

Developer

/ Documentation

/ Unreal Engine ▾

/ Unreal Engine 5.4 Documentation

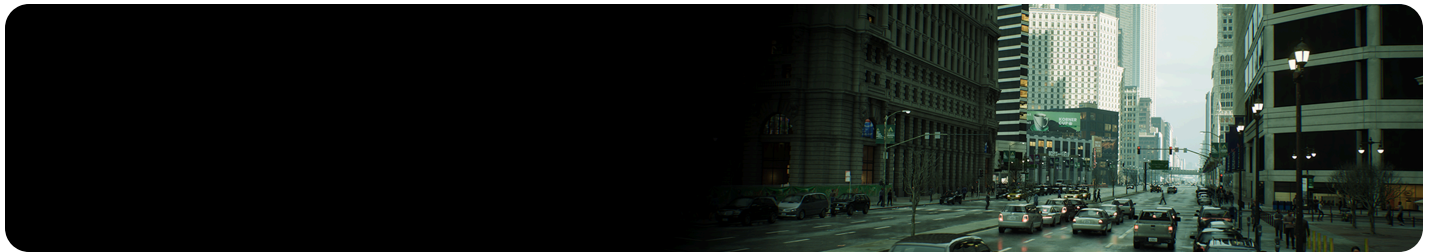
/ Designing Visuals, Rendering, and Graphics

/ Graphics Programming

/ Mesh Drawing Pipeline

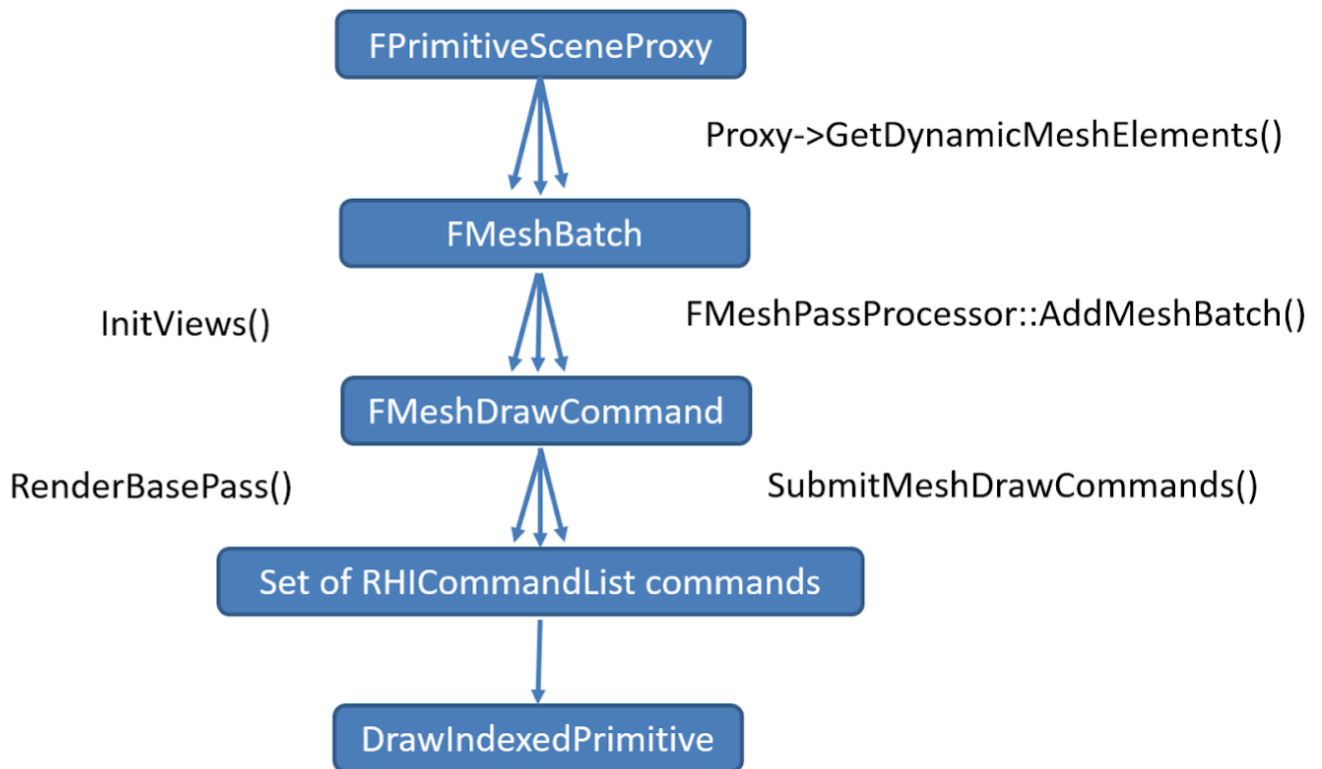
Mesh Drawing Pipeline

Guide to adding custom mesh passes and understanding Unreal Engine's mesh drawing performance characteristics.



