Overview of Shaders in Plugins

Going over creating shaders in Plugins.

Please note that this document **is not** a guide on how to write HLSL code or GPU efficient shaders but merely to show you how to create a new shader using the Plugin system.

Adding new shaders for use in Unreal Engine can now be achieved via the Plugin system. Creating a shader via the Plugin system allows you to quickly and easily share what you have created with the one you want. In the following document, we will take a high-level look at what needs to done to create shaders in plugins.

For additional help, you can directly look at **//Engine/Plugins/Compositing/LensDistortion's** Plugin.

Plugin Creation Tips

When creating a new Plugin, you should be aware of the following things:

- Use the Plugin Wizard to quickly create all of the needed files and folders for your Plugin.

- Right now, it is not possible to make drastic changes, like adding a new shader model, to the Material Editor via Plugins.
- Make sure you add all of your files and folder in the required location and then regenerate your Visual Studio solution files.
- In your ProjectName.uplugin, make sure that the module's LoadingPhase to PostConfigInit (only for the modules that will have shader implementations) like in the following example:

```
{
"FileVersion" : 3,
"Version" : 1,
"VersionName" : "1.0",
"FriendlyName" : "Foo",
"Description" : "Plugin to play around with shaders.",
"Category" : "Sandbox",
"CreatedBy" : "Epic Games, Inc.",
"CreatedByURL" : "http://epicgames.com",
"DocsURL" : "",
"MarketplaceURL" : "",
"SupportURL" : "",
"EnabledByDefault" : false,
"CanContainContent" : true,
"IsBetaVersion" : false,
"Installed" : false,
"Modules" :
[
{
"Name" : "Foo",
"Type" : "Developer",
"LoadingPhase" : "PostConfigInit"
}
]
}

```

Copy full snippet

Render Thread

As opposed to game-side API, the RHI render commands are (and should be) enqueued by a dedicated thread: the rendering thread. The game thread enqueues FIFO (first in, first out) commands through the ENQUEUE_RENDER_COMMAND to the rendering thread. The rendering thread, therefore, can be 0 or one frame behind the game thread. As a matter of CPU performance, the synchronization between them must be avoided at all cost in production runtime. To make sure your plugin's C++ function is called by the right thread, you can add multiple asserts such as `check (IsInGameThread());` or `check (IsInRenderingThread());` for improved threading robustness.

Unreal Shader Files

There are two different shader file types that you need to be aware of when developing new shaders for use in Unreal Engine. Each file has a different purpose which you will find listed below:

- **Unreal Shader Headers (.USH)**
  - Only included by other USH or USF files
- **Unreal Shader Format (.USF)**
  - Should be private data only
    - No backward compatibility is guaranteed in Private directories
  - Should contain shader entry points

Shader File Preprocessing and Virtual File Path

USF shader files, based on HLSL language, is Unreal Engine shader file format that contains the multi platform shader code. To achieve this multi platform support, the engine's shader compiler has added an extra platform-independent source file preprocessing pass before the platform-specific shader compiler (FXC, HLSLCC for GLSL cross compilation, etc.). As a result, all #define and #if are resolved at this very first preprocessing. Of course, each platform has built-in #define to know at shader preprocessing the targeted platform, such as VULKAN_PROFILE.

As same as C/C++ files, you can include usf files with #include "HelloWorld.usf," that would include the file named HelloWorld.usf stored in the same directory as the USF file you have

the #include written in. To avoid multiple includes of the same file, you can add the #pragma once pre processing directive at the top of your file. For instance:

- FooCommon.usf

```
// File shared across all Plugin's shaders
#pragma once

#include "/Engine/Public/Platform.ush"

// ...

```

Copy full snippet

- FooBar.usf

```
// File containing all foobar-related functions and structures.
#pragma once

#include "FooCommon.usf"

// ...

