Analysis of Unreal Rendering System (03) - Rendering Mechanism

Table of contents

3.1 Overview and foundation of this article

3.1.1 Overview of rendering mechanism

This article mainly describes how UE organizes the objects in the scene into Draw Calls, what optimizations and processing are done during the process, and how the scene renderer renders the entire scene. The main contents involved are:

- Model drawing process.
- Dynamic and static rendering paths.
- Scene Renderer.
- Basic concepts and optimization techniques involved.
- Code analysis of core classes and interfaces.

These techniques will be covered in detail in the following chapters.

3.1.2 Basics of Rendering Mechanism

As usual, in order to better get into the topic of this article, we first explain or review some basic concepts and types that will be covered in this article.

type	Analysis
UPrimitiveComponent	The primitive component is the parent class of all objects that can be rendered or have physical simulation. It is the smallest granularity unit for CPU layer clipping.
FPrimitiveSceneProxy	The primitive scene proxy is the representative of UPrimitiveComponent in the renderer, mirroring the state of UPrimitiveComponent in the rendering thread.
FPrimitiveSceneInfo	The internal state of the renderer (describing the implementation of FRendererModule), is equivalent to the fusion of UPrimitiveComponent and FPrimitiveSceneProxy. Only the renderer module exists, so the engine module cannot perceive its existence.
FScene	It is the representative of UWorld in the rendering module. Only objects added to FScene will be perceived by the renderer. The rendering thread has all the states of FScene (the game thread cannot modify it directly).
FSceneView	Describes a single view in FScene. The same FScene allows multiple views. In other words, a scene can be drawn by multiple views, or multiple views can be drawn at the same time. A new view instance is created for each frame.
FViewInfo	The view is represented inside the renderer. Only the renderer module exists, and the engine module is invisible.
FSceneRenderer	It is created for each frame, encapsulating temporary data between frames. derives FDeferredShadingSceneRenderer (deferred shading scene renderer) and FMobileSceneRenderer (mobile scene renderer), representing the default renderers for PC and mobile respectively.

type	Analysis
FMeshBatchElement	The data of a single mesh model includes some data required for mesh rendering, such as vertices, indices, UniformBuffer, and various identifiers.
FMeshBatch	Stores a set of FMeshBatchElement data that have the same material and vertex buffer.
FMeshDrawCommand	Completely describes all the states and data of a Pass Draw Call, such as shader binding, vertex data, index data, PSO cache, etc.
FMeshPassProcessor	The mesh rendering pass processor is responsible for processing the mesh objects of interest in the scene and converting them from FMeshBatch objects into one or more FMeshDrawCommands.

It should be pointed out that, except for UPrimitiveComponent, which belongs to the game thread, all other concepts belong to the rendering thread.

3.2 Model drawing pipeline

3.2.1 Overview of the model rendering pipeline

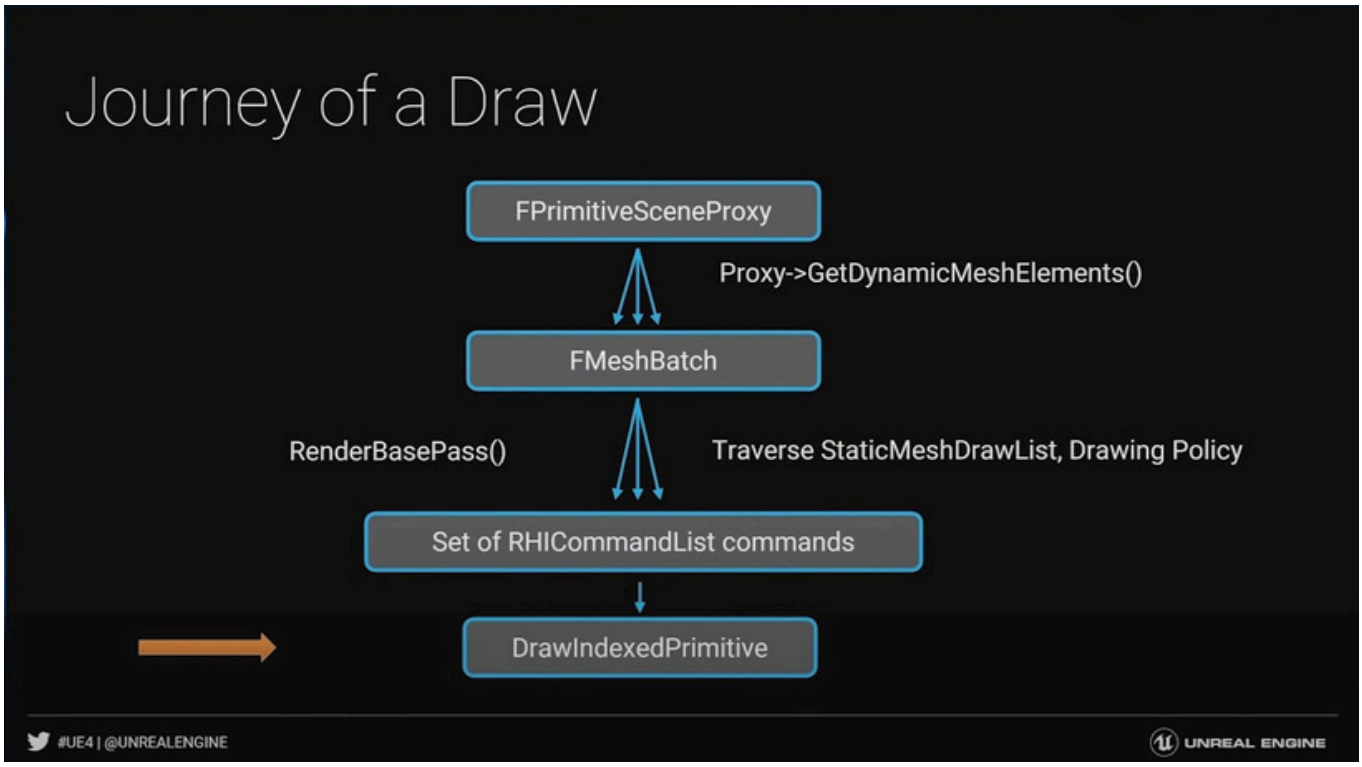
When learning graphics APIs such as OpenGL or DirectX, you must have come across similar code (taking OpenGL drawing a triangle as an example):

```
void DrawTriangle()
{
    //Construct triangle vertex and index data.
    floatvertices[] = {
        0.5f,    0.5f,   0.0f,    //  top right
        0.5f,  - - 0.5f,   0.0f,    //  bottom right
        0.5f,  - - 0.5f,   0.0f,    //  bottom left
        0.5f,    0.5f,   0.0f       //  top left
    };
    unsigned intindices[] = {
        0,1,    3,    // first    Triangle
        1,2,    3     // second Triangle
    };

    //createGPUSide resources and bind.
    unsigned intVBO, VAO, EBO;
    glGenVertexArrays(1,        &VAO);
    glGenBuffers(1,        &VBO);
    glGenBuffers(1,        &EBO);
    glBindVertexArray(VAO);
```

```
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    glBufferData(GL_ARRAY_BUFFER,sizeof(vertices), vertices, GL_STATIC_DRAW);

    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, EBO); glBufferData(GL_ELEMENT_ARRAY_BUFFER,sizeof(indices),
    indices, GL_STATIC_DRAW);

    glVertexAttribPointer(0,3, GL_FLOAT, GL_FALSE,3*sizeof(float), (void*)0glEnableVertexAttribArray(0);


    glBindBuffer(GL_ARRAY_BUFFER,0
    glBindVertexArray(0);

    //Clean up the background
    glClearColor(0.2f,0.3f,0.3f,1.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    //Draw a triangle
    glUseProgram(shaderProgram); glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES,6, GL_UNSIGNED_INT,0);

}
```

The above Hello Triangle roughly goes through several stages: constructing CPU resources, creating and binding GPU resources, and calling the drawing interface. For simple applications or learning graphics, directly calling the graphics API can simplify the process and get straight to the point.
However, for commercial game engines, which need to render complex scenes at dozens of frames per second (hundreds or thousands of Draw Calls, hundreds of thousands or even millions of triangles), it is definitely not possible to directly use simple graphics API calls.

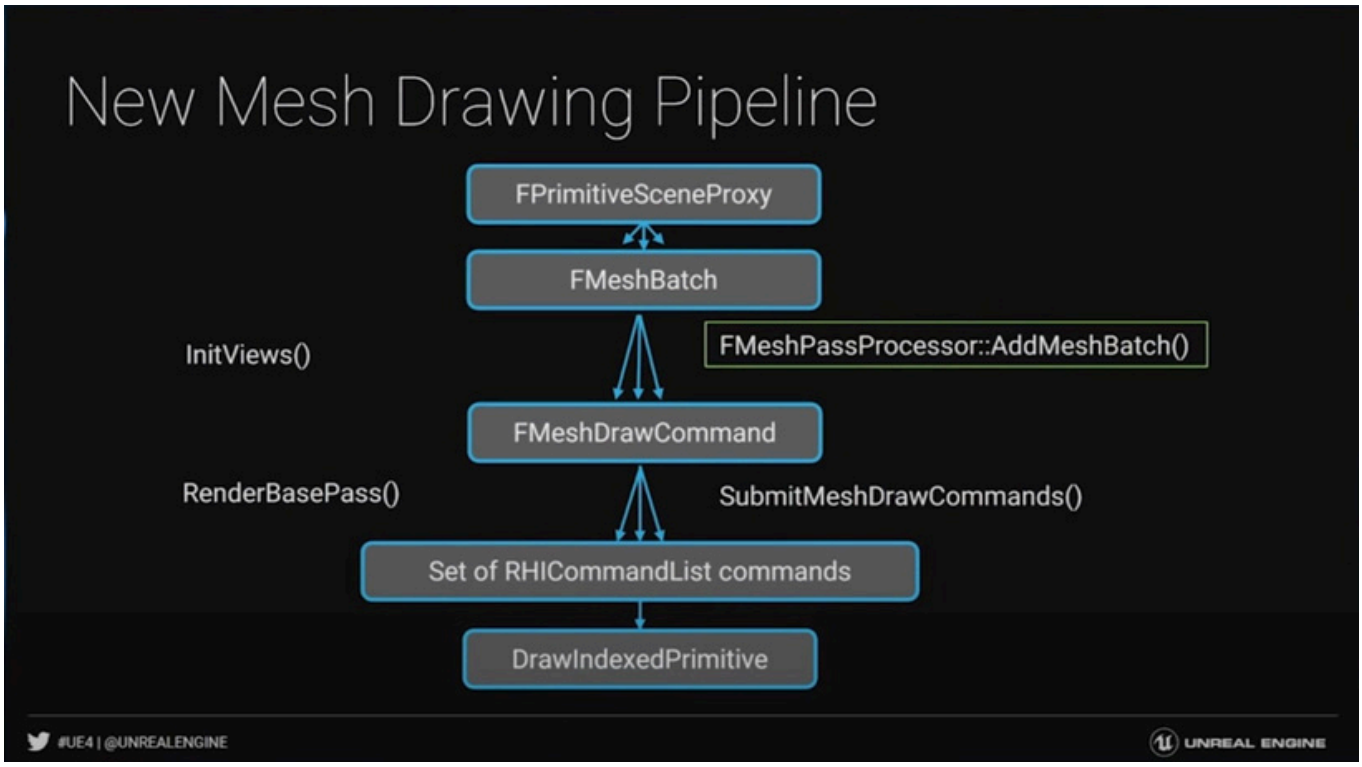
Commercial game engines need to perform a lot of operations and optimizations before actually calling the graphics API, such as occlusion culling, dynamic and static merging, dynamic instances, cache states and commands, generating intermediate instructions and then translating them into graphics API instructions, etc.
Before UE4.21, in order to achieve the above purpose, the mesh rendering process (Mesh Draw Pipeline) was adopted, as shown in the following diagram:



Mesh drawing process for UE4.21 and earlier versions.

The general process is that when rendering, the renderer will traverse all PrimitiveSceneProxy objects in the scene that have passed the visibility test, use its interface to collect different FMeshBatch, and then traverse these FMeshBatch in different rendering Passes, and use the DrawingPolicy corresponding to the Pass to convert it into a RHI command list. Finally, it will generate the corresponding graphics API instructions and submit them to the GPU hardware for execution.

On this basis, UE4.22 has made a major reconstruction of the mesh rendering pipeline in order to better optimize rendering. It abandoned the inefficient DrawingPolicy and replaced it with PassMeshProcessor. It added a concept FMeshDrawCommand between FMeshBatch and RHI commands to sort, cache, and merge drawing instructions in a more controllable manner:



UE4.22 refactored the new mesh drawing process. New concepts and operations such as FMeshDrawCommand and FMeshPassProcessor were added.

This has two main purposes:

- Supports RTX real-time ray tracing. Ray tracing requires traversing the objects in the entire scene and retaining the shader resources of the entire scene.
- GPU-driven rendering pipeline. Including GPU clipping, so the CPU cannot know the visibility of each frame, but it cannot create drawing instructions for the entire scene for each frame, otherwise real-time rendering cannot be achieved.

To achieve the above goals, the reconstructed pipeline adopts more aggregation caching measures, which are reflected in:

- When static primitives are added to the scene, drawing instructions are created and then cached.
- Allow the RHI layer to do as much preprocessing as possible.
    - shader binding table entry.
    - Graphics Pipeline State.
- Avoid rebuilding draw instructions for static meshes every frame.

After refactoring the model rendering pipeline, DepthPass and BasePass can reduce the number of Draw Calls by several times in most scenarios and cache a large number of commands:

A comparison of the rendering data of a test scene in Fortnite under the new and old mesh rendering pipelines. It can be seen that under the new mesh rendering process, Draw Calls have been greatly reduced and the number of command caches is also huge.

The subsequent chapters of this section will analyze the reconstructed mesh drawing process.

3.2.2 From FPrimitiveSceneProxy to FMeshBatch

In the previous article, we have explained that FPrimitiveSceneProxy is the mirror data of UPrimitiveComponent in the game thread in the rendering thread. FMeshBatch is a new concept that we will only touch upon in this section. It contains all the information needed to draw a Pass , decoupling the mesh Pass and FPrimitiveSceneProxy, so FPrimitiveSceneProxy does not know which Passes will draw it.

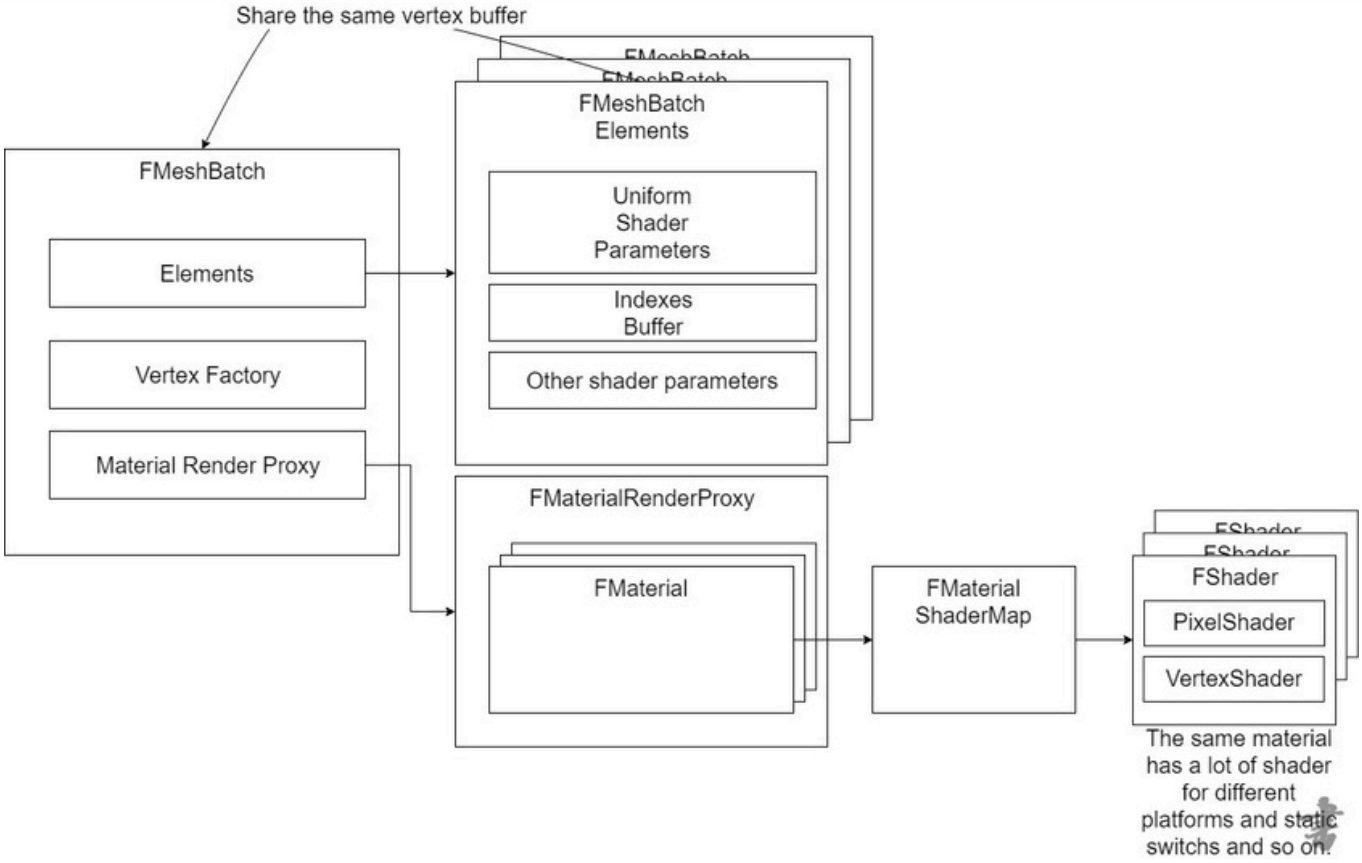
The main declarations of FMeshBatch and FMeshBatchElement are as follows:

```
// Engine\Source\Runtime\Engine\Public\MeshBatch.h

//Grid batch elements, storingFMeshBatchData required for a single
grid. structFMeshBatchElement {

    //GridUniformBuffer,If usingGPU Scene,You need tonull.
    FRHIUniformBuffer* PrimitiveUniformBuffer; //GridUniformBufferexist
    CPUSide data.
    constTUniformBuffer<FPrimitiveUniformShaderParameters>*
    PrimitiveUniformBufferResource;
    //Index buffer.
    constFIndexBuffer* IndexBuffer;

    union
    {
        uint32*     InstanceRuns;
        classFSplineMeshSceneProxy*          SplineMeshSceneProxy;
```

```
    };
    //    User data.
    const    void* UserData;
    void*    VertexFactoryUserData;

    FRHIVertexBuffer* IndirectArgsBuffer; uint32
    IndirectArgsOffset;

    //GraphicsIDMode, YesPrimID_FromPrimitiveSceneInfo(GPU Scenemode) and
PrimID_DynamicPrimitiveShaderData(Each grid has its ownUniformBuffer)
    //Can only be modified by the renderer.
    EPrimitiveIdMode PrimitiveIdMode : PrimID_NumBits +1; uint32
    DynamicPrimitiveShaderDataIndex:twenty four;

    uint32 FirstIndex;
    /** When 0, IndirectArgsBuffer will be used. */ uint32
    NumPrimitives;

    // Instancequantity
    uint32 NumInstances;
    uint32    BaseVertexIndex;
    uint32 MinVertexIndex;
    uint32 MaxVertexIndex;
    int32    UserIndex;
    float    MinScreenSize;
    float    MaxScreenSize;

    uint32 InstancedLODIndex :4; uint32
    InstancedLODRange :4;
    uint32 bUserDataIsColorVertexBuffer:1; uint32
    bIsSplineProxy :1; uint32 bIsInstanceRuns :1;


    //Get the number of primitives.
    int32GetNumPrimitives()const {

        if(bIsInstanceRuns && InstanceRuns) {

            int32 Count =0;
            for(uint32 Run =0; Run < NumInstances; Run++) {

                Count += NumPrimitives * (InstanceRuns[Run *2+1] -InstanceRuns[Run *
2]   +   1);
            }
            returnCount;
        }
        else
        {
            returnNumPrimitives * NumInstances;
        }
    }
};


//Grid batches.
struct    FMeshBatch
{
    //    This groupFMeshBatchElementThe data of the two layers have the same material and vertex buffer.
```

```
    // TInlineAllocator<1>showElementsThe array has at least1Elements.
    TArray<FMeshBatchElement,TInlineAllocator<1> > Elements; const
    FVertexFactory* VertexFactory;//Vertex Factory.
    constFMaterialRenderProxy* MaterialRenderProxy;//The material to use for rendering.


    uint16 MeshIdInPrimitive;//The grid where the primitive is locatedid,Used for stable sorting of identical
    primitives. int8 LODIndex;//GridLODIndex, used forLODof smooth transition. uint8 SegmentIndex;//The
    submodel index.

    //Crop marks.
    uint32 ReverseCulling :1; uint32
    bDisableBackfaceCulling:1;


    //Special RenderingPassThe associated
    tag of . uint32 CastShadow                  :1;//Is it in the shadowPassMedium Rendering. :1;//Is the
    uint32 bUseForMaterial                        material required?PassMedium Rendering.
    uint32 bUseForDepthPass :1;//Is it at depthPassMedium Rendering.
    uint32 bUseAsOccluder                       :1;//Indicates whether to occlude the body.
    uint32 bWireframe                           :1;//Whether to use wireframe mode.

    uint32 Type : PT_NumBits;//Primitive type, such asPT_TriangleList(default),PT_LineList, ... uint32
    DepthPriorityGroup : SDPG_NumBits;//Depth priority groups, such asSDPG_World (default),
    SDPG_Foreground

    //Other tags and data
    constFLightCacheInterface*              LCI;
    FHitProxyId BatchHitProxyId;
    floatTessellationDisablingShadowMapMeshSize;

    uint32 bCanApplyViewModeOverrides:1; uint32
    bUseWireframeSelectionColoring:1; uint32
    bUseSelectionOutline:1; uint32 bSelectable :1;

    uint32 bRequiresPerElementVisibility:1; uint32
    bDitheredLODTransition:1; uint32
    bRenderToVirtualTexture:1; uint32
    RuntimeVirtualTextureMaterialType :
    RuntimeVirtualTexture::MaterialType_NumBits;

    (.....)

    //Tool interface.
    bool IsTranslucent(ERHIFeatureLevel::Type InFeatureLevel)const; bool IsDecal
    (ERHIFeatureLevel::Type InFeatureLevel)const; bool IsDualBlend(ERHIFeatureLevel::Type
    InFeatureLevel)const; bool UseForHairStrands(ERHIFeatureLevel::Type InFeatureLevel)const;
    bool IsMasked(ERHIFeatureLevel::Type InFeatureLevel)const; int32

            GetNumPrimitives()const;
    bool    HasAnyDrawCalls()const;
};
```

It can be seen that FMeshBatch records a group of FMeshBatchElement data with the same material and vertex factory (as shown below), and also stores rendering Pass-specific tags and other required data for use and secondary processing in subsequent mesh rendering processes.



