## New shading models and changing the GBuffer

Implementing a Celshading model directly into UE5.1 source. This celshading use a linear color curve atlas to drive all the values. Learn how to set your own shading model, add a render target to GBuffer, storing information in the View struct, creating a global node for material graph. Some tips and tricks We will see restriction about mobile and raytracing.

作者 **0ne3y3** 12月 08, 2022 • 上次更新: **5月 26, 2024**

显示更多



在这个页面上



# SHADING MODELS AND GBUFFER

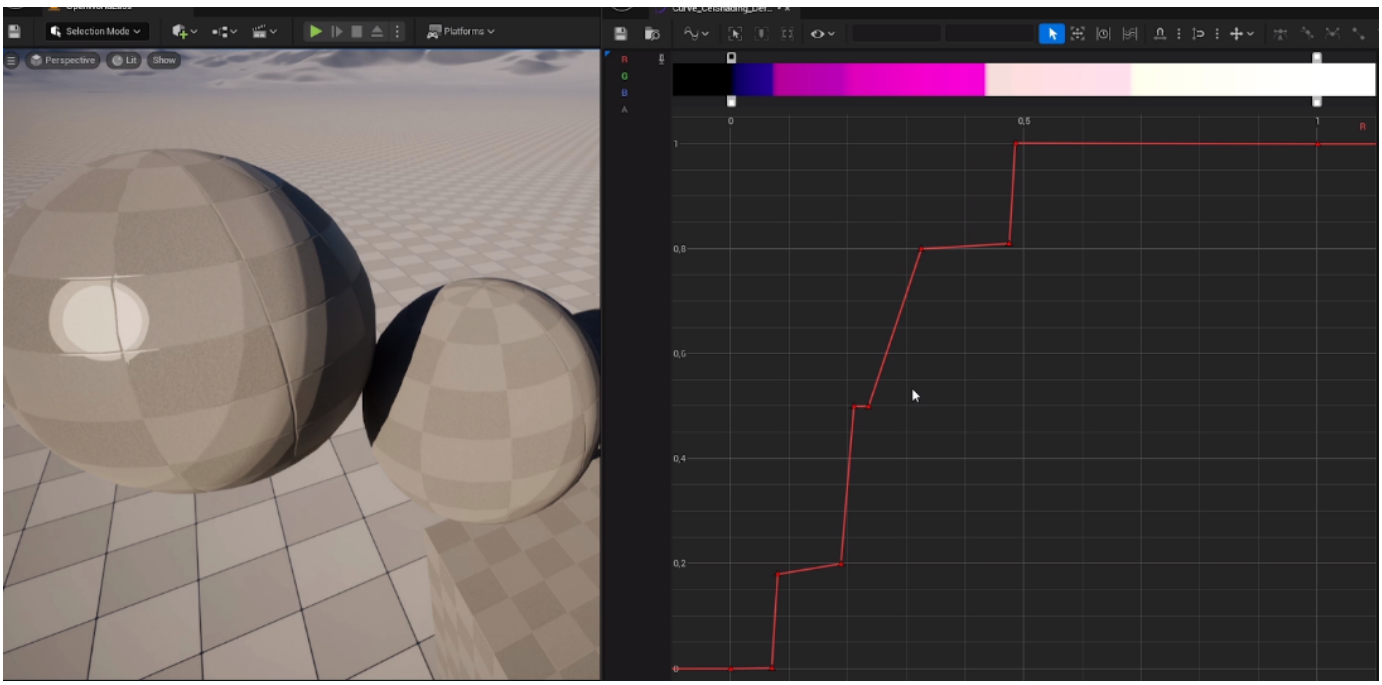
# 1. WELCOME

After seeing a lot of outdated article about this stuff, I decided to make this tutorial to have something a bit more up to date.
I will add 2 disclaimers about this tutorial :
**- We won't talk about forward rendering.**
**- I will give you some lead for the new 5.1 mobile deferred rendering, but I didn't try**

**myself, so you may need to do some search on your own.**

If you are still with me, welcome ! :)
In this article, we learn :

1. How the Gbuffer is generated in C++ with the help of preprocessors. Then add a new render target (Gbuffer scene texture) to store our celshading information. With, unfortunately, a constraint about raytracing.
2. How materials are generated and add 3 new shading models (Celshading default, Celshading subsurface and Celshading skin).
3. Add a new global parameter, sampler and texture to the View struct.
4. Creating a new output node for material graph.
5. Some tips and tricks and more information.



(Curve drive the Celshading)

# 2. REQUIREMENT

First and foremost, you need to download the source engine from GitHub. It's a private repo, so follow this to access it :
**Sign up to get access to unreal GitHub repo**

Install it, everything is explained on the GitHub (we are using 5.1 in this article).
After this point, I assume you have compiled the whole engine, launched it, verified

everything is fine and created a new blank project from whatever template you want.

# 3. REDUCING SHADER PERMUTATIONS

You should have seen for your first launch in a blank project, UE5 compiles a whole >8k shaders permutations. Normally, you don't need to compile them again, but since we're going to modify global shaders, it's possible it needs to recompile all of them.

Here are some tips to reduce those permutations.

## In your materials :

- In the details panel, you can set the usage of this material for a specific Vertex Factory. If "**Automatically Set Usage in Editor**" is checked, when you use it in a new VF it will generate that permutation automatically. For some big project, it may be nice to disable it, your artist will think twice before applying a material to something.

- Still in material, the Quality, Shader model (SM5/6/etc), Ray-Tracing, Nanite and Static switches will generate permutations, as it needs to compile each possible combination of them. You can imagine Static switches create an enormous amount of permutations as it goes exponential (only 3 statics switches mean 27 permutations). Well actually, switches are only triggering permutations when used. If only 1 material instance use a combination, it means just one permutation.

## In your project settings :

Now, in your Project settings, we can disable some permutations. You may need them later on, but for now as we develop our shaders, let's disable them. If there is a specific feature you must have as your shaders depends on it, keep it enabled.

Go to **Project Settings > Rendering**.

**Lighting**

- Allow Static Lighting

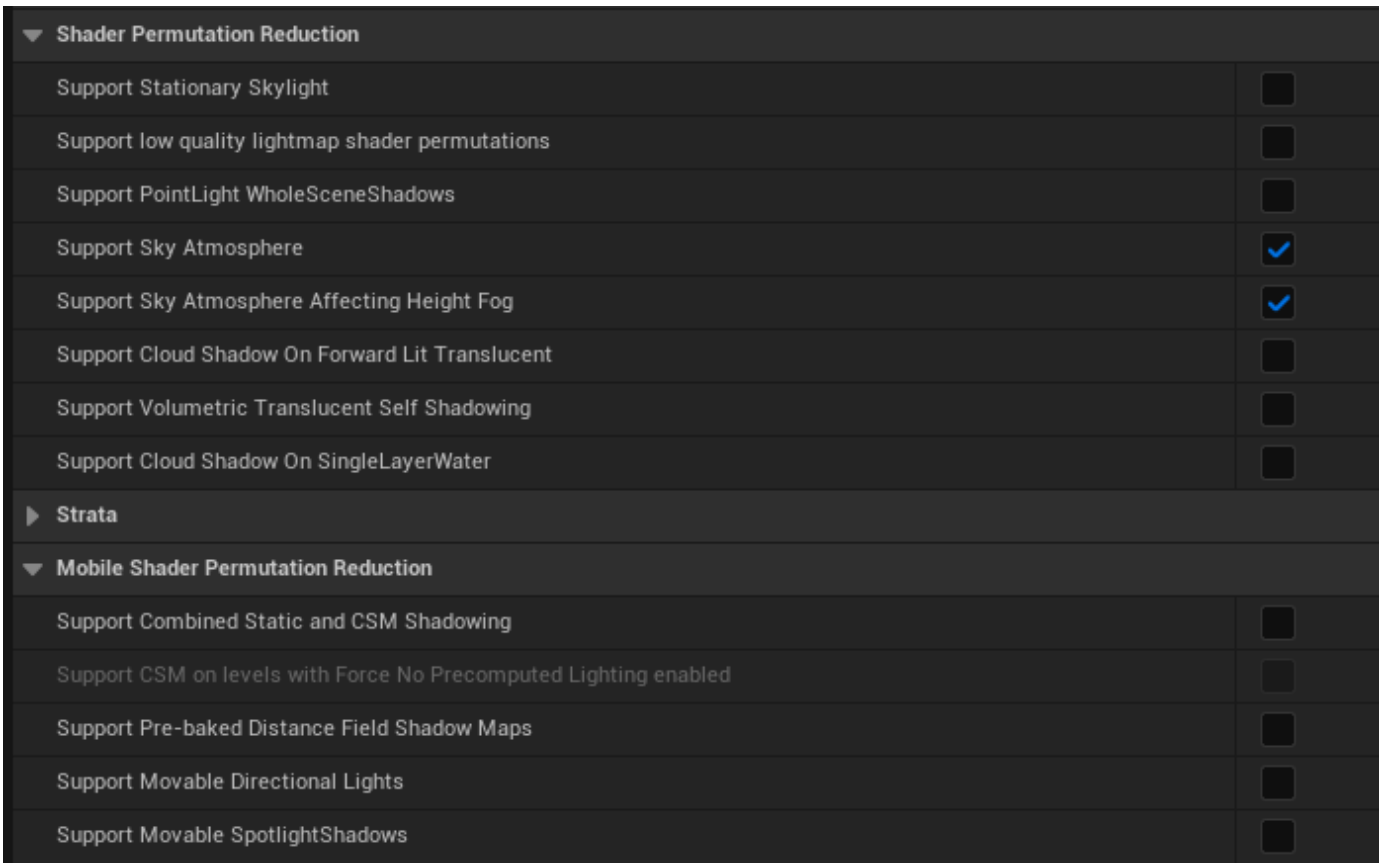
***Note:*** *Disabling this means lights marked as Static will have no influence on your materials. Set them to Movable for testing if this is disabled*

### Shader Permutation Reduction

- Support Stationary Skylight
  ***Note:*** *Disabling this means skylights marked as Stationary will have no influence on your materials. Set them to Movable for testing if this is disabled*
- Support Low Quality Lightmap Shader Permutations
- Support PointLight WholeSceneShadows
- Support Atmospheric Fog
- Support Atmosphere affecting height fog

### Mobile Shader Permutation Reduction

- Support Combined Static and CSM shadowing
- Support pre-baked Distance Field shadows map
- Support movable directional lights
- Support Movable spotlight shadows



(Nearly everything is disabled)

## In your ConsoleVariables.ini :

Now let's change some console variables (CVAR). Open **ConsoleVariables.ini** file. This contains all CVAR executed at the start of your editor. I recommend you to set your console commands here if you keep typing them every time you open Unreal Engine.

