

- Developer
- / Documentation
- / Unreal Engine ▾
- / Unreal Engine 5.4 Documentation
- / Designing Visuals, Rendering, and Graphics
- / Graphics Programming
- / Shader Development

Shader Development

Information for graphics programmers writing shaders.



Quick start

When working on shaders, be sure to enable `r.ShaderDevelopmentMode` by setting it to 1. The easiest way is to edit `ConsoleVariables.ini` so it is done every time you load. This will enable retry-on-error and shader development related logs and warnings.

Use **Ctrl+Shift+.**, which executes the *recompileshaders changed* command. Executing this command should be performed after you have saved your changes to the Unreal Shader (.usf) file.



If you change a file that is included in many shaders (such as, `common.usf`), this can take a while. If you want to iterate on a material, you can trigger recompile of the material by making a small change to the material (e.g. move node) and using the 'Apply' in the material editor.

Shaders and Materials

Global shaders

Global shaders are shaders which operate on fixed geometry (like a full screen quad) and do not need to interface with materials. Examples would be shadow filtering, or post processing. Only one shader of any given global shader type exists in memory.

Material and Mesh types

Materials are defined by a set of states that control how the material is rendered (blend mode, two sided, etc) and a set of material inputs that control how the material interacts with the various rendering passes (BaseColor, Roughness, Normal, etc).

Vertex Factories

Materials have to support being applied to different mesh types, and this is accomplished with vertex factories. A `FVertexFactoryType` represents a unique mesh type, and a `FVertexFactory` instance stores the per-instance data to support that unique mesh type. For example, `FGPUSkinVertexFactory` stores the bone matrices needed for skinning, as well as references to the various vertex buffers that the GPU skin vertex factory shader code needs as input. The vertex factory shader code is an implicit interface which is used by the various pass shaders to abstract mesh type differences. Vertex factories consist of mainly vertex shader code, but some pixel shader code as well. Some important components of the vertex factory shader code are:

Function	Description
<code>FVertexFactoryInput</code>	Defines what the vertex factory needs as input to the vertex shader. These must match the vertex declaration in the C++ side <code>FVertexFactory</code> .
<code>FVertexFactoryIntermediates</code>	Used to store cached intermediate data that will be used in multiple vertex factory functions. A common example is the TangentToLocal matrix, which had to be computed from unpacked vertex inputs.
<code>FVertexFactoryInterpolantsVSToPS</code>	Vertex factory data to be passed from the vertex shader to the pixel shader.

Function	Description
VertexFactoryGetWorldPosition	This is called from the vertex shader to get the world space vertex position. For Static Meshes this merely transforms the local space positions from the vertex buffer into world space using the LocalToWorld matrix. For GPU skinned meshes, the position is skinned first and then transformed to world space.
VertexFactoryGetInterpolantsVSToPS	Transforms the FVertexFactoryInput to FVertexFactoryInterpolants, which will be interpolated by the graphics hardware before getting passed into the pixel shader.
GetMaterialPixelParameters	This is called in the pixel shader and converts vertex factory specific interpolants (FVertexFactoryInterpolants) to the FMaterialPixelParameters structure which is used by the pass pixel shaders.

Material Shaders

Shaders using `FMaterialShaderType` are pass specific shaders which need access to some of the material's attributes, and therefore must be compiled for each material, but do not need to access any mesh attributes. The light function pass shaders are an example of `FMaterialShaderType`'s.

Shaders using `FMeshMaterialShaderType` are pass specific shaders which depend on the material's attributes AND the mesh type, and therefore must be compiled for each material/vertex factory combination. For example, `TBasePassVS` / `TBasePassPS` need to evaluate all of the material inputs in a forward rendering pass.

A material's set of required shaders is contained in a `FMaterialShaderMap`. It looks like this:

```

1 FMaterialShaderMap
2 FLightFunctionPixelShader - FMaterialShaderType
3 FLocalVertexFactory - FVertexFactoryType
4 TDepthOnlyPS - FMeshMaterialShaderType
5 TDepthOnlyVS - FMeshMaterialShaderType

```

```
6 TBasePassPS - FMeshMaterialShaderType
7 TBasePassVS - FMeshMaterialShaderType
8 Etc
9 FGPUSkinVertexFactory - FVertexFactoryType
10 Etc
11
```

 Copy full snippet

Vertex factories are included in this matrix based on their **ShouldCache** function, which depends on the material's usage. For example, `bUsedWithSkeletalMesh` being `true` will include the GPU skin vertex factories. `FMeshMaterialShaderType`'s are included in this matrix based on their `ShouldCache` function, which depends on material and vertex factory attributes. This is a sparse matrix approach to caching shaders and it adds up to a large number of shaders pretty quick which takes up memory and increases compile times. The major advantage over storing a list of actually needed shaders is that no list has to be generated, so needed shaders have always already been compiled before run time on consoles. Unreal Engine mitigates the shader memory problem with shader compression, and the compile time problem with multicore shader compilation.

