Threaded Rendering

Information for graphics programmers working with the threaded renderer.

Rendering thread

In Unreal Engine, the entire renderer operates in its own thread that is a frame or two behind the game thread.

When dealing with rendering things, you have to carefully consider every memory read and write to ensure not only thread safety, but also determinism in behavior. When functional behavior depends on execution speed differences between two threads, it is called a race condition. Avoiding race conditions is important because they are usually very difficult to reproduce, and may be machine, platform, debugger or configuration dependent because of speed differences. These kind of bugs can rarely be debugged and take something like 10x the time to fix compared to a normal reproducible bug.

Here is a simple example of a race condition / threading bug:

```
1 /** FStaticMeshSceneProxy Actor is called on the game thread when a
  component is registered to the scene. */
2 FStaticMeshSceneProxy::FStaticMeshSceneProxy(UStaticMeshComponent*
  InComponent):
3 FPrimitiveSceneProxy(...),
```

```
Owner(InComponent->GetOwner()) <======== Note: AActor pointer is cached
...

/** DrawDynamicElements is called on the rendering thread when the renderer is doing a pass over the scene. */
void FStaticMeshSceneProxy::DrawDynamicElements(...)
{
if (Owner->AnyProperty) <========== Race condition! The game thread owns all AActor / UObject state,
// and may be writing to it at any time. The UObject may even have been garbage collected, causing a crash.
// This could have been done safely by mirroring the value of AnyProperty in this proxy.
}

```

Copy full snippet

Development approach

There is no way to exhaustively test to find race conditions. It is important to realize that you cannot create reliable threaded code by guess-and-checking or retroactively fixing bugs. The best approach is to completely understand the interactions of the game thread and rendering thread and use mechanisms to ensure determinism. You should be able to explain the order of events that will make every interaction deterministic, or else you are almost certainly creating race conditions.

Thread specific data structures

For this reason, it is a good idea to have data in separate structures that are 'owned' by the different threads so that it is obvious who can modify what. This holds true for functions as well. It is best to always call each function from the same thread or things get really complicated. Most of Unreal Engine is structured this way, for example, **UPrimitiveComponent** is the base game thread class of anything that can be rendered, cast shadows, has its own visibility state, etc. The rendering thread can never touch the memory of UPrimitiveComponent directly since the game thread may be writing to its members at any time. The rendering thread has its own class to represent this functionality, which is **FPrimitiveSceneProxy**. The game thread can never touch the members of memory of an

FPrimitiveSceneProxy after it is created and registered.
UActorComponent::RegisterComponent adds a component to the scene and makes it visible to the renderer by creating a FPrimitiveSceneProxy. Once the component is registered, it will have **FPrimitiveSceneProxy::DrawDynamicElements** called on it for every pass that is needed if it is visible.

Performance considerations

The game thread blocks at the end of each **Tick()** until the rendering thread catches up to either one frame or two frames behind. Since the rendering thread is so far behind, it is never acceptable during gameplay to block the game thread until the rendering thread catches up completely. Blocking during loading or GC of individual objects is also a bad idea, since Unreal Engine supports async streaming levels. There are asynchronous mechanisms for various operations to avoid blocking.

Inter-thread communication

Asynchronous

The primary method of communication between the two threads is through the ENQUEUE_UNIQUE_RENDER_COMMAND_XXXPARAMETER macro. This macro creates a local class with a virtual **Execute** function that contains the code you enter into the macro. The game thread inserts the command into the rendering command queue, and the rendering thread calls the Execute function when it gets around to it.

FRenderCommandFence provides a convenient way to track the progress of the rendering thread on the game thread. The game thread calls **FRenderCommandFence::BeginFence** to begin the fence. The game thread can then call **FRenderCommandFence::Wait** to block until the rendering thread has processed the fence, or it can just poll the progress of the rendering thread by checking **GetNumPendingFences**. When GetNumPendingFences returns 0, the rendering thread has processed the fence.

Blocking

FlushRenderingCommands is the standard method of blocking the game thread until the rendering thread has caught up. This is useful for offline (editor) operations which modify

memory being accessed by the rendering thread.

Rendering resources

FRenderResource provides the base rendering resource interface and provides hooks for initialization and releasing. Anything that derives from FRenderResource (**FVertexBuffer**, **FIndexBuffer**, etc) needs to be initialized before it is used for rendering and released before being deleted. **FRenderResource::InitResource** can only be called from the rendering thread, so there is a helper function (**BeginInitResource**) that can be called on the game thread to enqueue a rendering command to call FRenderResource::InitResource. RHI functions can only be called from the rendering thread (with the exception of a few for creating devices, viewports, etc).

UObjects and Garbage Collection

Garbage Collection (GC) happens on the game thread and operates on **UObjects**. The game thread may delete a UObject while the rendering thread is processing a command that references it. For this reason, the rendering thread should never dereference a UObject pointer unless a mechanism is in place to make sure the UObject is not deleted until the rendering thread no longer references it. An example is UPrimitiveComponent, which uses a FRenderCommandFence called **DetachFence** to prevent GC from deleting the UObject before the rendering thread has processed the detach command.

Game thread FRenderResource handling

There is two common scenarios of game thread rendering thread resource interaction to consider, the case of static resources (only modified on load or in the editor, like an index buffer) and dynamic resources, which need to be updated every frame with the latest results of the game thread simulation.