Here we're going to add some CVAR :

**r.ForceDebugViewModes=2**

This command disables all debug views and materials (wireframe material, gbuffer debug visualization, etc).
***Note:*** *If you're using Nanite, The engine is going to crash on import / converting a mesh to Nanite. Somehow, it needs the wireframe material to generate collision, which is nullptr in this case.*

**r.Shaders.Optimize=0**

This doesn't help shader compilation time or reduce permutations. But it enables more debug information for RenderDoc.
***Note:*** *remember to enable it when profiling your shader.*

**r.ShaderDevelopmentMode=1**

Get detailed logs on shader compiles. Make debugging easier when a shader can't compile.
***Note:*** *remember to disable it when profiling your shader.*

**r.DumpShaderDebugInfo=1**

Get files saved out to disk for all the shaders that get compiled. Files are saved to **ProjectName/Saved/ShaderDebugInfo**.
(I won't activate this one, but good to know)

- Source files and includes
- A preprocessed version of the shader
- A batch file to compile the preprocessed version with equivalent command line

options to the compiler that were used

***/!\ Important Note: If you leave this setting on, it can fill your HD with many tiny files and folders.***

# 4. RECOMPILING SHADER AT RUNTIME

It is possible to recompile shaders at runtime directly inside Unreal Engine. Even the global one !
This is done by a command. This will search all unreal shaders file (.usf) changed and recompile them :

**RecompileShaders changed**

There is also more, for specific cases :

**RecompileShaders all**
Recompile everything like when you start the editor.

**RecompileShaders global**
Recompile all global shader (.usf).

**RecompileShaders material [name]**
Recompile a specific material.

**RecompileShaders platform [name]**
Recompile changed shaders for a specific platform.

Now we know everything to start. Let's dive into Unreal Engine !

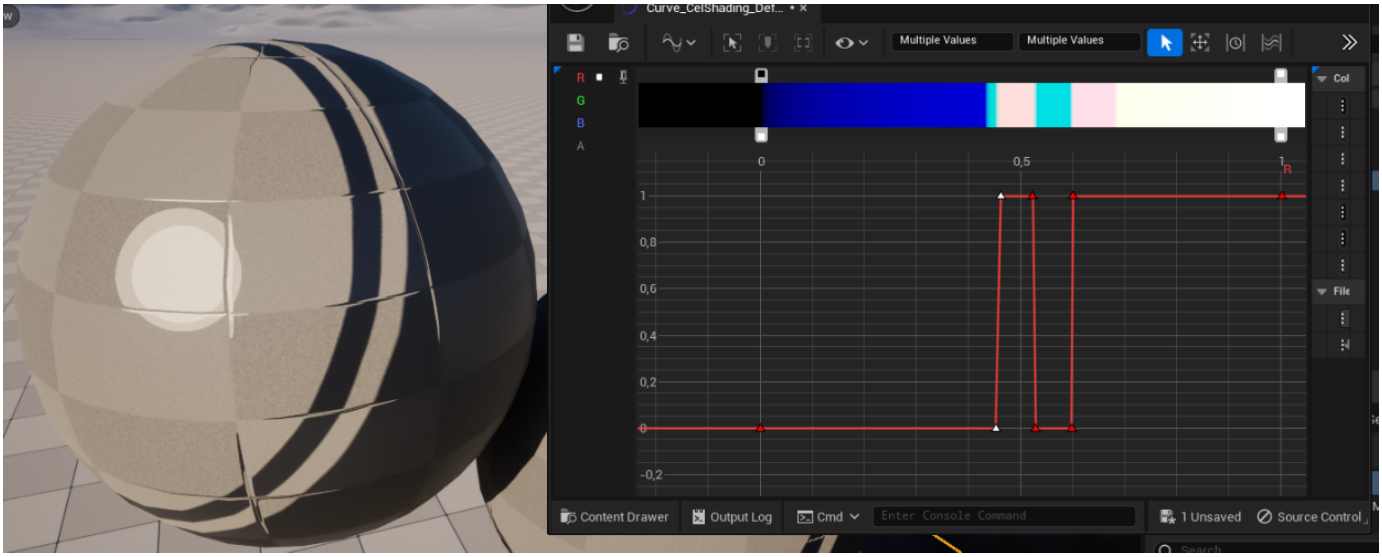
# 5. LINEAR COLOR CURVE ATLAS

For our celshading method, we are going to use a Linear color curve atlas to drive all the values. This atlas stores multiple LUT curves into one texture, and we sample it in our shader to get back our curve value. So we can have multiple curves to choose across all our materials.

From what I learned in this cool GDC Talk from Matt Oztalay is that on a small curve

texture, the cost isn't that big compared to pure math. Of course, it depends on the math and hardware used. **Don't take it as a pure exact science, profiling is always your best friend.**

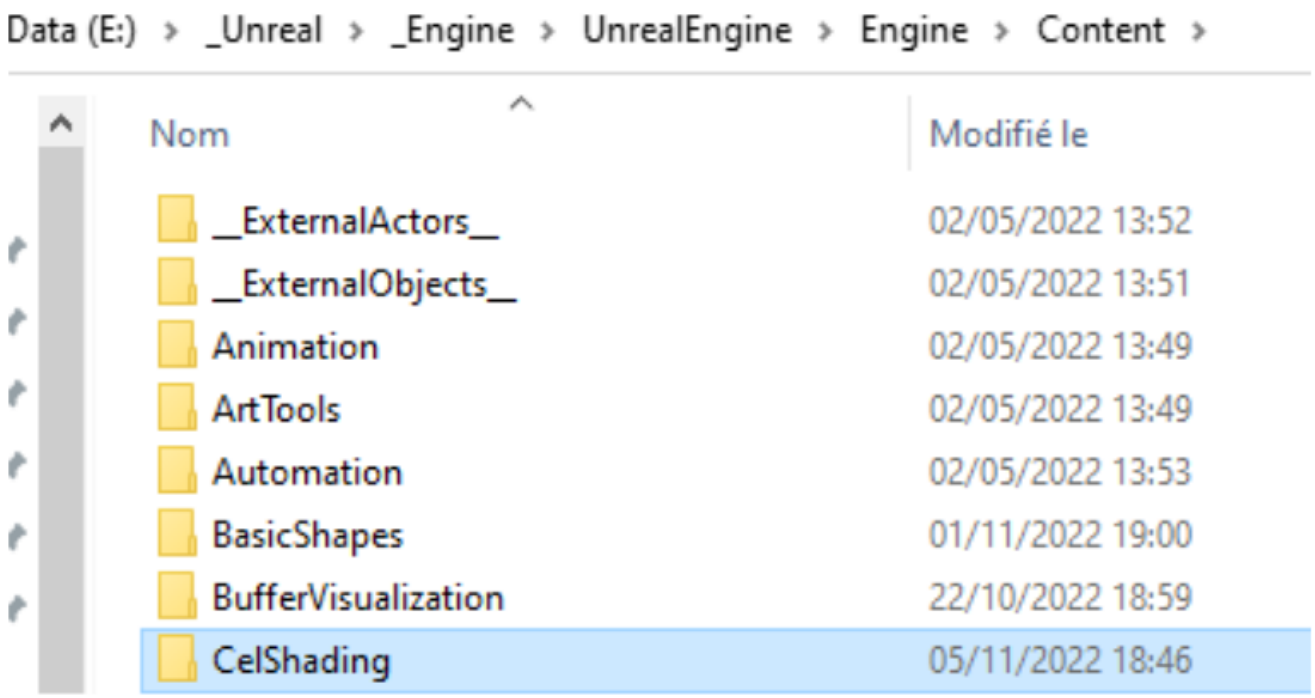
But compared to pure math, your artist / designer will have an absolute control on how light will react. Which give the possibility to make really funky light response :




(Really stylized shadow)

# A. Creating the Linear Color Curve Atlas

First, we need to add a folder to our engine content. This will be our default resources, shared for all our future projects. We add an option to change which atlas to use later, don't worry.
Go to your **Engine/Content** folder and just add a folder "**Celshading**" :




Inside the editor, in your newly created folder, create the **Linear Color Curve Atlas** and some **Linear Color curves** :


(I'm sorry, since I made you change some CVAR, you're going to recompile all shaders again)

# B. A new project settings

You may want to change which curve atlas to use per project. This is why we're going to add a new project settings, following [Tom Looman] tutorial.

In the folder **Engine/Source/Runtime/Engine/Classes/Engine/** , we create a new C++ header (.h) named **CelshadingSettings.h.**

Inside this file will contain our class **UCelshadingSettings**, derived from **UDevelopperSettings**. All **UDevelopperSettings** class are automatically registered inside the project settings menu. Our class look like this :

C++ **UCelshadingSettings header**

展开代码 复制完整片段 (30行长度)

We define our **UCLASS** as an **Engine** configuration, so the changes are saved inside **DefaultEngine.ini**
Here we set as a **defaultconfig**. Which mean our changes are global for all our project. If you need for one atlas for each project, remove this tag.
**DisplayName** is the name of the menu inside **project settings**.

Then we define our member **TSoftObjectPtr**. We use a soft reference, so we don't load the asset when opening the menu (as it's not needed). We use the tag "**config**" to tell this **UPROPERTY** need to be saved in the .ini.

At the end, only in editor, we overwrite the **PostEditChangeProperty()** function. This function is triggered every time a **UPROPERTY** is changed in the editor.

Now we create our CelshadingSettings.cpp inside the folder **Engine/Source/Runtime/Engine/Private/**, this is what it looks like :

C++ **UCelshadingSettings cpp file**

```
#include "Engine/CelshadingSettings.h"
#include "Curves/CurveLinearColorAtlas.h"

UCelshadingSettings::UCelshadingSettings(const FObjectInitializer& ObjectIniti
  : Super(ObjectInitializer)
{

}
```

展开代码 复制完整片段 (29行长度)

Here we just have a default constructor for the class (somehow it's needed, else it doesn't compile).

And our **PostEditChangeProperty()** function. The logic is empty because we need to create our celshading subsystem.

Finally, inside your **Config/BaseEngine.ini** add the following line :
*[/Script/Engine.CelshadingSettings]*
*CelshadingCurveAtlas=/Engine/CelShading*
*/CurveAtlas_DefaultCelShading.CurveAtlas_DefaultCelShading*

Unreal use the format **Variable=Value** for its configuration file, so here we say CelshadingCurveAtlas=PathToYourAtlas.
You can get the relative path of your celshading by right-clicking it and select "**Copy Reference**".

# C. Celshading Engine Subsystem

We need to load and save a reference to our Celshading atlas. To do this, we are using an **Engine Subsystem**. Subsystems works kinda like a classic C++ singleton. But the advantage is that their lifetime is automatically managed by the Engine itself. There is different type of subsystem, and the change is mostly when it is created and destroyed. For more information about that, check the wiki.
We're using an Engine subsystem, so it is created at the start of the engine (and editor) and destroyed at the end of the engine lifetime.

Inside the folder **Engine/Source/Runtime/Engine/Public/Subsystems/** we create our header. We name it **CelshadingSubsystem.h**.

This is what it looks like. I added some comments :

C++ **UCelshadingSubsystem class**

展开代码 复制完整片段 (37行长度)

***Note:*** *you can't create raw* ***UPROPERTY*** *pointer in subsystem. It must be* ***TObjectPtr<>****.*

I don't think it needs more comment, it is mostly a classic class.

Now in for our c++ file. We create it inside **Engine/Source/Runtime/Engine/Private /Subsystems/** and name it **CelshadingSubsystem.cpp**.
This is what it looks like, I also added comments :

C++ **UCelshadingSubsystem cpp file**

展开代码 复制完整片段 (39行长度)

Same here, nothing really heavy to understand. Now we have our subsystem done. We can modify our **PostEditChangeProperty()** inside the Celshading settings. We go back to the **UCelshadingSettings.cpp** file and add our subsystem :

C++ **UCelshadingSettings cpp file**

展开代码 复制完整片段 (21行长度)

**Important note :** If you add C++ file outside the engine like here, you must regenerate your visual studio project file. Doing so will update the UnrealBuildTool and create the **.generated.h** file associated with it. If even after that you get some compilation error because it's missing, you may need to delete the content inside the engine Intermediate folder.

# 6. VIEW UNIFORM SHADER STRUCT

## A. RDG and Proxies

We created our celshading curve atlas on the CPU side. Now we need to send its RHI texture to the GPU, so we will be able to sample it.