The information contained in this page is targeted at programmers who want to add custom mesh passes, or want to understand Unreal Engine's mesh drawing performance characteristics.

The **Mesh Drawing Pipeline** is based around a concept of retained mode, where all scene draws are prepared in advance instead of building them every frame. It also features aggressive caching and draw call merging in order to exploit properties of Static Meshes which change infrequently and can be reused across frames.



The journey of a draw.

Mesh rendering starts from `FPrimitiveSceneProxy`, which is the render thread representation for the game thread's `UPrimitiveComponent`. `FPrimitiveSceneProxy` is responsible for submitting `FMeshBatch`'s to the renderer through the callbacks to `GetDynamicMeshElements` and `DrawStaticElements`.

`FMeshBatch` decouples the `FPrimitiveSceneProxy` implementation (user code) from mesh passes (private renderer module). It contains everything the pass needs to figure out final shader bindings and render state, so the proxy never knows what passes it will be rendered in.

The next step is to convert `FMeshBatch` into a mesh pass specific `FMeshDrawCommand`. `FMeshDrawCommand` is an interface between `FMeshBatch` and the RHI. It's a fully stateless draw description that stores everything that the RHI needs to know about a mesh draw:

- Which shaders to use.
- Their resource bindings.
- Draw call parameters.

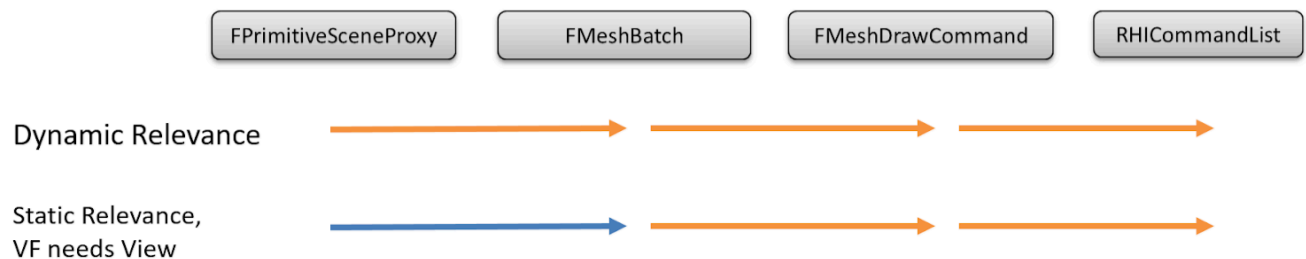
This enables caching and merging the draw calls just above the RHI level.

`FMeshDrawCommand` is created from a `FMeshBatch` by a mesh pass specific `FMeshPassProcessor`.

Finally, `SubmitMeshDrawCommands` is used to convert `FMeshDrawCommand` into a series of RHI commands set on a `RHICommandList`.

Cached and Dynamic Mesh Batches

`FPrimitiveSceneProxy` has two paths for generating `FMeshBatches` — a cached and a dynamic one. `FPrimitiveSceneProxy` implementations control which path is used each frame through the `GetViewRelevance()` function.



FMeshBatch code paths. Orange arrows are operations that have to be done every frame, while the blue arrow indicates an operation that is done once before being cached.

The Cached path builds and reuses `FMeshBatch` and is preferred for fast rendering of draws which don't change every frame, like Static Meshes. It is implemented by `DrawStaticElements`, which is called when a proxy is added to the scene.

Created `FMeshBatches` are stored inside `FPrimitiveSceneInfo::StaticMeshes` and reused every frame until the proxy is removed from the scene.

The Dynamic path recreates `FMeshBatch` every frame. This is the most flexible path and is used for draws which often change from frame to frame — for example, particles. It's implemented by `GetDynamicMeshElements`. This function is called every frame from `InitViews` and creates a temporary `FMeshBatch` for every view.

