Adding Global Shaders to Unreal Engine

An overview of adding and using your own Global Shaders.

Global Shaders are shaders that are not created using the Material Editor. Instead, they are created using C++, operate on fixed geometry and do not need to interface with materials or a mesh. Sometimes more advanced functionality is required to achieve a desired look and a custom shader pass is necessary to do this.

Some examples of Global Shaders would be rendering post-processing effects, dispatching compute shaders, and clearing the screen.

Unreal Shader Files

Unreal Engine uses **Unreal Shader** (.usf) files to store and read information about the shaders it uses. The source files of any new shaders created need to be stored in the `Engine/Shaders` folder. If a shader is part of a plugin, it should be stored in the `Plugin/Shaders` folder instead.

Use the command **r.ShaderDevelopmentMode=1** in your **ConsoleVariables.ini** file to get detailed logs on shader compiles.

See Shader Development for more information.

Example Global Shader

As an example, we will create a simple pass-through **Vertex Shader** and a **Pixel Shader** that returns a custom Color.

Creating and Adding a New Shader

Create your own shader by creating a new text file in your `Engine/Shaders` folder. Rename its file extension to **.usf** and give it a name. The following example uses **MyTest.usf**.

Next, add the following code your **MyTest.usf** file:

MyTest.usf

```
// Simple pass-through vertex shader

void MainVS(
in float4 InPosition : ATTRIBUTE0,
out float4 Output : SV_POSITION
)
{
Output = InPosition;
}

// Simple solid color pixel shader
float4 MyColor;
float4 MainPS() : SV_Target0
{
return MyColor;
}
```

Copy full snippet

Class Declaration

Now, to get Unreal Engine to recognize and start compiling the shader, you need to declare a C++ class. This example uses the Vertex Shader as that class:

MyTestVS.h

```
#include "GlobalShader.h"

// This can go on a header or cpp file
class FMyTestVS : public FGlobalShader
{
DECLARE_EXPORTED_SHADER_TYPE(FMyTestVS, Global, /*MYMODULE_API*/);

FMyTestVS() { }
FMyTestVS(const ShaderMetaType::CompiledShaderInitializerType& Initializer)
: FGlobalShader(Initializer)
{
}

static bool ShouldCache(EShaderPlatform Platform)
{
return true;
}
};
```

Copy full snippet

There are a few requirements when doing this:

- This is a subclass of `FGlobalShader`. As such, it will end up in the Global Shader Map, meaning it doesn't need a material to find it.
- Usage of `DECLARE_EXPORTED_SHADER_TYPE()` macro generates exports required for serialization of the shader type. The third parameter is a type for external linkage for the code module where the shader module will live, if required. For example, any C++ code that doesn't live in the Renderer Module.
- There are two constructors: the default constructor, and the serialization constructor.
- The `ShouldCache()` function is needed to determine if this shader should be compiled under certain circumstances. For example, you may not want to compile a compute shader on a non-compute shader capable RHI.

Registering a Shader Type

A **Shader Type** is a template or class that is specified by shader code, which maps to a physical C++ class. A Shader Type can be registered to Unreal Engine's list of types using the following code:

```
// This needs to go on a cpp file
IMPLEMENT_SHADER_TYPE(, FMyTestVS, TEXT("MyTest"), TEXT("MainVS"),
SF_Vertex);

```

Copy full snippet

This macro maps the type (`FMyTestVS`) to the .usf file (`MyTest.usf`), the shader entry point (`MainVS`), and the frequency/shader stage (`SF_Vertex`). It also causes the shader to be added to the compilation list, as long as its `ShouldCache()` method returns *true*.

Whichever module you add your `FGlobalShader` to *must* be loaded before the engine starts, or you will get an assert, such as:

```
> `Shader type was loaded after engine init. Use
`ELoadingPhase::PostConfigInit` on your module to cause it to load earlier.`

```

Copy full snippet

After a game or editor has launched, a dynamic module is not allowed to add its own shader type.

Declaring the Pixel Shader

Next, the Pixel Shader is declared using the following code:

```
class FMyTestPS : public FGlobalShader
{
DECLARE_EXPORTED_SHADER_TYPE(FMyTestPS, Global, /*MYMODULE_API*/);

FShaderParameter MyColorParameter;
```

```
6
7  FMyTestPS() { }
8  FMyTestPS(const ShaderMetaType::CompiledShaderInitializerType& Initializer)
9  : FGlobalShader(Initializer)
10 {
11 MyColorParameter.Bind(Initializer.ParameterMap, TEXT("MyColor"),
   SPF_Mandatory);
12 }
13
14 static void ModifyCompilationEnvironment(EShaderPlatform Platform,
   FShaderCompilerEnvironment& OutEnvironment)
15 {
16 FGlobalShader::ModifyCompilationEnvironment(Platform, OutEnvironment);
17 // Add your own defines for the shader code
18 OutEnvironment.SetDefine(TEXT("MY_DEFINE"), 1);
19 }
20
21 static bool ShouldCache(EShaderPlatform Platform)
22 {
23 // Could skip compiling for Platform == SP_METAL for example
24 return true;
25 }
26
27 // FShader interface.
28 virtual bool Serialize(FArchive& Ar) override
29 {
30 bool bShaderHasOutdatedParameters = FGlobalShader::Serialize(Ar);
31 Ar << MyColorParameter;
32 return bShaderHasOutdatedParameters;
33 }
34
35 void SetColor(FRHICommandList& RHICmdList, const FLinearColor& Color)
36 {
37 SetShaderValue(RHICmdList, GetPixelShader(), MyColorParameter, Color);
38 }
39 };
40
41 // Same source file as before, different entry point
42 IMPLEMENT_SHADER_TYPE(, FMyTestPS, TEXT("MyTest"), TEXT("MainPS"),
   SF_Pixel);
43
```

