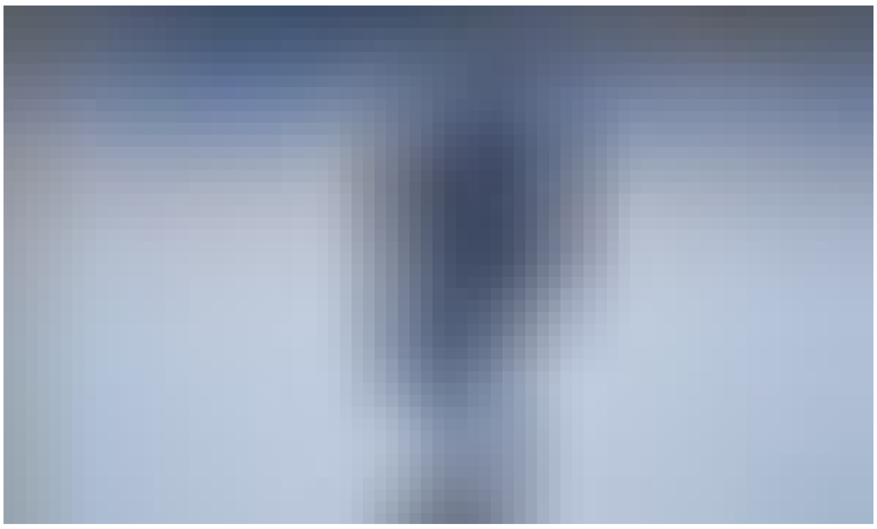
Matt Hoffman Follow
Dec 20, 2017 · 10 min read

Unreal Engine 4 Rendering Part 6: Adding a new Shading Model

(If you haven't read Part 5 in this series, it is available here)

Adding a new Shading Model

Unreal supports several common shading models out of the box which satisfy the needs of most games. Unreal supports a generalized microfacet specular as their default lighting model but has lighting models that support high end hair and eye effects as well. These shading models may not be the best fit for your game and you may wish to tweak them or add entirely new ones, especially for highly stylized games.


A work in progress shading model that uses stepped lighting

Integrating a new lighting model is surprisingly little code but requires some patience (as it will require a (nearly) full compile of the engine and all shaders). Make sure you check out the section on Iteration once you've decided to start making incremental changes on your own as this can help cut down on the ~10 minute iteration times you will find out of the box.

Most of the code in this post is based on the excellent (but somewhat outdated) information by FelixK on their blog series, plus some corrections from the commentators on the various posts. **It is highly encouraged that you read FelixK's blog as well**, as I have skimmed through some of the shader code changes in exchange for explaining more about the process and *why* we're doing it.

There are three different areas of the engine we need to modify to support a new shading model, the material editor, the material itself and the existing shader code. We're going to tackle these changes one area at a time.

