outside of their defined range. If you search online, you'll find a classic computer graphics attenuation function. However, it doesn't attenuate to zero at a finite range, meaning it doesn't help us here. Instead, I want you to use the following range-based attenuation function:

```
float Attenuate(Light light, float3 worldPos)
{
	float dist = distance(light.Position, worldPos);
	float att = saturate(1.0f - (dist * dist / (light.Range * light.Range)));
	return att * att;
}
```

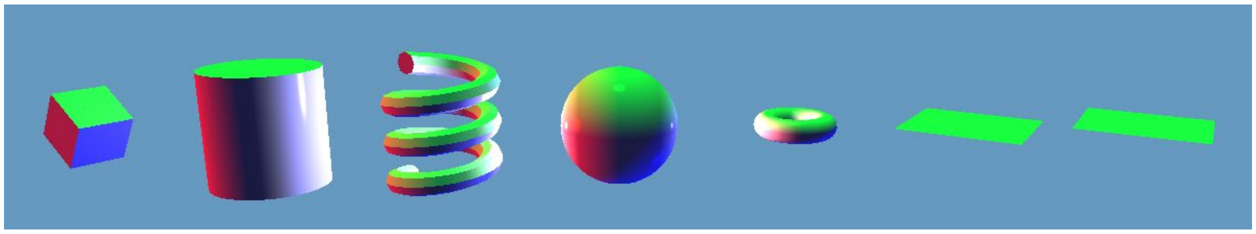
This function will properly return zero if the distance to the light is greater than the light's range, and it has a non-linear falloff which plausibly mimics real attenuation.

Putting it All Together

Once you have the proper direction for a point light, use the same diffuse and specular calculations as before. After you have those results for *this light*, multiply by the attenuation result from the function above. That's the total light for one point light. Do that for each one!

Final Example

All five lights, plus ambient, on several white objects.



Task 10: User Interface

Use ImGui to add controls for some aspect of your lights.

This could be as simple as changing the overall ambient term and the color of each light, or you could get fancier and allow all light data to be altered. This might include the light type, the direction and/or position, range for attenuation, etc. The exact interface is up to you, but you should be able to alter at least one thing (probably color) of *each light*, and changing values should have an obvious impact on the lighting of the scene.

Odds & Ends

Arrays in Constant Buffers?

An enticing thought: can we create an *array of lights* in your constant buffer, allowing us to easily pass in and process multiple lights? You bet! We've even set things up to support this: the Light struct's size is specifically a multiple of 16 bytes, so data packing rules aren't an issue, and we have a "light type" variable in the mix.

You could easily make a helper function that takes in a single light, checks the type and uses a switch statement (or other conditional) to call the proper function for either directional or point lights. Simply loop through the array in the shader, handling one light at a time, adding the results as you go.

The remaining issue, however, is that arrays in constant buffers need to have a constant size when declared. In other words, you must hard-code the number of lights your shader can handle:

```
cbuffer ExternalData : register(b0)
{
	float roughness;
	float3 colorTint;
	float3 ambientColor;
	float3 cameraPosition;

	Light lights[5]; // Array of exactly 5 lights
}
```

Then, in C++, you could have a matching vector<Light> that gets copied all at once (as long as it has no more lights than the shader can handle):

```
ps->SetData("lights", &lights[0], sizeof(Light) * (int)lights.size());
```

There are some ways to make this setup a little more dynamic, but you'll always have a maximum number of lights due to the array being a constant size. A future lecture will discuss ways of handling a more dynamic light loop.

Deliverables

Submit a zip of the entire project to the appropriate dropbox on MyCourses.