```

Copy full snippet

- FooBarComputeShader.usf

```
// Compute shader that does foobar on the GPU

#include "FooCommon.usf"
#include "FooBar.usf"

// ...
```

Copy full snippet

You can also do this from a Plugin or project module's shader to include a USF file by doing either of the following:

- In the engine with #include `/Engine/<FilePath>`, where `<FilePath>` is a file path relative to `//Engine/Shaders/` directory;
- Or another plugin with #include `/Plugin/<PluginName>/<PluginFilePath>`, where `<PluginName>` is the name of an **activated** Plugin, and `<PluginFilePath>` is a file path relative to the Plugin's `Shaders/` directory. It is the responsibility of the developer to add a dependency to the correct Plugin in the .uplugin.

First Global Shader

Global shaders inherit from the FGlobalShader in the following manner:

```
class FLensDistortionUVGenerationShader : public FGlobalShader
{
public:
// This function determines whether or not the shader should be compiled for a given platform.
// In this case, the shader will not be usable without SM4 support.
static bool ShouldCache(EShaderPlatform Platform)
{
return IsFeatureLevelSupported(Platform, ERHIFeatureLevel::SM4);
}

// Compile-time constants for shader can be defined in this function:
static void ModifyCompilationEnvironment(EShaderPlatform Platform, FShaderCompilerEnvironment& OutEnvironment)
{
FGlobalShader::ModifyCompilationEnvironment(Platform, OutEnvironment);
OutEnvironment.SetDefine(TEXT("GRID_SUBDIVISION_X"), kGridSubdivisionX);
OutEnvironment.SetDefine(TEXT("GRID_SUBDIVISION_Y"), kGridSubdivisionY);
}

// Default constructor.
FLensDistortionUVGenerationShader() {}

// Constructor using an initializer object. We can bind parameters here so that our C++ code
// can interface with USF, enabling us to set shader parameters from code.
FLensDistortionUVGenerationShader(const ShaderMetaType::CompiledShaderInitializerType& Initializer)
```

```
: FGlobalShader(Initializer)
{
PixelUVSize.Bind(Initializer.ParameterMap, TEXT("PixelUVSize"));
RadialDistortionCoefs.Bind(Initializer.ParameterMap,
TEXT("RadialDistortionCoefs"));
TangentialDistortionCoefs.Bind(Initializer.ParameterMap,
TEXT("TangentialDistortionCoefs"));
DistortedCameraMatrix.Bind(Initializer.ParameterMap,
TEXT("DistortedCameraMatrix"));
UndistortedCameraMatrix.Bind(Initializer.ParameterMap,
TEXT("UndistortedCameraMatrix"));
OutputMultiplyAndAdd.Bind(Initializer.ParameterMap,
TEXT("OutputMultiplyAndAdd"));
}

// All members must be serialized here. This function is run at load and
save time, and is used
// to put the shader into the DDC and pak files.
virtual bool Serialize(FArchive& Ar) override
{
bool bShaderHasOutdatedParameters = FGlobalShader::Serialize(Ar);
Ar << PixelUVSize << RadialDistortionCoefs << TangentialDistortionCoefs <<
DistortedCameraMatrix << UndistortedCameraMatrix << OutputMultiplyAndAdd;
return bShaderHasOutdatedParameters;
}

// This function is an example of how shader parameters can be precomputed
based on data specific
// to the shader. In this case, the shader requires several matrices that
can be computed from
// a few parameters, and that would be inefficient to compute inside the
shader itself. Note that
// this function is not an override; it is custom to this class, and is
called when this feature's
// specific implementation requires it.
template<typename TShaderRHIParamRef>
void SetParameters(
FRHICommandListImmediate& RHICmdList,
const TShaderRHIParamRef ShaderRHI,
const FCompiledCameraModel& CompiledCameraModel,
const FIntPoint& DisplacementMapResolution)
{
```

```
FVector2D PixelUVSizeValue(
1.f / float(DisplacementMapResolution.X), 1.f /
float(DisplacementMapResolution.Y));
FVector RadialDistortionCoefsValue(
CompiledCameraModel.OriginalCameraModel.K1,
CompiledCameraModel.OriginalCameraModel.K2,
CompiledCameraModel.OriginalCameraModel.K3);
FVector2D TangentialDistortionCoefsValue(
CompiledCameraModel.OriginalCameraModel.P1,
CompiledCameraModel.OriginalCameraModel.P2);

SetShaderValue(RHICmdList, ShaderRHI, PixelUVSize, PixelUVSizeValue);
SetShaderValue(RHICmdList, ShaderRHI, DistortedCameraMatrix,
CompiledCameraModel.DistortedCameraMatrix);
SetShaderValue(RHICmdList, ShaderRHI, UndistortedCameraMatrix,
CompiledCameraModel.UndistortedCameraMatrix);
SetShaderValue(RHICmdList, ShaderRHI, RadialDistortionCoefs,
RadialDistortionCoefsValue);
SetShaderValue(RHICmdList, ShaderRHI, TangentialDistortionCoefs,
TangentialDistortionCoefsValue);
SetShaderValue(RHICmdList, ShaderRHI, OutputMultiplyAndAdd,
CompiledCameraModel.OutputMultiplyAndAdd);
}

private:
// Shader parameters.
FShaderParameter PixelUVSize;
FShaderParameter RadialDistortionCoefs;
FShaderParameter TangentialDistortionCoefs;
FShaderParameter DistortedCameraMatrix;
FShaderParameter UndistortedCameraMatrix;
FShaderParameter OutputMultiplyAndAdd;
};

// This macro exposes the shader to the Engine. Note the absolute virtual
source file path.
IMPLEMENT_SHADER_TYPE(, FLensDistortionUVGenerationVS,
TEXT("/Plugin/LensDistortion/Private/UVGeneration.usf"), TEXT("MainVS"),
SF_Vertex)
```