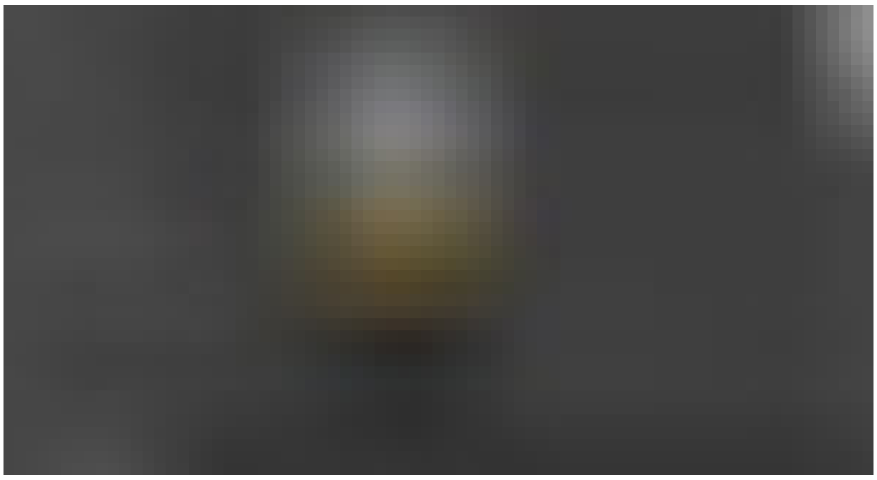
Modifying the Material Editor

Our first stop is the `EMaterialShadingModel` enum inside of *EngineTypes.h*. This enum determines what shows up in the **Shading Model** dropdown inside of the Material Editor. We're going to add our new enum entry `MSM_StylizedShadow` to the enum right before `MSM_MAX`

```
// Note: Check UMaterialInstance::Serialize if changed!
UENUM()
enum EMaterialShadingModel
{
// … Previous entries omitted for brevity
MSM_Eye UMETA(DisplayName="Eye"),
MSM_StylizedShadow UMETA(DisplayName="Stylized Shadow"),
MSM_MAX,
};
```

Enums appear to be serialized by name (if present?) but it's worth adding to the end of the list anyways for any parts of the engine that may serialize them by integer value.

Epic left a comment above the `EMaterialShadingModel` enum warning developers to check the `UMaterialInstance::Serialize` function if we change the enum. It doesn't look like there's anything in there that we need to change if adding a new shading model so we can ignore it and move on. (If you're curious about what that function does, it looks like they did change the order of enum values at one point so the function has some code to fix that up depending on the version of the asset that is being loaded.)



Hooray for easy to add dropdown options!

Having completed this change the new shading model would show up in the *Shading Model* dropdown inside the Material Editor if we were to compile it, but it wouldn't do anything! FelixK uses the `Custom Data 0` pin to allow artists to set the size of the range for light attenuation. We need to modify the code to make the `Custom Data 0` pin enabled for our custom shading model.

Open up *Material.cpp* (not to be confused with the identically named file in the Lightmass project) and look for the `UMaterial::IsPropertyActive` function. This function is called for each possible pin on the Material. If you are trying to modify a *material domain* (such as decal, post processing, etc.) you will need to pay careful attention to the first section of this function where they look at each domain and simply specify which pins should be enabled. If you are modifying the *Shading Model* like we are, then it's a little more complicated—there is a switch statement that returns true for each pin if it should be active given other properties.

In our case, we want to enable the `MP_CustomData0` pin, so we scroll down to the section on `MP_CustomData0` and add `|| ShadingModel == MSM_StylizedShadow` to the end of it. When you change the *Shading Model* to Stylized Shadow this pin should become enabled, allowing you to connect your material graph to it.

```
switch (InProperty)
{
// Other cases omitted for brevity
case MP_CustomData0:
Active = ShadingModel == MSM_ClearCoat || ShadingModel ==
MSM_Hair || ShadingModel == MSM_Cloth || ShadingModel ==
MSM_Eye || ShadingModel == MSM_StylizedShadow;
break;
}
```

It is important to understand that **this code only changes the UI in the material editor**, and you will still need to make sure you use the data that is supplied to these pins inside your shader.

Side Note: `Custom Data 0` and `Custom Data 1` are single channel floating point properties which may or may not be enough extra data for your custom shading model. Javad Kouchakzadeh pointed out to me that you can create brand new pins which will let you choose how the HLSL code gets generated for them. Unfortunately the scope of this is a little beyond this tutorial, but may be the subject of a future tutorial. If you're feeling adventurous, check out *MaterialShared.cpp* for the `InitializeAttributeMap()` function!

Modifying the HLSL Pre-Processor Defines

Once we have modified the Material Editor to be able to choose our new shading model we need to make sure our shaders know when they've been set to use our shading model!