Copy full snippet

In this class, the shader parameter **MyColor** from the .usf file is being exposed:

- The `FShaderParameter MyColorParameter` member is added to the class, which holds information for the runtime to be able to find the bindings. This, in turn, allows the value of the parameter to be set at runtime.
- In the serialization constructor, the `Bind()` function binds the parameter to the `ParameterMap` by name. This *must* match the .usf file's name.
- The `ModifyCompilationEnvironment()` function is used when the same C++ class define for different behaviors, and to be able to set up #define values in the shader.
- The `Serialize()` method is *required*. It is where the compile and cook time information from the shader's binding (matched during the serialization constructor) gets loaded and stored at runtime.
- Lastly, the custom `SetColor()` method shows an example of how to set the `MyColor` parameter at runtime with a specified value.

Writing a Simple Function

The following code writes a simple function to draw a fullscreen quad using the specified Shader Types:

```
1  void RenderMyTest(FRHICommandList& RHICmdList, ERHIFeatureLevel::Type
   FeatureLevel, const FLinearColor& Color)
2  {
3  // Get the collection of Global Shaders
4  auto ShaderMap = GetGlobalShaderMap(FeatureLevel);
5
6  // Get the actual shader instances off the ShaderMap
7  TShaderMapRef MyVS(ShaderMap);
8  TShaderMapRef MyPS(ShaderMap);
9
10 // Declare a bound shader state using those shaders and apply it to the
   command list
11 static FGlobalBoundShaderState MyTestBoundShaderState;
12 SetGlobalBoundShaderState(RHICmdList, FeatureLevel, MyTestBoundShaderState,
   GetVertexDeclarationFVector4(), *MyVS, *MyPS);
13
```

```
// Call our function to set up parameters
MyPS->SetColor(RHICmdList, Color);

// Setup the GPU in prep for drawing a solid quad
RHICmdList.SetRasterizerState(TStaticRasterizerState::GetRHI());
RHICmdList.SetBlendState(TStaticBlendState<>::GetRHI());
RHICmdList.SetDepthStencilState(TStaticDepthStencilState::GetRHI(), 0);

// Setup the vertices
FVector4 Vertices[4];
Vertices[0].Set(-1.0f, 1.0f, 0, 1.0f);
Vertices[1].Set(1.0f, 1.0f, 0, 1.0f);
Vertices[2].Set(-1.0f, -1.0f, 0, 1.0f);
Vertices[3].Set(1.0f, -1.0f, 0, 1.0f);

// Draw the quad
DrawPrimitiveUP(RHICmdList, PT_TriangleStrip, 2, Vertices,
sizeof(Vertices[0]));
}

```

Copy full snippet

Test this in your codebase by clearing a console variable that can be toggled at runtime. Use the following code to do this:

```
static TAutoConsoleVariable CVarMyTest(
TEXT("r.MyTest"),
0,
TEXT("Test My Global Shader, set it to 0 to disable, or to 1, 2 or 3 for
fun!"),
ECVF_RenderThreadSafe
);

void FDeferredShadingSceneRenderer::RenderFinish(FRHICommandListImmediate&
RHICmdList)
{
[...]
// ***
// Inserted code, just before finishing rendering, so we can overwrite the
screen's contents!
```

```
int32 MyTestValue = CVarMyTest.GetValueOnAnyThread();
if (MyTestValue != 0)
{
FLinearColor Color(MyTestValue == 1, MyTestValue == 2, MyTestValue == 3, 1);
RenderMyTest(RHICmdList, FeatureLevel, Color);
}
// End Inserted code
// ***
FSceneRenderer::RenderFinish(RHICmdList);
[...]
}

```

Copy full snippet

Test the functionality of your added console variable by running your project and opening the console window using the tilde (~) key. Then type in the any one of the following commands to set the variable:

- **r.MyTest** with a value of 1, 2, or 3, to change the color.
- **r.MyTest 0** to disable the shader pass.

Additional Notes

- For information on debugging the compilation of your .usf file and/or want to see the processed file, see Debugging the Shader Compiling Process.
- You can modify .usf files while an uncooked game or editor is running for quick iteration. Use the keyboard shortcut **Ctrl + Shift + .** (period), or open the console window and enter the command **recompileshaders changed**, to pick up and rebuild your shaders.