FMeshPassProcessor

Specific pass mesh processors are derived from the `FMeshPassProcessor` base class and are responsible for converting the `FMeshBatch` into mesh draw commands for a given pass. This is where final draw filtering happens, the appropriate shader is selected and shader bindings are collected.

In order to create a custom mesh pass processor, it must be derived from `FMeshPassProcessor` and `AddMeshBatch` function needs to be overridden.

`AddMeshBatch` implements:

- Draw filtering - For example, if a Material has a translucent draw mode then don't process it in `FDepthPassMeshProcessor`
- Selecting shaders and pipeline state (depth/stencil/blend state)
- Finally calling `BuildMeshDrawCommands()` which gathers shader bindings for the pass/material/vertex factory/primitive and adds the new draw command to the appropriate list.

Shader Bindings

Shader Bindings in Unreal Engine can be Uniform Buffers, Samplers, Textures, ShaderResourceViews or loose parameters (`FShaderParameter`).

A `FMeshPassProcessor` does not send shader bindings directly to the RHI with `RHICmdList.SetShaderParameter`, it merely records them into the `FMeshDrawSingleShaderBindings` class. The `BuildMeshDrawCommands()` function which is shared code between all passes will call `GetShaderBindings()` on the pass shaders.

Shader Bindings fall into a few categories:

- Pass-constant uniform buffers like `ViewUniformBuffer` or `DepthPassUniformBuffer`
- Vertex Factory bindings
- Material bindings
- Primitive bindings
- Pass specific bindings which change between draws.



Note that setting different bindings per-draw prevents draw call merging. Also, setting loose parameters — shader parameters that are not in a Uniform Buffer — prevents draw call merging, forcing slow constant buffer updates between draws.

Since each `FMeshPassProcessor` must go through `BuildMeshDrawCommands()` to call the pass shader's `GetShaderBindings()`, we need a mechanism to pass arbitrary data from the


`FMeshPassProcessor` to the `GetShaderBindings()` call. This is accomplished with the `ShaderElementData` parameter to `BuildMeshDrawCommands()`.

FMeshDrawCommand Performance Hazards

A number of inline allocators are used in `FMeshDrawCommand` to store variable length arrays without extra heap allocations. Overflowing these causes a performance hazard, as each mesh draw command must construct/destroy/copy the heap allocation along with cache misses on traversal of the commands.

`FMeshDrawShaderBindings` assumes **2** shader frequencies (Vertex + Pixel):

```
1 TArray<FMeshDrawShaderBindingsLayout, TInlineAllocator<2>>ShaderLayouts
2
```

 Copy full snippet

`FMeshDrawCommand` assumes **10** shader bindings among all frequencies:

```
1 const int32 NumInlineShaderBindings = 10;
2
```

 Copy full snippet

`FMeshDrawCommand` assumes **4** vertex streams from the vertex factory:

```
typedef TArray<FVertexInputStream, TInlineAllocator<4>>FVertexInputStreamArray
```

 Copy full snippet

Pass Types

There are three ways to use a `FMeshPassProcessor` for drawing:

Pass Types	Description
<code>EMeshPass::Type</code> enum	Adding an entry here allocates an <code>FParallelMeshDrawCommandPass</code> inside <code>FScene</code> . This enables the <code>FScene</code> to cache mesh draw commands for the pass at <code>AddToScene</code> time. An <code>FMeshPassProcessor</code> must be registered to their enum with <code>FRegisterPassProcessorCreateFunction</code> . Pass setup and dispatch happens in a task.
Manual Pass	Using a manual pass where <code>FParallelMeshDrawCommandPass</code> is stored as a variable in an arbitrary class. This is used when there are a variable number of passes each frame (for example, shadow depth pass). This type of pass cannot cache commands at <code>FScene::AddToScene</code> time, but still gets the benefit of pass set up and dispatch happening in a task.
<code>DrawDynamicMeshPass</code>	This is used for immediate mode drawing and is the slowest, but most convenient approach. Pass set up and dispatch happen immediately within the caller thread.



Note that the renderer has not been made extensible to plugins at this time, and with the exception `DrawDynamicMeshPass`, adding a new pass requires changing renderer module code.