Unreal Engine uses the Render Dependency Graph to manage the code generation and

connection between the rendering thread (c++ part) and shaders (HLSL). All the needed resources are automatically created with macros. We use specials structures, for example, this struct :

```
BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters, )
    SHADER_PARAMETER(FVector2D, ViewportSize)
    SHADER_PARAMETER(FVector4, Hello)
    SHADER_PARAMETER(float, World)
    SHADER_PARAMETER_ARRAY(FVector, FooBarArray, [16])

    SHADER_PARAMETER_TEXTURE(Texture2D, BlueNoiseTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState, BlueNoiseSampler)

    SHADER_PARAMETER_TEXTURE(Texture2D, SceneColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState, SceneColorSampler)

    SHADER_PARAMETER_UAV(Texture2D, SceneColorTexture)
END_SHADER_PARAMETER_STRUCT()
```

Becomes like this in HLSL :

```
float2 ViewportSize;
float4 Hello;
float World;
float3 FooBarArray[16];

Texture2D BlueNoiseTexture;
SamplerState BlueNoiseSampler;

Texture2D SceneColorTexture;
SamplerState SceneColorSampler;

RWTexture2D<float4> SceneColorOutput;
```

I won't talk about RDG in general, please read the documentation or **Froyok**'s articles (link

in **TIPS & STUDIES** chapter). Instead of creating a shaders struct from scratch, we use an already made struct which is perfect for our purpose. This is the **FViewUniformShaderParameters** struct.

This struct contains all the common variables for our rendering. Like skylight parameters, hair parameters, landscape settings, etc... the list is quite big.

We're going to add our celshading atlas texture here. This **FViewUniformShaderParameters** struct is on the rendering thread, so we can't directly load our celshading atlas from **CelshadingSettings** class (you get an assert, trust me...).

To send resources from the Gamethread to Renderthread, Unreal use what we call : a proxy. Proxies are created on the Gamethread, and then sent to the Renderthread. For example, a skylight component use this proxy class :

C++

展开代码 复制完整片段 (44行长度)

This class is created when a skylight component is created in the scene. In the constructor, it sends its self reference to the Renderthread via a lamba function and the macro command **ENQUEUE_RENDER_COMMAND** :

C++

```
// [...] There are too many arguments, I removed some for visibility

#if WITH_EDITOR
  , SecondsToNextIncompleteCapture(0.0f)
  , bCubemapSkyLightWaitingForCubeMapTexture(false)
  , bCaptureSkyLightWaitingForShaders(false)
```

展开代码 复制完整片段 (29行长度)

This macro command **ENQUEUE_RENDER_COMMAND** is important to avoid clashing between 2 threads, which would cause an access violation. Every time a property is updated in the component, we call this macro to update it on the Renderthread, like here, we get our scene proxy and send again the reference :

C++

展开代码 复制完整片段 (21行长度)

Now you learned all this, we're not going to do that. :D
I didn't manage to get this right (didn't try much, to be honest). **But this how you must do it or, even better, creating your own shader struct.**

***/!\ Disclaimer:*** *To easily get our celshading atlas, we're just going to get the reference directly from the subsystem.* ***This isn't thread safe****, but so far I never encountered a crash because of that. Most of my crash was due to curves calculation doing weird things when editing them (the error was caught inside curve's c++ file). It should be okay at runtime/game if it's only reading value.*

# B. View Uniform Shader Parameters Struct

We need to create our resources in the **FViewUniformShaderParameters** struct, It will generate our HLSL variables. Go in the file **SceneView.h** and search for :

*BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT_WITH_CONSTRUCTOR(FViewUniformShaderParameters, ENGINE_API)*

We add 3 new parameters at the end :

C++

```
1 //Celshading Lucas : Add necessary ressources to view structure
2 SHADER_PARAMETER_TEXTURE(Texture2D, CelshadingAtlas)
3 SHADER_PARAMETER_SAMPLER(SamplerState, CelshadingSampler)
4 SHADER_PARAMETER(uint32, CelshadingTextureHeight)
```

复制完整片段 (4行长度)

Our atlas RHI texture, its sampler and a global uint for the texture height (used later to determine which curve to choose).

Now we need to initialize our values in the constructor. It's **MANDATORY** because the texture and values won't be sent at the start of the engine and the Renderthread really doesn't like nullptr (the engine will just crash).

In the file SceneManagement.cpp, go to the constructor **FViewUniformShaderParameters::FViewUniformShaderParameters()** and add at the end :

C++

```
    FRHITexture* BlackVolume = (GBlackVolumeTexture &&  GBlackVolumeTexture->Tex
    FRHITexture* BlackUintVolume = (GBlackUintVolumeTexture &&  GBlackUintVolume
    check(GBlackVolumeTexture);
  [...]

    //Celshading Lucas: Initialize Celshading member of FViewUniformShaderParame
```

展开代码 复制完整片段 (14行长度)

We use the template to create a global sampler using **point sampling**, we have to use point sampling because bilinear blends pixels at top and down, so our lines blend into each other. We clamp because we don't want values before 0 and after 1. Finally, we initialize the curve atlas with the global white texture.

Now to update our Celshading atlas we have to go to **SceneRendering.cpp** and search for the function **FViewInfo::SetupUniformBufferParameters()**, add the following lines :

C++

展开代码 复制完整片段 (34行长度)

At the top of the file, don't forget to include the curve atlas and our custom subsystem header :
**#include "Subsystems/CelshadingSubsystem.h"**
**#include "Curves/CurveLinearColorAtlas.h"**

**SetupUniformBufferParameters()** is executed every frame to update multiple values of the View struct, so every change to our atlas will be instantly updated.

Our atlas is ready to be used in shaders. Now we have 2 choices. Create a new gbuffer render target to store our celshading information or using the existing **CustomData**. Both have it's up and down.

# 7. GBUFFER (OPTIONAL)

## A. Warning about this chapter

When doing this engine modification, I wanted to create a new GBuffer render target to store my Celshading information. It was easy at first, but as I dig more into it, after activating renderer option, I came across a lot of problems that are beyond my knowledge.

If you go down this path, I will teach you the basic stuff that work with 90% of the case. The final boss to understand is : **raytracing**.

***/!\ If you add a new GBuffer render target following this tutorial, you won't be able to use :***

- ***Raytracing shadow***
- ***Raytracing skylight***
- ***Mobile deferred rendering (without some changes)***

I didn't manage to make it work with those options. BUT, **you can still activate raytracing and use hardware lumen though**. Which is still cool, I guess...

**For mobile, this is a hardware restriction**, as you can't add more gbuffer render target since it's already fully using the GPU storage. GBuffer aren't saved in memory for mobile to save bandwidth, it's stored directly in the **Tiled GPU memory.** So stick with the buffer **CustomData.**

**/!\ By not using a new Gbuffer render target, you won't be able to do :**

- **Using multiple curve for Subsurface and Skin shading model**

Yes, only that, it's forced to use the curve at index 0 and 1. Now it's your choice to use it or not. If not, skip this whole chapter and directly do to **Shading Models**.

Also, we're going to only use one channel on the new GBuffer scene texture, which is kinda a waste. Unfortunately, it doesn't seem we can create a one channel texture for a Gbuffer (there is no value for that in the **EBufferType** enum). So you may want to think twice before doing this.

# B. How the GBuffer is created ?

I assume if you're reading this, you already know what the Gbuffer is and its work in a deferred light rendering. It is handled and generated the same way as any other resources, with the **RGD** model. The Gbuffer consists of multiple render targets (dynamic textures created at runtime). It is generated from the current view, it's storing each pixel's information.

***Note:*** *the size of the render target =/= to the viewport resolution, in some cases, it may be bigger or smaller.*

I won't go into detail for the whole generation process, but I'll give you the most important functions. It's quite difficult to get which part to modify because there is still the old legacy generation of it. It happens a lot of times, I change something, but it's actually not used after.

## The Gbuffer logic and basepass

What the c++ code is generating ? Well we need to see how the basepass works, UE uses only one basepass for every shading model, vertex factories and blend modes. So when you open the file, you're stormed with a ton of *#if / #endif* preprocessor (Strata didn't help for that :p ). Let's try to find our essential information there.

***Note:*** *UE uses* ***.usf*** *and* ***.ush*** *(**U**nreal **S**hader **F**ile and **U**nreal **S**hader **H**eader) for HLSL. It's the same as plain HLSL but with extra macro.*

In the file **BasePassPixelShader.usf**, the big function **FPixelShaderInOut_MainPS()**.

C++

```
  FVertexFactoryInterpolantsVSToPS Interpolants,
  FBasePassInterpolantsVSToPS BasePassInterpolants,
  in FPixelShaderIn In,
  inout FPixelShaderOut Out)
{

#if INSTANCED_STEREO
  const uint EyeIndex = Interpolants.EyeIndex;
```

展开代码 复制完整片段 (40行长度)

At the start of the basepass. We get our scene view (stereo if VR), and then all vertex / lightmap / pixel information.
We do some calculation (pixel offset, etc..). And then we create a struct **FGBufferData** :

C++

展开代码 复制完整片段 (32行长度)

This struct stores all our data after sampling the Gbuffer's render targets in others passes (lighting / shading / post process). Here, we use it to encode those data to the GBuffer's render targets.

We'll get back to it when we do the Gbuffer HLSL chapter. Right under the initialization, we give it our material information based on the shading model used, with the function **SetGBufferForShadingMaterial()** :

C++

展开代码 复制完整片段 (24行长度)

Same, we get back to it later for our custom shading model. Then it's doing some shading calculations, for example the diffuse color (base color - metal) :

```
GBuffer.DiffuseColor = BaseColor - BaseColor * Metallic;
```

Finally, at the end, all the data is encoded into MRT textures. MRT Textures are the HLSL equivalent to the GBufferXTextures in the c++ shader struct. We'll see this struct later. This is the function **EncodeGBufferToMRT()** :

C++

展开代码 复制完整片段 (42行长度)

This function is generated from our c++, same as **DecodeGBufferDataDirect()** and other functions to sample a pixel, like **DecodeGBufferDataUV()**. You can take a look in the file ***YourProjectName***|*Intermediate*|*ShaderAutogen*|*PCD3D_SM6*|*AutogenShaderHeaders.ush* to find the generated code (this is generated for each project and each shader models (SM5/SM6)), it looks like this for my generated Gbuffer :

```
void EncodeGBufferToMRT(inout FPixelShaderOut Out, FGBufferData GBuffer, float QuantizationBias)
{
    float4 MrtFloat1 = 0.0f;
    float4 MrtFloat2 = 0.0f;
    uint4 MrtUint2 = 0;
    float4 MrtFloat3 = 0.0f;
    float4 MrtFloat4 = 0.0f;
    float4 MrtFloat5 = 0.0f;
    float4 MrtFloat6 = 0.0f;

    float3 WorldNormal_Compressed = EncodeNormalHelper(GBuffer.WorldNormal, 0.0f);

    MrtFloat1.x = WorldNormal_Compressed.x;
    MrtFloat1.y = WorldNormal_Compressed.y;
    MrtFloat1.z = WorldNormal_Compressed.z;
    MrtFloat1.w = GBuffer.PerObjectGBufferData.x;
    MrtFloat2.x = GBuffer.Metallic.x;
    MrtFloat2.y = GBuffer.Specular.x;
    MrtFloat2.z = GBuffer.Roughness.x;
    MrtUint2.w |= ((((GBuffer.ShadingModelID.x) >> 0) & 0x0f) << 0);
    MrtUint2.w |= ((((GBuffer.SelectiveOutputMask.x) >> 0) & 0x0f) << 4);
    MrtFloat3.x = GBuffer.BaseColor.x;
    MrtFloat3.y = GBuffer.BaseColor.y;
    MrtFloat3.z = GBuffer.BaseColor.z;
    MrtFloat3.w = GBuffer.GenericAO.x;
    MrtFloat4.x = GBuffer.Velocity.x;
    MrtFloat4.y = GBuffer.Velocity.y;
    MrtFloat4.z = GBuffer.Velocity.z;
    MrtFloat4.w = GBuffer.Velocity.w;
    MrtFloat5.x = GBuffer.CustomData.x;
    MrtFloat5.y = GBuffer.CustomData.y;
    MrtFloat5.z = GBuffer.CustomData.z;
    MrtFloat5.w = GBuffer.CustomData.w;
    MrtFloat6.w = GBuffer.CelShading.x;

    Out.MRT[1] = MrtFloat1;
    Out.MRT[2] = float4(MrtFloat2.x, MrtFloat2.y, MrtFloat2.z, (float(MrtUint2.w) + .5f) / 255.0f);
    Out.MRT[3] = MrtFloat3;
    Out.MRT[4] = MrtFloat4;
    Out.MRT[5] = MrtFloat5;
    Out.MRT[6] = MrtFloat6;
    Out.MRT[7] = float4(0.0f, 0.0f, 0.0f, 0.0f);
    Out.MRT[8] = float4(0.0f, 0.0f, 0.0f, 0.0f);
}
```