Creating a material shader


A material shader type is created with the `DECLARE_SHADER_TYPE` macro:

```
1
2 class FLightFunctionPixelShader : public FShader {
3     DECLARE_SHADER_TYPE(FLightFunctionPixelShader,Material);
4 }
```

 Copy full snippet

This declares the necessary metadata and functions for a material shader type. The material shader type is instantiated with `IMPLEMENT_MATERIAL_SHADER_TYPE`:

```
1
2 IMPLEMENT_MATERIAL_SHADER_TYPE(,FLightFunctionPixelShader,TEXT("LightFunctionP
3
```



This generates the material shader type's global metadata, which allows us to do things like iterate through all shaders using a given shader type at runtime.

A typical material pixel shader type will first create a `FMaterialPixelParameters` struct by calling the **GetMaterialPixelParameters** vertex factory function. `GetMaterialPixelParameters` transforms the vertex factory specific inputs into properties like `WorldPosition`, `TangentNormal`, etc that any pass might want to access. Then a material shader will call **CalcMaterialParameters**, which writes out the rest of the members of `FMaterialPixelParameters`, after which `FMaterialPixelParameters` is fully initialized. The material shader will then access some of the material's inputs through functions in `MaterialTemplate.usf` (**GetMaterialEmissive** for the material's emissive input for example), do some shading and output a final color for that pass.

Special Engine Materials

UMaterial has a setting called `bUsedAsSpecialEngineMaterial` that allows the material to be used with any vertex factory type. This means all vertex factories are compiled with the material, which will be a very large set. `bUsedAsSpecialEngineMaterial` is useful for:

- Materials used with rendering viewmodes like lighting only.
- Materials used as fallbacks when there is a compilation error (`DefaultDecalMaterial`, `DefaultMaterial`, etc).
- Materials whose shaders are used when rendering other materials in order to cut down on the number of shaders that have to be cached. For example, an opaque material's depth-only shaders will produce the same depth output as the `DefaultMaterial`, so the `DefaultMaterial`'s shaders are used instead and the opaque material skips caching the depth-only shader.

Shader compilation

Unreal Engine compiles shaders asynchronously using a streaming system. Compile requests are enqueued when materials load that do not have a cached shader map, and compile results are applied as they become available, without blocking the engine. This is optimal in

terms of load time and compile throughout, but it does mean that there are quite a few layers between the actual platform shader compile and the material that requested it.

The actual compiling work is done in helper processes called the Shader Compile Workers, because the platform shader compile functions (D3DCompile) often contain critical sections within them that prevent multi-core scaling within a single process.

Debugging shader compilers

There are some settings to control how compilation is done which can simplify debugging of the shader compilers. These can be found in the [DevOptions.Shaders] section of BaseEngine.ini.

Setting	Description
bAllowCompilingThroughWorkers	Whether to launch the SCW to call the compiler DLL's or whether Unreal Engine should call the compiler DLL's directly. If disabled, compiling will be single-core.
bAllowAsynchronousShaderCompiling	Whether compiling should be done on another thread within Unreal Engine.

If you want to step into the shader compiler DLL's directly from Unreal Engine (Compiled3D11Shader for example), you should set both of these to *false*. Compilation will take a long time though, so make sure all other shaders have been cached.