Open up *MaterialShared.cpp* and look for the somewhat massive `FMaterial::SetupMaterialEnvironment(EShaderPlatform Platform, const FUniformExpressionSet& InUniformExpressionSet, FShaderCompilerEnvironment& OutEnvironment) const` function. This function lets you look at various configuration factors (such as properties on your material) and then modify the `OutEnvironment` variable by adding additional defines.

In our particular case we'll scroll down to the section which switches on `GetShadingModel()` and add our `MSM_StylizedShadow` case (from *EngineTypes.h*) and give it a string name following the existing pattern.

```
switch(GetShadingModel())
{
// Other cases omitted for brevity
case MSM_Eye:
OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_EYE"),
TEXT("1")); break;
case MSM_StylizedShadow:
OutEnvironment.SetDefine(TEXT
("MATERIAL_SHADINGMODEL_STYLIZED_SHADOW"), TEXT("1"));
break;
}
```

Now, when the Shading Model for the material is set to `MSM_StylizedShadow` the HLSL compiler will set `MATERIAL_SHADINGMODEL_STYLIZED_SHADOW` as a pre-processor define. This will allow us to later go `#if MATERIAL_SHADINGMODEL_STYLIZED_SHADOW` within the HLSL code to make things that only work on shader permutations that use our shading model.

Review

This concludes the modifications needed to the C++ code. We've added our shading model to the drop down in the editor, we've changed which pins users can plug data into, and we've made sure that the resulting shader can tell when we're in that mode. Compile the engine and get a cup of coffee—modifying *EngineTypes.h* is going to cause a large portion of the C++ code to be recompiled. We don't want to run the editor until we've made changes to the .ush/.usf files though as modifying them will cause all of our shaders to recompile!

Updating the GBuffer Shading Model ID

Now that it is possible to tell when we are building a permutation of the shader that uses our lighting model (via the `MATERIAL_SHADINGMODEL_STYLIZED_SHADOW` we can start making changes to the shaders. The first thing we need to do is write a new Shading Model ID into the GBuffer. This allows the *DeferredLightPixelShader* to know which shading model to try and use when it runs lighting calculations.

Open *DeferredShadingCommon.ush* and there is a section in the middle that starts with `#define SHADINGMODELID_UNLIT` . We're going to add our own Shading Model ID to the end of it, and then update `SHADINGMODELID_NUM` .

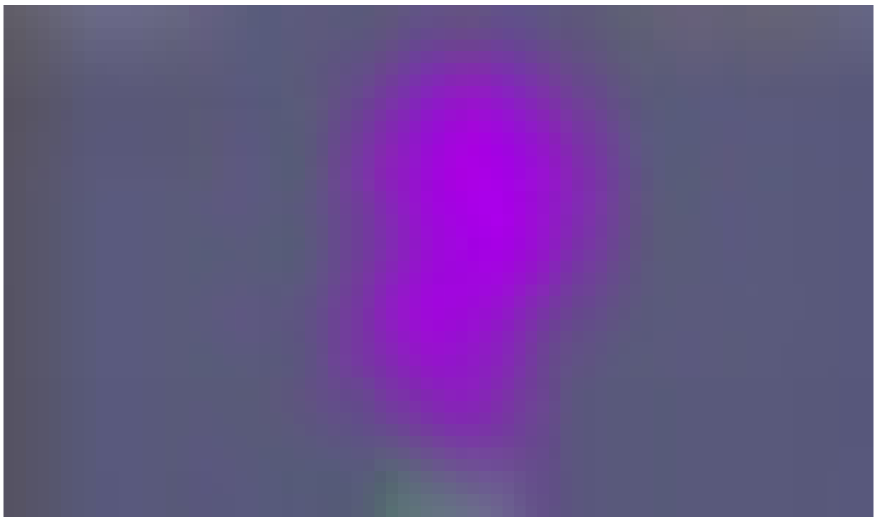
```
#define SHADINGMODELID_EYE 9
#define SHADINGMODELID_STYLIZED_SHADOW 10
#define SHADINGMODELID_NUM 11
```

We'll need to tell the shaders to write this Shading Model ID into the GBuffer, but before we leave this file we should update the *Buffer*

Visualization > Shading Model color so that you can tell which pixels in your scene are rendered with your shading model. At the bottom of the file should be `float3 GetShadingModelColor(uint ShadingModelID)` .

