Graphics Programming Overview

Information for graphics programmers working with the rendering systems and writing shaders.

Getting Started

There is a lot of rendering code in Unreal Engine so it is hard to get a quick high level view of what is going on. A good place to start reading through the code is `FDeferredShadingSceneRenderer::Render`, which is where a new frame is rendered on the rendering thread. It is also useful to do a profilegpu command and look through the draw events. You can then do a **Find in Files** in Visual Studio on the draw event name to find the corresponding C++ implementation.

- See Shader Development for information on working with shaders.
- See Coordinate Spaces for an explanation of Coordinate Space terminology used in Unreal Engine.

Useful console commands when working on rendering (Usually you get help when using **?** as parameter and the current state with no parameters):

Console Command	Description
stat unit	Shows overall frame time as well as the game thread, rendering thread, and GPU times. Whichever is the

Console Command	Description
	longest is the bottleneck. However, GPU time contains idle time, so is only the bottleneck if it is the longest and stands alone.
Ctrl+Shift+. or **recompileshaders changed**	Recompile shaders that have changed since you last saved the .usf file. This will automatically happen on load.
Ctrl+Shift+; or **profilegpu**	Measure GPU timings for the view being rendered. You can view the results in the UI that pops up or in the engine log.
Vis or **VisualizeTexture**	Visualize the contents of various render targets with the ability to save as bmp.
show x	Toggles specified show flag. Use show to get the list of showflags and their current state. In the editor, use the viewport UI instead.
pause	Pauses the game, but continues rendering. Any simulation rendering work will stop.
slomo x	Changes the game speed. Can be very useful for slowing down time without skipping simulation work, when profiling. For example slomo .01
debugcreateplayer 1	For testing splitscreen.
r.CompositionGraphDebug	Execute to get a single frame dump of the composition graph of one frame (post processing and lighting).
r.DumpShaderDebugInfo	When set to 1, will cause any shaders that are then compiled to dump debug info to GameName/Saved/ShaderDebugInfo.

Console Command	Description
r.RenderTargetPoolTest	Clears the texture returned by the rendertarget pool with a special color to track down color leaking bugs.
r.SetRes	Set the display resolution for the current game view. Has no effect in the editor.
r.ViewportTest	Allows to test different Viewport rectangle configuations (in game only) as they can happen when using Matinee/Editor.

Useful command lines when working on rendering:

Commandline	Description
-d3ddebug	Enables the D3D11 debug layer, useful for catching API errors.
-sm4	Forces Feature Level SM4 with the D3D11 RHI.
-opengl3 / -opengl4	Forces use of OpenGL RHI at the specified feature level.
-dx11	currently the default on windows
-dx12	experimental
-featureleveles2	Ignored when running with editor, there the UI has to be used
-featureleveles31	Ignored when running with editor, there it needs to be enabled in Editor Preferences
-ddc=noshared	Prevents the use of network (shared) Derived Data Cache. Can be useful when debugging shader caching issues.

Modules

The renderer code exists in its own module, which is compiled to a dll for non-monolithic builds. This allows faster iteration as we do not have to relink the entire application when rendering code changes. The Renderer module depends on Engine because it has many callbacks into Engine. However, when the Engine needs to call some code in the Renderer, this happens through an interface, usually IRendererModule or FSceneInterface.

Scene representation

The scene as the renderer sees it is defined by primitive components and the lists of various other structures stored in FScene. An Octree of primitives is maintained for accelerated spatial queries.

Primary scene classes

There is a Rendering Thread, which operates in parallel with the game thread. Most classes that bridge the gap between the game thread and rendering thread are split into two parts based on which thread has ownership of that state.

The primary classes are:

Class	Description
UWorld	A world contains a collection of Actors and Components that can interact with each other. Levels can be streamed in and out of the world, and multiple worlds can be active in the program.
ULevel	Collection of Actors and Components that are loaded / unloaded together and saved in a single map file.
USceneComponent	Base class of any object that needs to be added to an FScene, like lights, meshes, fog, etc.
UPrimitiveComponent	Base class of anything that can be rendered or interact with physics. Also acts as the granularity of visibility culling and rendering property