Static resources

Here is how the static resource interaction is handled in Unreal Engine, using **USkeletalMesh** as an example.

- **USkeletalMesh::PostLoad** gets called on load, which calls **InitResources**. This calls **BeginInitResource** on any static **FRenderResources** that it has like the index buffer. BeginInitResource enqueues a rendering command to call **FRenderResource::InitResource**. From this point on, the game thread can no longer modify the index buffer memory until it does something to take back ownership.
- A component registers which starts rendering with the USkeletalMesh's index buffer.
- GC determines that the component is no longer referenced at some point (level unload or no longer referenced) and detaches the component. Note that at this point, the game thread cannot delete the index buffer memory, because the rendering thread may not have processed the detach yet and may still be rendering with the index buffer.
- GC calls **USkeletalMesh::BeginDestroy**, which is the game thread object's chance to enqueue commands to release the rendering resources, so it does **BeginReleaseResource(&IndexBuffer)**; The game thread still cannot delete the memory of **IndexBuffer** because the rendering thread has not necessarily processed the release yet. We could block the game thread until the rendering thread catches up, but this would cause hitches and be slow, so we have an asynchronous mechanism instead. In order to track the rendering thread's progress of processing the release command we initiate a fence.
- GC calls **USkeletalMesh::IsReadyForFinishDestroy**, and will not destroy the UObject until this function returns `true`. The function only returns `true` once the fence has been passed by the rendering thread, which means it is now safe to delete the index buffer memory from the game thread.
- GC finally calls **UObject::FinishDestroy** which can be used to release memory in a central location. In the case of the index buffer, its memory gets freed when the USkeletalMesh destructor calls **FRawStaticIndexBuffer**'s destructor, which calls the destructor of the **TArray** holding the index buffer memory, which frees the memory.

This mechanism works well because it is efficient (never blocks either thread, initializes in a central location instead of checking for whether initialization is needed every frame), and is deterministic.

Dynamic resources

The skeletal mesh bone transforms which are produced by the game thread animation each frame are a good example of dynamic resource updating. The goal is to get the transforms from the game thread after each animation update into an array on the rendering thread

where they can be set as shader constants. The same would be true if you were updating an index or vertex buffer each frame. Here is the order of operations:

- **USkinnedMeshComponent::CreateRenderState_Concurrent** allocates **USkinnedMeshComponent::MeshObject**. From this point on, the game thread can only write to the **MeshObject** pointer, but not to the memory of the **FSkeletalMeshObject**.
- **USkinnedMeshComponent::UpdateTransform** gets called to update the component's movement at least once per frame. This calls **FSkeletalMeshObjectGPUSkin::Update** in the case of GPU skinning. At this point, we have up to date transforms on the game thread and need to get them over to the rendering thread. This is done by first allocating memory on the heap (**FDynamicSkelMeshObjectData**), then copying the bone transforms into it, and then passing off this copy to the rendering thread using ENQUEUE_UNIQUE_RENDER_COMMAND_TWOPARAMETER. The rendering thread now owns the copy and is responsible for deleting it. The ENQUEUE_UNIQUE_RENDER_COMMAND_TWOPARAMETER macro contains code to copy the transforms to their final destination so they can be set as shader constants. This is where you would lock and update a vertex buffer if updating vertex positions.
- At some point, the component gets detached. The game thread enqueues rendering commands to release all of the dynamic FRenderResources and can now set the MeshObject pointer to NULL, however the actual memory is still being referenced by the rendering thread and cannot be deleted. This is where the deferred deletion mechanism comes in to play. Classes that derive from **FDeferredCleanupInterface** can be deleted in an asynchronous way that is thread safe. FSkeletalMeshObject implements this interface. The game thread wants to kick off the deferred deletion of the FSkeletalMeshObject so it calls **BeginCleanup(MeshObject)**. The memory will eventually be deleted when it is safe to do so and cleanup is complete.

Updating state vs Traversing the scene for rendering

When developing a system that has distinct update and render operations, it is tempting to combine the two in **DrawDynamicElements**, however this is a poor design choice. A better solution is to separate the update out of the rendering traversal, for example enqueue the update command from within the game thread Tick.

DrawDynamicElements is called by the high level rendering code to draw the elements of a primitive component. The high level code assumes that no RHI state is being changed, and that it can call DrawDynamicElements as many times as it needs each frame, depending on shading passes, number of views, and scene captures in the scene. DrawDynamicElements may even be called, but then the underlying drawing policy discards the results for various reasons (for example a translucent FMeshElement submitted during the depth pass will be discarded). If the primitive component is actually not visible, the occlusion system may or may not actually call DrawDynamicElements, depending on the heuristic it is using. All of these factors can conflict with state updating which should happen once per frame.

A better solution is to separate the update from the rendering traversal. The game thread Tick can enqueue a rendering command to do the update operation. The rendering command can optionally skip updating based on visibility, if this is acceptable for the use case, by using **LastRenderTime** of the primitive scene info. If the update operation is enqueued separately in this manner, any RHI functions can be used including setting different render targets.

State caching (as opposed to updating) is an exception to this rule. State caching is storing an intermediate result of the rendering traversal as an optimization. It is closely tied with the traversal, and does not change RHI state, so it does not suffer the downsides mentioned before (as long as the determination of when to cache is done correctly).