An FMeshBatch has a set of FMeshBatchElements, a vertex factory, and a material instance. All FMeshBatchElements in the same FMeshBatch share the same material and vertex buffer (which can be regarded as a Vertex Factory). But usually (most of the time), an FMeshBatch will only have one FMeshBatchElement.

At the beginning of rendering, the scene renderer FSceneRenderer will perform visibility testing and culling to remove obscured and hidden objects. At the end of this stage, it will call GatherDynamicMeshElementsthe FPrimitiveSceneProxy that collects all the current scene. The process

The schematic code is as follows:

```
void FSceneRender::Render(FRHICommandListImmediate& RHICmdList) {

	bool FDeferredShadingSceneRenderer::InitViews((FRHICommandListImmediate& RHICmdList,
. . . )
	{
		voidFSceneRender::ComputeViewVisibility(FRHICommandListImmediate& RHICmdList,
. . . )
		{
			FSceneRender::GatherDynamicMeshElements(Views, Scene, ViewFamily,
DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer, HasDynamicMeshElementsMasks,
HasDynamicEditorMeshElementsMasks, HasViewCustomDataMasks, MeshCollector);
		}
	}
}
```

Let's go inFSceneRender::GatherDynamicMeshElementsand see what logic is executed:

```
// Engine\Source\Runtime\Renderer\Private\SceneVisibility.cpp
```

```
void FSceneRenderer::GatherDynamicMeshElements(
        TArray<FViewInfo>&         InViews,
        const    FScene* InScene,  const
        FGlobalDynamicIndexBuffer& InViewFamily,
        FGlobalDynamicVertexBuffer&         DynamicIndexBuffer,
                                              DynamicVertexBuffer,
        FGlobalDynamicReadBuffer& constDynamicReadBuffer,
                FPrimitiveViewMasks&        HasDynamicMeshElementsMasks,
        const    FPrimitiveViewMasks&       HasDynamicEditorMeshElementsMasks,
        const FPrimitiveViewMasks&          HasViewCustomDataMasks,
        FMeshElementCollector&         Collector
{
        (.....)

        int32 NumPrimitives = InScene->Primitives.Num();

        int32 ViewCount = InViews.Num(); {

                //deal withFMeshElementCollector.
                Collector.ClearViewMeshArrays();
                for(int32 ViewIndex =0; ViewIndex < ViewCount; ViewIndex++) {

                        Collector.AddViewMeshArrays(
                                &InViews[ViewIndex],
                                &InViews[ViewIndex].DynamicMeshElements,
                                &InViews[ViewIndex].SimpleElementCollector,
                                &InViews[ViewIndex].DynamicPrimitiveShaderData,
                                InViewFamily.GetFeatureLevel(),
                                &DynamicIndexBuffer,
                                &DynamicVertexBuffer,
                                &DynamicReadBuffer);

                }

                const boolbIsInstancedStereo = (ViewCount >0) ?
(InViews[0].IsInstancedStereoPass() || InViews[0].bIsMobileMultiViewEnabled):false;
                constEShadingPath ShadingPath = Scene->GetShadingPath();

                //Traverse all the primitives in the scene.
                for(int32 PrimitiveIndex =0; PrimitiveIndex < NumPrimitives; ++PrimitiveIndex) {

                        constuint8 ViewMask = HasDynamicMeshElementsMasks[PrimitiveIndex];

                        if(ViewMask !=0)//Only process objects that are not occluded or hidden {

                                // Don't cull a single eye when drawing a stereo pair
                                constuint8 ViewMaskFinal = (bIsInstancedStereo) ? ViewMask |0x3:
ViewMask;

                                FPrimitiveSceneInfo* PrimitiveSceneInfo = InScene-
> Primitives[PrimitiveIndex];
                                constFPrimitiveBounds& Bounds = InScene->PrimitiveBounds[PrimitiveIndex]; //Will
                                FPrimitiveSceneProxyThe information is set to the collector.
                                Collector.SetPrimitive(PrimitiveSceneInfo->Proxy, PrimitiveSceneInfo-
>DefaultDynamicHitProxyId);
                                //Set dynamic grid custom data.
                                SetDynamicMeshElementViewCustomData(InViews, HasViewCustomDataMasks,
PrimitiveSceneInfo);
```

```
                    //markDynamicMeshEndIndicesThe
                    beginning of. if(PrimitiveIndex >0) {

                        for(int32 ViewIndex =0; ViewIndex < ViewCount; ViewIndex++) {

                            InViews[ViewIndex].DynamicMeshEndIndices[PrimitiveIndex -1] =
Collector.GetMeshBatchCount(ViewIndex);
                        }
                    }

                    //Get the data of a dynamic grid element.
                    PrimitiveSceneInfo->Proxy->GetDynamicMeshElements(InViewFamily.Views, ViewMaskFinal,
InViewFamily,       Collector);

                    //markDynamicMeshEndIndicesThe end of.
                    for(int32 ViewIndex =0; ViewIndex < ViewCount; ViewIndex++) {

                        InViews[ViewIndex].DynamicMeshEndIndices[PrimitiveIndex] =
Collector.GetMeshBatchCount(ViewIndex);
                    }

                    //deal withMeshPassRelated data and tags.
                    for(int32 ViewIndex =0; ViewIndex < ViewCount; ViewIndex++) {

                        if(ViewMaskFinal & (1<< ViewIndex)) {

                            FViewInfo& View = InViews[ViewIndex];
                            const boolbAddLightmapDensityCommands = View.Family-
> EngineShowFlags.LightMapDensity && AllowDebugViewmodes();
                            constFPrimitiveViewRelevance& ViewRelevance =
View.PrimitiveViewRelevanceMap[PrimitiveIndex];

                            constint32 LastNumDynamicMeshElements =
View.DynamicMeshElementsPassRelevance.Num();

View.DynamicMeshElementsPassRelevance.SetNum(View.DynamicMeshElements.Num());

                            for(int32 ElementIndex = LastNumDynamicMeshElements; ElementIndex
<  View.DynamicMeshElements.Num(); ++ElementIndex)
                            {
                                constFMeshBatchAndRelevance& MeshBatch =
View.DynamicMeshElements[ElementIndex];
                                FMeshPassMask& PassRelevance =
View.DynamicMeshElementsPassRelevance[ElementIndex];
                                //This will calculate the currentMeshBatchWhich ones will beMeshPassReference, thus adding toviewCorrespondence
MeshPassin the array of .
                                ComputeDynamicMeshRelevance(ShadingPath,
bAddLightmapDensityCommands, ViewRelevance, MeshBatch, View, PassRelevance,
PrimitiveSceneInfo, Bounds);
                            }
                        }
                    }
                }
            }
        }

        (.....)
```

```
        //Collectors execute tasks.
        MeshCollector.ProcessTasks();
}
```

From the above code, we can see that when collecting dynamic graphics metadata, an FMeshElementCollector object will be created for each FSceneRenderer to collect all visible FPrimitiveSceneProxy mesh data in the scene. The key code in the middle PrimitiveSceneInfo->Proxy->GetDynamicMeshElements()is to give each graphics object the opportunity to add visible graphics elements to the renderer (collector). Let's expand this function to see (since FPrimitiveSceneProxythis interface of the base class is an empty function body and does not perform any operation, this collection operation is implemented by the specific subclass, and the implementation of the subclass is taken asFSkeletalMeshSceneProxyan example here):

```
// Engine\Source\Runtime\Engine\Private\SkeletalMesh.cpp

void FSkeletalMeshSceneProxy::GetDynamicMeshElements(constTArray<constFSceneView*>& Views,const
FSceneViewFamily& ViewFamily, uint32 VisibilityMap, FMeshElementCollector& Collector)const

{
        GetMeshElementsConditionallySelectable(Views, ViewFamily,true, VisibilityMap, Collector);

}

void FSkeletalMeshSceneProxy::GetMeshElementsConditionallySelectable(constTArray<const FSceneView*>& Views,
constFSceneViewFamily& ViewFamily,boolbInSelectable, uint32 VisibilityMap, FMeshElementCollector& Collector)const
{

        (......)

        constint32 LODIndex = MeshObject->GetLOD();
        constFSkeletalMeshLODRenderData& LODData = SkeletalMeshRenderData-
> LODRenderData[LODIndex];

        if( LODSections.Num() >0&& LODIndex >= SkeletalMeshRenderData->CurrentFirstLODIdx ) {

                constFLODSectionElements& LODSection = LODSections[LODIndex]

                //according toLODTraverse all sub-models and add them tocollectormiddle.
                for(FSkeletalMeshSectionIter Iter(LODIndex, *MeshObject, LODData, LODSection);
Iter;      + + Iter)
                {
                        constFSkelMeshRenderSection& Section = Iter.GetSection(); constint32
                        SectionIndex = Iter.GetSectionElementIndex();
                        constFSectionElementInfo& SectionElementInfo = Iter.GetSectionElementInfo();

boolbSectionSelected              =false;              if(MeshObject-
>IsMaterialHidden(LODIndex,
SectionElementInfo.UseMaterialIndex) || Section.bDisabled)
                        {
                                continue;
                        }
                        //Will specifyLODIndexandSectionIndexJoinCollectormiddle. GetDynamicElementsSection(Views,
                        ViewFamily, VisibilityMap, LODData, LODIndex,
SectionIndex, bSectionSelected, SectionElementInfo, bInSelectable, Collector);
```

```
            }
        }

        (......)
}

void FSkeletalMeshSceneProxy::GetDynamicElementsSection(constTArray<constFSceneView*>& Views,const
FSceneViewFamily& ViewFamily, uint32 VisibilityMap,const FSkeletalMeshLODRenderData& LODData,constint32
LODIndex,constint32 SectionIndex,bool bSectionSelected,constFSectionElementInfo& SectionElementInfo,bool
bInSelectable, FMeshElementCollector& Collector )const {


        constFSkelMeshRenderSection& Section = LODData.RenderSections[SectionIndex]; const bool
        bIsSelected =false;
        const boolbIsWireframe = ViewFamily.EngineShowFlags.Wireframe;

        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                if(VisibilityMap & (1<< ViewIndex)) {

                        constFSceneView* View = Views[ViewIndex];

                        //fromColloctorAssign oneFMeshBatch. FMeshBatch& Mesh
                        = Collector.AllocateMesh();

                        //Create the base grid batch object (FMeshBatchElementExample).
                        CreateBaseMeshBatch(View, LODData, LODIndex, SectionIndex, SectionElementInfo,
Mesh);

                        if(!Mesh.VertexFactory) {

                                // hide this part
                                continue;
                        }

                        Mesh.bWireframe |= bForceWireframe;
                        Mesh.Type = PT_TriangleList; Mesh.bSelectable
                        = bInSelectable;

                        //Set the firstFMeshBatchElementObject. FMeshBatchElement&
                        BatchElement = Mesh.Elements[0];
                        const boolbRequiresAdjacencyInformation = RequiresAdjacencyInformation(
SectionElementInfo.Material, Mesh.VertexFactory->GetType(), ViewFamily.GetFeatureLevel() );

                        if( bRequiresAdjacencyInformation ) {

                                check(LODData.AdjacencyMultiSizeIndexContainer.IsIndexBufferValid() );
                                BatchElement.IndexBuffer =
LODData.AdjacencyMultiSizeIndexContainer.GetIndexBuffer();
                                Mesh.Type = PT_12_ControlPointPatchList;
                                BatchElement.FirstIndex *=4;
                        }

                        BatchElement.MinVertexIndex = Section.BaseVertexIndex;
                        Mesh.ReverseCulling = IsLocalToWorldDeterminantNegative();
                        Mesh.CastShadow = SectionElementInfo.bEnableShadowCasting;
                        Mesh.bCanApplyViewModeOverrides =true;
                        Mesh.bUseWireframeSelectionColoring = bIsSelected;
```

```
                (......)

                if( ensureMsgf(Mesh.MaterialRenderProxy, TEXT("GetDynamicElementsSection with
invalid MaterialRenderProxy. Owner:%s LODIndex:%d UseMaterialIndex:%d"),
* GetOwnerName().ToString(), LODIndex, SectionElementInfo.UseMaterialIndex) &&
                        ensureMsgf(Mesh.MaterialRenderProxy->GetMaterial(FeatureLevel),
TEXT("GetDynamicElementsSection with invalid FMaterial. Owner:%s LODIndex:%d UseMaterialIndex:
%d"), *GetOwnerName().ToString(), LODIndex, SectionElementInfo.UseMaterialIndex) )

                {
                    //WillFMeshBatchAdd to the collector.
                    Collector.AddMesh(ViewIndex, Mesh);
                }

                (......)
            }
        }
}
```

It can be seen that FSkeletalMeshSceneProxy will add an FMeshBatch to each Section mesh according to different LOD indexes, and each FMeshBatch has only one FMeshBatchElement instance. In addition, FSceneRender::GatherDynamicMeshElementsthere is a key sentence in the logic ComputeDynamicMeshRelevance,which is used to calculate which MeshPasses will reference the current MeshBatch, and then add it to the count of the corresponding MeshPass of the view:

```
// Engine\Source\Runtime\Renderer\Private\SceneVisibility.cpp

void ComputeDynamicMeshRelevance(EShadingPath ShadingPath,bool bAddLightmapDensityCommands,
constFPrimitiveViewRelevance& ViewRelevance,const FMeshBatchAndRelevance& MeshBatch, FViewInfo&
View, FMeshPassMask& PassMask, FPrimitiveSceneInfo* PrimitiveSceneInfo,constFPrimitiveBounds&
Bounds) {

    constint32 NumElements = MeshBatch.Mesh->Elements.Num();

    //depthPass/hostPasscount.
    if(ViewRelevance.bDrawRelevance && (ViewRelevance.bRenderInMainPass ||
ViewRelevance.bRenderCustomDepth || ViewRelevance.bRenderInDepthPass))
    {
        PassMask.Set(EMeshPass::DepthPass);
        View.NumVisibleDynamicMeshElements[EMeshPass::DepthPass] += NumElements;

        if(ViewRelevance.bRenderInMainPass || ViewRelevance.bRenderCustomDepth) {

            PassMask.Set(EMeshPass::BasePass);
            View.NumVisibleDynamicMeshElements[EMeshPass::BasePass] += NumElements;

            if(ShadingPath == EShadingPath::Mobile) {

                PassMask.Set(EMeshPass::MobileBasePassCSM);
                View.NumVisibleDynamicMeshElements[EMeshPass::MobileBasePassCSM]                 + =
NumElements;
            }

            if(ViewRelevance.bRenderCustomDepth)
```

```
                {
                    PassMask.Set(EMeshPass::CustomDepth);
                    View.NumVisibleDynamicMeshElements[EMeshPass::CustomDepth]  += NumElements;
                }

                if(bAddLightmapDensityCommands) {

                    PassMask.Set(EMeshPass::LightmapDensity);
                    View.NumVisibleDynamicMeshElements[EMeshPass::LightmapDensity]  +=
NumElements;
                }


                if(ViewRelevance.bVelocityRelevance) {

                    PassMask.Set(EMeshPass::Velocity);
                    View.NumVisibleDynamicMeshElements[EMeshPass::Velocity]  += NumElements;
                }

                if(ViewRelevance.bOutputsTranslucentVelocity) {

                    PassMask.Set(EMeshPass::TranslucentVelocity);
                    View.NumVisibleDynamicMeshElements[EMeshPass::TranslucentVelocity]  +=
NumElements;
                }

                if(ViewRelevance.bUsesSingleLayerWaterMaterial) {

                    PassMask.Set(EMeshPass::SingleLayerWaterPass);
                    View.NumVisibleDynamicMeshElements[EMeshPass::SingleLayerWaterPass]  +=
NumElements;
                }
            }
        }

        // Translucent and othersPasscount.
        if (ViewRelevance.HasTranslucency() &&
                !ViewRelevance.bEditorPrimitiveRelevance
            && ViewRelevance.bRenderInMainPass)
        {
            if (View.Family->AllowTranslucencyAfterDOF())
            {
                if(ViewRelevance.bNormalTranslucency) {

                    PassMask.Set(EMeshPass::TranslucencyStandard);
                    View.NumVisibleDynamicMeshElements[EMeshPass::TranslucencyStandard]  +=
NumElements;
                }

                if(ViewRelevance.bSeparateTranslucency) {

                    PassMask.Set(EMeshPass::TranslucencyAfterDOF);
                    View.NumVisibleDynamicMeshElements[EMeshPass::TranslucencyAfterDOF]  +=
NumElements;
                }

                if(ViewRelevance.bSeparateTranslucencyModulate) {
```

```
                    PassMask.Set(EMeshPass::TranslucencyAfterDOFModulate);

View.NumVisibleDynamicMeshElements[EMeshPass::TranslucencyAfterDOFModulate] NumElements;  + =

                }
            }
            else
            {
                PassMask.Set(EMeshPass::TranslucencyAll);
                View.NumVisibleDynamicMeshElements[EMeshPass::TranslucencyAll]                  +=NumElements;
            }

            if(ViewRelevance.bDistortion) {

                PassMask.Set(EMeshPass::Distortion);
                View.NumVisibleDynamicMeshElements[EMeshPass::Distortion]                + = NumElements;
            }

            if(ShadingPath == EShadingPath::Mobile && View.bIsSceneCapture) {

                PassMask.Set(EMeshPass::MobileInverseOpacity);
                View.NumVisibleDynamicMeshElements[EMeshPass::MobileInverseOpacity]                  + =
NumElements;
            }
        }

        (.....)
}
```

The above code also involves a collectorFMeshElementCollector,which is used to collect all visible MeshBatch information of the specified view. The declaration is as follows:

```
// Engine\Source\Runtime\Engine\Public\SceneManagement.h

classFMeshElementCollector {

public:
    //Interface for drawing points, lines, surfaces, and sprites.
    FPrimitiveDrawInterface*GetPDI(int32 {                    ViewIndex)

        returnSimpleElementCollectors[ViewIndex];
    }
    //Assign oneFMeshBatchObject.
    FMeshBatch&AllocateMesh() {

        constint32 Index = MeshBatchStorage.Add(1); return
        MeshBatchStorage[Index];
    }

    //IncreaseMeshBatchto the collector. When adding, the relevant data will be initialized and set, and then added to
    MeshBatchesList. void AddMesh(int32 ViewIndex, FMeshBatch& MeshBatch);

    //Data acquisition interface.
    FGlobalDynamicIndexBuffer& FGlobaGlDetyDnyanmaimcViceIrntedxeBxuBfuffefre&r( );
    FGlobalDynamicReadBuffer&GetDynamGeictRDeyandaBmuifcfVeerr()t;exBuffer();
```

```
        uint32GetMeshBatchCount(uint32 ViewIndex)const; uint32
        GetMeshElementCount(uint32 ViewIndex)const;
        ERHIFeatureLevel::TypeGetFeatureLevel()const;

        void RegisterOneFrameMaterialProxy(FMaterialRenderProxy* Proxy);
        template<typename T, typename... ARGS>
        T&AllocateOneFrameResource(ARGS&&...Args); bool
        ShouldUseTasks()const;

        //Task interface.
        void AddTask(TFunction<void()>&& Task) {

                ParallelTasks.Add(new (FMemStack::Get())                TFunction<void()>(MoveTemp(Task)));
        }
        void AddTask(constTFunction<void()>& Task) {

                ParallelTasks.Add(new (FMemStack::Get())                TFunction<void()>(Task));
        }
        void ProcessTasks();

protected:
        FMeshElementCollector(ERHIFeatureLevel::Type InFeatureLevel);

        //set upFPrimitiveSceneProxyof data.
        void SetPrimitive(constFPrimitiveSceneProxy* InPrimitiveSceneProxy, FHitProxyId DefaultHitProxyId)

        {
                check(InPrimitiveSceneProxy); PrimitiveSceneProxy =
                InPrimitiveSceneProxy;

                for(int32 ViewIndex =0; ViewIndex < SimpleElementCollectors.Num(); ViewIndex++) {

                        SimpleElementCollectors[ViewIndex]->HitProxyId = DefaultHitProxyId;
                        SimpleElementCollectors[ViewIndex]->PrimitiveMeshId =0;
                }

                for(int32 ViewIndex =0; ViewIndex < MeshIdInPrimitivePerView.Num(); ++ViewIndex) {

                        MeshIdInPrimitivePerView[ViewIndex] =0;
                }
        }

        void ClearViewMeshArrays();

        //TowardsViewAdd a groupMesh.
        void AddViewMeshArrays(
                FSceneView* InView,
                TArray<FMeshBatchAndRelevance,SceneRenderingAllocator>*                 ViewMeshes,
                FSimpleElementCollector* ViewSimpleElementCollector,
                TArray<FPrimitiveUniformShaderParameters>*                 InDynamicPrimitiveShaderData,
                ERHIFeatureLevel::Type InFeatureLevel,
                FGlobalDynamicIndexBuffer*          InDynamicIndexBuffer,
                FGlobalDynamicVertexBuffer* FGlobalIDnyDnyanmamicRiceVaedrBteuxfBfeurf*fe r,
                InDynamicReadBuffer);

        TChunkedArray<FMeshBatch> MeshBatchStorage;//Save all assignedFMeshBatchExamples.
        TArray<TArray<FMeshBatchAndRelevance, SceneRenderingAllocator>*, TInlineAllocator<2> > MeshBatches;//Need
to be renderedFMeshBatchExamples
```

```
    TArray<int32, TInlineAllocator<2> > NumMeshBatchElementsPerView;//EachviewCollected MeshBatchElement
    quantity.
    TArray<FSimpleElementCollector*, TInlineAllocator<2> > SimpleElementCollectors;//A collector of simple objects such as
point, line, and surface sprites.

    TArray<FSceneView*, TInlineAllocator<2> > Views;//Collected by the collectorFSceneViewExamples. TArray<uint16,
    TInlineAllocator<2> > MeshIdInPrimitivePerView;// Current Mesh Id In Primitive per view

    TArray<TArray<FPrimitiveUniformShaderParameters>*, TInlineAllocator<2> >
DynamicPrimitiveShaderDataPerView;//viewDynamic metadata for updating toGPU Scenemiddle.

    TArray<FMaterialRenderProxy*, SceneRenderingAllocator> TemporaryProxies;
    TArray<FOneFrameResource*, SceneRenderingAllocator> OneFrameResources;

    constFPrimitiveSceneProxy* PrimitiveSceneProxy;//Currently collectingPrimitiveSceneProxy

    //Global dynamic buffering.
    FGlobalDynamicIndexBuffer* FGlobaDlDyynnaammicicInVdeertxeBxuBfufeffre;r*
    FGlobalDynamicReadBuffer* DynamicDReyandaBmuicffVeerr;texBuffer;


    ERHIFeatureLevel::Type FeatureLevel;

    const boolbUseAsyncTasks;//Whether to use asynchronous tasks.
    TArray<TFunction<void()>*, SceneRenderingAllocator> ParallelTasks;//A list of tasks waiting to be processed after dynamic mesh
data has been collected.
};
```

There is a one-to-one correspondence between FMeshElementCollector and FSceneRenderer, and each FSceneRenderer has a collector. After the collector collects the list of visible elements of the corresponding view, it usually has a list of FMeshBatch that needs to be rendered, as well as their management data and status, to collect and prepare enough for the subsequent process.