Class	Description
	specification (casts shadows, etc). Just like all UObjects, the game thread owns all variables and state and the rendering thread should not access it directly.
ULightComponent	Represents a light source. The Renderer is responsible for computing and adding its contribution to the scene.
FScene	Renderer version of the UWorld. Objects only exist to the renderer once they are added to the FScene, which is called registering a component. The rendering thread owns all states in the FScene -the game thread cannot modify it directly.
FPrimitiveSceneProxy	Renderer version of UPrimitiveComponent, mirrors UPrimitiveComponent state for the rendering thread. Exists in the engine module and intended to be subclassed to support different types of primitives (skeletal, rigid, BSP, etc). Implements some very important functions like GetViewRelevance, DrawDynamicElements, etc.
FPrimitiveSceneInfo	Internal renderer state (specific to the FRendererModule implementation) that corresponds to a UPrimitiveComponent and FPrimitiveSceneProxy. Exists in the renderer module, so the engine cannot see it.
FSceneView	Engine representation of a single view into an FScene. A scene can be rendered with different views in different calls to FSceneRenderer::Render (multiple editor viewports) or with multiple views in the same call to FSceneRenderer::Render (splitscreen in game). A new View is constructed for each frame.
FViewInfo	Internal renderer representation of a view, exists in the renderer module.
FSceneViewState	The ViewState stores private renderer information about a view which is needed across frames. In game, there is one view state per ULocalPlayer.

Class	Description
FSceneRenderer	A class created each frame to encapsulate inter-frame temporaries.

Here is a list of the primary classes arranged by which module they are in. This becomes important when you are trying to figure out how to solve linker issues.

Engine Module	Renderer Module
UWorld	FScene
UPrimitiveComponent / FPrimitiveSceneProxy	FPrimitiveSceneInfo
FSceneView	FViewInfo
ULocalPlayer	FSceneViewState
ULightComponent / FLightSceneProxy	FLightSceneInfo

And the same classes arranged by which thread has ownership of their state. It is important to always be mindful of what thread owns the state you are writing code for, to avoid causing race conditions.

Game Thread	Rendering Thread
UWorld	FScene
UPrimitiveComponent	FPrimitiveSceneProxy / FPrimitiveSceneInfo
	FSceneView / FViewInfo
ULocalPlayer	FSceneViewState
ULightComponent	FLightSceneProxy / FLightSceneInfo

Material classes

Class	Description
FMaterial	An interface to a material used for rendering. Provides access to material properties (e.g. blend mode). Contains a shader map used by the renderer to retrieve individual shaders.
FMaterialResource	UMaterial's implementation of the FMaterial interface.
FMaterialRenderProxy	A material's representation on the rendering thread. Provides access to an FMaterial interface and the current value of each scalar, vector, and texture parameter.
UMaterialInterface	[abstract] Game thread interface for material functionality. Used to retrieve the FMaterialRenderProxy used for rendering and the UMaterial that is used as the source.
UMaterial	A material asset. Authored as a node graph. Computes material attributes used for shading, sets blend mode, etc.
UMaterialInstance	[abstract] An instance of a UMaterial. Uses the node graph in the UMaterial but provides different parameters (scalars, vectors, textures, static switches). Each instance has a parent UMaterialInterface. Therefore a material instance's parent may be a UMaterial or another UMaterialInstance. This creates a chain that will eventually lead to a UMaterial.
UMaterialInstanceConstant	A UMaterialInstance that may only be modified in the editor. May provide scalar, vector, texture, and static switch parameters.
UMaterialInstanceDynamic	A UMaterialInstance that may be modified at runtime. May provide scalar, vector, and texture parameters. It cannot provide static switch parameters and it cannot be the parent of another UMaterialInstance.

Primitive components and proxies

Primitive components are the basic unit of visibility and relevance determination. For example, occlusion and frustum culling happen on a per-primitive basis. Therefore it is important when designing a system to think about how big to make components. Each component has a bounds that is used for various operations like culling, shadow casting, and light influence determination.

Components only become visible to the scene (and therefore the renderer) when they are registered. Game thread code that changes a component's properties must call **MarkRenderStateDirty()** on the component to propagate the change to the rendering thread.

FPrimitiveSceneProxy and FPrimitiveSceneInfo

FPrimitiveSceneProxy is the rendering thread version of UPrimitiveComponent that is intended to be subclassed depending on the component type. It lives in the Engine module and has functions called during rendering passes. FPrimitiveSceneInfo is the primitive component state that is private to the renderer module.