(The GBuffer.Celshading is the new Gbuffer member :) )

**DecodeGBufferDataDirect()** is the same but reverse, from MRT textures to **FGBufferData** struct.
For sampling a pixel, there are multiple functions, but here's one :

```
// @param PixelPos relative to left top of the rendertarget (not viewport)
FGBufferData DecodeGBufferDataUV(float2 UV, bool bGetNormalizedNormal = true)
{
    float CustomNativeDepth = Texture2DSampleLevel(SceneTexturesStruct.CustomDepthTexture, SceneTexturesStruct_CustomDepthTextureSampler, UV, 0).r;
    int2 IntUV = (int2)trunc(UV * View.BufferSizeAndInvSize.xy * View.BufferToSceneTextureScale.xy);
    uint CustomStencil = SceneTexturesStruct.CustomStencilTexture.Load(int3(IntUV, 0)) STENCIL_COMPONENT_SWIZZLE;
    float SceneDepth = CalcSceneDepth(UV);
    float4 AnisotropicData = Texture2DSampleLevel(SceneTexturesStruct.GBufferFTexture, SceneTexturesStruct_GBufferFTextureSampler, UV, 0).xyzw;

    float4 InMRT1 = Texture2DSampleLevel(SceneTexturesStruct.GBufferATexture, SceneTexturesStruct_GBufferATextureSampler, UV, 0).xyzw;
    float4 InMRT2 = Texture2DSampleLevel(SceneTexturesStruct.GBufferBTexture, SceneTexturesStruct_GBufferBTextureSampler, UV, 0).xyzw;
    float4 InMRT3 = Texture2DSampleLevel(SceneTexturesStruct.GBufferCTexture, SceneTexturesStruct_GBufferCTextureSampler, UV, 0).xyzw;
    float4 InMRT4 = Texture2DSampleLevel(SceneTexturesStruct.GBufferVelocityTexture, SceneTexturesStruct_GBufferVelocityTextureSampler, UV, 0).xyzw;
    float4 InMRT5 = Texture2DSampleLevel(SceneTexturesStruct.GBufferDTexture, SceneTexturesStruct_GBufferDTextureSampler, UV, 0).xyzw;
    float4 InMRT6 = Texture2DSampleLevel(SceneTexturesStruct.GBufferGTexture, SceneTexturesStruct_GBufferGTextureSampler, UV, 0).xyzw;

    FGBufferData Ret = DecodeGBufferDataDirect(InMRT1,
        InMRT2,
        InMRT3,
        InMRT4,
        InMRT5,
        InMRT6,

        CustomNativeDepth,
        AnisotropicData,
        CustomStencil,
        SceneDepth,
        bGetNormalizedNormal,
        CheckerFromSceneColorUV(UV));

    return Ret;
}
```

(GBufferGTexture was the old name I gave it, ours will be GBufferCelshadingTexture)

Anyway, back to the basepass. After encoding **FGbufferData** to the MRT textures, down at the end of the file, we have our **MainPS()** included, the entry point of the basepass :

C++

展开代码 复制完整片段 (12行长度)

Functions order matter in HLSL, so we need to declare our basepass functions before the **MainPS()**, which is why it's included at the end.

If we take a look at **PixelShaderOutputCommon.ush**, the function **MainPS()** is filled with

*#if / #endif* to define which **OutTarget : SVTarget** we need :

C++

展开代码 复制完整片段 (51行长度)

**PIXELSHADEROUTPUT_MRT0** to **MRT7** preprocessors are set in C++ function. So we can get exactly how many render targets we need and avoid empty ones.

Inside **MainPS()**, we get our MRT values from the basepass function **FPixelShaderInOut_MainPS()** and add it to the **OutTargetX** :

C++

展开代码 复制完整片段 (19行长度)

Finally, our render targets are set, other passes can sample them.

A TLDR would be : **MainPS()** start → **PixelShaderInOut_MainPS()** start (basepass) → Get Material information → Encode to MRTs → Encode MRTs To OutTargets → done.

From this, I hope you can understand how the basepass works and how we feed our GBuffer's render targets for others passes. Now let's see in c++ how to enable a "MRT" texture for our Celshading information.

## The Gbuffer generation from c++

To set all those *#ifndef* **PIXELSHADEROUTPUT_MRT0** to **MRT7** we need to take a look at the function **FShaderCompileUtilities::ApplyDerivedDefines()** in the file **ShaderGenerationUtil.cpp**

We first get all our global defines, like the maximum shader model (SM5/SM6) supported for the platform, do we use forward shading, is this virtual texture, is lightmap activated, etc...

C++

展开代码 复制完整片段 (26行长度)

Nothing much important here for us. The next function **CalculateDerivedMaterialParameters()** defines which Gbuffer we need for a given material shading model. But, from the comment under, it's actually not used anymore, since the new Gbuffer refactor (new GBuffer generation), except for some global defines

(**IS_BASE_PASS** and **USES_GBUFFER**):

C++

展开代码 复制完整片段 (15行长度)

Next, we fetch our Gbuffer information with **FetchFullGBufferInfo()**. This is the most important function, as it defines how many render targets we have, and channel bits packing for the GBuffer slots. A Gbuffer slot is the c++ equivalent of a **FGbufferData** HLSL struct member. **FGbufferData** we saw earlier in the basepass.

```
// [...]
    EGBufferLayout Layout = (EGBufferLayout)MaterialDefines.GBUFFER_LAYOUT;
    FGBufferParams Params = FShaderCompileUtilities::FetchGBufferParamsRuntime(P
    FGBufferInfo BufferInfo = FetchFullGBufferInfo(Params);
// [...]
```

Then, we check all Gbuffer slots and get which shading model is writing in it :

C++

```
if (DerivedDefines.USES_GBUFFER)
{
  bTargetUsage[0] = true;
  bool Slots[GBS_Num] = {};
```

展开代码 复制完整片段 (13行长度)

Finally, for each slot, we check if it's initialized correctly and find which MRT index it's writing into (-1 if unused) :

C++

展开代码 复制完整片段 (27行长度)

And that's it. Our Gbuffer is now generated. Now let's modify it.
If you don't want to use a new gbuffer render target, skip this part and go directly at "SHADING MODELS" chapter.

# C. Changing the GBuffer (C++ side)

## Creating a GBuffer texture

To generate Gbuffer's scene textures, Unreal use the shader struct **FSceneTextureUniformParameters** in **SceneTexturesConfig.h**, this will be our first change :

C++

展开代码 复制完整片段 (26行长度)

Then inside the file **SceneTexturesConfig.cpp**, in the function **FSceneTexturesConfig::Init()**, we initialize our celshading GBuffer. Also in the function **FSceneTexturesConfig::GetGBufferRenderTargetsInfo()** right under :

C++

展开代码 复制完整片段 (65行长度)

We get an error, but that’s normal, don’t worry. We haven't implemented the **GBufferCelshading** yet.

Now, we need to initialize texture at the start of the engine. Because, like the View struct, the rendering thread doesn’t like *nullptr*. :)

Go to **SceneTextures.cpp** in the function **SetupSceneTextureUniformParameters()** and add our **GBufferCelshadingTexture** :

C++

展开代码 复制完整片段 (45行长度)

Now it seems UE have another struct to pass parameters from basepass to another later. It's the **FSceneTextureParameters,** in the file **SceneTextureParameters.h :**

C++

展开代码 复制完整片段 (13行长度)

Now we need to refer this new parameter in every function it's used. First in the file **PlanarReflectionRendering.cpp**, in the function **FDeferredShadingSceneRenderer::RenderDeferredPlanarReflections().** This is

the entry point of planar reflections rendering :

C++

展开代码 复制完整片段 (24行长度)

Then we have the function **AddPixelInspectorPass()** in the file **PostProcessBufferInspector.cpp** :

C++

展开代码 复制完整片段 (26行长度)

And lastly in **SceneTextureParameters.cpp**, both getters **FSceneTextureParameters::GetSceneTextureParameters() :**

C++

展开代码 复制完整片段 (42行长度)

Our new texture is now available and initialized. Now we need to create and initialize the **GbufferCelshading**.

## Creating and initializing the GBufferCelshading

First, we need to create our RDG Texture that we reference in the **GBufferCelshadingTexture**.
We need to go in the class **FSceneTextures** in **SceneTextures.h** and declare our **GBufferCelshading** member :

C++

展开代码 复制完整片段 (20行长度)

We already set the reference to our **GBufferCelshadingTexture** earlier. But there is still an error for the struct **ESceneTextureSetupMode**. So let's declare our celshading GBuffer there too.

In the file **SceneRenderTargetParameters.h**, this is what I came with :

C++

展开代码 复制完整片段 (19行长度)

Set my **GbufferCelshading** flag between **GBufferF** and **SSAO**. And add it to the end of **GBuffers** flag.

Now, in the file **SceneRendering.h**, inside the struct **FFastVramConfig**, we add a flag determining if we use fast VRAM or not for this buffer :

C++

展开代码 复制完整片段 (22行长度)

Back to the file **SceneRendering.cpp**. We add a CVAR for our **GBufferCelshading** to tell if we are using fast VRAM or not. By default, we don't, because this is different for each platform, so this set later in your project :

C++

展开代码 复制完整片段 (15行长度)

The global **GBufferCelshading** member is now created and initazed, but there are still a lot of files to change. I don't really dig what those functions do. But better add it there to avoid problems.

For Virtual production, in the file **TextureShareSceneViewExtension.cpp**, the function **FTextureShareSceneViewExtension::ShareSceneViewColors_RenderThread()**. And declare it in the struct **TextureShareStrings::SceneTextures**, file **TextureShareString.h** :

C++

```
        InTextureRef, InView.GPUIndex, &InView.UnconstrainedViewRect);
};
```

展开代码 复制完整片段 (51行长度)

For post process inspector, in the file **PostProcessBufferInspector.cpp**, add a member in the struct **FPixelInspectorParameters** and set in the function **ProcessPixelInspectorRequests()** :

C++

展开代码 复制完整片段 (53行长度)

## Modifying the GBuffer generation

Now we created everything needed for the Gbuffer. Let's dig into the generation of it.