In addition, after collecting the mesh data, FMeshElementCollector can also specify a list of tasks that need to be processed to achieve synchronization of multi-threaded parallel processing.

3.2.3 From FMeshBatch to FMeshDrawCommand

The previous section talked about collecting dynamic MeshElements. In fact, it will be called SetupMeshPassto createFMeshPassProcessor:

```
void FSceneRender::Render(FRHICommandListImmediate& RHICmdList) {

    bool FDeferredShadingSceneRenderer::InitViews((FRHICommandListImmediate& RHICmdList,
. . . )
    {
        voidFSceneRender::ComputeViewVisibility(FRHICommandListImmediate& RHICmdList,
. . . )
        {
            //Collect dynamicsMeshElement
            FSceneRender::GatherDynamicMeshElements(Views, Scene, ViewFamily,
DynamicIndexBuffer, DynamicVertexBuffer, DynamicReadBuffer, HasDynamicMeshElementsMasks,
HasDynamicEditorMeshElementsMasks, HasViewCustomDataMasks, MeshCollector);
```

```
                //Process allviewofFMeshPassProcessor.
                for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                    FViewInfo& View = Views[ViewIndex]; if(!
                    View.ShouldRenderView()) {

                        continue;
                    }

                    //Processing AssignmentviewofFMeshPassProcessor.
                    FViewCommands& ViewCommands = ViewCommandsPerView[ViewIndex];
                    SetupMeshPass(View, BasePassDepthStencilAccess, ViewCommands);
                }
            }
        }
}
```

TheFSceneRenderer::SetupMeshPasslogic and explanation are as follows:

```
void FSceneRenderer::SetupMeshPass(FViewInfo& View, FExclusiveDepthStencil::Type
BasePassDepthStencilAccess, FViewCommands& ViewCommands)
{
    constEShadingPath ShadingPath = Scene->GetShadingPath();

    //TraversalEMeshPassAll definedPass.
    for(int32 PassIndex =0; PassIndex < EMeshPass::Num; PassIndex++) {

        constEMeshPass::Type PassType = (EMeshPass::Type)PassIndex;

        if((FPassProcessorManager::GetPassFlags(ShadingPath, PassType) &
EMeshPassFlags::MainView) != EMeshPassFlags::None)
        {
            (.....)

            //createFMeshPassProcessor PassProcessorCreateFunction
            CreateFunction =
FPassProcessorManager::GetCreateFunction(ShadingPath, PassType);
            FMeshPassProcessor* MeshPassProcessor = CreateFunction(Scene, &View, nullptr);

            //Get the specifiedPassofFParallelMeshDrawCommandPassobject.
            FParallelMeshDrawCommandPass& Pass          =
View.ParallelMeshDrawCommandPasses[PassIndex];

            if(ShouldDumpMeshDrawCommandInstancingStats()) {

                Pass.SetDumpInstancingStats(GetMeshPassName(PassType));
            }

            //Processing visiblePassThe processing task creates thisPassAll drawing
            commands. Pass.DispatchPassSetup(
                Scene,
                View,
                PassType,
                BasePassDepthStencilAccess,
                MeshPassProcessor,
                View.DynamicMeshElements,
```

```
                    &View.DynamicMeshElementsPassRelevance,
                    View.NumVisibleDynamicMeshElements[PassType],
                    ViewCommands.DynamicMeshCommandBuildRequests[PassType],
                    ViewCommands.NumDynamicMeshCommandBuildRequestElements[PassType],
                    ViewCommands.MeshCommands[PassIndex]);
            }
        }
}
```

The enumeration definitions involved in the above codeEMeshPassare as follows:

```
// Engine\Source\Runtime\Renderer\Public\MeshPassProcessor.h

namespace   EMeshPass
{
    enum Type
    {
        DepthPass,                  //    depth
        BasePass,                   //    Geometry/Basics
        SkyPass,                    // Sky
        SingleLayerWaterPass,       //    Single layer of water
        CSMShadowDepth,             //    Cascade Shadow Depth
        Distortion,                 //    Perturbation
        Velocity,                   //speed

        //TransparentPass     Translucent
        Velocity,    TranslucencyStandard,
        TranslucencyAfterDOF,
        TranslucencyAfterDOFModulate,
        TranslucencyAll,


        LightmapDensity,            //Lightmap Intensity
        DebugViewMode,              //Debug View Mode
        CustomDepth,                //Custom Depth
        MobileBasePassCSM,
        MobileInverseOpacity,
        VirtualTexture,                 //Virtual Texturing

        //Special in editor modePass
#if   WITH_EDITOR
        HitProxy,
        HitProxyOpaqueOnly,
        EditorSelection,
#endif

        Num,
        NumBits =5,
    };
}
```

It can be seen that UE lists all the Passes that may need to be drawn in advance, and generates DrawCommand for the required Passes in parallel during the SetupMeshPass stage. The FParallelMeshDrawCommandPass::DispatchPassSetupmain logic and analysis are as follows:

```
// Engine\Source\Runtime\Renderer\Private\MeshDrawCommands.cpp

void FParallelMeshDrawCommandPass::DispatchPassSetup(
	FScene* Scene,
	constFViewInfo& View,
	EMeshPass::Type PassType, FExclusiveDepthStencil::Type
	BasePassDepthStencilAccess, FMeshPassProcessor* MeshPassProcessor,

	constTArray<FMeshBatchAndRelevance, SceneRenderingAllocator>& DynamicMeshElements, const
	TArray<FMeshPassMask, SceneRenderingAllocator>* DynamicMeshElementsPassRelevance,

	int32 NumDynamicMeshElements,
	TArray<constFStaticMeshBatch*, SceneRenderingAllocator>&
	InOutDynamicMeshCommandBuildRequests,
	int32 NumDynamicMeshCommandBuildRequestElements,
	FMeshCommandOneFrameArray& InOutMeshDrawCommands, FMeshPassProcessor*
	MobileBasePassCSMMeshPassProcessor, FMeshCommandOneFrameArray*
	InOutMobileBasePassCSMMeshDrawCommands
)
{
	MaxNumDraws = InOutMeshDrawCommands.Num() + NumDynamicMeshElements +
	NumDynamicMeshCommandBuildRequestElements;

	//set upTaskContextData collected and generatedMeshCommandRequired data. TaskContext.MeshPassProcessor
	= MeshPassProcessor; TaskContext.MobileBasePassCSMMeshPassProcessor =
	MobileBasePassCSMMeshPassProcessor; TaskContext.DynamicMeshElements = &DynamicMeshElements;

	TaskContext.DynamicMeshElementsPassRelevance = DynamicMeshElementsPassRelevance;

	TaskContext.View = &View;
	TaskContext.ShadingPath =	Scene->GetShadingPath(); = Scene-
	TaskContext.ShaderPlatform	>GetShaderPlatform();
	TaskContext.PassType = PassType;
	TaskContext.bUseGPUScene = UseGPUScene(GMaxRHIShaderPlatform, View.GetFeatureLevel());
	TaskContext.bDynamicInstancing	= IsDynamicInstancingEnabled(View.GetFeatureLevel());
	TaskContext.bReverseCulling =	View.bReverseCulling;
	TaskContext.bRenderSceneTwoSided	= View.bRenderSceneTwoSided;
	TaskContext.BasePassDepthStencilAccess	= BasePassDepthStencilAccess;
	TaskContext.DefaultBasePassDepthStencilAccess	= Scene-
> DefaultBasePassDepthStencilAccess; TaskContext.NumDynamicMeshElements =
	NumDynamicMeshElements;
	TaskContext.NumDynamicMeshCommandBuildRequestElements =
	NumDynamicMeshCommandBuildRequestElements;

	// Only apply instancing for ISR to main view passes const boolbIsMainViewPass = PassType !=
	EMeshPass::Num && (FPassProcessorManager::GetPassFlags(TaskContext.ShadingPath,
TaskContext.PassType) EMeshPassFlags::MainView) != EMeshPassFlags::None;	&

	TaskContext.InstanceFactor = (bIsMainViewPass && View.IsInstancedStereoPass()) ?2:
1;

	//Setting based onviewTransparent sort keys
	TaskContext.TranslucencyPass = ETranslucencyPass::TPT_MAX;
	TaskContext.TranslucentSortPolicy	= View.TranslucentSortPolicy;
	TaskContext.TranslucentSortAxis	=View.TranslucentSortAxis;
	TaskContext.ViewOrigin	=View.ViewMatrices.GetViewOrigin();
	TaskContext.ViewMatrix	= View.ViewMatrices.GetViewMatrix();
```

```
	TaskContext.PrimitiveBounds = &Scene->PrimitiveBounds;

	switch(PassType)
	{
		caseEMeshPass::TranslucencyStandard: TaskContext.TranslucencyPass =
ETranslucencyPass::TPT_StandardTranslucency;break;
		caseEMeshPass::TranslucencyAfterDOF: TaskContext.TranslucencyPass =
ETranslucencyPass::TPT_TranslucencyAfterDOF;break;
		caseEMeshPass::TranslucencyAfterDOFModulate: TaskContext.TranslucencyPass =
ETranslucencyPass::TPT_TranslucencyAfterDOFModulate;break;
		caseEMeshPass::TranslucencyAll: TaskContext.TranslucencyPass =
ETranslucencyPass::TPT_AllTranslucency;break;
		caseEMeshPass::MobileInverseOpacity: TaskContext.TranslucencyPass =
ETranslucencyPass::TPT_StandardTranslucency;break;
	}

	//Exchange command list
	FMemory::Memswap(&TaskContext.MeshDrawCommands,		&InOutMeshDrawCommands,
sizeof(InOutMeshDrawCommands));
	FMemory::Memswap(&TaskContext.DynamicMeshCommandBuildRequests,
&InOutDynamicMeshCommandBuildRequests,sizeof(InOutDynamicMeshCommandBuildRequests));

	if(TaskContext.ShadingPath == EShadingPath::Mobile && TaskContext.PassType == EMeshPass::BasePass)

	{
		FMemory::Memswap(&TaskContext.MobileBasePassCSMMeshDrawCommands,
InOutMobileBasePassCSMMeshDrawCommands,sizeof(*InOutMobileBasePassCSMMeshDrawCommands));
	}
	else
	{
		check(MobileBasePassCSMMeshPassProcessor == nullptr &&
InOutMobileBasePassCSMMeshDrawCommands == nullptr);
	}

	if(MaxNumDraws >0) {

		//According to the maximum number of draws (MaxNumDraws)Preallocate resources on the rendering thread.
		bPrimitiveIdBufferDataOwnedByRHIThread =false; TaskContext.PrimitiveIdBufferDataSize =
		TaskContext.InstanceFactor * MaxNumDraws * sizeof(int32);

		TaskContext.PrimitiveIdBufferData =
		FMemory::Malloc(TaskContext.PrimitiveIdBufferDataSize);
		PrimitiveIdVertexBufferPoolEntry =
		GPrimitiveIdVertexBufferPool.Allocate(TaskContext.PrimitiveIdBufferDataSize);
		TaskContext.MeshDrawCommands.Reserve(MaxNumDraws);
		TaskContext.TempVisibleMeshDrawCommands.Reserve(MaxNumDraws);

		const boolbExecuteInParallel = FApp::ShouldUseThreadingForPerformance()
			&& CVarMeshDrawCommandsParallelPassSetup.GetValueOnRenderThread() >0 && GRenderingThread;
			// Rendering thread is required to safely use rendering
resources in parallel.

		//	If it is a parallel mode, create a parallel task instance and joinTaskGraphSystem
		if	execution. (bExecuteInParallel)
		{
			FGraphEventArray DependentGraphEvents;

DependentGraphEvents.Add(TGraphTask<FMeshDrawCommandPassSetupTask>::CreateTask(nullptr,
```

```
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(TaskContext));
                TaskEventRef =
TGraphTask<FMeshDrawCommandInitResourcesTask>::CreateTask(&DependentGraphEvents,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(TaskContext);
            }
            else
            {
                QUICK_SCOPE_CYCLE_COUNTER(STAT_MeshPassSetupImmediate);
                FMeshDrawCommandPassSetupTaskTask(TaskContext);
                Task.AnyThreadTask();
                FMeshDrawCommandInitResourcesTaskDependentTask(TaskContext);
                DependentTask.AnyThreadTask();
            }
        }
}
```

The above code involves several key concepts:FMeshPassProcessor, FMeshDrawCommandPassSetupTaskContext,FMeshDrawCommandPassSetupTask, FMeshDrawCommandInitResourcesTask.The definitions and analysis of the following three concepts are as follows:

```
// Engine\Source\Runtime\Renderer\Private\MeshDrawCommands.h

//Parallel mesh drawing command channel setting task (FMeshDrawCommandPassSetupTask)The
required context. classFMeshDrawCommandPassSetupTaskContext {

public:
    //viewRelated data.
    constFViewInfo* View;
    EShadingPath ShadingPath;
    EShaderPlatform     ShaderPlatform;
    EMeshPass::Type     PassType;
    bool bUseGPUScene;
    bool bDynamicInstancing;
    bool bReverseCulling;
    bool bRenderSceneTwoSided;
    FExclusiveDepthStencil::Type            BasePassDepthStencilAccess;
    FExclusiveDepthStencil::Type            DefaultBasePassDepthStencilAccess;

    //Grid Channel Processor (Mesh pass processor).
    FMeshPassProcessor*     MeshPassProcessor;
    FMeshPassProcessor*     MobileBasePassCSMMeshPassProcessor;
    constTArray<FMeshBatchAndRelevance, SceneRenderingAllocator>* DynamicMeshElements; const
    TArray<FMeshPassMask, SceneRenderingAllocator>* DynamicMeshElementsPassRelevance;


    //Command related data.
    int32   InstanceFactor;
    int32   NumDynamicMeshElements;
    int32 NumDynamicMeshCommandBuildRequestElements;
    FMeshCommandOneFrameArray MeshDrawCommands;
    FMeshCommandOneFrameArray TAMrroabyi<leBasePassCSMMeshDrawCommands;
    constFStaticMeshBatch*,                 SceneRenderingAllocator>
    DynamicMeshCommandBuildRequests;
    TArray<constFStaticMeshBatch*,          SceneRenderingAllocator>
    MobileBasePassCSMDynamicMeshCommandBuildRequests;
```

```
    FDynamicMeshDrawCommandStorage MeshDrawCommandStorage;
    FGraphicsMinimalPipelineStateSet MinimalPipelineStatePassSet; bool
    NeedsShaderInitialisation;

    //Resources that need to be pre-allocated in the rendering thread.
    void* PrimitiveIdBufferData;
    int32 PrimitiveIdBufferDataSize;
    FMeshCommandOneFrameArray TempVisibleMeshDrawCommands;

    //Required for sorting transparent objects.
    ETranslucencyPass::Type TranslucencyPass;
    ETranslucentSortPolicy::Type            TranslucentSortPolicy;
    FVector     TranslucentSortAxis;
    FVector     ViewOrigin;
    FMatrix     ViewMatrix;
    constTArray<structFPrimitiveBounds>* PrimitiveBounds;

    // For logging instancing stats. int32
            VisibleMeshDrawCommandsNum;
    int32     NewPassVisibleMeshDrawCommandsNum;
    int32 MaxInstances;
};


// Engine\Source\Runtime\Renderer\Private\MeshDrawCommands.cpp

//Conversion AssignmentEMeshPassEach ofFMeshBatchTo a groupFMeshDrawCommand. FMeshDrawCommandPassSetupTaskTo be
used.
void GenerateDynamicMeshDrawCommands(
    constFViewInfo& View,
    EShadingPath ShadingPath, EMeshPass::Type
    PassType, FMeshPassProcessor*
    PassMeshProcessor,
    constTArray<FMeshBatchAndRelevance, SceneRenderingAllocator>& DynamicMeshElements, const
    TArray<FMeshPassMask, SceneRenderingAllocator>* DynamicMeshElementsPassRelevance,

    int32 MaxNumDynamicMeshElements,
    constTArray<constFStaticMeshBatch*, SceneRenderingAllocator>&
    DynamicMeshCommandBuildRequests,
    int32 MaxNumBuildRequestElements, FMeshCommandOneFrameArray&
    VisibleCommands, FDynamicMeshDrawCommandStorage&
    MeshDrawCommandStorage, FGraphicsMinimalPipelineStateSet&
    MinimalPipelineStatePassSet, bool& NeedsShaderInitialisation


)
{
    (.....)

    //BuildFDynamicPassMeshDrawListContextInstance, used to passPassMeshProcessorThe generated drawing
    commands. FDynamicPassMeshDrawListContextDynamicPassMeshDrawListContext(
        MeshDrawCommandStorage,
        VisibleCommands,
        MinimalPipelineStatePassSet,
        NeedsShaderInitialisation
    ) ;
    PassMeshProcessor->SetDrawListContext(&DynamicPassMeshDrawListContext);
```

```
    {       //Process dynamic mesh batches.
            constint32 NumCommandsBefore = VisibleCommands.Num(); constint32
            NumDynamicMeshBatches = DynamicMeshElements.Num();

            //Iterate over all dynamic grid batches.
            for(int32 MeshIndex =0; MeshIndex < NumDynamicMeshBatches; MeshIndex++) {

                    if(!DynamicMeshElementsPassRelevance || (*DynamicMeshElementsPassRelevance)
[MeshIndex].Get(PassType))
                    {
                            constFMeshBatchAndRelevance& MeshAndRelevance =
DynamicMeshElements[MeshIndex];
                            check(!MeshAndRelevance.Mesh->bRequiresPerElementVisibility); constuint64
                            BatchElementMask = ~0ull;

                            //WillFMeshBatchJoinPassMeshProcessorto be processed.
                            PassMeshProcessor->AddMeshBatch(*MeshAndRelevance.Mesh,          BatchElementMask,
MeshAndRelevance.PrimitiveSceneProxy);
                    }
            }

            (......)
    }

    //Process batches of static meshes.
    {
            constint32 NumCommandsBefore = VisibleCommands.Num();
            constint32 NumStaticMeshBatches = DynamicMeshCommandBuildRequests.Num();

            for(int32 MeshIndex =0; MeshIndex < NumStaticMeshBatches; MeshIndex++) {

                    constFStaticMeshBatch* StaticMeshBatch =
DynamicMeshCommandBuildRequests[MeshIndex];
                    constuint64 BatchElementMask = StaticMeshBatch->bRequiresPerElementVisibility
? View.StaticMeshBatchVisibility[StaticMeshBatch->BatchVisibilityId]: ~0ull;

                    //WillFMeshBatchJoinPassMeshProcessorto be processed. PassMeshProcessor-
                    >AddMeshBatch(*StaticMeshBatch, BatchElementMask,
StaticMeshBatch->PrimitiveSceneInfo->Proxy, StaticMeshBatch->Id);
            }

            (......)
    }
}

//Parallelize the task of setting up mesh drawing instructions. Includes dynamic mesh drawing command generation,
sorting, merging, etc. classFMeshDrawCommandPassSetupTask {

public:
    FMeshDrawCommandPassSetupTask(FMeshDrawCommandPassSetupTaskContext&          InContext
            : Context(InContext)
    {
    }

    (......)

    void AnyThreadTask()
    {
```

```
        const boolbMobileShadingBasePass = Context.ShadingPath == EShadingPath::Mobile && Context.PassType ==
        EMeshPass::BasePass;
        const boolbMobileVulkanSM5BasePass =
IsVulkanMobileSM5Platform(Context.ShaderPlatform) && Context.PassType ==
EMeshPass::BasePass;

        if(bMobileShadingBasePass) {

            (.....)
        }
        else
        {
            //Generates dynamic and static mesh drawing instructions (viaMeshPassProcessorWillFMeshBatchConvert to
MeshDrawCommand).
            GenerateDynamicMeshDrawCommands(
                * Context.View,
                Context.ShadingPath,
                Context.PassType,
                Context.MeshPassProcessor,
                * Context.DynamicMeshElements,
                Context.DynamicMeshElementsPassRelevance,
                Context.NumDynamicMeshElements,
                Context.DynamicMeshCommandBuildRequests,
                Context.NumDynamicMeshCommandBuildRequestElements,
                Context.MeshDrawCommands,
                Context.MeshDrawCommandStorage,
                Context.MinimalPipelineStatePassSet,
                Context.NeedsShaderInitialisation

            ) ;
        }

        if(Context.MeshDrawCommands.Num() >0) {

            if(Context.PassType != EMeshPass::Num) {

                //applicationviewAlready existing inMeshDrawCommand,     For example: rendering a planar reflection with inverse clipping mode (reverse
culling     mode)
                ApplyViewOverridesToMeshDrawCommands(
                    Context.ShadingPath,
                    Context.PassType,
                    Context.bReverseCulling,
                    Context.bRenderSceneTwoSided,
                    Context.BasePassDepthStencilAccess,
                    Context.DefaultBasePassDepthStencilAccess,
                    Context.MeshDrawCommands,
                    Context.MeshDrawCommandStorage,
                    Context.MinimalPipelineStatePassSet,
                    Context.NeedsShaderInitialisation,
                    Context.TempVisibleMeshDrawCommands

                ) ;
            }

            //Update the sort key.
            if(bMobileShadingBasePass || bMobileVulkanSM5BasePass) {

                (.....)
            }
            else if(Context.TranslucencyPass != ETranslucencyPass::TPT_MAX)
```

```
                {
                    //useviewUpdate the grid with the relevant data sort key. The sort key type isFMeshDrawCommandSortKey,Included
BasePassand transparent keys, where transparent objects are sorted based on their distance from the camera.
                    UpdateTranslucentMeshSortKeys(
                        Context.TranslucentSortPolicy,
                        Context.TranslucentSortAxis,
                        Context.ViewOrigin,
                        Context.ViewMatrix,                *
                        Context.PrimitiveBounds,
                        Context.TranslucencyPass,
                        Context.MeshDrawCommands

                    ) ;
                }

                {
                    QUICK_SCOPE_CYCLE_COUNTER(STAT_SortVisibleMeshDrawCommands); //
                    implementMeshDrawCommandSorting,FCompareFMeshDrawCommandsFirst,
FMeshDrawCommandSortKeyAs the basis for sorting, then useStateBucketId.
                    Context.MeshDrawCommands.Sort(FCompareFMeshDrawCommands());
                }

                if(Context.bUseGPUScene) {

                    //generateGPUScene-related data (mainly all the rendering scenesPrimitive
                    data). BuildMeshDrawCommandPrimitiveIdBuffer(
                        Context.bDynamicInstancing,
                        Context.MeshDrawCommands,
                        Context.MeshDrawCommandStorage,
                        Context.PrimitiveIdBufferData,
                        Context.PrimitiveIdBufferDataSize,
                        Context.TempVisibleMeshDrawCommands,
                        Context.MaxInstances,
                        Context.VisibleMeshDrawCommandsNum,
                        Context.NewPassVisibleMeshDrawCommandsNum,
                        Context.ShaderPlatform,
                        Context.InstanceFactor

                    ) ;
                }
            }
        }

        void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
        MyCompletionGraphEvent)
        {
            AnyThreadTask();
        }

private:
        FMeshDrawCommandPassSetupTaskContext& Context;//The device context.
};


// MeshDrawCommandRequired pre-allocated resources.
classFMeshDrawCommandInitResourcesTask {

public:

        (.....)
```

```
    void AnyThreadTask()
    {
        TRACE_CPUPROFILER_EVENT_SCOPE(MeshDrawCommandInitResourcesTask); if
        (Context.NeedsShaderInitialisation) {

            //Initialize all boundshaderresource.
            for(constFGraphicsMinimalPipelineStateInitializer& Initializer:
Context.MinimalPipelineStatePassSet)
            {
                Initializer.BoundShaderState.LazilyInitShaders();
            }
        }
    }

    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        AnyThreadTask();
    }

private:
    FMeshDrawCommandPassSetupTaskContext&       Context;
};
```