Important FPrimitiveSceneProxy methods

Function	Description
GetViewRelevance	Called from InitViews at the beginning of the frame, and returns a populated FPrimitiveViewRelevance.
DrawDynamicElements	Called to draw the proxy in any passes which the proxy is relevant to. Only called if the proxy indicated it has dynamic relevance.
DrawStaticElements	Called to submit StaticMesh elements for the proxy when the primitive is being attached on the game thread. Only called if the proxy indicated it has static relevance.

Scene rendering order

The renderer processes the scene in the order that it wants to composite data to the render targets. For example, the Depth only pass is rendered before the Base pass, so that Heirarchical Z (HiZ) will be populated to reduce shading cost in the base pass. This order is statically defined by the order pass functions are called in C++.

Relevance

FPrimitiveViewRelevance is the information on what effects (and therefore passes) are relevant to the primitive. A primitive may have multiple elements with different relevance, so FPrimitiveViewRelevance is effectively a logical OR of all the element's relevancies. This means that a primitive can have both opaque and translucent relevance, or dynamic and static relevance; they are not mutually exclusive.

FPrimitiveViewRelevance also indicates whether a primitive needs to use the dynamic and/or static rendering path with bStaticRelevance and bDynamicRelevance.

Drawing Policies

Drawing policies contain the logic to render meshes with pass specific shaders. They use the FVertexFactory interface to abstract the mesh type, and the FMaterial interface to abstract the material details. At the lowest level, a drawing policy takes a set of mesh material shaders and a vertex factory, binds the vertex factory's buffers to the Rendering Hardware Interface (RHI), binds the mesh material shaders to the RHI, sets the appropriate shader parameters, and issues the RHI draw call.

Drawing Policy methods

Function	Description
Constructor	Finds the appropriate shader from the given vertex factory and material shader map, stores these references.
CreateBoundShaderState	Creates an RHI bound shader state for the drawing policy.
Matches/Compare	Provides methods to sort the drawing policy with others in the static draw lists. Matches must compare on all the factors that

Function	Description
	DrawShared depends on.
DrawShared	Sets RHI state that is constant between drawing policies that return true from Matches. For example, most drawing policies sort on material and vertex factory, so shader parameters depending only on the material can be set, and the vertex buffers specific to the vertex factory can be bound. State should always be set here if possible instead of SetMeshRenderState, since DrawShared is called less times in the static rendering path.
SetMeshRenderState	Sets RHI state that is specific to this mesh, or anything not set in DrawShared. This is called many more times than DrawShared so performance is especially critical here.
DrawMesh	Actually issues the RHI draw call.

Rendering paths

Unreal Engine has a dynamic path which provides more control but is slower to traverse, and a static rendering path which caches scene traversal as close to the RHI level as possible. The difference is mostly high level, since they both use drawing policies at the lowest level. Each rendering pass (drawing policy) needs to be sure to handle both rendering paths if needed.

Dynamic rendering path

The dynamic rendering path uses TDynamicPrimitiveDrawer and calls DrawDynamicElements on each primitive scene proxy to render. The set of primitives that need to use the dynamic path to be rendered is tracked by FViewInfo::VisibleDynamicPrimitives. Each rendering pass needs to iterate over this array, and call DrawDynamicElements on each primitive's proxy. DrawDynamicElements of the proxy then needs to assemble as many FMeshElements as it needs and submit them with DrawRichMesh or TDynamicPrimitiveDrawer::DrawMesh. This ends up creating a new temporary drawing policy, calling CreateBoundShaderState, DrawShared, SetMeshRenderState, and finally DrawMesh.

The dynamic rendering path provides a lot of flexibility because each proxy has a callback in DrawDynamicElements where it can execute logic specific to that component type. It also has minimal insertion cost but high traversal cost, because there is no state sorting, and nothing is cached.

Static rendering path

The static rendering path is implemented through static draw lists. Meshes are inserted into the draw lists when they are attached to the scene. During this insertion, DrawStaticElements on the proxy is called to collect the FStaticMeshElements. A drawing policy instance is then created and stored, along with the result of CreateBoundShaderState. The new drawing policy is sorted based on its Compare and Matches functions and inserted into the appropriate place in the draw list (see TStaticMeshDrawList::AddMesh). In InitViews, a bitarray containing visibility data for the static draw list is initialized and passed into TStaticMeshDrawList::DrawVisible where the draw list is actually drawn. DrawShared is only called once for all the drawing policies that match each other, while SetMeshRenderState and DrawMesh are called for each FStaticMeshElement (see TStaticMeshDrawList::DrawElement).