We'll add an entry in both the `#if PS4_PROFILE` section, as well as the `switch(ShadingModelID)` following the existing patterns. We've chosen purple simply because the original tutorial did as well.

```
switch(ShadingModelID)
{
// Omitted for brevity
case SHADINGMODELID_EYE: return float3(0.3f, 1.0f, 1.0f);
case SHADINGMODELID_STYLIZED_SHADOW: return float3(0.4f,
0.0f, 0.8f); // Purple
}
```



float3(0.4f, 0.0f, 0.8f)

Now we need to tell the *BasePassPixelShader* to write the correct ID to the Shading Model ID texture. Open up *ShadingModelsMaterial.ush* and look at the `SetGBufferForShadingModel` function. This function allows each shading model to choose how the various PBR data channels are written to the `FGBufferData` struct. The only thing you *have to do* is ensure GBuffer.ShadingModelID is assigned. If we wished to use the `Custom Data 0` channel from the Material Editor this is where you would query the value and write it into the GBuffer as well.

```
#elif MATERIAL_SHADINGMODEL_EYE
GBuffer.ShadingModelID = SHADINGMODELID_EYE;
// Omitted for brevity
```

```
#elif MATERIAL_SHADINGMODEL_STYLIZED_SHADOW
GBuffer.ShadingModelID = SHADINGMODELID_STYLIZED_SHADOW;
GBuffer.CustomData.x = GetMaterialCustomData0
(MaterialParameters);
#else
// missing shading model, compiler should report
ShadingModelID is not set
#endif
```

We enabled the `Custom Data 0` pin in the Editor earlier by changing the C++ code. Calling `GetMaterialCustomData0(…)` is what actually gets the value and stores it in the GBuffer so that it can be read later in our shading model. If you are using the `CustomData` section of the GBuffer you will need to open *BasePassCommon.ush* and add your `MATERIAL_SHADINGMODEL_STYLIZED_SHADOW` to the end of the `#define WRITES_CUSTOMDATA_TO_GBUFFER` section. This is an optimization that lets Unreal omit writing to or sampling the custom data buffer if the shading model doesn't use it.

Changing Attenuation Calculations

Up until now we've only been focusing on adding a new shading model and taking care of the various boilerplate code needed to add it. Now we're going to look at modifying how light attenuation is calculated when using our shading model. To do this, we're going to open *DeferredLightingCommon.ush* and find the `GetDynamicLighting` function.

Unreal uses the following calculation to determine final light multiplier: `LightColor * (NoL * SurfaceAttenuation)`. The NoL (N dot L) produces a smooth gradient which isn't what we want here. We're going to create a new light attenuation variable and modify the value depending on our shading model. Then we'll update the existing function calls to use our new attenuation variable to avoid duplicating code. Most of the way through the function should be a section that calls the `AreaLightSpecular` function right before two calls to `LightAccumulator_Add` (once to accumulate surface, once to accumulate subsurface). We'll add this block of code:

```
float3 AttenuationColor = 0.f;

BRANCH
if(ShadingModelID == SHADINGMODELID_STYLIZED_SHADOW)
{
float Range = GBuffer.CustomData.x * 0.5f;
AttenuationColor = LightColor * ((DistanceAttenuation *
LightRadiusMask * SpotFalloff) * smoothstep(0.5f — Range,
0.5f + Range, SurfaceShadow) * 0.1f);
}
else
{
AttenuationColor = LightColor * (NoL * SurfaceAttenuation);
}
```