It can be seenFMeshDrawCommandPassSetupTaskthat it plays a very important role in the mesh rendering pipeline, including the generation, sorting, and merging of dynamic mesh drawing and static drawing commands. The key value of the sorting stage isFMeshDrawCommandSortKeydetermined by, and its definition is as follows:

```
// Engine\Source\Runtime\Renderer\Public\MeshPassProcessor.h

// FVisibleMeshDrawCommandThe sort key value of . class
RENDERER_API FMeshDrawCommandSortKey {

public:
    union
    {
        uint64 PackedData;              //After packaging64Bit key value data

        //Geometry channel sort key
        struct
        {
            uint64 VertexShaderHash                 :16;//Low:VSThe hash value of the address.
            uint64 PixelShaderHash                  :32;//Median:PSThe hash value of the
            uint64 Masked                           address. :16;//High: YesMaskedMaterial
        } BasePass;

        //Transparent channel sort key value
        struct
        {
            uint64 MeshIdInPrimitive uint64     :16;//Low: Share the samePrimitiveStable gridid
            Distance                                :32;//Median: distance to the camera :16;//
            uint64 Priority                         High: Priority (as specified by the material)
        } Translucent;
```

```
            //Normal sort key value
            struct
            {
                uint64 VertexShaderHash    :32;//Low:VSThe hash value of the address.
                uint64 PixelShaderHash     :32;//High position:PSThe hash value of the address.
            }Generic;
        };

        //Not equal operator
        FORCEINLINEbooloperator!=(FMeshDrawCommandSortKey B)const {

            returnPackedData != B.PackedData;
        }

        //Less than operator, used for sorting.
        FORCEINLINEbooloperator<(FMeshDrawCommandSortKey B)const {

            returnPackedData < B.PackedData;
        }

        static constFMeshDrawCommandSortKey Default;
};
```

A few additional points need to be made aboveFMeshDrawCommandSortKey:

- FMeshDrawCommandSortKeyAlthough three key values, BasePass, TransparentPass, and OrdinaryPass, can be stored, only one type of data is valid at the same time.
- The key value calculation logic is scattered in different files and stages. For example, the key value of BasePass can occur in the BasePassRendering, DepthRendering and MeshPassProcessor stages. The key value calculation logic and analysis are as follows:

Key Name	Calculation code	Analysis
VertexShaderHash	PointerHash(VertexShader)	A pointer to a hash of the VS used by the material.
PixelShaderHash	PointerHash(PixelShader)	Pointer hash value of the PS used by the material.
Masked	BlendMode == EBlendMode::BLEND_Masked ? 0 : 1	Whether the material's blending mode is Masked.
MeshIdInPrimitive	MeshIdInPrimitivePerView[ViewIndex]	Stable mesh ids that share the same primitive based on the view.
Distance	(uint32)~BitInvertIfNegativeFloat(((*uint32*	Calculate Distance according to ETranslucentSortPolicy, and then reverse

Key Name	Calculation code	Analysis
	) &Distance))	the negative distance.
Priority	-	Derived directly from the transparent sorting priority assigned to the material.

- operator<Directly comparing with PackedData shows that the higher the data bit, the higher the priority. Specifically, the sorting basis of BasePass is first to determine whether it is a Masked material, and then to determine the address hash values of PS and VS. Similarly, the sorting priority of the transparent channel is: the priority specified by the material, the distance from the mesh to the camera, and the mesh ID.

  > Generally speaking, when sorting meshes, the factors that have the greatest impact on performance are given the highest priority.
  > For example, in the BasePass stage, Masked materials will seriously hinder parallel efficiency and throughput on some GPU devices, and are ranked at the highest position; and PS is often higher than VS in terms of the number of instructions and computational complexity, so it is reasonable to rank before VS.
  > However, the order of transparent channels is special, that is, the distance between the object and the camera. To correctly draw the front and back relationship of semitransparent objects, they must be drawn from far to near, otherwise the front and back relationship will be disordered. Therefore, the transparent channel must put the distance in the highest position (highest priority).

- PackedData packs several groups of data into a single oneuint64,so only one comparison is needed, which can improve the efficiency of sorting. Otherwise, according to the traditional writing method, using severalif-elsestatements will inevitably increase the number of CPU instructions and reduce the sorting efficiency.
- Modify key values and related sorting logic to customize sorting priorities and algorithms. For example, add several sorting dimensions: texture, vertex data, rendering status, etc.

Next, we will explain some important concepts:FMeshPassProcessorandFMeshDrawCommands,which appear many times in the above code. TheyFMeshPassProcessorplay the role ofFMeshBatch conversion .FMeshDrawCommandsHere are the definitions and analysis of them and related concepts:

```
// Engine\Source\Runtime\Renderer\Public\MeshPassProcessor.h


//Does not include render textures (Render Target) of the rendering pipeline state.RTIts size affects the performance of the mesh drawing instructions traversal.
classFGraphicsMinimalPipelineStateInitializer {

public:
    //RTRelated data: pixel format, tags.
```

```
using TRenderTargetFormats = TStaticArray<uint8/*EPixelFormat*/,
MaxSimultaneousRenderTargets>;
using TRenderTargetFlags = TStaticArray<uint32, MaxSimultaneousRenderTargets>;

(……)

//Make a copy of your own value and pass it out.
FGraphicsPipelineStateInitializerAsGraphicsPipelineStateInitializer() {                                    const

    returnFGraphicsPipelineStateInitializer
    (BoundShaderState.AsBoundShaderState()
        , BlendState
        , RasterizerState
        , DepthStencilState
        , ImmutableSamplerState
        , PrimitiveType
        ,   0
        , FGraphicsPipelineStateInitializer::TRenderTargetFormats(PF_Unknown)
        , FGraphicsPipelineStateInitializer::TRenderTargetFlags(0)
        , PF_Unknown
        ,   0
        , ERenderTargetLoadAction::ENoAction
        , ERenderTargetStoreAction::ENoAction
        , ERenderTargetLoadAction::ENoAction
        , ERenderTargetStoreAction::ENoAction
        , FExclusiveDepthStencil::DepthNop
        ,   0
        , ESubpassHint::None
        ,   0
        ,   0
        , bDepthBounds
        , bMultiView
        , bHasFragmentDensityAttachment
    ) ;
}

(……)

//calculateFGraphicsMinimalPipelineStateInitializerThe hash value of .
inlinefriend uint32GetTypeHash(constFGraphicsMinimalPipelineStateInitializer& Initializer)

{
    //add and initialize any leftover padding within the struct to avoid unstable key structFHashKey

    {
        uint32    VertexDeclaration;
        uint32    VertexShader;
        uint32    PixelShader;
        uint32    RasterizerState;
    } HashKey;
    HashKey.VertexDeclaration           =
    PointerHash(Initializer.BoundShaderState.VertexDeclarationRHI);
    HashKey.VertexShader =
GetTypeHash(Initializer.BoundShaderState.VertexShaderIndex);
    HashKey.PixelShader = GetTypeHash(Initializer.BoundShaderState.PixelShaderIndex);
    HashKey.RasterizerState = PointerHash(Initializer.RasterizerState);

    returnuint32(CityHash64((const char*)&HashKey,sizeof(FHashKey)));
```

```
    }

    //Compare interfaces.
    booloperator==(constFGraphicsMinimalPipelineStateInitializer& rhs)const; booloperator!=(const
    FGraphicsMinimalPipelineStateInitializer& rhs)const booloperator<(const
    FGraphicsMinimalPipelineStateInitializer& rhs)const; booloperator>(const
    FGraphicsMinimalPipelineStateInitializer& rhs)const;


    //Rendering Pipeline States
    FMinimalBoundShaderStateInput BoundShaderState;  //  Boundshaderstate.
    FRHIBlendState*                 BlendState;          //Mixed state.
    FRHIRasterizerState*          RasterizerState;  //  Rasterization status.
    FRHIDepthStencilState*        DepthStencilState;  //  Deep target state.
    FImmutableSamplerState        ImmutableSamplerState;  //Immutable sampler state.

    //Other status.
    bool                    bDepthBounds  =false;
    bool                    bMultiView =  false;
    bool                    bHasFragmentDensityAttachment Padding[=1] f=a {ls};e//;
    uint8                    Data added due to memory alignment.

    EPrimitiveType          PrimitiveType;
};


//The only representativeFGraphicsMinimalPipelineStateInitializerAn exampleid, used for quick sort.
classFGraphicsMinimalPipelineStateId {

public:
    uint32GetId()const {

        returnPackedId;
    }

    //Judgment and comparison interface.
    inlinebool IsValid()const
    inlinebooloperator==(constFGraphicsMinimalPipelineStateId& rhs)const; inlinebooloperator!=(
    constFGraphicsMinimalPipelineStateId& rhs)const;

    //Get the associatedFGraphicsMinimalPipelineStateInitializer.
    inlineconstFGraphicsMinimalPipelineStateInitializer&GetPipelineState(const
FGraphicsMinimalPipelineStateSet& InPipelineSet)const
    {
        if(bComesFromLocalPipelineStateSet) {

            returnInPipelineSet.GetByElementId(SetElementIndex);
        }

        {
            FScopeLockLock(&PersistentIdTableLock);
            returnPersistentIdTable.GetByElementId(SetElementIndex).Key;
        }
    }

    static void InitializePersistentIds();
    //GetFGraphicsMinimalPipelineStateInitializerThe corresponding permanentpipeline state id. static
    FGraphicsMinimalPipelineStateIdGetPersistentId(const FGraphicsMinimalPipelineStateInitializer&
    InPipelineState);
```

```
    static void RemovePersistentId(FGraphicsMinimalPipelineStateId Id);

    //Get it in the following orderpipeline state id: Global and permanentidTable andPassSetIf none of the parameters are found, a blank instance is created and added toPassSetparameter.
    RENDERER_APIstaticFGraphicsMinimalPipelineStateIdGetPipelineStateId(const FGraphicsMinimalPipelineStateInitializer& InPipelineState, FGraphicsMinimalPipelineStateSet& InOutPassSet, bool& NeedsShaderInitialisation);


private:
    //Packaged key value.
    union
    {
        uint32 PackedId =0;

        struct
        {
            uint32 SetElementIndex                                  :   30;
            uint32 bComesFromLocalPipelineStateSet:1; uint32 bValid
                                                                    :1;
        };
    };

    structFRefCountedGraphicsMinimalPipelineState {

        FRefCountedGraphicsMinimalPipelineState() : {               RefNum(0)

        }
        uint32 RefNum;
    };

    static      FCriticalSection PersistentIdTableLock;
    using    PersistentTableType = Experimental::TRobinHoodHashMap<FGraphicsMinimalPipelineStateInitializer, FRefCountedGraphicsMinimalPipelineState>;
    //lastingidsurface.
    staticPersistentTableType PersistentIdTable;

    static      int32    LocalPipelineIdTableSize;
    static      int32    CurrentLocalPipelineIdTableSize;
    static      bool     NeedsShaderInitialisation;
};

//Grid drawing instructions, which record the drawing of a singleMeshAll resources and data required, and there should be no redundant data, if necessaryInitView Transfer data, availableFVisibleMeshDrawCommand.
//AllFMeshDrawCommandThe referenced resources must have a guaranteed lifecycle, becauseFMeshDrawCommandIt does not manage the lifecycle of resources. classFMeshDrawCommand
{
public:
    //Resource Binding
    FMeshDrawShaderBindings      ShaderBindings;
    FVertexInputStreamArray      VertexStreams;
    FRHIIndexBuffer* IndexBuffer;

    //Cached Render Pipeline State (PSO)
    FGraphicsMinimalPipelineStateId CachedPipelineId;

    //Draw command parameters.
```

```
    uint32    FirstIndex;
    uint32    NumPrimitives;
    uint32    NumInstances;

    //Vertex data, including normal mode and indirect mode.
    union
    {
        struct
        {
            uint32    BaseVertexIndex;
            uint32 NumVertices;
        }VertexParams;

        struct
        {
            FRHIVertexBuffer*        Buffer;
            uint32 Offset;
        } IndirectArgs;
    };

    int8 PrimitiveIdStreamIndex;

    //Non-rendering state parameters.
    uint8 StencilRef;

    //Determine whether it is consistent with the specifiedFMeshDrawCommandMatch, if matched, can be merged into the same instanceto draw. bool MatchesForDynamicInstancing(constFMeshDrawCommand& Rhs)const {

        returnCachedPipelineId == Rhs.CachedPipelineId
            && StencilRef == Rhs.StencilRef
            && ShaderBindings.MatchesForDynamicInstancing(Rhs.ShaderBindings) &&
            VertexStreams == Rhs.VertexStreams
            && PrimitiveIdStreamIndex == Rhs.PrimitiveIdStreamIndex &&
            IndexBuffer == Rhs.IndexBuffer
            && FirstIndex == Rhs.FirstIndex && NumPrimitives
            == Rhs.NumPrimitives && NumInstances ==
            Rhs.NumInstances
            && ((NumPrimitives >0&& VertexParams.BaseVertexIndex ==
Rhs.VertexParams.BaseVertexIndex && VertexParams.NumVertices ==
Rhs.VertexParams.NumVertices)
                || (NumPrimitives ==0&& IndirectArgs.Buffer == Rhs.IndirectArgs.Buffer
&& IndirectArgs.Offset == Rhs.IndirectArgs.Offset));
    }

    //Gets the hash value of a dynamic instance.
    uint32GetDynamicInstancingHash()const {

        //add and initialize any leftover padding within the struct to avoid unstable keys structFHashKey

        {
            uint32 IndexBuffer;
            uint32 VertexBuffers =0; uint32
            VertexStreams =0; uint32
                PipelineId;
            uint32 DynamicInstancingHash;
            uint32    FirstIndex;
            uint32 NumPrimitives;
            uint32 NumInstances;
```

```
                uint32    IndirectArgsBufferOrBaseVertexIndex;
                uint32    NumVertices;
                uint32    StencilRefAndPrimitiveIdStreamIndex;

                //Pointer address hashing
                staticinlineuint32PointerHash(const void* Key) {

#if    PLATFORM_64BITS
                    // Ignoring the lower 4 bits since they are likely zero anyway. // Higher bits are more
                    significant in 64 bit builds.
                    returnreinterpret_cast<UPTRINT>(Key) >>4;
#else
                    returnreinterpret_cast<UPTRINT>(Key);
#endif
                };

                //Hash combination
                staticinlineuint32HashCombine(uint32 A, uint32 B) {

                    returnA ^ (B +0x9e3779b9+ (A <<6) + (A >>2));
                }
            }   HashKey;

            //WillFMeshDrawCommandAll member variables are filled with values    FHashKey HashKey.PipelineId =
            CachedPipelineId.GetId(); HashKey.StencilRefAndPrimitiveIdStreamIndex = StencilRef |
            (PrimitiveIdStreamIndex
<< 8);
            HashKey.DynamicInstancingHash = ShaderBindings.GetDynamicInstancingHash();

            for(intindex =0; index < VertexStreams.Num(); index++) {

                constFVertexInputStream& VertexInputStream = VertexStreams[index]; constuint32
                StreamIndex = VertexInputStream.StreamIndex; constuint32 Offset =
                VertexInputStream.Offset;

                uint32 Packed = (StreamIndex <<28) | Offset;
                HashKey.VertexStreams = FHashKey::HashCombine(HashKey.VertexStreams, Packed);
                HashKey.VertexBuffers = FHashKey::HashCombine(HashKey.VertexBuffers,
FHashKey::PointerHash(VertexInputStream.VertexBuffer));
            }

            HashKey.IndexBuffer        =  FHashKey::PointerHash(IndexBuffer);
            HashKey.FirstIndex         =    FirstIndex;
            HashKey.NumPrimitives          =  NumPrimitives;
            HashKey.NumInstances       =  NumInstances;

            if(NumPrimitives >0) {

                HashKey.IndirectArgsBufferOrBaseVertexIndex                =  VertexParams.BaseVertexIndex;
                HashKey.NumVertices = VertexParams.NumVertices;
            }
            else
            {
                HashKey.IndirectArgsBufferOrBaseVertexIndex =
FHashKey::PointerHash(IndirectArgs.Buffer);
                HashKey.NumVertices = IndirectArgs.Offset;
            }
```

```
        //FilledHashKeyConverted to hash value, the data is exactly the sameHashKeyAlways have the same hash value, which makes it easy to determine whether batch rendering is possible.
        returnuint32(CityHash64((char*)&HashKey,sizeof(FHashKey)));
    }

    (......)

    //WillFMeshBatchThe relevant data is processed and passed toFMeshDrawCommand
    middle. void SetDrawParametersAndFinalize(
        const    FMeshBatch& MeshBatch,
        int32 BatchElementIndex,
        FGraphicsMinimalPipelineStateId              PipelineId,
        constFMeshProcessorShaders* ShadersForDebugging)
    {
        constFMeshBatchElement& BatchElement = MeshBatch.Elements[BatchElementIndex];

        IndexBuffer = BatchElement.IndexBuffer ? BatchElement.IndexBuffer-
> IndexBufferRHI.GetReference()              : nullptr;
        FirstIndex =         BatchElement.FirstIndex; =
        NumPrimitives          BatchElement.NumPrimitives;
        NumInstances = BatchElement.NumInstances;

        if(NumPrimitives >0) {

            VertexParams.BaseVertexIndex VertexP=araBmatsc.hNEulemmVenrti.cBeass e=V ertexIndex;
            BatchElement.MaxVertexIndex -
BatchElement.MinVertexIndex +1;
        }
        else
        {
            IndirectArgs.Buffer          =  BatchElement.IndirectArgsBuffer;
            IndirectArgs.Offset          =  BatchElement.IndirectArgsOffset;
        }

        Finalize(PipelineId, ShadersForDebugging);
    }

    //savePipelineIdandshaderDebug information.
    void Finalize(FGraphicsMinimalPipelineStateId PipelineId,constFMeshProcessorShaders* ShadersForDebugging)

    {
        CachedPipelineId = PipelineId;
        ShaderBindings.Finalize(ShadersForDebugging);
    }

    /** Submits commands to the RHI Commandlist to draw the MeshDrawCommand. */ static void
    SubmitDraw(
        constFMeshDrawCommand& RESTRICT MeshDrawCommand,
        constFGraphicsMinimalPipelineStateSet& FRHIVertexBGufrfaeprh*icsMinimalPipelineStateSet,
                                    ScenePrimitiveIdsBuffer,
        int32 PrimitiveIdOffset,
        uint32     InstanceFactor,
        FRHICommandList& CommandList,
        class FMeshDrawCommandStateCache& RESTRICT StateCache);

    (......)
};
```

```
//  Visible mesh drawing instructions. Stores the information required for mesh drawing instructions that have been determined to be visible for subsequent visibility processing.
//  andFMeshDrawCommandThe difference is,FVisibleMeshDrawCommandShould only be storedInitViewsRequired for operation (visibility/sorting)
Book, and there should be no data related to drawing submissions.
classFVisibleMeshDrawCommand {

public:
    (......)

    //RelatedFMeshDrawCommandExample.
    constFMeshDrawCommand* MeshDrawCommand; //Key-value based on
    stateless sorting. (e.g. transparent drawing instructions based on depth
    sorting) FMeshDrawCommandSortKey SortKey;
    //Drawing Primitivesid, can be used fromPrimitiveSceneDataofSRVGets the metadata of the image.DrawPrimitiveIdCan be traced back
FPrimitiveSceneInfoExample.
    int32 DrawPrimitiveId;
    //ProductionFVisibleMeshDrawCommandScene primitivesid,in the case of-1It means noFPrimitiveSceneInfo, can be traced
backFPrimitiveSceneInfoExample.
    int32 ScenePrimitiveId;
    // Offset into the buffer of PrimitiveIds built for this pass, in int32's. int32 PrimitiveIdBufferOffset;


    //dynamicinstancingStatus Bucketid  (Dynamic instancing state bucket ID). //
        All the sameStateBucketIdDrawing instructions can be combined into the sameinstancingmiddle.
    //    - 1Indicates that other factors replaceStateBucketId
    int32     Sort. StateBucketId;

    // Needed for view overrides
    ERasterizerFillMode MeshFillMode : ERasterizerFillMode_NumBits +1; ERasterizerCullMode
    MeshCullMode : ERasterizerCullMode_NumBits +1;
};


//Grid Channel Processor
classFMeshPassProcessor {

public:

    //The following scenario,view,contextThe data is passed in
    by the construction function. constFScene* RESTRICT
    Scene; ERHIFeatureLevel::Type FeatureLevel; const
    FSceneView* ViewIfDynamicMeshCommand;
    FMeshPassDrawListContext* DrawListContext;


    (......)

    //IncreaseFMeshBatchInstances, by specific subclassesPassaccomplish.
    virtualvoid AddMeshBatch(constFMeshBatch& RESTRICT MeshBatch, uint64 BatchElementMask,const
FPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy, int32 StaticMeshId =-1) =0;



    //Mesh drawing strategy override settings.
    structFMeshDrawingPolicyOverrideSettings {

        EDrawingPolicyOverrideFlags                MeshOverrideFlags       =
        EDrawingPolicyOverrideFlags::None;
        EPrimitiveType                             MeshPrimitiveType       =  PT_TriangleList;
    };
```

```
(……)

//Will1indivualFMeshBatchConvert to1or moreMeshDrawCommands.
template<typename PassShadersType, typename ShaderElementDataType> void
BuildMeshDrawCommands(
    constFMeshBatch& RESTRICT MeshBatch, uint64
    BatchElementMask,
    constFPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy, const
    FMaterialRenderProxy& RESTRICT MaterialRenderProxy, constFMaterial&
    RESTRICT MaterialResource,
    constFMeshPassProcessorRenderState& RESTRICT DrawRenderState,
    PassShadersType PassShaders,

    ERasterizerFillMode        MeshFillMode,
    ERasterizerCullMode        MeshCullMode,
    FMeshDrawCommandSortKey SortKey,
    EMeshPassFeatures MeshPassFeatures,
    constShaderElementDataType& ShaderElementData)
{
    constFVertexFactory* RESTRICT VertexFactory = MeshBatch.VertexFactory; constFPrimitiveSceneInfo*
    RESTRICT PrimitiveSceneInfo = PrimitiveSceneProxy ? PrimitiveSceneProxy->GetPrimitiveSceneInfo() :
    nullptr;

    // FMeshDrawCommandInstance, used to collect various rendering resources and
    data. FMeshDrawCommand SharedMeshDrawCommand;

    //deal withFMeshDrawCommandTemplate data for .
    SharedMeshDrawCommand.SetStencilRef(DrawRenderState.GetStencilRef());

    //Rendering state instance.
    FGraphicsMinimalPipelineStateInitializer PipelineState; PipelineState.PrimitiveType =
    (EPrimitiveType)MeshBatch.Type; PipelineState.ImmutableSamplerState =
    MaterialRenderProxy.ImmutableSamplerState;

    //deal withFMeshDrawCommandThe vertex data,shaderand render status.
    EVertexInputStreamType InputStreamType = EVertexInputStreamType::Default; if
    ((MeshPassFeatures & EMeshPassFeatures::PositionOnly) != EMeshPassFeatures::Default)
                                                InputStreamType =
EVertexInputStreamType::PositionOnly;
    if((MeshPassFeatures & EMeshPassFeatures::PositionAndNormalOnly) !=
EMeshPassFeatures::Default) InputStreamType                =
EVertexInputStreamType::PositionAndNormalOnly;

    FRHIVertexDeclaration* VertexDeclaration = VertexFactory-
> GetDeclaration(InputStreamType);
    SharedMeshDrawCommand.SetShaders(VertexDeclaration,
PassShaders.GetUntypedShaders(), PipelineState);

    PipelineState.RasterizerState = GetStaticRasterizerState<true>(MeshFillMode, MeshCullMode);

    PipelineState.BlendState = DrawRenderState.GetBlendState(); PipelineState.DepthStencilState =
    DrawRenderState.GetDepthStencilState();

    VertexFactory->GetStreams(FeatureLevel, InputStreamType,
    SharedMeshDrawCommand.VertexStreams);

    SharedMeshDrawCommand.PrimitiveIdStreamIndex = VertexFactory-
> GetPrimitiveIdStreamIndex(InputStreamType);
```

```
            //deal withVS/PS/GSwaitshaderThe binding
            data. int32 DataOffset =0;
            if(PassShaders.VertexShader.IsValid()) {

                    FMeshDrawSingleShaderBindings ShaderBindings              =
 SharedMeshDrawCommand.ShaderBindings.GetSingleShaderBindings(SF_Vertex,                  DataOffset);
                    PassShaders.VertexShader->GetShaderBindings(Scene, FeatureLevel,
PrimitiveSceneProxy, MaterialRenderProxy, MaterialResource, DrawRenderState, ShaderElementData,
ShaderBindings);
            }

            if(PassShaders.PixelShader.IsValid()) {

                    FMeshDrawSingleShaderBindings ShaderBindings =
SharedMeshDrawCommand.ShaderBindings.GetSingleShaderBindings(SF_Pixel,                  DataOffset);
                    PassShaders.PixelShader->GetShaderBindings(Scene, FeatureLevel,
PrimitiveSceneProxy, MaterialRenderProxy, MaterialResource, DrawRenderState, ShaderElementData,
ShaderBindings);
            }

            (......)

            constint32 NumElements = MeshBatch.Elements.Num();

            //Traverse theFMeshBatchAllMeshBatchElement,Get from materialFMeshBatchElementAll relatedshader The type of
bound data.
            for(int32 BatchElementIndex =0; BatchElementIndex < NumElements;
 BatchElementIndex++)
            {
                    if((1ull<< BatchElementIndex) & BatchElementMask) {

                            constFMeshBatchElement& BatchElement =
MeshBatch.Elements[BatchElementIndex];
                            FMeshDrawCommand& MeshDrawCommand = DrawListContext-
> AddCommand(SharedMeshDrawCommand, NumElements);

                            DataOffset =0;
                            if(PassShaders.VertexShader.IsValid()) {

                                    FMeshDrawSingleShaderBindings VertexShaderBindings              =
MeshDrawCommand.ShaderBindings.GetSingleShaderBindings(SF_Vertex, DataOffset);

FMeshMaterialShader::GetElementShaderBindings(PassShaders.VertexShader, ViewIfDynamicMesShcCeonme,mand,
VertexFactory, InputStreamType, FeatureLevel, PrimitiveSceneProxy, MeshBatch, BatchElement, ShaderElementData,
VertexShaderBindings, MeshDrawCommand.VertexStreams);

                            }

                            if(PassShaders.PixelShader.IsValid()) {

                                    FMeshDrawSingleShaderBindings PixelShaderBindings =
MeshDrawCommand.ShaderBindings.GetSingleShaderBindings(SF_Pixel, DataOffset);
                                    FMeshMaterialShader::GetElementShaderBindings(PassShaders.PixelShader,
Scene, ViewIfDynamicMeshCommand, VertexFactory, EVertexInputStreamType::Default, FeatureLevel,
PrimitiveSceneProxy, MeshBatch, BatchElement, ShaderElementData, PixelShaderBindings,
MeshDrawCommand.VertexStreams);
                            }
```

```
                (.....)

                //Process and obtainPrimitiveId.
                int32 DrawPrimitiveId;
                int32 ScenePrimitiveId;
                GetDrawCommandPrimitiveId(PrimitiveSceneInfo,          BatchElement,
DrawPrimitiveId,  ScenePrimitiveId);

                //Final treatmentMeshDrawCommand
                FMeshProcessorShaders ShadersForDebugging          =
PassShaders.GetUntypedShaders();
                DrawListContext->FinalizeCommand(MeshBatch,          BatchElementIndex,
DrawPrimitiveId, ScenePrimitiveId, MeshFillMode, MeshCullMode, SortKey, PipelineState, &ShadersForDebugging,
MeshDrawCommand);
            }
        }
    }

protected:
    RENDERER_APIvoid GetDrawCommandPrimitiveId(
        constFPrimitiveSceneInfo* RESTRICT PrimitiveSceneInfo, const
            FMeshBatchElement&  BatchElement,
        int32& DrawPrimitiveId,
        int32& ScenePrimitiveId)  const;
};
```