The static rendering path moves a lot of work to attach time, which significantly speeds up scene traversal at rendering time. Static draw list rendering is about 3x faster on the rendering thread for Static Meshes, which allows a lot more Static Meshes in the scene. Because static draw lists cache data at attach time, they can only cache view independent state. Primitives that are rarely reattached but often rendered are good candidates for the static draw lists.

The static rendering path can expose bugs because of the way it only calls DrawShared once per state bucket. These bugs can be difficult to detect, since they depend on the rendering order and the attach order of meshes in the scene. Special view modes such as lighting only, unlit, etc will force all primitives to use the dynamic path, so if a bug goes away when forcing the dynamic rendering path, there is a good chance it is due to an incorrect implementation of a drawing policy's DrawShared and/or the Matches function.

High level Rendering order

Here is a description of the control flow when rendering a frame starting from FDeferredShadingSceneRenderer::Render:

Operation	Description
GSceneRenderTargets.Allocate	Reallocates the global scene render targets to be large enough for the current view, if needed.
InitViews	Initializes primitive visibility for the views through various culling methods, sets up dynamic shadows that are visible this frame, intersects shadow frustums with the world if necessary (for whole scene shadows or preshadows).
PrePass / Depth only pass	RenderPrePass / FDepthDrawingPolicy. Renders occluders, outputting only depth to the depth buffer. This pass can operate in several modes: disabled, occlusion only, or complete depths, depending on what is needed by active features. The usual purpose of this pass is to initialize Hierarchical Z to reduce the shading cost of the Base pass, which has expensive pixel shaders.
Base pass	RenderBasePass / TBasePassDrawingPolicy. Renders opaque and masked materials, outputting material attributes to the GBuffer. Lightmap contribution and sky lighting is also computed here and put in scene color.
Issue Occlusion Queries / BeginOcclusionTests	Kicks off latent occlusion queries that will be used in the next frame's InitViews. These are done by rendering bounding boxes around the objects being queried, and sometimes grouping bounding boxes together to reduce draw calls.
Lighting	Shadowmaps are rendered for each light and light contribution is accumulated to scene color, using a mix of standard deferred and tiled deferred shading. Light is also accumulated in the translucency lighting volumes.

Operation	Description
Fog	Fog and atmosphere are computed per-pixel for opaque surfaces in a deferred pass.
Translucency	Translucency is accumulated into an offscreen render target where it has fogging applied per-vertex so it can integrate into the scene. Lit translucency computes final lighting in a single pass to blend correctly.
Post Processing	Various post process effects are applied using the GBuffers. Translucency is composited into the scene.

This is a fairly simplified and high level view. To get more details, look through the relevant code or the log output of a 'profilegpu'.

Render Hardware Interface (RHI)

The RHI is a thin layer above the platform specific graphics API. The RHI abstraction level in Unreal Engine is as low level as possible, with the intention that most features can be written in platform independent code and 'just work' on all platforms that support the required feature level.

Feature sets are quantized into ERHIFeatureLevel to keep the complexity low. If a platform cannot support all of the features required for a Feature Level, it must drop down in levels until it can.

Feature Level	Description
SM5	Generally corresponds with D3D11 Shader Model 5, except only 16 textures can be used because of OpenGL 4.3 limits. Supports tessellation, compute shaders and cubemap arrays. The deferred shading path is supported.
SM4	Corresponds to D3D11 Shader Model 4, which is generally the same as SM5, except no tessellation, compute shaders or cubemap arrays. The deferred

Feature Level	Description
	shading path is supported. Eye Adaptation is not supported as it uses compute shaders.
ES3_1	Corresponds to the features supported by OpenGL ES3.1, Vulkan, and Metal.

Rendering state grouping

Render states are grouped based on what part of the pipeline they affect. For example, RHISetDepthState sets all state relevant to depth buffering.

Rendering state defaults

Since there are so many rendering states, it is not practical to set them all every time we want to draw something. Instead, Unreal Engine has an implicit set of states which are assumed to be set to the defaults (and therefore must be restored to those defaults after they are changed), and a much smaller set of states which have to be set explicitly. The set of states that do not have implicit defaults are:

- RHISetRenderTargets
- RHISetBoundShaderState
- RHISetDepthState
- RHISetBlendState
- RHISetRasterizerState
- Any dependencies of the shaders set by RHISetBoundShaderState

All other states are assumed to be at their defaults (as defined by the relevant TStaticState, for example the default stencil state is set by `RHISetStencilState(TStaticStencilState<>::GetRHI())`.