First, we need to add a **GBufferCelshading** binding in the struct **FGbufferBindings** inside the file **GBufferInfo.h** :

C++

```
FGBufferBinding GBufferE;
FGBufferBinding GBufferVelocity;
FGBufferBinding GBufferCelshading;
```

展开代码 复制完整片段 (11行长度)

Along with the struct **FGBufferParams**, this is used to control allocation and slotting of base pass textures. **FGBufferParams** contain layout parameters : platform used, has velocity, has tangent, has static light enabled, etc...

Now we need to add our binding to the getter **FSceneTextures::GetGBufferRenderTargets()** function in the file **SceneTextures.cpp**. But also, in the same file, search the function **FSceneTextures::InitializeViewFamily()** at the end, we have to create our RDG texture if the **GBufferCelshading** is bind to one shading model :

C++

展开代码 复制完整片段 (60行长度)

Back in **GBufferInfo.h**, In the enum **EGBufferSlot**, we set our celshading slot right before **GBS_num** :

C++

```
    GBS_SceneColor, // RGB 11.11.10
    GBS_WorldNormal, // RGB 10.10.10
    GBS_PerObjectGBufferData, // 2
    GBS_Metallic, // R8
    GBS_Specular, // R8
    GBS_Roughness, // R8
    GBS_ShadingModelId, // 4 bits
```

展开代码 复制完整片段 (34行长度)

Gbuffer slots determines how we manage texture channel and bits packing. And for shading models, this is used to kinda tell "*this shading model writes in this slot*". If the slot is not used, discard it, like we saw earlier in the gbuffer theory chapter.

Inside **ShaderGenerationUtil.cpp**, add this slot at the end of the switch in **GetSlotTextName()** function, this will be the name of our member in the HLSL struct **GBufferData** we saw in the previous chapter :

C++

展开代码 复制完整片段 (70行长度)

Now it's time to change the most difficult part. I hope I won't lose you. Let's go to the file **GBufferInfo.cpp** in the function **FetchLegacyGBufferInfo()** :

First, we initialize our **GbufferCelshading** at index -1 (not used). We set the real index later depending on the GBuffer configuration :

C++

展开代码 复制完整片段 (20行长度)

Then we have an if condition for setting the maximum targets our Gbuffer contains. Since we added a new one, we need to increment numbers by one :

C++

展开代码 复制完整片段 (12行长度)

Under this, we finally set our MRT index for our GBuffer and its parameter. I just copied the **CustomData** parameters. Don't forget to increment **TargetSeparatedMainDirLight** and **TargetGBufferE** since we added our **GbufferCelshading** before.

C++

展开代码 复制完整片段 (62行长度)

And right at the end, we have the bits packing for our slot. It determines how our slot is written in memory. First we create a **FGBufferItem**, and set the buffer compression (Slot uses X channels of X bits) and if the bit packing is even, odd or both.

Our celshading information just need one uncompressed 8bits channel. And then, per channel, we set our packing.
If we pass in 2 values, it means we use the entire channel. If we pass in 4 values, it means we also need to pack the bits.

For more information about bit packing, watch the other slots, especially **GBS_ShadingModelId** / **GBS_SelectiveOutputMask**, which are packed together.

C++

```
// [...]

// Celshading Lucas: set slot bit packing
Info.Slots[GBS_Celshading] = FGBufferItem(GBS_Celshading, GBC_Raw_Unorm_8, GBCH_Both);
Info.Slots[GBS_Celshading].Packing[0] = FGBufferPacking(TargetCelshading, 0, 3);

// [...]

}
```

复制完整片段 (9行长度)

GBuffer max target

The max number of render target the GBuffer have is hard coded in C++ and HLSL. For my configuration, I don't use static light / precalculated shadow, so I have one render target available. You may need to increase this number if you need all of them.

In the file **GBufferInfo.h**, the struct **FGBufferInfo** has a *static const* **MaxTargets** member :

C++

```
struct FGBufferInfo
{
static const int MaxTargets = 8; // Increase this if in need of more GBuffer render target.

int32 NumTargets;
FGBufferTarget Targets[MaxTargets];

FGBufferItem Slots[GBS_Num];

};
```

复制完整片段 (10行长度)

That's it for the c++ part.

# D. GBuffer (HLSL side)

Now let's change some HLSL. First, we declare our new GBuffer render target, and its sampler. We also add our new **CelShading** member inside the struct **FGbufferData.** All in the file **DeferredShadingCommon.ush**.

C++

展开代码 复制完整片段 (49行长度)

In **SceneTexturesCommon.ush** file, there is another sampler to define :

C++

展开代码 复制完整片段 (12行长度)

Finally, for deferred decals, inside their **FPixelShaderInOut_MainPS()** function, we need to add a dummy value. I'm not sure if this is useful, but let's add it in case. In the file **DeferredDecal.usf** :

C++

展开代码 复制完整片段 (28行长度)

## GBuffer max target (again)

in the file **Common.ush**, in the struct **FPixelShaderOut**, the MRT array is set manually :

C++

展开代码 复制完整片段 (14行长度)

Then you need to add more defines **PIXELSHADEROUTPUT_MRT** in the file **ShaderOutputCommon.ush**. But also, for each entry point **MainPS()**, you need to add your new MRT output, for writing inside MRT, for example inside **PixelShaderOutputCommon.ush**, to add a new output :

C++

展开代码 复制完整片段 (77行长度)

It should be enough. But...
***Important note: I don't know if there is something special to change about the deferred rendering function in C++ to take account of this new OutputTarget. Maybe it's automated, maybe not.***

## Undefeated final boss : raytracing

As I said at the start of this chapter, if you want raytracing, you're going to need to alter some file. There are 2 major changes to do.
The first change is the raytracing function for sampling Gbuffer scene texture. For some reason, it is not using the new refactored GBuffer. And everything is needed to be added manually. Like the good ol' time. :)

In the file **RayTracingDeferredShadingCommon.ush**, there is the function **GetGBufferDataFromSceneTexturesLoad()**. Here we need to add our sampling for the Celshading scene texture and then add the result to **DecodeGBufferData().** We also have

a function right under **GetGBufferDataFromPayload()**, we add our change there too. I'll explain the "payload" after :

C++

展开代码 复制完整片段 (48行长度)

Now we change the **DecodeGBufferData()**.

C++

展开代码 复制完整片段 (27行长度)

Now this is done, you should be able to activate raytracing and hardware lumen in project settings. But not raytracing shadow / skylight.

This is because of what come next. Raytracing have 2 structs used to store information on what the ray hit. It's called "**Payload**". This is probably the part I did badly, and this is why

it crashes. So you're on your own...

We have 2 structs : **FMaterialClosestHitPayload** and **F** Packed **MaterialClosestHitPayload**. Those structs contain all the information of our material. **FPackedMaterialClosestHitPayload** is used when **TraceRay()** function is called, we can't see the function itself, it seems to be generated from C++. It stores all the material information, then it gets unpacked to **FMaterialClosestHitPayload** and lastly, we use **GetGBufferDataFromPayload()** we changed earlier.

The most difficult one to understand is the packed one. It seems all our information are tightly packed into 64 bytes. And adding more byte make the engine throw an error directX with "invalid argument". For the unpacked payload, I'm sure my change doesn't hurt anything, but for the packed payload, as it's exactly a power of 2, it's most likely intentional.

If a renderer engineer pass by, please enlighten us :)

C++

展开代码 复制完整片段 (60行长度)

And then we have **PackRayTracingPayload()** and **UnpackRayTracingPayload()** functions :

C++

```
FPackedMaterialClosestHitPayload Output = (FPackedMaterialClosestHitPayload)
Output.HitT = Input.HitT;
Output.SetRayCone(RayCone);
Output.RadianceAndNormal[0]  = f32tof16(Input.Radiance.x);
Output.RadianceAndNormal[0] |= f32tof16(Input.Radiance.y) << 16;
Output.RadianceAndNormal[1]  = f32tof16(Input.Radiance.z);
Output.RadianceAndNormal[1] |= f32tof16(Input.WorldNormal.x) << 16;
Output.RadianceAndNormal[2]  = f32tof16(Input.WorldNormal.y);
```

展开代码 复制完整片段 (68行长度)

Maybe we can pack **MipBias** into 4 bytes and use the other 4 for our celshading...

# 8. SHADING MODELS

**Note:** I will add some **#IF USE_NEW_GBUFFER** to separate code from using the new gbuffer render target or not using it, this is not to add in the final code, just take the part you need.

# A. Shading models (C++ side)

## Creating the models

Now it is time for us to create our 2 new shading models. To start, we go to the enum **EMaterialShadingModel** in the file **EngineTypes.h** and add our new shading models right under strata :

C++

```
{
  MSM_Unlit       UMETA(DisplayName="Unlit"),
  MSM_DefaultLit        UMETA(DisplayName="Default Lit"),
  MSM_Subsurface        UMETA(DisplayName="Subsurface"),
```

展开代码 复制完整片段 (30行长度)

This enum is used by the combo box inside materials to choose our shading model.
***Note:*** *Order matters between this enum and the HLSL declaration we do later.*

In the file **MaterialShader.cpp** inside the function **GetShadingModelString()**, we add our cases. In the same file, inside the function **UpdateMaterialShaderCompilingStats()**. There is a condition for all the lit shading model, we add ours at the end. This is to get better instruction count and stats for our celshading models.

C++

展开代码 复制完整片段 (46行长度)

Next, in **ShaderMaterial.h**, we add our defines in the struct **FShaderMaterialPropertyDefines**, This is used later to define which GBuffer slots we write into :

C++

```
// [...]

  uint8 MATERIAL_SHADINGMODEL_DEFAULT_LIT : 1;
  uint8 MATERIAL_SHADINGMODEL_SUBSURFACE : 1;
  uint8 MATERIAL_SHADINGMODEL_PREINTEGRATED_SKIN : 1;
  uint8 MATERIAL_SHADINGMODEL_SUBSURFACE_PROFILE : 1;
  uint8 MATERIAL_SHADINGMODEL_CLEAR_COAT : 1;
```

展开代码 复制完整片段 (24行长度)

Materials has this node to choose a shading model inside the logic, when using **MakeMaterialAttribute**, we also need to add our Celshading ones there. In the file **MaterialExpressionShadingModel.h**, the UPROPERTY **ShadingModel** has hard coded restrictions for which shading model you can choose. We add ours at the end :

C++

展开代码 复制完整片段 (28行长度)

Now we can choose our shading model in the material and the node. But if you choose it, nothing happens because we didn't add them to the HLSL material translator. We have to set up the environment for compiling this shader. In the file **HLSLMaterialTranslator.cpp** search the function **FHLSLMaterialTranslator::GetMaterialEnvironment()** and add :

C++

```
    bool bMaterialRequestsDualSourceBlending = false;

  // [...]

    // Celshading Lucas: Initialize shading model's defines for shaders HLSL Tra
    if (ShadingModels.HasShadingModel(MSM_CS_Default))
    {
      OutEnvironment.SetDefine(TEXT("MATERIAL_SHADINGMODEL_CS_DEFAULT"), TEXT("1
```

展开代码 复制完整片段 (28行长度)

There is also another **GetMaterialEnvironment()** function in the file **MaterialHLSLEmitter.cpp**, which looks like the exact copy of the one above, so add our changes there too.