> It is used several times when calculating key values aboveCityHash64.CityHash64It is an algorithm for calculating the hash value of any number of strings. It is a fast non-encrypted hash function. Its implementation code is in Engine\Source\Runtime\Core\Private\Hash\CityHash.cpp.
> Those who are interested can study it by themselves.
> Similar hash algorithms include: HalfMD5, MD5, SipHash64, SipHash128, IntHash32, IntHash64, SHA1, SHA224, SHA256, etc.

FMeshDrawCommandIt holds all the information needed by the RHI to draw the grid. This information is platform-independent and graphics API-independent (stateless), and is based on a data-driven design so that its device context can be shared.

FMeshPassProcessor::AddMeshBatchImplemented by subclasses, each subclass usually corresponds to EMeshPassa channel of the enumeration. Its common subclasses are:

- FDepthPassMeshProcessor: Depth channel mesh processor, corresponding EMeshPass::DepthPass.
- FBasePassMeshProcessor: Geometry pass mesh processor, correspondingEMeshPass::BasePass .
- FCustomDepthPassMeshProcessor: Custom depth channel mesh processor, corresponding EMeshPass::CustomDepth.
- FShadowDepthPassMeshProcessor: Shadow channel mesh processor, corresponding EMeshPass::CSMShadowDepth.

- FTranslucencyDepthPassMeshProcessor: Transparent depth channel mesh processor, no corresponding oneEMeshPass.
- FLightmapDensityMeshProcessor: Light map mesh processor, corresponding EMeshPass::LightmapDensity.
- .....

Different PassesFMeshBatchare processed differently. Here is the most common FBasePassMeshProcessorone:

```
// Engine\Source\Runtime\Renderer\Private\BasePassRendering.cpp

void FBasePassMeshProcessor::AddMeshBatch(constFMeshBatch& RESTRICT MeshBatch, uint64 BatchElementMask,
constFPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy, int32 StaticMeshId)

{
	if(MeshBatch.bUseForMaterial) {

		(......)

		if(bShouldDraw
			&& (!PrimitiveSceneProxy || PrimitiveSceneProxy->ShouldRenderInMainPass()) &&
				ShouldIncludeDomainInMeshPass(Material.GetMaterialDomain())
			&& ShouldIncludeMaterialInDefaultOpaquePass(Material))
		{
			(......)

			//  Handling simple forward rendering
			if
(IsSimpleForwardShadingEnabled(GetFeatureLevelShaderPlatform(FeatureLevel)))
			{
				AddMeshBatchForSimpleForwardShading(
					MeshBatch,
					BatchElementMask,
					StaticMeshId,
					PrimitiveSceneProxy,
					MaterialRenderProxy,
					Material,
					LightMapInteraction,
					bIsLitMaterial,
					bAllowStaticLighting,
					bUseVolumetricLightmap,
					bAllowIndirectLightingCache,
					MeshFillMode,
					MeshCullMode);

			}
			//Rendering volumetric transparent self-shadowing objects
			else if	(bIsLitMaterial
				&& bIsTranslucent
				&& PrimitiveSceneProxy
				&& PrimitiveSceneProxy->CastsVolumetricTranslucentShadow())
			{
				(......)

				if(bIsLitMaterial
```

```
                && bAllowStaticLighting
                && bUseVolumetricLightmap
                && PrimitiveSceneProxy)
            {
                Process<    FSelfShadowedVolumetricLightmapPolicy           > (
                    MeshBatch,
                    BatchElementMask,
                    StaticMeshId,
                    PrimitiveSceneProxy,
                    MaterialRenderProxy,
                    Material,
                    BlendMode,
                    ShadingModels,
                    FSelfShadowVolumetricLightmapPolicy(),
                    ElementData,
                    MeshFillMode,
                    MeshCullMode);

            }

            (......)
        }
        //Depending on the lightmap options and quality levels, callProcessto be processed. else


        {
             static constautoCVarSupportLowQualityLightmap =
 IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.SupportLowQualityLightmaps"));
            const boolbAllowLowQualityLightMaps = (!CVarSupportLowQualityLightmap) ||
(CVarSupportLowQualityLightmap->GetValueOnAnyThread() !=0);

            switch(LightMapInteraction.GetType()) {

            caseLMIT_Texture:
                if(bAllowHighQualityLightMaps) {

                    constFShadowMapInteraction ShadowMapInteraction =
(bAllowStaticLighting && MeshBatch.LCI && bIsLitMaterial)
                        ?MeshBatch.LCI->GetShadowMapInteraction(FeatureLevel)
                        : FShadowMapInteraction();

                    if(ShadowMapInteraction.GetType() == SMIT_Texture) {

                        Process< FUniformLightMapPolicy >(
                            MeshBatch,
                            BatchElementMask,
                            StaticMeshId,
                            PrimitiveSceneProxy,
                            MaterialRenderProxy,
                            Material,
                            BlendMode,
                            ShadingModels,

FUniformLightMapPolicy(LMP_DISTANCE_FIELD_SHADOWS_AND_HQ_LIGHTMAP),
                            MeshBatch.LCI,
                            MeshFillMode,
                            MeshCullMode);
                    }

                    (......)
```

```
                        }

                        (.....)

                        break;
                    default:
                        if(bIsLitMaterial
                            && bAllowStaticLighting
                            && Scene
                            &&Scene->VolumetricLightmapSceneData.HasData()
                            && PrimitiveSceneProxy
                            && (PrimitiveSceneProxy->IsMovable()
                                ||PrimitiveSceneProxy->NeedsUnbuiltPreviewLighting()
                                ||PrimitiveSceneProxy->GetLightmapType() ==
ELightmapType::ForceVolumetric))
                        {
                            Process<    FUniformLightMapPolicy >(
                                MeshBatch,
                                BatchElementMask,
                                StaticMeshId,
                                PrimitiveSceneProxy,
                                MaterialRenderProxy,
                                Material,
                                BlendMode,
                                ShadingModels,

FUniformLightMapPolicy(LMP_PRECOMPUTED_IRRADIANCE_VOLUME_INDIRECT_LIGHTING),
                                MeshBatch.LCI,
                                MeshFillMode,
                                MeshCullMode);
                        }

                        (.....)

                        break;
                    };
                }
            }
        }
}

// FBasePassMeshProcessorHandling different lightmap types (shaderbindings, rendering state, sort keys, vertex data,
etc.), and finally callBuildMeshDrawCommandsWillFMeshBatchConvert toFMeshDrawCommands. template<typename
LightMapPolicyType> void FBasePassMeshProcessor::Process(

    constFMeshBatch& RESTRICT MeshBatch, uint64
        BatchElementMask,
    int32 StaticMeshId,
    constFPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy, const
    FMaterialRenderProxy& RESTRICT MaterialRenderProxy, constFMaterial&
    RESTRICT MaterialResource, EBlendMode BlendMode,

    FMaterialShadingModelField ShadingModels, const
    LightMapPolicyType& RESTRICT LightMapPolicy,
    consttypename LightMapPolicyType::ElementDataType& RESTRICT LightMapElementData,
    ERasterizerFillMode     MeshFillMode,
    ERasterizerCullMode     MeshCullMode)
{
```

```
	constFVertexFactory* VertexFactory = MeshBatch.VertexFactory;

	const boolbRenderSkylight = Scene && Scene->ShouldRenderSkylightInBasePass(BlendMode) ShadingModels.IsLit();
&&
	const boolbRenderAtmosphericFog = IsTranslucentBlendMode(BlendMode) && (Scene && Scene-
>HasAtmosphericFog() && Scene->ReadOnlyCVARCache.bEnableAtmosphericFog);

	TMeshProcessorShaders<
		TBasePassVertexShaderPolicyParamType<LightMapPolicyType>, FBaseHS,

		FBaseDS,
		TBasePassPixelShaderPolicyParamType<LightMapPolicyType>>		BasePassShaders;

	//Gets the specified lightmap strategy type.shader.
	GetBasePassShaders<LightMapPolicyType>(
		MaterialResource,
		VertexFactory->GetType(),
		LightMapPolicy,
		FeatureLevel,
		bRenderAtmosphericFog,
		bRenderSkylight,
		Get128BitRequirement(),
		BasePassShaders.HullShader,
		BasePassShaders.DomainShader,
		BasePassShaders.VertexShader,
		BasePassShaders.PixelShader );



	//Rendering state handling.
	FMeshPassProcessorRenderStateDrawRenderState(PassDrawRenderState);

	SetDepthStencilStateForBasePass(
		ViewIfDynamicMeshCommand,
		DrawRenderState,
		FeatureLevel,
		MeshBatch,
		StaticMeshId,
		PrimitiveSceneProxy,
		bEnableReceiveDecalOutput);


	if(bTranslucentBasePass) {

		SetTranslucentRenderState(DrawRenderState, MaterialResource,
GShaderPlatformForFeatureLevel[FeatureLevel], TranslucencyPassType);
	}

	//initializationShaderMaterial book.
	TBasePassShaderElementData<LightMapPolicyType>ShaderElementData(LightMapElementData);
	ShaderElementData.InitializeMeshMaterialData(ViewIfDynamicMeshCommand, PrimitiveSceneProxy, MeshBatch,
StaticMeshId,true);

	//Processes sort key values.
	FMeshDrawCommandSortKey SortKey = FMeshDrawCommandSortKey::Default;

	if(bTranslucentBasePass) {

		SortKey = CalculateTranslucentMeshStaticSortKey(PrimitiveSceneProxy,
		MeshBatch.MeshIdInPrimitive);
```

```
        }
        else
        {
            SortKey = CalculateBasePassMeshStaticSortKey(EarlyZPassMode, BlendMode,
BasePassShaders.VertexShader.GetShader(), BasePassShaders.PixelShader.GetShader());
        }

        //WillFMeshBatchThe elements are converted into
        FMeshDrawCommands. BuildMeshDrawCommands(
            MeshBatch,
            BatchElementMask,
            PrimitiveSceneProxy,
            MaterialRenderProxy,
            MaterialResource,
            DrawRenderState,
            BasePassShaders,
            MeshFillMode,
            MeshCullMode,        SortKey,
            EMeshPassFeatures::Default,
            ShaderElementData);


}
```

It can be seen thatFMeshPassProcessorThe main functions of are:

- Pass filtering: Filter out MeshBatch that is irrelevant to the Pass, such as filtering out transparent objects in the depth Pass.
- Select the Shader and rendering state (depth, template, blending state, rasterization state, etc.) required for the drawing command.
  Collect Shader resource bindings involved in drawing commands.
-
    - Pass's Uniform Buffer, such as ViewUniformBuffer, DepthPassUniformBuffer.
    - Vertex factory bindings (vertex data and indices).
    - Material binding.
    - The bindings associated with the drawing instructions of the Pass.
- Collect Draw Call related parameters.