FParallelMeshDrawCommandPass

In order to add a custom mesh pass, first we need to add a new entry to `EMeshPass` enum. Next, inside `FRelevancePacket::MarkRelevant()`, and based on relevance flags, add Static Meshes to the list of visible mesh draw commands. For example this snippet adds a mesh draw command into depth pass if it is relevant for depth pass:

```

1 if (StaticMeshRelevance.bUseForDepthPass)
2 {
3 DrawCommandPacket.AddCommandsForMesh(PrimitiveIndex, PrimitiveSceneInfo,
  StaticMeshRelevance, StaticMesh, Scene, bCanCache, EMeshPass::DepthPass);

```

```
4 }  
5
```

 Copy full snippet


Mark `EMeshPass` relevance for dynamic draws inside `ComputeDynamicMeshRelevance`:

```
1 if (ViewRelevance.bDrawRelevance && (ViewRelevance.bRenderInMainPass ||  
   ViewRelevance.bRenderCustomDepth))  
2 {  
3   PassMask.Set(EMeshPass::DepthPass);  
4   View.NumVisibleDynamicMeshElements[EMeshPass::DepthPass] += NumElements;  
5 }  
6
```

 Copy full snippet

Use `FParallelMeshDrawCommandPass::DispatchDraw` to draw this specific pass:


```
1 View.ParallelMeshDrawCommandPasses[EMeshPass::DepthPass].DispatchDraw(nullptr,  
   RHICmdList);  
2
```



 Copy full snippet

It is also possible to set up a Parallel Command List set in order to draw this pass in parallel:

```
1 FPrePassParallelCommandListSet ParallelCommandListSet(View, this, ParentCmdList  
   DrawRenderState);  
2 View.ParallelMeshDrawCommandPasses[EMeshPass::DepthPass].DispatchDraw(&Parallel  
   ParentCmdList);  
3
```



 Copy full snippet

DrawDynamicMeshPass

`FParallelMeshDrawCommandPass` is the default path for common mesh passes. It should be used for performance critical mesh passes, as it's the only path which supports mesh draw command caching and parallel rendering. On the other hand performance requirements enforce a very strict design — for example, it is not possible to modify mesh draw commands or shader bindings after `InitViews`.

For certain use cases, like drawing a few meshes inside the Editor, `DrawDynamicMeshPass` may be a simpler solution. It provides immediate mode mesh drawing and is the most flexible rendering path. Unreal Engine uses `DrawDynamicMeshPass` for some Editor-only passes and Canvas rendering.

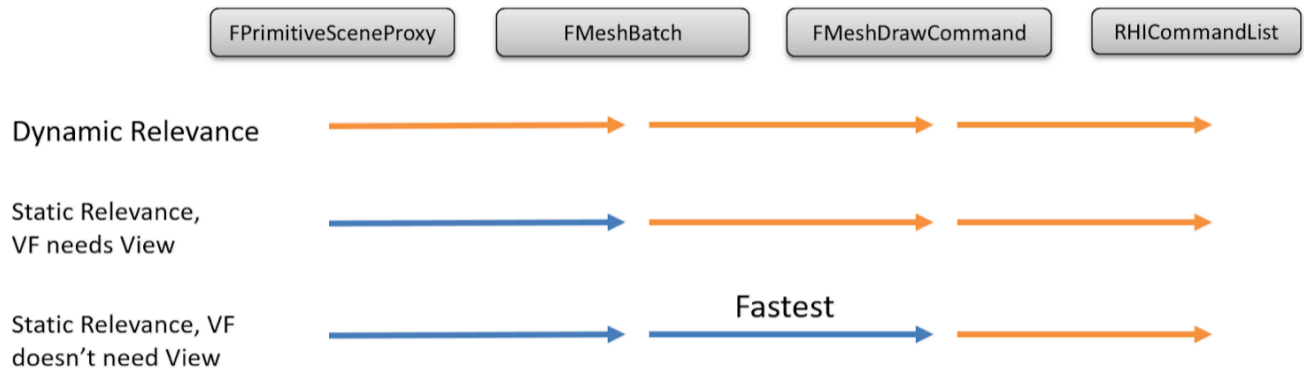
Drawing with `DrawDynamicMeshPass` is quite simple, it just requires to pass a lambda which will fill a temporary list of mesh draw commands:

```
1 DrawDynamicMeshPass(View, RHICmdList, [&View, CurrentDecalStage,
  RenderTargetMode](FDynamicPassMeshDrawListContext* DynamicMeshPassContext)
2 {
3   FMeshDecalMeshProcessor PassMeshProcessor(
4     View.Family->Scene->GetRenderScene(),
5     &View,
6     CurrentDecalStage,
7     RenderTargetMode,
8     DynamicMeshPassContext);
9
10  for (int32 MeshBatchIndex = 0; MeshBatchIndex < View.MeshDecalBatches.Num();
    ++MeshBatchIndex)
11  {
12    const FMeshBatch* Mesh = View.MeshDecalBatches[MeshBatchIndex].Mesh;
13    const FPrimitiveSceneProxy* PrimitiveSceneProxy =
      View.MeshDecalBatches[MeshBatchIndex].Proxy;
14    const uint64 DefaultBatchElementMask = ~0ull;
15
16    PassMeshProcessor.AddMeshBatch(*Mesh, DefaultBatchElementMask,
      PrimitiveSceneProxy);
17  }
18  });
19
```