Now in the file **ShaderGenerationUtil.cpp**, inside the function **FShaderCompileUtilities::ApplyFetchEnvironment()**, we add our new HLSL defines. Under it, there is the function **DetermineUsedMaterialSlots()**, we add our new shading models and tell the engine it writes in all basic slots (base color, specular, roughness, etc...), **CustomData** for subsurface and skin. There is another function in this file, **SetSlotsForShadingModelType()**, but it doesn't seem to be used, no call at all. Just in case, let's add our shading models too.

C++

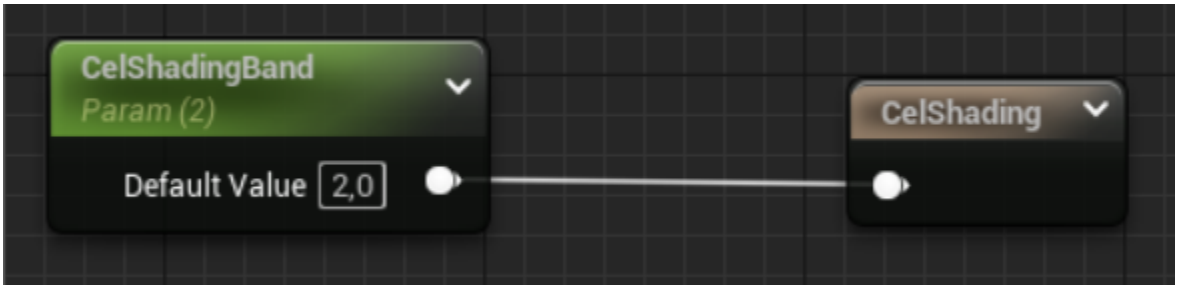
展开代码 复制完整片段 (128行长度)

Now our shading models are all set and they writes into the GBuffer. Let's create the

celshading material node for them.

## Creating a new material custom output node

To pass our values from our material to the GBuffer, we have 2 choices. Using existing material pin (CustomData0) or creating a new custom output node.




We're going to use the latter. Why ? You may ask. Because after choosing the first option (using Customdata0), I saw that the nodes **MakeMaterialAttribute** and **BreakMaterialAttribute** also need to be changed too. And from the words of an acquaintance, it's hell to do so. So let's do it the easy way.

If you check the folder **Engine/Source/Runtime/Engine/Classes/Materials/**, you'll see all the material expression nodes. There is one file for each node.

So we will need to add ours in it. I created a c++ header file and named it **MaterialExpressionCelshadingCustomOutput.h**. To create this node, I copied **BendNormal** configuration, you can find it if you search the class **UMaterialExpressionBentNormalCustomOutput**.

Here what it looks like :

C++

```
#include "Materials/MaterialExpressionCustomOutput.h"
#include "MaterialExpressionCelshadingCustomOutput.generated.h"
```

展开代码 复制完整片段 (29行长度)

First, we include all the needed dependencies. Then we create a new UCLASS(), named **UMaterialExpressionCelShadingOutput** which is a child of the class **UMaterialExpressionCustomOutput**. **UMaterialExpressionCustomOutput** is an amazing class to automatically generate HLSL getters. It will define a new function and add your math logic in it. Then you can just call it in the basepass wherever you want, like any classic material pin.

I think the name of the methods are self-explanatory. The important one is **GetFunctionName()**. This will be the name of your function inside HLSL. So mine will be **GetCelShadingSelection0()**. Since this node can contain multiple pins. It adds the index of them at the end.

Now we need to define the content of these methods. All Material expression nodes are defined in the file **MaterialExpressions.cpp**. I just copied BentNormal. Don't forget to **#include "Materials/MaterialExpressionCelShadingCustomOutput.h"** at the top of the file.

C++

展开代码 复制完整片段 (53行长度)

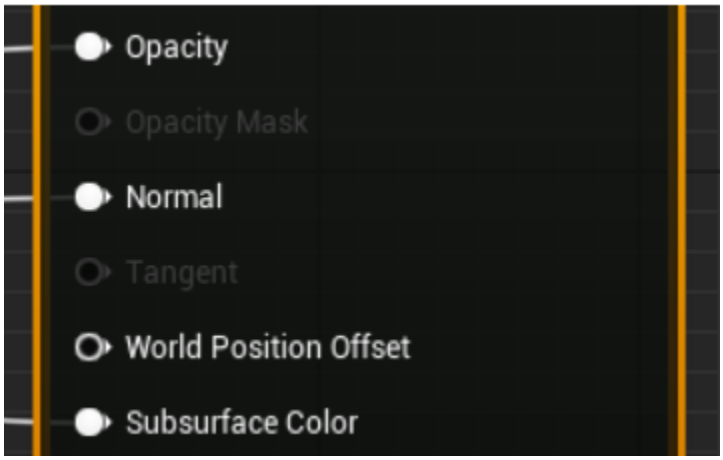
First we have the constructor.
Then, when the material compiles, it calls this function **Compile()**. We check if the pin is

connected, else we throw an error.
Next, the function **GetCaption()**, this is the name of the node.
And then, **GetInput()** is just a getter of input. Here I only have one, so I just return the Input member.

**Reminder :** Don't forget to regenerate your Visual Studio project file.

## Subsurface color and opacity pins

For our Celshading subsurface model, we need to unlock the subsurface color and opacity pins. It’s okay to use them because they’re available in the **MakeMaterialAttribute** node.




We activate them inside the file **Material.cpp**, in the function **IsPropertyActive_Internal()** :

C++

展开代码 复制完整片段 (39行长度)

For **Opacity**, it seems to be handled with a getter **IsSubsurfaceShadingModel()**. We add our subsurface Celshading in it (**MaterialShared.h**) :

C++

```
inline bool IsSubsurfaceShadingModel(FMaterialShadingModelField ShadingModel)
{
return ShadingModel.HasShadingModel(MSM_Subsurface) || ShadingModel.HasShadingModel(MSM_PreintegratedSkin) || ShadingModel.HasShadingModel(MSM_SubsurfaceProfile) || ShadingModel.HasShadingModel(MSM_TwoSidedFoliage) || ShadingModel.HasShadingModel(MSM_Cloth) || ShadingModel.HasShadingModel(MSM_Eye) || ShadingModel.HasShadingModel(MSM_CS_Subsurface) || ShadingModel.HasShadingModel(MSM_CS_Skin);
}
```

复制完整片段 (6行长度)

## Good to know about pins

This section is just for information, we're not doing any change. If you need to rename a pin (like **customdata0**) to a better name, it is inside the file **MaterialAttributeDefinitionMap.cpp** and the function **FMaterialAttributeDefinitionMap::GetAttributeOverrideForMaterial()** :

C++

展开代码 复制完整片段 (25行长度)

If you want to borrow a pin like **Refraction**, it will be available yes, but nothing is written in it. This is because when translating materials, UE will check if there is something redundant or not supported, and then the “unsupported” part will be discarded, or be replaced with the default value. For that you need to change, inside the **HLSLMaterialTranslator.cpp** file, this condition in the function **Translate()** :

C++

展开代码 复制完整片段 (17行长度)

Right now, only translucent and single layer water materials can write into the refraction pin.

We’re done with the c++ part. If you compile, you’ll be able to change the shading model and see the pin activate/deactivate, but of course, everything appears black. Now let’s attack the HLSL side.

# B. Shading Models (HLSL side)

## Initialize the shading model

If you remember, we used the enum **EMaterialShadingModel** for declaring our shading models, where I said it needed to match the HLSL code. In the file **ShadingCommon.ush**

we add our new defines :

C++

展开代码 复制完整片段 (23行长度)

Still in this file, there is a function **GetShadingModelColor()** to get the color when using the debug view “Shading model”, choose the color you want :

C++

展开代码 复制完整片段 (46行长度)

Lastly, in the file **Definitions.usf** we add some undefined defines to 0 as Cg can't handle them :

C++

展开代码 复制完整片段 (12行长度)

We're ready to dig into materials and deferred lighting.

## Feeding the FGBufferData

As seen earlier in the Gbuffer chapter, we call the function **SetGBufferForShadingModel()** to set our data into the GBuffer. Let's open the file **ShadingModelsMaterial.ush**.

Here we have a big if/else condition with all the shading models. Right at the end, we add ours :

C++

展开代码 复制完整片段 (31行长度)

We get our celshading material parameter with **GetCelShadingSelection0()** and divide it by the height of our Celshading curve atlas texture. This will give us the coordinate Y for the UV to sample the texture later.

We need to add our new shading models to some getter in the file **DeferredShadingCommon.ush**. The functions **IsSubsurfaceModel()** and **HasCustomGBufferData()** need to have our subsurface celshading. I also create a quick **IsCelShading()** function under them, to use it later :

C++

展开代码 复制完整片段 (32行长度)

## The deferred light pass

The entry point of this is the function **DeferredLightPixelMain()** which is located in the file **DeferredLightPixelShaders.usf**. Inside this function, the lights are all calculated by the function **GetDynamicLighting()**, which calls **GetDynamicLightingSplit()**, which again calls **AccumulateDynamicLighting()** in the file **DeferredLightingCommon.ush** :

C++

```
  float4 InScreenPosition : TEXCOORD0,
#else
  float2 ScreenUV     : TEXCOORD0,
  float3 ScreenVector   : TEXCOORD1,
#endif
```

展开代码 复制完整片段 (53行长度)

This **AccumulateDynamicLighting()** function is the one used for light and shadow calculation for each light and pixel. It is quite long, so I'm just taking important parts to understand. For one pixel, first we get the light attenuation. If the pixel is outside the light mask, we discard the rest of the calculation because it is useless.

C++

展开代码 复制完整片段 (14行长度)

If the pixel is inside its mask, we get the shadow term. It's the shadow calculation, like if the pixel position is inside a shadow cast by another object (**SurfaceShadow** member) :

C++

```
1 FShadowTerms Shadow;
2 Shadow.SurfaceShadow = AmbientOcclusion;
3 Shadow.TransmissionShadow = 1;
4 Shadow.TransmissionThickness = 1;
5 Shadow.HairTransmittance.OpaqueVisibility = 1;
```

```
const float ContactShadowOpacity = GBuffer.CustomData.a;
GetShadowTerms(GBuffer.Depth, GBuffer.PrecomputedShadowFactors,
GBuffer.ShadingModelID, ContactShadowOpacity, LightData,
TranslatedWorldPosition, L, LightAttenuation, Dither, Shadow);
SurfaceShadow = Shadow.SurfaceShadow;
```

复制完整片段 (8行长度)

Then, the important function **IntegrateBxDF()**. In this function, different shading methods are executed according to the shading model. This calculates the diffuse, specular and transmission color (subsurface). This is where we'll do our celshading calculation :

C++

展开代码 复制完整片段 (21行长度)

And finally, the function **LightAccumulator_AddSplit()** accumulates the lighting result (applying **SurfaceShadow** and **TransmissionShadow** too). It can be called multiple time:

C++

```
Lighting.Specular *= LightData.SpecularScale;

LightAccumulator_AddSplit( LightAccumulator, Lighting.Diffuse,
Lighting.Specular, Lighting.Diffuse, LightData.Color *
SurfaceShadowMultiplier, bNeedsSeparateSubsurfaceLightAccumulation );
LightAccumulator_AddSplit( LightAccumulator, Lighting.Transmission, 0.0f,
Lighting.Transmission, MaskedLightColor * Shadow.TransmissionShadow,
```

```
    bNeedsSeparateSubsurfaceLightAccumulation );
```

复制完整片段 (4行长度)

Now we understand how the light pass works. Let's make our celshading calculation.