FMeshPassProcessor::BuildMeshDrawCommands It will be called in the final stage FMeshPassDrawListContext::FinalizeCommand . FMeshPassDrawListContextIt provides two basic interfaces, which are abstract classes. The derived classes are FDynamicPassMeshDrawListContext and FCachedPassMeshDrawListContext,which represent the context of dynamic mesh drawing instructions and cached mesh drawing instructions respectively. Their interfaces and analysis are as follows:

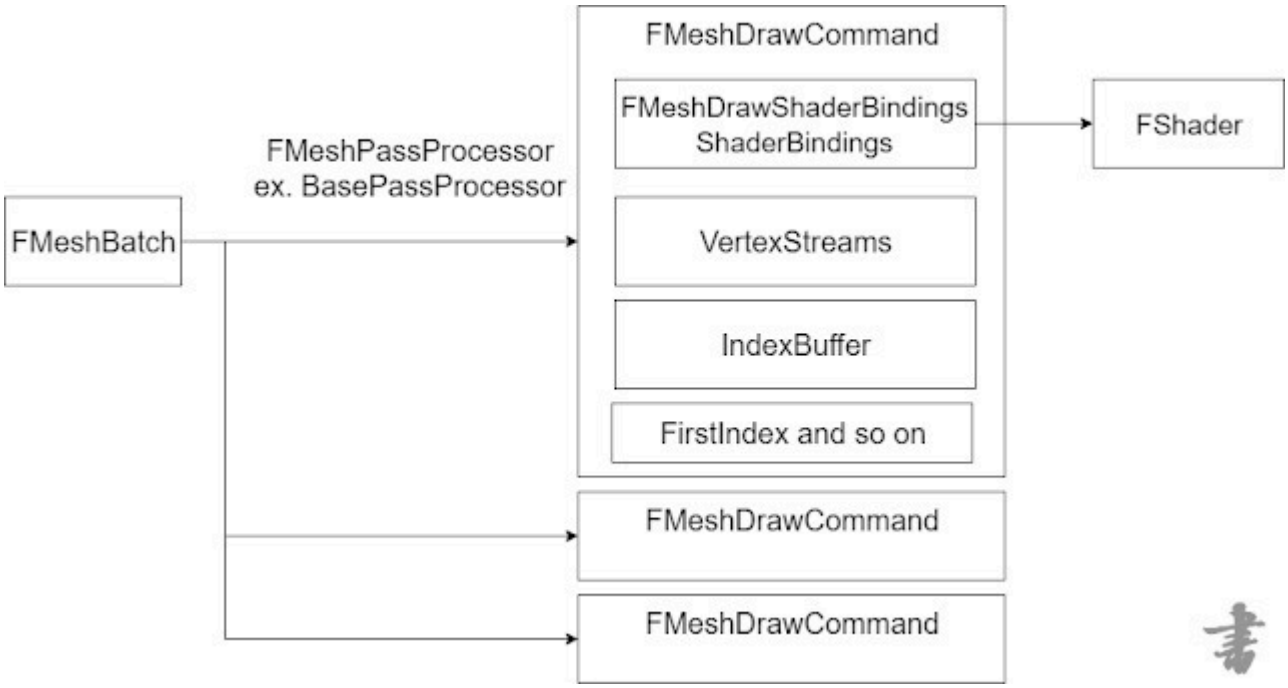
```
// Engine\Source\Runtime\Renderer\Public\MeshPassProcessor.h

//Grid channel draw list context.
classFMeshPassDrawListContext {

public:
    virtual FMeshDrawCommand&AddCommand(FMeshDrawCommand& Initializer, uint32
```

```
NumElements) =0;
    virtual    void FinalizeCommand(
        const   FMeshBatch& MeshBatch,
        int32 BatchElementIndex,
        int32 DrawPrimitiveId,
        int32 ScenePrimitiveId,
        ERasterizerFillMode     MeshFillMode,
        ERasterizerCullMode     MeshCullMode,
        FMeshDrawCommandSortKey SortKey,
        const   FGraphicsMinimalPipelineStateInitializer&          PipelineState,
        const   FMeshProcessorShaders* ShadersForDebugging,
        FMeshDrawCommand& MeshDrawCommand) =0;
};

//[Dynamic] Grid channel drawing list context.
classFDynamicPassMeshDrawListContext: public FMeshPassDrawListContext {

public:
    (……)

    virtual FMeshDrawCommand&AddCommand(FMeshDrawCommand& Initializer, uint32
NumElements) override final
    {
        //WillFMeshDrawCommandAdd to the list and return its index in the array.
        constint32 Index = DrawListStorage.MeshDrawCommands.AddElement(Initializer);
        FMeshDrawCommand& NewCommand = DrawListStorage.MeshDrawCommands[Index]; return
        NewCommand;
    }

    virtualvoid        FinalizeCommand(
        const   FMeshBatch& MeshBatch,
        int32 BatchElementIndex,
        int32 DrawPrimitiveId,
        int32 ScenePrimitiveId,
        ERasterizerFillMode     MeshFillMode,
        ERasterizerCullMode     MeshCullMode,
        FMeshDrawCommandSortKey SortKey,
        const   FGraphicsMinimalPipelineStateInitializer&          PipelineState,
        const   FMeshProcessorShaders* ShadersForDebugging,
        FMeshDrawCommand& MeshDrawCommand) override final
    {
        //Get the rendering pipelineId
FGraphicsMinimalPipelineStateId PipelineId =
FGraphicsMinimalPipelineStateId::GetPipelineStateId(PipelineState,
GraphicsMinimalPipelineStateSet, NeedsShaderInitialisation);

        //rightFMeshBatchThe data is processed and saved toMeshDrawCommandmiddle.
        MeshDrawCommand.SetDrawParametersAndFinalize(MeshBatch, BatchElementIndex, PipelineId,
        ShadersForDebugging);

        //createFVisibleMeshDrawCommand,and willFMeshDrawCommandWaiting for data to be
        filled into it. FVisibleMeshDrawCommand NewVisibleMeshDrawCommand;
        NewVisibleMeshDrawCommand.Setup(&MeshDrawCommand, DrawPrimitiveId,
ScenePrimitiveId,-1, MeshFillMode, MeshCullMode, SortKey);
        //Join directly toTArrayIndicates that the dynamic mode is not merged and instantiated
        MeshDrawCommand. DrawList.Add(NewVisibleMeshDrawCommand);
    }
```

```
private:
    //saveFMeshDrawCommandThe data structure used isTChunkedArray.
    FDynamicMeshDrawCommandStorage& DrawListStorage;
    // FVisibleMeshDrawCommandList, the data structure used isTArray, which internally referencesFMeshDrawCommandPointer, the data pointed to is stored inDrawListStorage.
    FMeshCommandOneFrameArray& DrawList; //
    PSOgather.
    FGraphicsMinimalPipelineStateSet& GraphicsMinimalPipelineStateSet;

    bool& NeedsShaderInitialisation;
};


//[Cache] Grid channel drawing list context.
classFCachedPassMeshDrawListContext: public FMeshPassDrawListContext {

public:
    FCachedPassMeshDrawListContext(FCachedMeshDrawCommandInfo& InCommandInfo, FCriticalSection& InCachedMeshDrawCommandLock, FCachedPassMeshDrawList& InCachedDrawLists, FStateBucketMap& InCachedMeshDrawCommandStateBuckets,constFScene& InScene);

    virtual FMeshDrawCommand&AddCommand(FMeshDrawCommand& Initializer, uint32 NumElements) override final
    {
        if(NumElements ==1) {

            returnInitializer;
        }
        else
        {
            MeshDrawCommandForStateBucketing = Initializer; return
            MeshDrawCommandForStateBucketing;
        }
    }

    virtualvoid         FinalizeCommand(
        const    FMeshBatch& MeshBatch,
        int32 BatchElementIndex,
        int32 DrawPrimitiveId,
        int32 ScenePrimitiveId,
        ERasterizerFillMode         MeshFillMode,
        ERasterizerCullMode         MeshCullMode,
        FMeshDrawCommandSortKey SortKey,
        const    FGraphicsMinimalPipelineStateInitializer&                PipelineState,
        const    FMeshProcessorShaders* ShadersForDebugging,
        FMeshDrawCommand& MeshDrawCommand) override final
    {
        FGraphicsMinimalPipelineStateId PipelineId =
        FGraphicsMinimalPipelineStateId::GetPersistentId(PipelineState);

        MeshDrawCommand.SetDrawParametersAndFinalize(MeshBatch, BatchElementIndex, PipelineId,
        ShadersForDebugging);

        if(UseGPUScene(GMaxRHIShaderPlatform, GMaxRHIFeatureLevel)) {

            Experimental::FHashElementId SetId;
            autohash = CachedMeshDrawCommandStateBuckets.ComputeHash(MeshDrawCommand); {
```

```
                FScopeLockLock(&CachedMeshDrawCommandLock);

                (……)

                //Lookup from cache hash tablehashofid,If it does not exist, add a new one. Thus achieving a mergeFMeshDrawCommand
purpose.
                SetId = CachedMeshDrawCommandStateBuckets.FindOrAddIdByHash(hash,
MeshDrawCommand, FMeshDrawCommandCount());
                //Count plus1
                CachedMeshDrawCommandStateBuckets.GetByElementId(SetId).Value.Num++;

                (……)
            }

            CommandInfo.StateBucketId = SetId.GetIndex();
        }
        else
        {
            FScopeLock Lock(&CachedMeshDrawCommandLock);
            // Only one FMeshDrawCommand supported per FStaticMesh in a pass
            // Allocate at lowest free index so that 'r.DoLazyStaticMeshUpdate' can shrink
the TSparseArray more effectively
            CommandInfo.CommandIndex =
CachedDrawLists.MeshDrawCommands.EmplaceAtLowestFreeIndex(CachedDrawLists.LowestFreeIndexSearchStart,
MeshDrawCommand);
}

        //Store other data.
        CommandInfo.SortKey = SortKey;
        CommandInfo.MeshFillMode    =  MeshFillMode;
        CommandInfo.MeshCullMode    =  MeshCullMode;
    }

private:
    FMeshDrawCommand  MeshDrawCommandForStateBucketing;  FCachedMeshDrawCommandInfo&  CommandInfo;
    FCriticalSection& CachedMeshDrawCommandLock; FCachedPassMeshDrawList& CachedDrawLists;
    FStateBucketMap& CachedMeshDrawCommandStateBuckets;//Robin Hood hash table, automatically merges and counts hashes with the


same hash valueFMeshDrawCommand.
    constFScene& Scene;
};
```

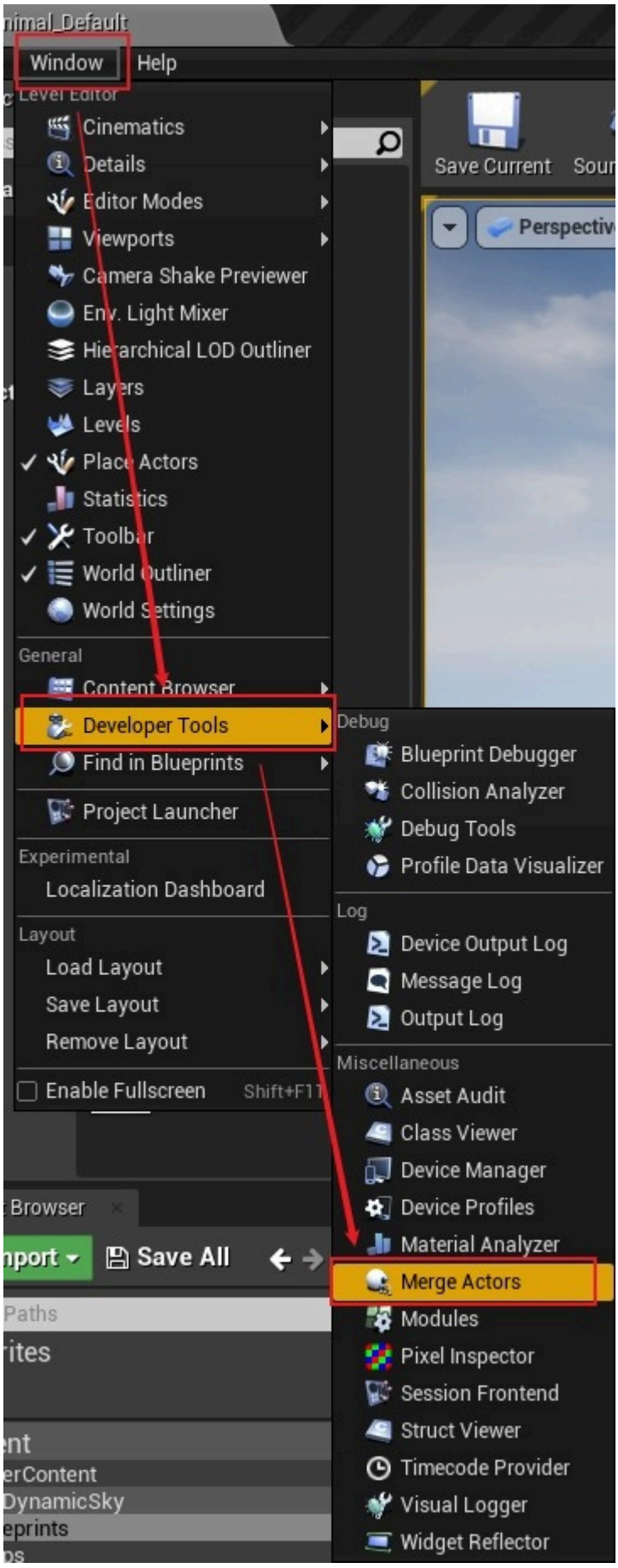
It can be seen that fromFMeshBatchthe toFMeshDrawCommandstage, the renderer does a lot of processing in order toFMeshBatchconvert toFMeshDrawCommandand save it to the FMeshPassDrawListContext member variable of FMeshPassProcessor. During this period, all the data
required for the mesh drawing instructions are collected or processed from various objects in order to enter the subsequent rendering process. The following figure shows these key processes:

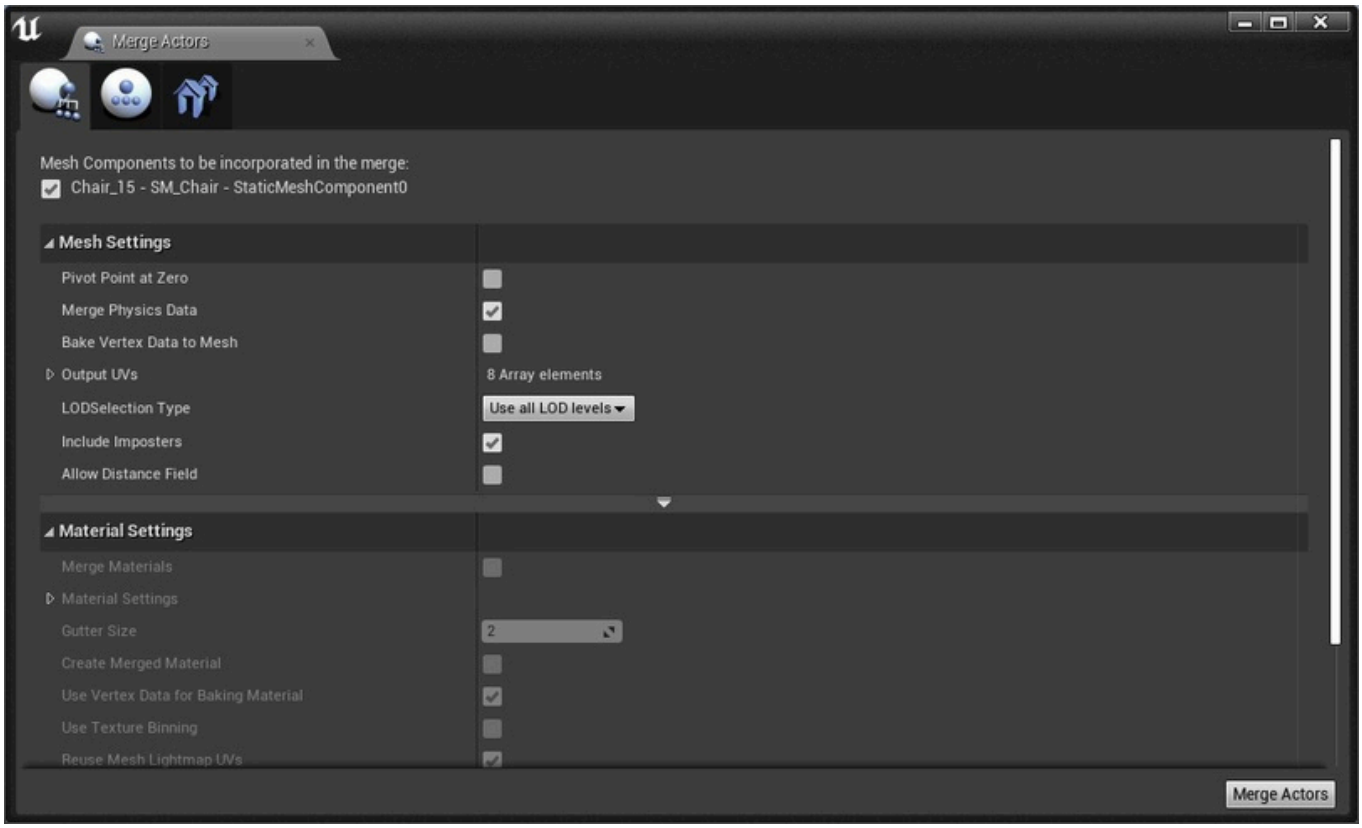


RegardingFMeshDrawCommandthe merging, it is necessary to add that the dynamic drawing path modeFDynamicPassMeshDrawListContextisFMeshDrawCommandstoredTArrayin the structure and will not be mergedFMeshDrawCommand,nor will the grid be dynamically instantiated, but it can improve the robustness of state-based sorting.

The cached (static) drawing path modeFCachedPassMeshDrawListContextonFStateBucketMap merging and counting functionality to implement instantiate drawing during the draw commit phase.

In addition, UE does not have a dynamic batch merging function like Unity, only manual merging of meshes in the editor stage (see the figure below).


Window
Help
Cinematics
Details
Editor Modes
Viewports
Camera Shake Previewer
Env. Light Mixer
Hierarchical LOD Outliner
Layers
Levels
Place Actors
Statistics
Toolbar
World Outliner
World Settings
General
Content Browser
Developer Tools
Find in Blueprints
Project Launcher
Experimental
Localization Dashboard
Layout
Load Layout
Save Layout
Remove Layout
Enable Fullscreen
Shift+F11
Debug
Blueprint Debugger
Collision Analyzer
Debug Tools
Profile Data Visualizer
Log
Device Output Log
Message Log
Output Log
Miscellaneous
Asset Audit
Class Viewer
Device Manager
Device Profiles
Material Analyzer
Merge Actors
Modules
Pixel Inspector
Session Frontend
Struct Viewer
Timecode Provider
Visual Logger
Widget Reflector
Save Current
Save All
Paths



How to open the built-in Actor Merge tool in the UE Editor and its interface preview.

3.2.4 From FMeshDrawCommand to RHICommandList

The previous section has explained in detail how to convert FMeshBatch to FMeshDrawCommand. This section will explain the subsequent steps, that is, how to convert FMeshDrawCommand to RHICommandList, and what processing and optimization are done during the process.