 Copy full snippet

Cached Mesh Draw Commands

Cached mesh draw commands are built-in `FPrimitiveSceneInfo::AddToScene` inside `FPrimitiveSceneInfo::CacheMeshDrawCommands`. Drawing with these is very efficient, as we simply need to select the appropriate pre-built commands each frame (`FDrawCommandRelevancePacket::AddCommandsForMesh`). Cached draw commands can only be used when draw state doesn't change every frame and all shader bindings can be set up inside `AddToScene`.



Mesh Draw Command caching paths. Orange arrows are operations that have to be done every frame, while blue arrows indicates an operation that is done once and cached.

In order to support cached mesh draw commands:

- The pass must be using an entry in `EMeshPass::Type`
- `EMeshPassFlags::CachedMeshCommands` flag must be passed when registering custom mesh processor
- The mesh pass processor needs to be able to set up all shader bindings without relying on `FSceneView`, as during caching it will be null

In order for the shader to access per-frame data with cached mesh draw commands we bind scene-wide uniform buffers (see `FScene::UniformBuffers`) and then use `RHIUpdateUniformBuffer` to change their contents before the draw.

Currently, only `FLocalVertexFactory (UStaticMeshComponent)` can be cached because all other vertex factories require a view to set up for their shader bindings.

Cache Invalidation

Any data that a Mesh Pass Processor reads in `AddMeshBatch` is a dependency of the cached mesh draw commands. When that dependency changes, the cached commands must be invalidated. A single primitive's cached commands can be invalidated with `FPrimitiveSceneInfo::BeginDeferredUpdateStaticMeshes`. The entire scene's cached commands can be invalidated by setting `Scene->bScenesPrimitivesNeedStaticMeshElementUpdate` to `true`. This is a heavyweight operation and should be avoided during gameplay as it will cause a hitch in larger scenes.

For example, `FBasePassMeshProcessor::AddMeshBatch` uses `Scene->SkyLight` to decide whether to select the Skylight shader permutation or not. When `Scene-SkyLight` changes, we must invalidate cached mesh draw commands.

To achieve good performance with this caching scheme, it's important to put data in persistent Uniform Buffers. Then, you need to update those buffers rather than invalidating the cached commands frequently. For example, the skylight case could be changed to a dynamic branch in the shader based on `PassUniformBuffer` contents rather than selecting a different shader permutation.

Resource Lifetime Management

`FMeshDrawCommand` is not responsible for maintaining the lifetime of any of the resources it references, so special care must be taken with cached mesh draw commands to invalidate the commands which might reference a particular resource. For example, when recreating a Uniform Buffer referenced by a cached mesh draw command will cause a crash when traversing the cached mesh draw commands for rendering. The Uniform Buffer should either be updated or the cached mesh draw commands must be invalidated.



`VALIDATE_UNIFORM_BUFFER_LIFETIME` can be used to track down cases where a Uniform Buffer is deleted which is still referenced by a cached mesh draw command.

Draw Call Merging

Since `FMeshDrawCommands` capture all of the state a draw needs just above the RHI level, we can easily compare them for compatibility with draw call merging. The only form of draw call

merging currently implemented is based around the D3D11 feature set, which enables merging of draw calls which have identical shader bindings into an instanced draw. More advanced RHIs like D3D12 enable more aggressive merging of draws but this is not yet implemented.

Dynamic Instancing

In order to merge two draws into an instanced one, they must have identical shader bindings (`FMeshDrawCommand::MatchesForDynamicInstancing`). Only InstanceID in the shader will vary between them, or vertex streams setup at instance frequency.