## Celshading calculation

Our first change will be inside the **IntegrateBxDF()** function. You can find it in the file **ShadingModels.ush**, this is a big switch with all shading models, let's add ours at the end :

C++

展开代码 复制完整片段 (38行长度)

We declare our 2 new functions **CelShadingDefaultBxDF()**, **CelShadingSubsurfaceBxDF()** and **CelshadingPreintegratedSkinBxDF()** right above, remember, function order matters in HLSL.

***Note:*** *For the default shading, I just copied the DefaultLitBxDF(), this is my own solution because I like UE realistic color and want to go in this art direction. In those functions, you can do whatever you like to make it more stylized / more contrasted / more colorful.*

This section is just to show how I do it, feel free to experiment on your own.

The start of the function is the same as default, getting the context and calculating NoL/NoH/etc values. The first part changing is the **Lighting.Diffuse** calculation :

C++

展开代码 复制完整片段 (17行长度)

Here I create my UV coordinate with the NoL (dot product Normal over Light) for X, and my curve index for Y. Then I use those UVs to sample my CelShading curve atlas.

Right under, we have our specular. I didn't do any change for Anisotropy nor rectangle lights, those will do the default calculation. If you have an idea how to do it, please hit me :)

I used the method from **YiChen Lin**'s tutorial (link in **REFERENCES**). Basically, we do the same calculation as **SpecularGGX()** but instead of returning the final color, we return a scalar. Then we celshade this scalar along with NoL and multiply our final specular color with this celshaded result. Add 2 functions above : **SpecularGGX_ScaleTerm()** and **F_Schlick_ScaleTerm()** :

C++

展开代码 复制完整片段 (18行长度)

The **Lighting.Specular** calculation look like :

C++

展开代码 复制完整片段 (27行长度)

At the end of the function, we have the default PBR energy conservation.

Now for our **CelShadingSubsurfaceBxDF()**, well it's the same as the **SubsurfaceBxDF()**. But instead, I use my **CelshadingDefaultBxDF()** first, to get the diffuse/specular. And then I celshade the **WrappedDiffuse** NoL value. I noticed the surface lighting is cel shaded, but the transmission lighting isn't. And it seems that this NoL can drive the transition between diffuse and subsurface color.

C++

展开代码 复制完整片段 (29行长度)

And that's it. We're done with our celshading calculation. For the skin, I copied **PreintegratedSkinBxDF()** and celshade the *dot(N, L) * .5 + .5 .*
***Note:*** *If you aren't using the new gbuffer. To get the second curve of our atlas, I just did: "* ***float CurveIndex = 1.0f / View.CelshadingTextureHeight*** *". Somehow 1.0f is necessary instead of 1, else it output 0... (weird HLSL thing)*

## Celshading shadow cast by other object

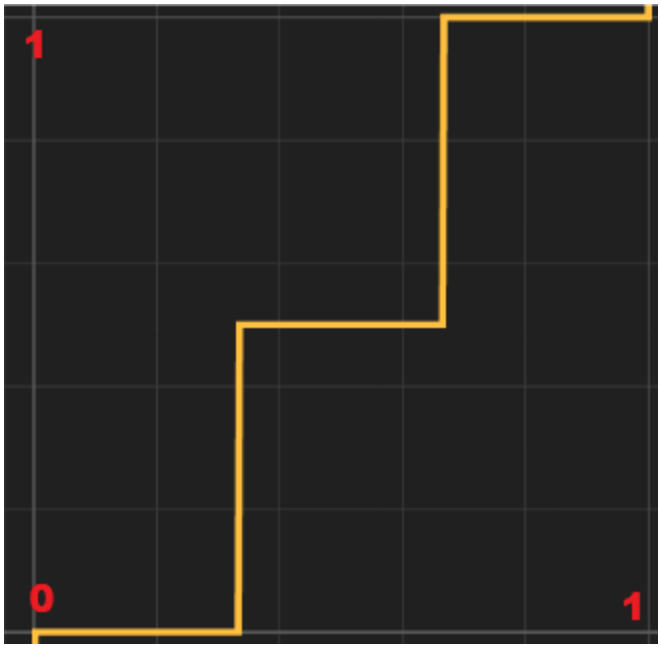
If you start the engine, you see that shadows cast from other objects are still smooth. This is because this value comes from the **SurfaceShadow** member, in the deferred light pass. We're going to celshade this, but using pure math this time. We don't want to bother with sampling a texture because this light pass is done multiple times (number of light the pixel is in), and we're adding our celshading calculation right in it. Anyway, let's go back to the file **DeferredLightingCommon.ush** in the function **AccumulateDynamicLighting()**.

Right after getting the shadow terms, let's add our celshading :

C++

展开代码 复制完整片段 (13行长度)

Those number are hard coded, but it gives a nice 3 bands shadow for a [0,1] range :




And then for all **LightAccumulator_AddSplit()** functions below, we change our surface shadow to **SurfaceShadowMultiplier** :

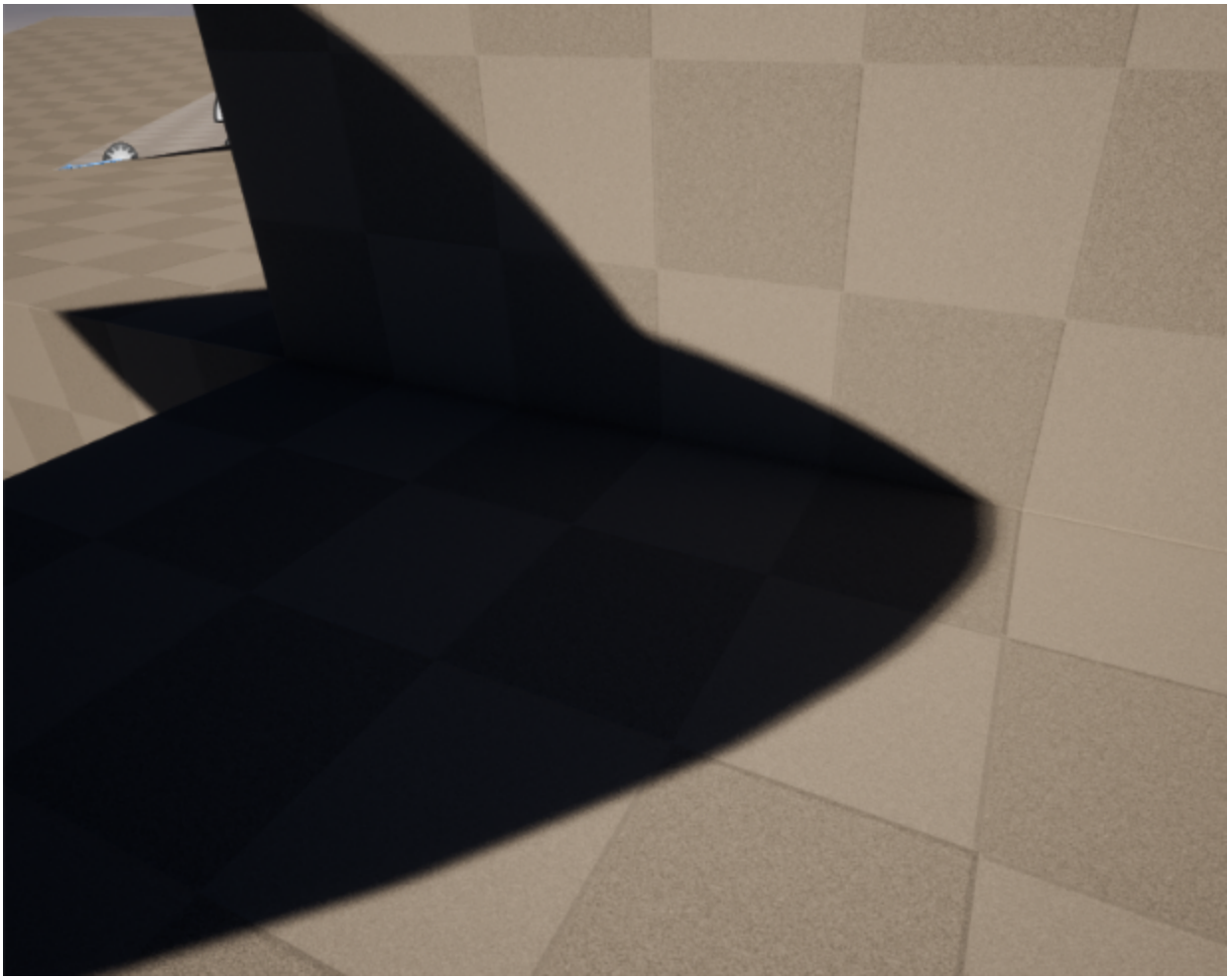
C++

```
1 LightAccumulator_AddSplit( LightAccumulator, Lighting.Diffuse,
  Lighting.Specular, Lighting.Diffuse, LightData.Color *
  SurfaceShadowMultiplier, bNeedsSeparateSubsurfaceLightAccumulation );
2 LightAccumulator_AddSplit( LightAccumulator, Lighting.Transmission, 0.0f,
  Lighting.Transmission, MaskedLightColor * Shadow.TransmissionShadow,
  bNeedsSeparateSubsurfaceLightAccumulation );
```

复制完整片段 (2行长度)

**TransmissionShadow** seems to be something else, I tried to celshade it too, but nothing good happened. :)

You can slightly see the banding when you increase the **Source Angle** of a light. This is why more than 3 bands is kinda useless :



## Subsurface data

Subsurface datas are used a lot across passes. For example, its opacity is used for calculating the thickness of an object. Used in **DistanceFieldShadowingCS()** function, file **DistanceFieldShadowing.usf**

C++

```
BRANCH
if (IsSubsurfaceModel(GBufferData.ShadingModelID))
{
SubsurfaceDensity = SubsurfaceDensityFromOpacity(GBufferData.CustomData.a);
bUseSubsurfaceTransmission = true;
}
```

复制完整片段 (6行长度)

Some conditions aren't resolved with **IsSubsurfaceData()**. My best bet to find all of them

was to do a big search with "**SHADINGMODELID_SUBSURFACE**" or "**MATERIAL_SHADINGMODEL_SUBSURFACE**". Here is a list of file with some condition to change :

- *AmbientCubemapComposite.usf*
- *BasePassPixelShader.usf*
- *ClusteredDeferredShadingPixelShader.usf*
- *DiffuseIndirectComposite.usf*
- *PostProcessGBufferHint.usf*
- *SkyLightingDiffuseShared.ush*
- *LumenCardPixelShader.usf*
- *LumenMaterial.ush*
- *LumenScreenProbeTracing.usf*
- *RaytracingMaterialHitShaders.usf*
- *RaytracingOcclusionRGS.usf*
- *VirtualShadowMapTransmissionCommon.ush*

And with all that, **we're finally done.** I hope I helped you to understand the UE5 rendering pipeline better. Read the **TIPS & STUDIES** section for more information.

# 9. TIPS AND STUDIES

## A. GBuffer and Mobile 5.1 deferred rendering

Since 5.1, Epic introduced a new deferred rendering for Mobile (and switch). The problem is, we are limited of how many Gbuffer render target we can have. Currently locked at 4. So we won't be able to add a new one. Stick to **CustomData**.