Then we need to replace the call to `LightAccumulator_Add` to use our new `AttenuationColor` variable.

```
// accumulate surface
{
float3 SurfaceLighting = SurfaceShading(GBuffer,
LobeRoughness, LobeEnergy, L, V, N, Random);
LightAccumulator_Add(LightAccumulator, SurfaceLighting,
(1.0/PI), AttenuationColor,
bNeedsSeparateSubsurfaceLightAccumulation);
}
```

You'll notice here that we have to use a dynamic branch (an if statement) instead of a pre-processor define. The pixel shader in *DeferredLightingCommon.ush* is run for each light which can affect multiple objects with multiple shading models; this prevents us from using a pre-processor define so we're forced to use a dynamic branch to check the ID from the GBuffer texture channel.

Changing the Surface Shading

You should also modify the surface shading function now that we've declared a new lighting model. If you do not declare a new surface shading model it will treat the pixels as black, so you need to at least add the case inside the switch function to use the standard shading.

Open up *ShadingModels.ush* and go to the `SurfaceShading` function at the bottom. We'll add a new entry in the switch, and then also declare the function for use.

```
float3 SurfaceShading( FGBufferData GBuffer, float3
LobeRoughness, float3 LobeEnergy, float3 L, float3 V, half3
N, uint2 Random )
{
 switch( GBuffer.ShadingModelID )
 {
  case SHADINGMODELID_UNLIT:
  return StandardShading( GBuffer.DiffuseColor,
GBuffer.SpecularColor, LobeRoughness, LobeEnergy, L, V, N );
  // Others omitted for brevity
  case SHADINGMODELID_STYLIZED_SHADOW:
  return StylizedShadowShading(GBuffer, LobeRoughness, L, V,
N);
 }
}
```

And then we declare our StylizedShadowShading function:

```
float3 StylizedShadowShading( FGBufferData GBuffer, float3
Roughness, float3 L, float3 V, half3 N)
{
 float Range = GBuffer.CustomData.x * 0.5f;
 float3 H = normalize(V+L);
 float NoH = saturate( dot(N, H));
 return GBuffer.DiffuseColor + saturate(smoothstep(0.5f —
Range, 0.5f + Range, D_GGX(Roughness.y, NoH)) *
GBuffer.SpecularColor);
}
```

Supporting Lit Translucency

If we want our shader to work for translucent objects that have to add specific support for that—open *BasePassPixelShader.usf* and find the section with the comment "// Volume lighting for lit translucency" and add your shading model to the #if statement.

```
//Volume lighting for lit translucency
#if (MATERIAL_SHADINGMODEL_DEFAULT_LIT ||
MATERIAL_SHADINGMODEL_SUBSURFACE ||
MATERIAL_SHADINGMODEL_STYLIZED_SHADOW) &&
(MATERIALBLENDING_TRANSLUCENT || MATERIALBLENDING_ADDITIVE)
&& !SIMPLE_FORWARD_SHADING && !FORWARD_SHADING)

Color += GetTranslucencyVolumeLighting(MaterialParameters,
PixelMaterialInputs, BasePassInterpolants, GBuffer,
IndirectIrradiance);
#endif
```

It's a good idea to re-launch the editor at this point and recompile all shaders. If you want to continue tweaking shaders from here it's a good idea to check the **Iteration** article on how to cut down on shader recompile times!

Review

We modified the BasePassPixelShaders so that it writes the correct ID into the GBuffer's ID channel and then we modified Light Attenuation and Surface Shading to use our new shading model by sampling the ID from the GBuffer.

The full modified code for this is available here on GitHub, but you will need access to Unreal Engine's main repo first.

Next Post

In our next tutorial we will cover creating an outline shader by adding a Geometry Shader to the deferred base pass. This covers how to add a shader stage and how to modify the existing base pass shaders to handle going from a Vertex Shader to a Pixel Shader, but also handle Vertex Shader to Geometry Shader to Pixel Shader.