After FMeshBatch is converted into FMeshDrawCommand, each Pass corresponds to an FMeshPassProcessor. Each FMeshPassProcessor saves all FMeshDrawCommands that need to be drawn in this Pass, so that the renderer can trigger and render at the appropriate time. Take the simplest PrePass (depth Pass) as an example:

```
void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

    (.....)

    // FMeshBatchConvert toFMeshDrawCommandThe logic isInitViewsFinish
    InitViews(RHICmdList, BasePassDepthStencilAccess, ILCTaskData,
    UpdateViewCustomDataEvents);

    (.....)

    //RenderingPrePass(depthPass)
    RenderPrePass(FRHICommandListImmediate&          RHICmdList,      TFunctionRef<void()>
    AfterTasksAreStarted)
    {
        boolbParallel = GRHICommandList.UseParallelAlgorithms() &&
        CVarParallelPrePass.GetValueOnRenderThread();
```

```
        (......)

        if(EarlyZPassMode != DDM_None) {

            const boolbWaitForTasks = bParallel &&
(CVarRHICmdFlushRenderThreadTasksPrePass.GetValueOnRenderThread() >0||
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0);

            //Traverse allview, eachviewRender depth oncePass.
            for(int32 ViewIndex =0;ViewIndex < Views.Num();ViewIndex++) {

                constFViewInfo& View = Views[ViewIndex];

                //Processing DepthPassRendering resources and status.
                TUniformBufferRef<FSceneTexturesUniformParameters> PassUniformBuffer;
                CreateDepthPassUniformBuffer(RHICmdList, View, PassUniformBuffer);

                FMeshPassProcessorRenderStateDrawRenderState(View, PassUniformBuffer);

                SetupDepthPassState(DrawRenderState);

                if(View.ShouldRenderView()) {

                    Scene->UniformBuffers.UpdateViewUniformBuffer(View);

                    if(bParallel)
                    {
                        //Parallel rendering depthPass.
                        bDepthWasCleared = RenderPrePassViewParallel(View, RHICmdList,
DrawRenderState, AfterTasksAreStarted, !bDidPrePre) || bDepthWasCleared;
                        bDidPrePre =true;
                    }
                    (......)
                }

                (......)
            }
        }

        (......)
    }
}

// Engine\Source\Runtime\Renderer\Private\DepthRendering.cpp

//Parallel rendering depthPassinterface
bool FDeferredShadingSceneRenderer::RenderPrePassViewParallel(constFViewInfo& View,
FRHICommandListImmediate& ParentCmdList,constFMeshPassProcessorRenderState& DrawRenderState,
TFunctionRef<void()> AfterTasksAreStarted,boolbDoPrePre) {

    boolbDepthWasCleared =false;

    {
        //Constructs a drawing instruction storage container.
        FPrePassParallelCommandListSetParallelCommandListSet(View, this, ParentCmdList,
            CVarRHICmdPrePassDeferredContexts.GetValueOnRenderThread() >0,
            CVarRHICmdFlushRenderThreadTasksPrePass.GetValueOnRenderThread() ==0&&
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() ==0,
```

```
                DrawRenderState);

            //Triggers parallel drawing.

View.ParallelMeshDrawCommandPasses[EMeshPass::DepthPass].DispatchDraw(&ParallelCommandList Set,
ParentCmdList);

            (......)
        }

        (......)

        returnbDepthWasCleared;
}


// Engine\Source\Runtime\Renderer\Private\MeshDrawCommands.cpp

void FParallelMeshDrawCommandPass::DispatchDraw(FParallelCommandListSet*
ParallelCommandListSet, FRHICommandList& RHICmdList)const {

        (......)

        FRHIVertexBuffer* PrimitiveIdsBuffer = PrimitiveIdVertexBufferPoolEntry.BufferRHI; constint32
        BasePrimitiveIdsOffset =0;

        if(ParallelCommandListSet) {

            (......)

            constENamedThreads::Type RenderThread = ENamedThreads::GetRenderThread();

            //Process the preceding tasks.
            FGraphEventArray       Prereqs;
            if    (ParallelCommandListSet->GetPrereqs())
            {
                Prereqs.Append(*ParallelCommandListSet->GetPrereqs());
            }
            if    (TaskEventRef.IsValid())
            {
                Prereqs.Add(TaskEventRef);
            }

            //Constructs the same number of parallel drawing tasks as the
            number of worker threads. constint32 NumThreads =
FMath::Min<int32> (FTaskGraphInterface::Get().GetNumWorkerThreads(),       ParallelCommandListSet->Width);
            constint32 NumTasks = FMath::Min<int32>(NumThreads, FMath::DivideAndRoundUp(MaxNumDraws,
ParallelCommandListSet->MinDrawsPerCommandList));
            constint32 NumDrawsPerTask = FMath::DivideAndRoundUp(MaxNumDraws, NumTasks);

            //TraversalNumTasksSecond, constructionNumTasksDrawing tasks (FDrawVisibleMeshCommandsAnyThreadTask)Reality
example.
            for(int32 TaskIndex =0; TaskIndex < NumTasks; TaskIndex++) {

                constint32 StartIndex = TaskIndex * NumDrawsPerTask;
                constint32 NumDraws = FMath::Min(NumDrawsPerTask, MaxNumDraws - StartIndex);
                checkSlow(NumDraws >0);
```

```
                FRHICommandList* CmdList = ParallelCommandListSet->NewParallelCommandList();

                //structureFDrawVisibleMeshCommandsAnyThreadTaskInstance and joinTaskGraphAmong them
TaskContext.MeshDrawCommandsThis is what was explained in the previous section.FMeshPassProcessorGenerated.
                FGraphEventRef AnyThreadCompletionEvent =
TGraphTask<FDrawVisibleMeshCommandsAnyThreadTask>::CreateTask(&Prereqs,
RenderThread).ConstructAndDispatchWhenReady(*CmdList, TaskContext.MeshDrawCommands,
TaskContext.MinimalPipelineStatePassSet, PrimitiveIdsBuffer, BasePrimitiveIdsOffset,
TaskContext.bDynamicInstancing, TaskContext.Instance Factor, TaskIndex, NumTasks);
                //Add the eventParallelCommandListSet, in order to track the depthPassWhether the parallel
                drawing is completed. ParallelCommandListSet->AddParallelCommandList(CmdList,
AnyThreadCompletionEvent, NumDraws);
            }
        }

        (......)
}


// Engine\Source\Runtime\Renderer\Private\MeshDrawCommands.cpp

void    FDrawVisibleMeshCommandsAnyThreadTask::DoTask(ENamedThreads::Type CurrentThread,
const    FGraphEventRef& MyCompletionGraphEvent)
{
        //    Calculate the drawing range
        constint32 DrawNum = VisibleMeshDrawCommands.Num();
        constint32 NumDrawsPerTask = TaskIndex < DrawNum ? FMath::DivideAndRoundUp(DrawNum, TaskNum) :0;

        constint32 StartIndex = TaskIndex * NumDrawsPerTask;
        constint32 NumDraws = FMath::Min(NumDrawsPerTask, DrawNum - StartIndex);

        //Pass the data required for drawing to the drawing interface
        SubmitMeshDrawCommandsRange(VisibleMeshDrawCommands, GraphicsMinimalPipelineStateSet,
PrimitiveIdsBuffer, BasePrimitiveIdsOffset, bDynamicInstancing, StartIndex, NumDraws, InstanceFactor, RHICmdList);


        RHICmdList.EndRenderPass();
        RHICmdList.HandleRTThreadTaskCompletion(MyCompletionGraphEvent);
}

//Submits grid drawing instructions for the specified range.
void SubmitMeshDrawCommandsRange(
        const    FMeshCommandOneFrameArray& VisibleMeshDrawCommands,
        const FGraphicsMinimalPipelineStateSet&                GraphicsMinimalPipelineStateSet,
        FRHIVertexBuffer*        PrimitiveIdsBuffer,
        int32    BasePrimitiveIdsOffset,
        bool    bDynamicInstancing,
        int32    StartIndex,
        int32    NumMeshDrawCommands,
        uint32    InstanceFactor,
        FRHICommandList& RHICmdList)
{
        FMeshDrawCommandStateCache    StateCache;

        //Iterate over the given range of drawing instructions, submitting them one by one.
        for(int32 DrawCommandIndex = StartIndex; DrawCommandIndex < StartIndex +
NumMeshDrawCommands; DrawCommandIndex++)
        {
```

```
            constFVisibleMeshDrawCommand& VisibleMeshDrawCommand =
            VisibleMeshDrawCommands[DrawCommandIndex];
            constint32 PrimitiveIdBufferOffset = BasePrimitiveIdsOffset + (bDynamicInstancing ?
VisibleMeshDrawCommand.PrimitiveIdBufferOffset : DrawCommandIndex) *sizeof(int32);
            //Submit a singleMeshDrawCommand.
            FMeshDrawCommand::SubmitDraw(*VisibleMeshDrawCommand.MeshDrawCommand,
GraphicsMinimalPipelineStateSet, PrimitiveIdsBuffer, PrimitiveIdBufferOffset, InstanceFactor,
RHICmdList, StateCache);
        }
}

//Submit a singleMeshDrawCommandarrive
RHICommandList. void FMeshDrawCommand::SubmitDraw(
        constFMeshDrawCommand& RESTRICT MeshDrawCommand,
        constFGraphicsMinimalPipelineStateSet& FRHIVertexBGufrfaeprh*icsMinimalPipelineStateSet,
                                    ScenePrimitiveIdsBuffer,
        int32 PrimitiveIdOffset,
        uint32     InstanceFactor,
        FRHICommandList& RHICmdList,
        FMeshDrawCommandStateCache&     RESTRICT StateCache)
{
        (……)

        constFGraphicsMinimalPipelineStateInitializer& MeshPipelineState =
        MeshDrawCommand.CachedPipelineId.GetPipelineState(GraphicsMinimalPipelineStateSet);

        //Settings and CachePSO.
        if(MeshDrawCommand.CachedPipelineId.GetId() != StateCache.PipelineId) {

            FGraphicsPipelineStateInitializer GraphicsPSOInit =
            MeshPipelineState.AsGraphicsPipelineStateInitializer();
            RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);
            SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);
            StateCache.SetPipelineState(MeshDrawCommand.CachedPipelineId.GetId());
        }

        //Set and cache template values.
        if(MeshDrawCommand.StencilRef != StateCache.StencilRef) {

            RHICmdList.SetStencilRef(MeshDrawCommand.StencilRef);
            StateCache.StencilRef = MeshDrawCommand.StencilRef;
        }

        //Set vertex data.
        for(int32 VertexBindingIndex =0; VertexBindingIndex <
MeshDrawCommand.VertexStreams.Num(); VertexBindingIndex++)
        {
            constFVertexInputStream& Stream =
            MeshDrawCommand.VertexStreams[VertexBindingIndex];

            if(MeshDrawCommand.PrimitiveIdStreamIndex !=-1&& Stream.StreamIndex ==
            MeshDrawCommand.PrimitiveIdStreamIndex)
            {
                RHICmdList.SetStreamSource(Stream.StreamIndex,              ScenePrimitiveIdsBuffer,
PrimitiveIdOffset);
                StateCache.VertexStreams[Stream.StreamIndex] =              Stream;
            }
            else if(StateCache.VertexStreams[Stream.StreamIndex] != Stream)
```

```
            {
                RHICmdList.SetStreamSource(Stream.StreamIndex,                Stream.VertexBuffer,
Stream.Offset);
                StateCache.VertexStreams[Stream.StreamIndex] =                Stream;
            }
        }

        //set upshaderBound resources. MeshDrawCommand.ShaderBindings.SetOnCommandList(RHICmdList,
        MeshPipelineState.BoundShaderState.AsBoundShaderState(), StateCache.ShaderBindings);


        //  Call different types of drawing instructions according to different data
        if  RHICommandList. (MeshDrawCommand.IndexBuffer)
        {
            if(MeshDrawCommand.NumPrimitives >0) {

                RHICmdList.DrawIndexedPrimitive(
                    MeshDrawCommand.IndexBuffer,
                    MeshDrawCommand.VertexParams.BaseVertexIndex, 0,

                    MeshDrawCommand.VertexParams.NumVertices,
                    MeshDrawCommand.FirstIndex,
                    MeshDrawCommand.NumPrimitives,
                    MeshDrawCommand.NumInstances * InstanceFactor
                ) ;
            }
            else
            {
                RHICmdList.DrawIndexedPrimitiveIndirect(
                    MeshDrawCommand.IndexBuffer,
                    MeshDrawCommand.IndirectArgs.Buffer,
                    MeshDrawCommand.IndirectArgs.Offset );

            }
        }
        else
        {
            if(MeshDrawCommand.NumPrimitives >0) {

                RHICmdList.DrawPrimitive(
                    MeshDrawCommand.VertexParams.BaseVertexIndex          +  MeshDrawCommand.FirstIndex,
                    MeshDrawCommand.NumPrimitives,
                        MeshDrawCommand.NumInstances *InstanceFactor);
            }
            else
            {
                RHICmdList.DrawPrimitiveIndirect(
                    MeshDrawCommand.IndirectArgs.Buffer,
                    MeshDrawCommand.IndirectArgs.Offset);
            }
        }
    }
```

The above code has already elaborated on the drawing process of PrePass (depth channel). The following explanations need to be added about the transition from FMeshDrawCommand to RHICommandList:

- Each Pass will execute a similar process as above, and the same frame will be executed multiple times, but not all Passes will be enabled. They can be dynamically enabled and disabled through the view's PassMask.
- DispatchDraw and SubmitMeshDrawCommandsRange intentionally use flat arrays, taking the following into account:

    - Only through the visibility set can the array of FVisibleMeshDrawCommand be easily and quickly divided, so as to submit FMeshDrawCommand drawing instructions to the multithreaded system TaskGraph in a flat manner.
    - By sorting the FMeshDrawCommand list and adding a StateCache to reduce the number of commands submitted to the RHICommandList, the load of RHICommandList conversion and execution is reduced. After adding this step, Fortnite can reduce the RHI execution time by 20%.
    - Cache consistency traversal. Tightly pack FMeshDrawCommand, lightweight, flattened, and continuously store the data required for SubmitDraw in memory, which can improve cache and pre-fetch hit rates.

        - TChunkedArray<FMeshDrawCommand> MeshDrawCommands;
        typedef TArray<FVisibleMeshDrawCommand, SceneRenderingAllocator>
        - FMeshCommandOneFrameArray;
        TArray<FMeshDrawShaderBindingsLayout, TInlineAllocator<2>>ShaderLayouts;
        - typedef TArray<FVertexInputStream, TInlineAllocator<4>>FVertexInputStreamArray;
        - const int32 NumInlineShaderBindings = 10;
        -

- When converting MeshDrawCommandPasses to RHICommandList commands, parallel mode is supported. The parallel allocation strategy is to simply divide the array equally into the number of worker threads, and then each worker thread executes the drawing instructions in the specified range. The advantage of this is that it is simple, fast and easy to understand, and improves the CPU cache hit rate. The disadvantage is that the execution time of tasks in each group may vary greatly, so the overall execution time is determined by the longest group, which is bound to prolong the time and reduce parallel efficiency. To address this problem, the author has come up with some strategies:

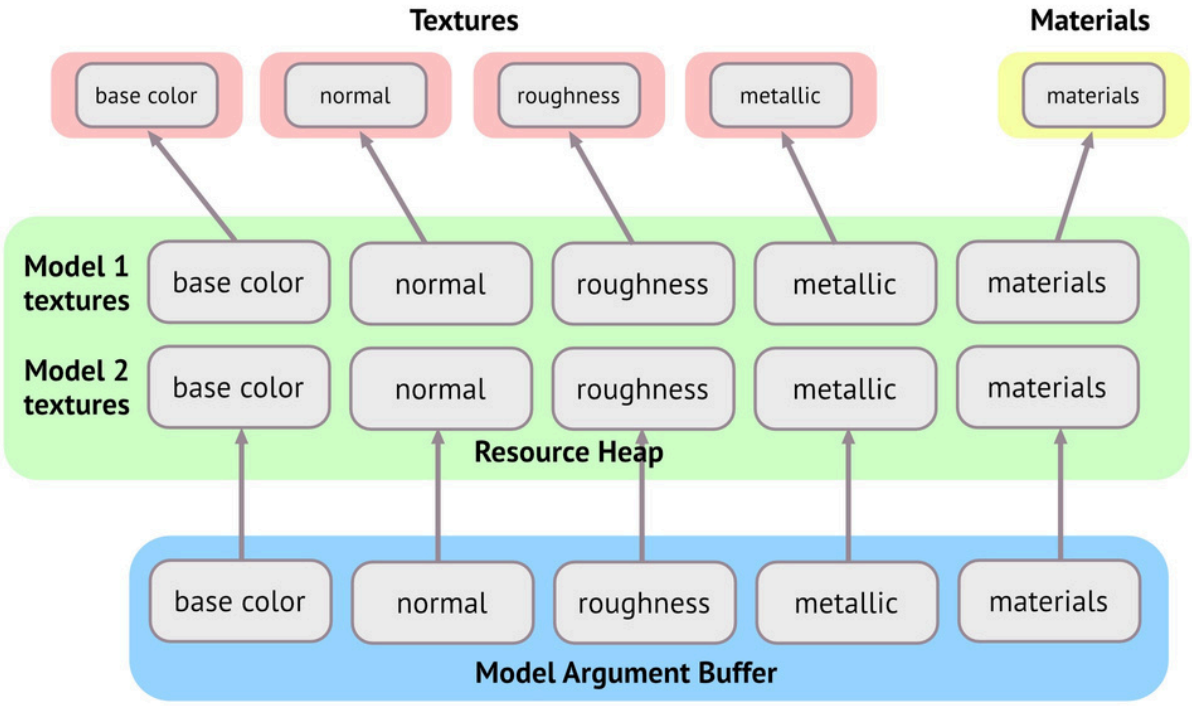
    - Heuristic strategy: Record the execution time of each MeshDrawCommand in the previous frame, and add up the adjacent MeshDrawCommands according to their execution time in the next frame. When their sum approaches the average value of each group, it is regarded as a group of execution bodies.
    - Check one or several properties of MeshDrawCommand. For example, group the meshes based on the number of faces or materials, and make the sum of the properties of each group of MeshDrawCommand roughly the same.

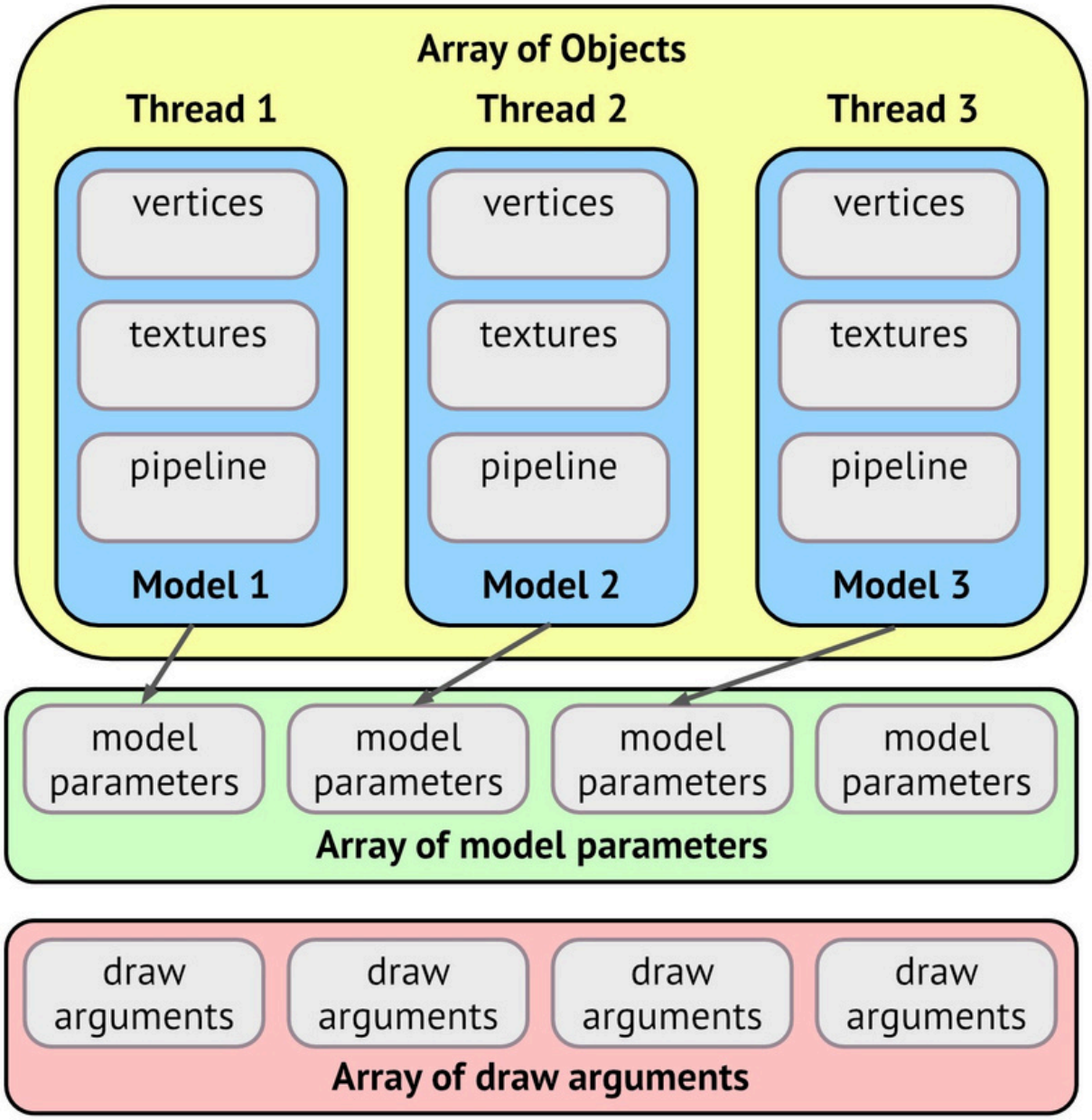
    Of course, the above strategy will increase the logic complexity and may also reduce the CPU cache hit rate. The actual effect depends on the operating environment.

- FMeshDrawCommand::SubmitDrawThe process caches PSO and template values to prevent duplicate data and instructions from being submitted to RHICommandList, reducing the IO interaction between the CPU and GPU.

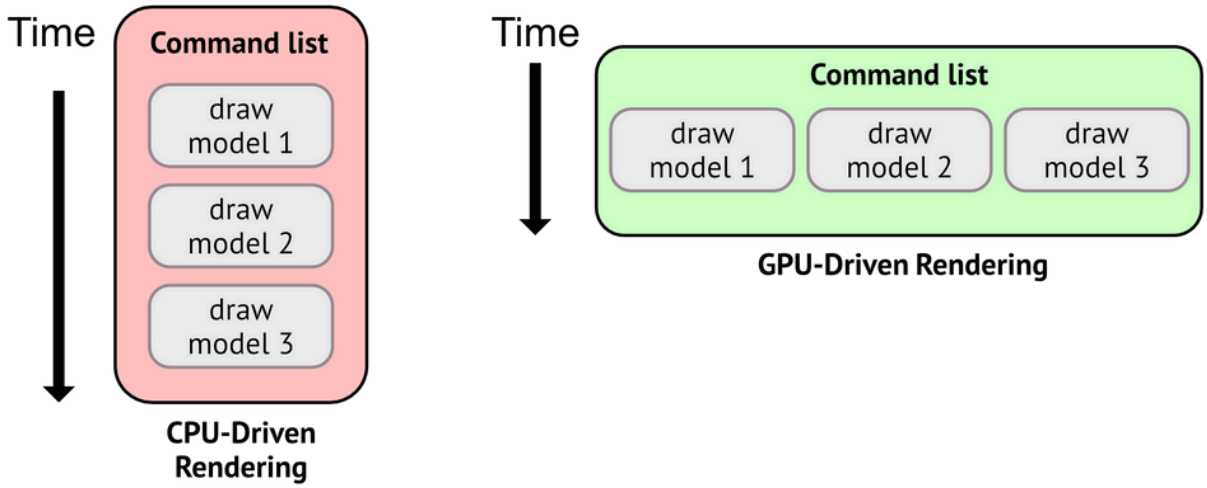
  > The switching of IO and rendering states between the CPU and GPU has always been a problem that has plagued the field of real-time rendering, especially in systems with heterogeneous CPUs and GPUs. Therefore, reducing the data interaction between the CPU and GPU is a major measure to optimize rendering performance. After adopting states such as caching PSO, in extreme cases, it can bring several times the performance improvement.

- FMeshDrawCommand::SubmitDrawFour drawing models are supported, one dimension is whether there is a vertex index, and the other dimension is whether it is indirect drawing.

  > **Introduction to Indirect Draw**
  > Before Indirect Draw, if an application wanted to use the same Draw Call to draw multiple objects, it could only use GPU Instance, but GPU Instance has many limitations, such as requiring the same vertices, indices, rendering states, and material data, and only allowing different Transforms. Even though textures can be packed into Atlas, and material attributes and model meshes can be packed into StructuredBuffer, there is no way to avoid the fatal limitation that the number of vertices must be the same each time you draw. To implement GPU Driven Rendering Pipeline, it must be broken into Clusters with the same number of vertices.
  > With the emergence of Indirect Draw technology, GPU-driven rendering pipelines will become simpler and more efficient. Its core idea is to allow resource references required by the same mesh to be placed in an Argument Buffer:
  >
  > 
  >
  >
  > Argument Buffers from different grids can form longer Buffers:



Since the data of each grid can be stored in different GPU threads, the drawing of multiple grids can be performed in parallel, which is bound to have a significant efficiency improvement compared to traditional serial drawing:



However, Indirect Draw is only supported in modern graphics APIs such as DirectX11, DirecXt12, Vulkan, and Metal.