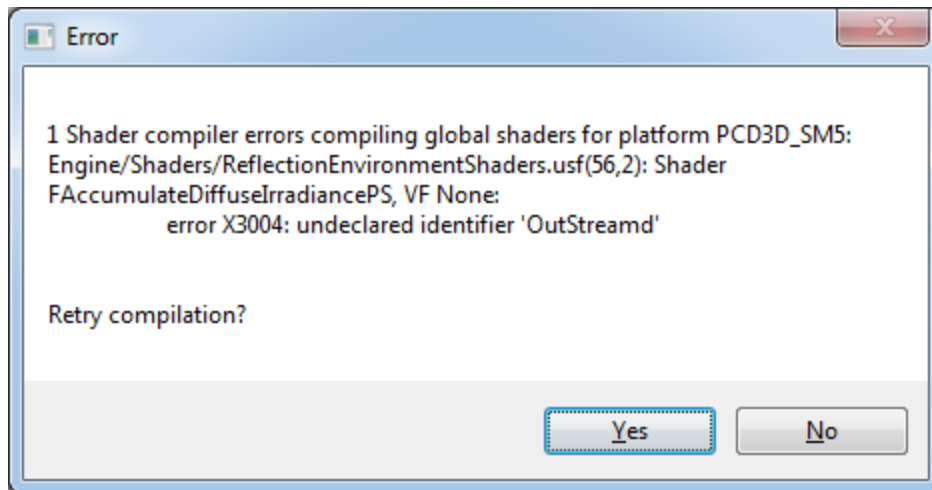
Retrying on compile errors

With r.ShaderDevelopmentMode enabled, you will get the opportunity to retry on shader compile error. This is especially important for global shaders since it is a fatal error if they do not compile successfully.

In debug, with the debugger attached, you will hit a breakpoint and get the compile error in the Visual Studio output window. You can then **Double-click** the error log to be taken directly to the offending line.

```
[2014.03.18-18.05.01:677][ 0]LogShaderCompilers:Warning: 1 Shader compiler errors compiling global shaders for platform PCD3D_SM5:  
Engine/Shaders/ReflectionEnvironmentShaders.usf(56,2): Shader FAccumulateCubeFacesPS, VF None:  
error X3004: undeclared identifier 'OutStreamd'  
UE4Editor-Win64-Debug.exe has triggered a breakpoint.
```

Otherwise you will get a Yes/No dialog



Shader caching and cooking

Once shaders are compiled, they are stored in the Derived Data Cache. They contain, in their key, a hash of all the inputs to the compile, including shader source files. That means that changes to shader source files are automatically picked up every time you re-launch the engine or do a 'recompileshaders changed'.

When you are modifying FShader Serialize functions, there is no need to handle backward compatibility, just add a space to a shader file that is included by that shader.

When cooking assets, material shaders are inlined into the material's package, and global shaders are stored separately in a global shader file which allows them to be loaded early in the engine startup.

Debugging

The primary method of debugging shaders is to modify a shader to output an intermediate, then visualize that with the appropriate VisualizeTexture command. This can allow fast iteration since you can compile on the fly without restarting the engine. For example, you can verify that WorldPosition is correct with something like:

```
1
2 OutColor = frac(WorldPosition / 1000);
3
```

 Copy full snippet

Then verify that the scale is right, and the result is not view dependent. However this method does not scale very well to more complex shaders that build data structures.

Dumping debug info

You can also use `r.DumpShaderDebugInfo=1` to get files saved out to disk for all the shaders that get compiled. It can be useful to set this in `ConsoleVariables.ini` just like `r.ShaderDevelopmentMode`. Files are saved to `GameName/Saved/ShaderDebugInfo`, including

- Source files and includes
- A preprocessed version of the shader
- A batch file to compile the preprocessed version with equivalent commandline options to the compiler that were used



If you leave this setting on, it can fill your HD with many tiny files and folders.

Iteration best practices

If you are working on a global shader, recompiles shaders changed or **Ctrl+Shift+.** is the fastest way to iterate. If the shader takes a long time to compile, you might consider specifying `CFLAG_StandardOptimization` as a compile flag in the shader's `ModifyCompilationEnvironment`.

If you are working on a material shader, like `BasePassPixelShader.usf`, it is much faster to iterate on a single material. Every time you click the Apply button in the material editor, it will re-read shader files from disk and recompile just that material.

Cross compiler

The [HLSL Cross Compiler](#) is used to automatically convert HLSL into GLSL for OpenGL platforms, allowing shaders to be authored only once for all platforms. It is run during offline shader compilation and performs various optimizations to the code that OpenGL drivers are frequently missing.

AsyncCompute

[AsyncCompute](#) is a hardware feature that is available on some APIs that work with certain GPUs. It allows interleaving to better utilize the hardware units in the GPU more efficiently.