Another limitation is the Gbuffer layout :

- **SceneColor**: RG11B10, stores Indirect Lighting + Emissive
- **GBufferA** : RGB10A2, stores encoded normals.xy, ShadingModelID/PerObjectData.
- **GBufferB**: RGBA8, stores Metallic, Specular, Roughness, Indirect Irradiance
- **GBufferC**, RGBA8 stores base color and precomputed shadow

Everything is already packed to use all channel and all bitties.
**Material AO** is not supported in dynamic lighting, and normal blending of decals is broken because of the low precision (10 bits).
Most importantly, if we want to use our shading model, we have to disable static light. To free some space on the buffer to store our **CustomData** and **ShadingModelID / Mask**. This will remove the **precomputed shadow** and **Indirect Irradiance.**

There are 2 new CVARs :

- **r.Mobile.AllowPerPixelShadingModels** : shading models are resolved at pixel level, which means a material can contain multiple shading models. This is enabled by default.
- **r.Mobile.EnabledShadingModelMask** : control which Shading Models are enabled on the mobile terminal. If not enabled, DefaultLit is used instead of the disabled shading model.

***Note:*** *If you are using the new gbuffer render target. You can totally add existing preprocessor* ***#if SHADING_PATH_MOBILE*** *to change behavior in the function* ***SetGBufferForShadingModel()****. Don't forget in C++ to tell the engine that "Celshading default" write into* ***CustomData****.*

# B. Celshading the lumen reflection ?

Following **YiChen Lin** tutorial, it is possible to “hack-celshading” UE reflection. For basic reflection (non lumen) the calculation is in the file **ReflectionEnvironmentPixelShader.usf** at the function ReflectionEnvironment(). It is a little difficult to celshade reflection as it doesn’t do any dot product calculation. This is mostly based on capturing color and intensity of the reflection for the current pixel with NoV (Normal over View).

His method to celshade the reflection is to convert the captured scene color RGB color space to LAB color space, so this will give us a light intensity value in a [0,100] range. Then we celshade this light intensity, and convert back the result to RGB.

In the function **ReflectionEnvironment()**, we add this branch :

C++

展开代码 复制完整片段 (14行长度)

***Note:*** *Since we have access to our GBuffer and View struct, we could probably use an extra channel in our curve atlas to get more flexibility (for example the unused alpha), just don't forget to multiply by 0.01 before sampling and 100 after sampling.*

This give us this flat "painting" look (left is normal reflection, right is celshaded) :

If you make this change, you may see that Lumen reflections are left untouched. This is because Lumen's features (GI and reflection) are done in parallel threads. So how do we

change it ? Well, we go to the file **LumenReflection.usf** at the function **ReflectionResolveCS()**. Here we search the **TonemappedSampleRadiance** variable, this is our sampled scene color.

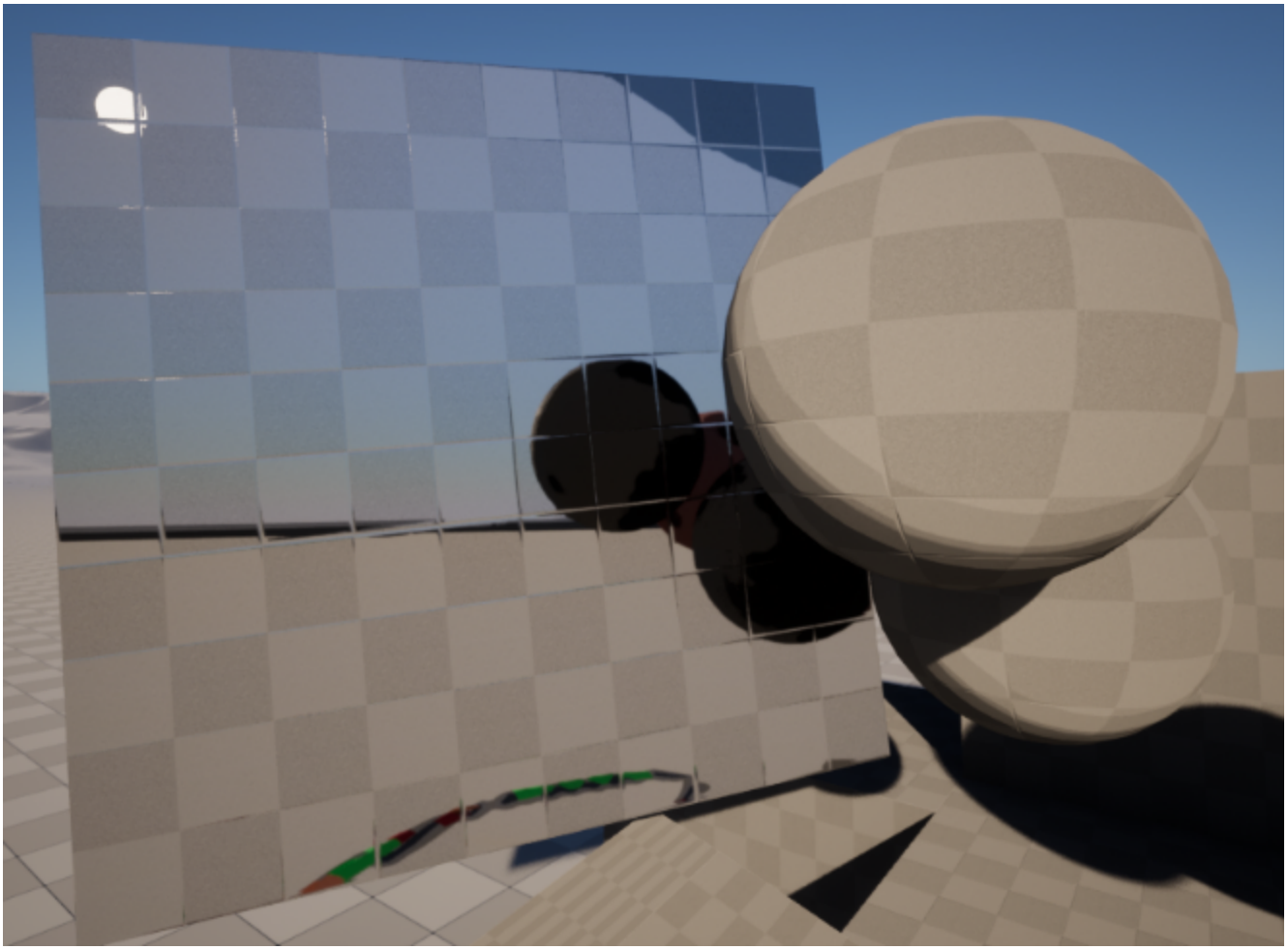
First, we declare our LAB variable at the start. Then under **TonemappedSampleRadiance** declaration, we add the LAB celshading. There is 2 **TonemappedSampleRadiance**. First one is necessary, you can try changing the second one or not, it gives different result (It seems to affect volumetric cloud / fog ?) :

C++

展开代码 复制完整片段 (54行长度)

We have one problem with this technique. Since we can't access our GBuffer and the material shading model, **this change will affect ALL materials**.

This give us this flat color result :



You may have some weird banding, maybe there is some lumen CVAR to change for improving the result ?

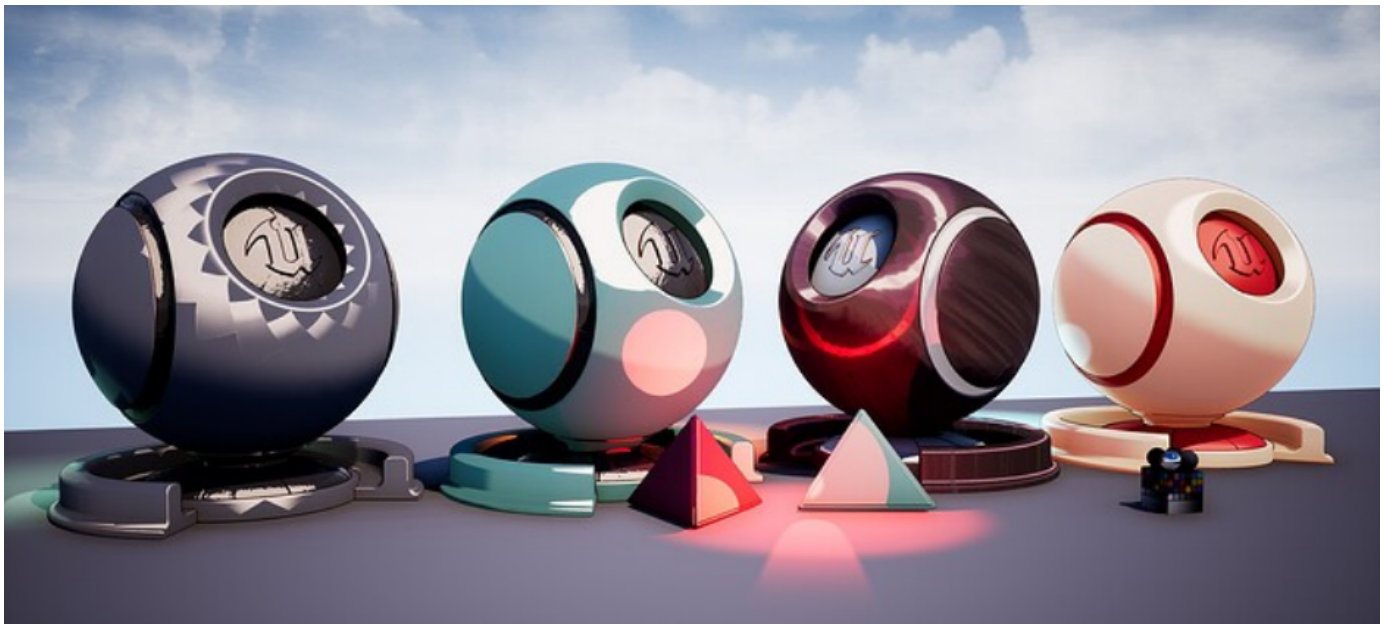
As an effect improvement, I was wondering if it was possible to apply some kind of directional Kuwahara filter during sampling the tonemapped color. This would give us a better paint style for our reflection. Something to dig further.

## Interesting stylized fork and articles

Doing this engine modification made me find some interesting fork about people doing celshading/stylized. The most interesting one is **SelShader** by **Envieous**. You can find his Unreal forum post here.
Their changes are interesting because they manage to apply a shadow color/texture.


261

Another interesting fork from **Doomfest** & **kusogaki77** can be found here. This is only for 4.27, but changes between 4 and 5 are really minimal. Including a lot of celshading models and most importantly a **stylized Anisotropic**. This is interesting to dig for the light calculation and see what algorithm they come up with.



There is not much article about the UE rendering pipeline and how to modify it. The most noticeable is **Froyok**'s article. It is about changing the default lens flare and bloom of Unreal Engine. You can find it here : https://www.froyok.fr/blog/2021-09-ue4-custom-lens-flare/

You can find a lot of useful information about adding a whole new post process pass.

# 10. REFERENCES

All articles that helped me write this.

- Custom shading model for UE4.22, **YiChen Lin** : https://medium.com/@solaslin/learning-unreal-engine-4-implement-cel-shading-w-outline-using-custom-shading-model-in-ue4-22-1-775bccdb9ffb
- Crash course for RGD, **UE official doc** : https://epicgames.ent.box.com/s/ul1h44ozs0t2850ug0hrohlzm53kxwrz
- The whole graphic programming section, **UE official doc** : https://docs.unrealengine.com/5.0/en-US/graphics-programming-for-unreal-engine

- Shader permutation tips, UE4.19, **Lordned** : https://medium.com/@lordned/unreal-engine-4-rendering-part-5-shader-permutations-2b975e503dd4
- Easily Add Custom 'Project Settings' to Unreal Engine (.INI), **Tom Looman** : https://www.tomlooman.com/unreal-engine-developer-settings/

Thanks to people's suggestion to improve this long tutorial. This is the end. See you in space cowboy !

分类: **渲染, 平台和构建** 行业: **游戏** engine source builds