Engine/Public/Platform.usf

To have your shader compiling on all Unreal Engine platforms, you are required to include /Engine/Public/Platform.usf in all your shader files (directly or indirectly).

Shader Development Tips

You can customize locally using ConsoleVariables.ini to change some settings in the renderer to accelerate iteration process when writing shaders. For example, the following Console Variables will help you get detailed debug information about what your shader is doing:

- **r.ShaderDevelopmentMode = 1** To get detailed logs on shader compiles and the opportunity to retry on errors.
- **r.DumpShaderDebugInfo = 1** To dump preprocessed shaders in the Saved folder.

> Warning: leaving this on for a while will fill your hard drive with many small files and folders so make sure to disable it when you are done.

Troubleshooting

If you are having issues getting your shader to compile or show up in the Unreal Engine editor, try the following:

- If you get the error **Can't compile:** `/Plugin/<MyPluginName>/<MyFile>` **not found.**

  Make sure to check your Plugin's module's LoadingPhase is set to PostConfigInit, and that there are no typo's in the Plugin's Shaders/ directory name.

- If you get the error **Can't #include** `"/Plugin/<ParentPluginName>/<MyFile>":`

  Make sure to check that the parent Plugin is activated and also check your Plugin dependency as this error means you are missing a Plugin dependency in your .uplugin or .uproject.

Existing Renderer Convention

In the renderer, we tend to have a convention on naming shader classes and shader entry point, especially with a shader domain suffix as shown in the following table:

Shader Domain	Suffix
Vertex Shader	VS
Hull Shader	HS
Domain Shader	DS
Geometry Shader	GS
Pixel Shader	PS
Compute Shader	CS

For example, in a C++ file, the call to **FLensDistortionUVGenerationVS** has VS at the end signaling that it is a Vertex Shader. Inside of a USF file the **void MainVS(...)** ends with a VS signaling that it is going to use the Vertex Shader. When dealing with Struct's in HLSL, the struct name should start with **F** like FBasePassInterpolators for example.

To read more about coding standards in Unreal Engine check out the Unreal Engine Coding Standards document for more information.

Additional Links

The following links contain more information about developing Global Shaders inside Unreal Engine.

- FGlobalShader Base Class
- Debugging the Shader Compiling Process
- Adding Global Shaders to Unreal Engine