Shader parameters must be carefully crafted to enable dynamic instancing. This is achieved through various means depending on the parameter frequency:

Pass Types	Description
Pass Parameters	These are placed in the pass uniform buffer, where any draws in the pass can merge.
FLocalVertexFactory Parameters	These are placed in a uniform buffer owned by <code>UStaticMesh</code> where any draws with the same <code>UStaticMesh</code> can merge.
Material Instance Parameters	These are placed in a material uniform buffer where any draws using the same Material Instance can merge.
Lightmap Resource Parameters	These are placed in a <code>LightmapResourceCluster</code> uniform buffer where any draws using the same <code>LightmapTexture</code> can merge.
Primitive Parameters	These are placed in a scene-wide primitive data buffer called <code>GPUScene</code> and indexed in the shader using <code>PrimitiveID</code> .

GPU Scene

In order to have different primitives in the same instanced draw with primitive-specific parameters, supporting platforms (`UseGPUScene`) upload them to a scene-wide buffer (`UpdateGPUScene`) and index into it with a `PrimitiveId`. For `FLocalVertexFactory`, `PrimitiveId` comes from a instance-frequency vertex input stream. This must be passed to the pixel shader, which must use `GetPrimitiveData(Parameters.PrimitiveId).Member` to access Primitive shader parameters, instead of accessing the primitive uniform buffer directly (`Primitive.Member`).

Instancing Efficiency

Currently, only cached mesh draw commands can be merged with dynamic instancing, which limits dynamic instancing to `FLocalVertexFactory`.

Certain edge cases also prevent merging:

- Lightmaps making small textures — adjust **MaxLightmapRadius** in `DefaultEngine.ini`
- Per-component vertex colors
- SpeedTree **Wind** node

To investigate dynamic instancing efficiency in a level, use the **r.MeshDrawCommands.LogDynamicInstancingStats 1** console command and inspect the output in the log.



Note that the **Depth Prepass** and **Shadow Depth** passes achieve higher merging efficiency because they frequently override with the default material's shaders when possible.

Mesh Drawing Parallelism

Most of the work of mesh drawing is in tasks to stay off the critical path of the Rendering Thread. In `InitViews` at the beginning of the RT frame, `FParallelMeshDrawCommandPass` issues one task per pass for pass setup (dynamic command generation, sorting and draw call merging). As the `RenderingThread` progresses through the frame and arrives at a mesh pass (for example, `RenderBasePass`), it kicks off multiple `FDrawVisibleMeshCommandsAnyThreadTasks` per pass for draw dispatch (recording the

`RHICmdList`), depending on how many cores the system has and how many draws there are to be dispatched.

- Setting **`r.MeshDrawCommands.ParallelPassSetup`** to **0** will disable the pass setup task and cause the work to be done on the `RenderingThread`, which can be useful for debugging.
- Setting **`r.RHICmdBasePassDeferredContexts`** to **0** will disable the parallel tasks for base pass draw dispatch, causing those to happen on the `RenderingThread`.

These tasks are kicked off as early as possible with a dependency chain so they can be executed in parallel with the `Rendering Thread` for a frame. The `Rendering Thread` only blocks on the completion of these tasks at the end of a frame in

`FSceneRenderer::WaitForTasksClearSnapshotsAndDeleteSceneRenderer`.

Console Variables

These are some useful console variable for diagnosing issues inside the mesh drawing pipeline:

Console Variable	Description
<code>r.MeshDrawCommands.ParallelPassSetup</code>	Toggles mesh draw command processing tasks, which is useful for diagnosing mesh pass threading issues.
<code>r.MeshDrawCommands.UseCachedCommands</code>	Forces all mesh draw commands to be dynamic when disabled, which is useful for diagnosing issues with stale data inside cached mesh draw commands.
<code>r.MeshDrawCommands.DynamicInstancing</code>	This toggles dynamic instancing. It's useful for diagnosing dynamic instancing issues.
<code>r.MeshDrawCommands.LogDynamicInstancingStats</code>	This is useful for inspecting dynamic instancing efficiency.
<code>r.GPUScene.UploadEveryFrame</code>	Forces GPU Scene to be fully updated every frame, which is useful for diagnosing issues with stale GPU

Console Variable

Description

	Scene data.
<code>r.GPUScene.ValidatePrimitiveBuffer</code>	This downloads GPU Scene to the CPU and validates its contents against primitive uniform buffers.