3.2.5 From RHICommandList to GPU

RHIstands for**Rendering Hardware Interface**,which is an abstract layer of different graphics APIs, and RHICommandList is responsible for collecting intermediate layer drawing instructions and data that are not related to the graphics API.

After RHICommandList collects a series of intermediate drawing commands, it will be converted one by one to the interface of the corresponding target graphics API in the RHI thread. The following takeFRHICommandList::DrawIndexedPrimitivethe interface as an example:

```
// Engine\Source\Runtime\RHI\Public\RHICommandList.h

void FRHICommandList::DrawIndexedPrimitive(FRHIIndexBuffer* IndexBuffer, int32 BaseVertexIndex, uint32 FirstInstance, uint32 NumVertices, uint32 StartIndex, uint32 NumPrimitives, uint32 NumInstances)

{
	if(!IndexBuffer)
	{
		UE_LOG(LogRHI, Fatal, TEXT("Tried to call DrawIndexedPrimitive with null IndexBuffer!"));

	}

	//  BypassRHIThreads are executed directly.
	if  (Bypass())
	{
		GetContext().RHIDrawIndexedPrimitive(IndexBuffer, NumVertices, BaseVertexIndex, FirstInstance, StartIndex, NumPrimitives, NumInstances);
		return;
	}

	//Create drawing instructions.
	ALLOC_COMMAND(FRHICommandDrawIndexedPrimitive)(IndexBuffer, BaseVertexIndex, FirstInstance, NumVertices, StartIndex, NumPrimitives, NumInstances);
}

// FRHICommandDrawIndexedPrimitiveThe declaration body
FRHICOMMAND_MACRO(FRHICommandDrawIndexedPrimitive) {

	//The data required by the command.
	FRHIIndexBuffer*	IndexBuffer;
	int32	BaseVertexIndex;
	uint32	FirstInstance;
	  uint32 NumVertices;
	uint32	StartIndex;
	uint32 NumPrimitives;
	uint32 NumInstances;

	FRHICommandDrawIndexedPrimitive(FRHIIndexBuffer* InIndexBuffer, int32 InBaseVertexIndex, uint32 InFirstInstance, uint32 InNumVertices, uint32 InStartIndex, uint32 InNumPrimitives, uint32 InNumInstances)

		:	IndexBuffer(InIndexBuffer)	,
		BaseVertexIndex(InBaseVertexIndex)	,
		FirstInstance(InFirstInstance)	,
		NumVertices(InNumVertices)	,
		StartIndex(InStartIndex)
```

```
        , NumPrimitives(InNumPrimitives)
        , NumInstances(InNumInstances)
    {
    }

    //The interface on which this command is executed.
    RHI_APIvoid Execute(FRHICommandListBase& CmdList);
};


// Engine\Source\Runtime\RHI\Public\RHICommandListCommandExecutes.inl

// FRHICommandDrawIndexedPrimitiveImplementation of the execution interface.
void FRHICommandDrawIndexedPrimitive::Execute(FRHICommandListBase& {                CmdList)

    RHISTAT(DrawIndexedPrimitive); INTERNAL_DECORATOR(RHIDrawIndexedPrimitive)(IndexBuffer,
    BaseVertexIndex, FirstInstance, NumVertices, StartIndex, NumPrimitives, NumInstances); }



// INTERNAL_DECORATORThe macro actually callsRHICommandListInsideIRHICommandContextThe corresponding interface.
#if   !defined(INTERNAL_DECORATOR)
    #defineINTERNAL_DECORATOR(Method) CmdList.GetContext().Method
#endif


// Engine\Source\Runtime\RHI\Public\RHICommandList.h

//distributeRHICommand macro definition
#defineALLOC_COMMAND(...) new ( AllocCommand(sizeof(__VA_ARGS__), alignof(__VA_ARGS__)) ) __VA_ARGS__


//distributeRHICommand interface.
void*FRHICommandListBase::AllocCommand(int32 AllocSize, int32 Alignment) {

    checkSlow(!IsExecuting()); //Allocates memory
    from the command memory manager.
    FRHICommandBase* Result = (FRHICommandBase*) MemManager.Alloc(AllocSize, Alignment);
    + +NumCommands;
    //Add the newly assigned command to the end of the linked list.

    * CommandLink   =  Result;
    CommandLink   =  &Result->Next;
    returnResult;
}
```

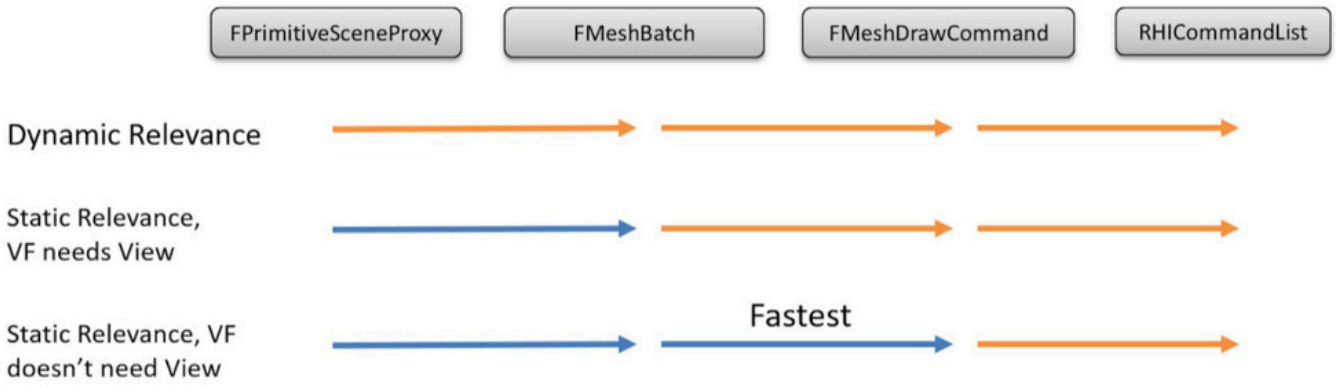
From the above, we can know that through the pre-defined macrosFRHICOMMAND_MACRO,the RHICommandList intermediate layer drawing instructions are converted to the corresponding graphics API so that the drawing instructions can be submitted to the GPU later.INTERNAL_DECORATOR

ALLOC_COMMAND IRHICommandContext

3.3 Static and dynamic drawing paths

3.3.1 Overview of drawing paths

In fact, there are already some shadows of static paths and dynamic paths in Chapter 3.2, but the dynamic path is used more for explanation. In fact, in order to optimize the drawing of static meshes, UE separates the static drawing path so as to perform customized performance optimization on it. Static paths are divided into two types, one requires View information, and the other does not require View information, which can perform more cache optimization:



UE has three kinds of mesh drawing paths (orange is dynamically generated every frame, blue is generated only once and then cached): the first is a dynamic drawing path, which is dynamically created every frame from FPrimitiveSceneProxy to RHICommandList, with the lowest efficiency but the strongest controllability; the second is a static path that requires a View, which can cache FMeshBatch data, with medium efficiency and controllability; the third is a static drawing path that does not require a view, which can cache FMeshBatch and FMeshDrawCommand, with the highest efficiency, but poor controllability and many conditions to be met.

The cache data of the static drawing path only needs to be generated once, so it can reduce the execution time of the rendering thread and improve the running efficiency. For example, static meshes are injected into FStaticMeshBatch by implementing the DrawStaticElements interface, and DrawStaticElements is usually called when SceneProxy is added to the scene.

3.3.2 Dynamic drawing path

The dynamic drawing path rebuilds FMeshBatch data every frame without caching, so it has the strongest scalability but the lowest efficiency. It is often used for particle effects, skeletal animation, program dynamic meshes, and meshes that need to update data every frame. Collect FMeshBatch through the GetDynamicMeshElements interface. For details, see [3.2 Model Drawing Pipeline](#3.2 Model Drawing Pipeline).

FParallelMeshDrawCommandPass is a general mesh pass. It is recommended to be used only in performance-critical mesh passes because it only supports parallel and cache rendering. If you want to use parallel or cache paths, you must go through a strict design because the mesh drawing command and any data bound to the shader cannot be modified after InitViews. The code of FParallelMeshDrawCommandPass has appeared in Chapter 3.2, but to further illustrate its use, here is a relatively concise shadow rendering example:

```
// Engine\Source\Runtime\Renderer\Private\ShadowRendering.h

classFProjectedShadowInfo: public FRefCountedObject {

	(……)

	//statementFParallelMeshDrawCommandPassExamples
	FParallelMeshDrawCommandPass ShadowDepthPass;

	(……)
};


// Engine\Source\Runtime\Renderer\Private\ShadowDepthRendering.cpp

void FProjectedShadowInfo::RenderDepthInner(FRHICommandListImmediate& RHICmdList, FSceneRenderer*
SceneRenderer, FBeginShadowRenderPassFunction BeginShadowRenderPass,bool bDoParallelDispatch)

{
	(……)

	//  Parallel Mode
	if  (bDoParallelDispatch)
	{
		boolbFlush = CVarRHICmdFlushRenderThreadTasksShadowPass.GetValueOnRenderThread()
>  0
			|| CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0;
		FScopedCommandListWaitForTasksFlusher(bFlush);

		{
			//Build a parallel processing set to store the generatedRHICommandListList.
			FShadowParallelCommandListSetParallelCommandListSet(*ShadowDepthView,
SceneRenderer, RHICmdList, CVarRHICmdShadowDeferredContexts.GetValueOnRenderThread() >0, !bFlush,
DrawRenderState, *this, BeginShadowRenderPass);

			//Send drawing instructions
			ShadowDepthPass.DispatchDraw(&ParallelCommandListSet,		RHICmdList);
		}
	}
	//  Non-parallel mode
	else
	{
		ShadowDepthPass.DispatchDraw(nullptr,		RHICmdList);
	}
}
```

It is very simple and convenient to use, isn't it? This is because UE has done a lot of encapsulation and detail processing for us behind the scenes.

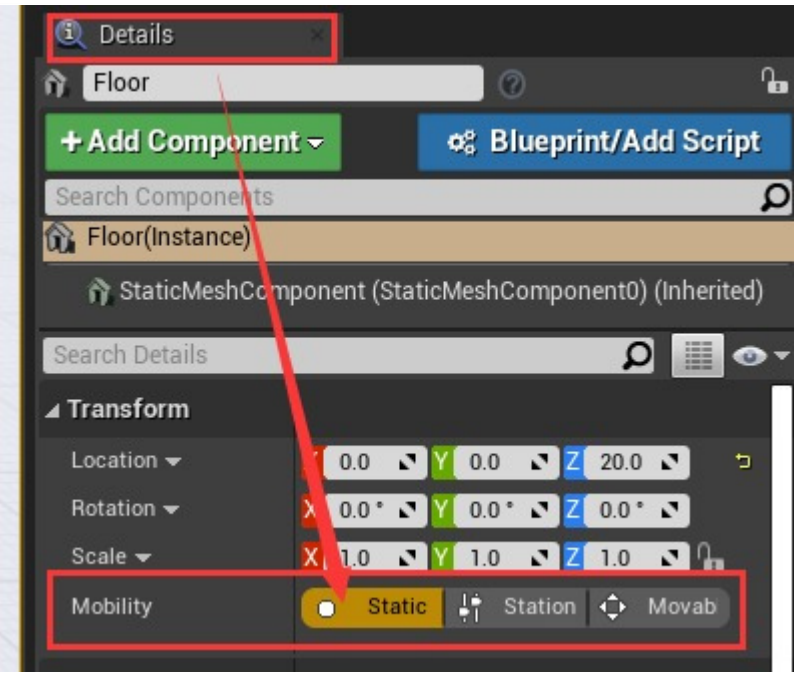
In addition to FParallelMeshDrawCommandPass, there is a simpler way to call drawing commands: DrawDynamicMeshPass. DrawDynamicMeshPass only needs to pass in view/RHICommandList and a lambda anonymous function. Its declaration and usage examples are as follows:

```
// Engine\Source\Runtime\Renderer\Public\MeshPassProcessor.inl
```

```
// DrawDynamicMeshPassStatement
template<typename LambdaType>
void DrawDynamicMeshPass(constFSceneView& View, FRHICommandList& RHICmdList,const LambdaType&
BuildPassProcessorLambda,boolbForceStereoInstancingOff =false);


// Engine\Source\Runtime\Renderer\Private\DepthRendering.cpp

void FDeferredShadingSceneRenderer::RenderPrePassEditorPrimitives(FRHICommandList& RHICmdList,const
FViewInfo& View,constFMeshPassProcessorRenderState& DrawRenderState, EDepthDrawingMode
DepthDrawingMode,boolbRespectUseAsOccluderFlag) {

    (......)

    boolbDirty =false;
    if(!View.Family->EngineShowFlags.CompositeEditorPrimitives) {

        const boolbNeedToSwitchVerticalAxis =
        RHINeedsToSwitchVerticalAxis(ShaderPlatform);
        constFScene* LocalScene = Scene;

        //CallDrawDynamicMeshPassProcessing DepthPass
        . DrawDynamicMeshPass(View, RHICmdList,
            [&View, &DrawRenderState, LocalScene, DepthDrawingMode,
bRespectUseAsOccluderFlag](FDynamicPassMeshDrawListContext* DynamicMeshPassContext)
            {
                FDepthPassMeshProcessor PassMeshProcessor(
                    LocalScene,
                    &View,
                    DrawRenderState,
                    bRespectUseAsOccluderFlag,
                    DepthDrawingMode,
                    false,
                    DynamicMeshPassContext);


                constuint64 DefaultBatchElementMask = ~0ull;

                for(int32 MeshIndex =0; MeshIndex < View.ViewMeshElements.Num();
MeshIndex++)
                {
                    constFMeshBatch& MeshBatch = View.ViewMeshElements[MeshIndex];
                    PassMeshProcessor.AddMeshBatch(MeshBatch, DefaultBatchElementMask,
nullptr);
                }
            });

        (......)
    }
}
```

3.3.3 Static drawing path

Static drawing paths can usually be cached, so they are also called cached drawing paths. Applicable objects can be static models (which can be specified in the mesh properties panel of the UE editor, see the figure below).



The static model is cached when its corresponding FPrimitiveSceneInfo calls AddToScene. The following is the specific processing code and analysis:

```
// Engine\Source\Runtime\Renderer\Private\PrimitiveSceneInfo.cpp

void FPrimitiveSceneInfo::AddToScene(FRHICommandListImmediate& RHICmdList, FScene* Scene, const
TArrayView<FPrimitiveSceneInfo*>& SceneInfos,boolbUpdateStaticDrawLists,bool bAddToStaticDrawLists,bool
bAsyncCreateLPIs) {

	(......)

	{
		SCOPED_NAMED_EVENT(FPrimitiveSceneInfo_AddToScene_AddStaticMeshes,
		FColor::Magenta);
		//	Working with static models
		if	(bUpdateStaticDrawLists)
		{
			AddStaticMeshes(RHICmdList, Scene, SceneInfos, bAddToStaticDrawLists);
		}
	}

	(......)
}

void FPrimitiveSceneInfo::AddStaticMeshes(FRHICommandListImmediate& RHICmdList, FScene* Scene,const
TArrayView<FPrimitiveSceneInfo*>& SceneInfos,boolbAddToStaticDrawLists) {

	LLM_SCOPE(ELLMTag::StaticMesh);

	{
		//Process static primitives in parallel.
		ParallelForTemplate(SceneInfos.Num(), [Scene, &SceneInfos](int32 Index) {

			SCOPED_NAMED_EVENT(FPrimitiveSceneInfo_AddStaticMeshes_DrawStaticElements,
FColor::Magenta);
			FPrimitiveSceneInfo* SceneInfo = SceneInfos[Index]; //Caches the static
			elements of a primitive.
```

```
                FBatchingSPDI BatchingSPDI(SceneInfo); BatchingSPDI.SetHitProxy(SceneInfo-
                >DefaultDynamicHitProxy);
                //CallProxyofDrawStaticElementsinterface, the collectedFStaticMeshBatchAdd toSceneInfo-
> StaticMeshesmiddle.
                SceneInfo->Proxy->DrawStaticElements(&BatchingSPDI); SceneInfo-
                >StaticMeshes.Shrink();
                SceneInfo->StaticMeshRelevances.Shrink();

                check(SceneInfo->StaticMeshRelevances.Num() == SceneInfo->StaticMeshes.Num());
            });
        }

        {
            //AllPrimitiveSceneInfoofstaticMeshBatchAdd to sceneStaticMesheList.
            SCOPED_NAMED_EVENT(FPrimitiveSceneInfo_AddStaticMeshes_UpdateSceneArrays, FColor::Blue);

            for(FPrimitiveSceneInfo* SceneInfo : SceneInfos) {

                for(int32 MeshIndex =0; MeshIndex < SceneInfo->StaticMeshes.Num();
MeshIndex++)
                {
                    FStaticMeshBatchRelevance&MeshRelevance = SceneInfo-
> StaticMeshRelevances[MeshIndex];
                    FStaticMeshBatch& Mesh = SceneInfo->StaticMeshes[MeshIndex];

                    // Add the static mesh to the scene's static mesh list. //Adds a StaticMesh
                    element to the scene's list of StaticMeshes.
                    FSparseArrayAllocationInfo SceneArrayAllocation = Scene-
> StaticMeshes.AddUninitialized();
                    Scene->StaticMeshes[SceneArrayAllocation.Index] = &Mesh; Mesh.Id =
                    SceneArrayAllocation.Index;
                    MeshRelevance.Id = SceneArrayAllocation.Index;

                    //  Handle element-by-element visibility (if necessary).
                    if  (Mesh.bRequiresPerElementVisibility)
                    {
                        // Use a separate index into StaticMeshBatchVisibility, since most
meshes don't use it
                        Mesh.BatchVisibilityId = Scene-
> StaticMeshBatchVisibility.AddUninitialized().Index;
                        Scene->StaticMeshBatchVisibility[Mesh.BatchVisibilityId] =true;
                    }
                }
            }
        }

        //  Cache staticMeshDrawCommand
        if  (bAddToStaticDrawLists)
        {
            CacheMeshDrawCommands(RHICmdList,       Scene,     SceneInfos);
        }
}

void FPrimitiveSceneInfo::CacheMeshDrawCommands(FRHICommandListImmediate& RHICmdList, FScene* Scene,
constTArrayView<FPrimitiveSceneInfo*>& SceneInfos) {

    //@todo - only need material uniform buffers to be created since we are going to cache pointers to them
```

```
// Any updates (after initial creation) don't need to be forced here
FMaterialRenderProxy::UpdateDeferredCachedUniformExpressions();

SCOPED_NAMED_EVENT(FPrimitiveSceneInfo_CacheMeshDrawCommands, FColor::Emerald);

QUICK_SCOPE_CYCLE_COUNTER(STAT_CacheMeshDrawCommands);
FMemMarkMark(FMemStack::Get());

//Counts the number of threads in parallel.
staticconstexprintBATCH_SIZE =64;
const intNumBatches = (SceneInfos.Num() + BATCH_SIZE -1) / BATCH_SIZE;

//Thread callback.
autoDoWorkLambda = [Scene, SceneInfos](int32 Index) {

    SCOPED_NAMED_EVENT(FPrimitiveSceneInfo_CacheMeshDrawCommand,        FColor::Green);

    structFMeshInfoAndIndex {

        int32    InfoIndex;
        int32    MeshIndex;
    };

    TArray<FMeshInfoAndIndex, TMemStackAllocator<>> MeshBatches;
    MeshBatches.Reserve(3* BATCH_SIZE);

    //Traverse the scope of the current thread and process them one by onePrimitiveSceneInfo
    intLocalNum = FMath::Min((Index * BATCH_SIZE) + BATCH_SIZE, SceneInfos.Num()); for(intLocalIndex =
    (Index * BATCH_SIZE); LocalIndex < LocalNum; LocalIndex++) {

        FPrimitiveSceneInfo* SceneInfo = SceneInfos[LocalIndex]; check(SceneInfo-
        >StaticMeshCommandInfos.Num() ==0); SceneInfo-
        >StaticMeshCommandInfos.AddDefaulted(EMeshPass::Num * SceneInfo-
> StaticMeshes.Num());
        FPrimitiveSceneProxy* SceneProxy = SceneInfo->Proxy;

        //  Volumetric transparent shadows need to be updated every frame and cannot be cached.
        if  (!SceneProxy->CastsVolumetricTranslucentShadow())
        {
            //WillPrimitiveSceneInfoAdd all static meshes toMeshBatchList.
            for(int32 MeshIndex =0; MeshIndex < SceneInfo->StaticMeshes.Num();
MeshIndex++)
            {
                FStaticMeshBatch& Mesh = SceneInfo->StaticMeshes[MeshIndex]; //
                    Check whether caching is supportedMeshDrawCommand
                if  (SupportsCachingMeshDrawCommands(Mesh))
                {
                    MeshBatches.Add(FMeshInfoAndIndex{ LocalIndex, MeshIndex });
                }
            }
        }
    }

    //Traverse all predefinedPass, each static element generatesMeshDrawCommandAdd to correspondencePass
    in the cache list. for(int32 PassIndex =0; PassIndex < EMeshPass::Num; PassIndex++) {

        constEShadingPath ShadingPath = Scene->GetShadingPath();
        EMeshPass::Type PassType = (EMeshPass::Type)PassIndex;
```

```
            if((FPassProcessorManager::GetPassFlags(ShadingPath, PassType) &
EMeshPassFlags::CachedMeshCommands) != EMeshPassFlags::None)
            {
                //Declare cache drawing command instance
                FCachedMeshDrawCommandInfoCommandInfo(PassType);

                //Get the corresponding from the scenePassVarious containers to build
                FCachedPassMeshDrawListContext. FCriticalSection& CachedMeshDrawCommandLock = Scene-
>CachedMeshDrawCommandLock[PassType];
                FCachedPassMeshDrawList& SceneDrawList = Scene->CachedDrawLists[PassType];
                FStateBucketMap& CachedMeshDrawCommandStateBuckets = Scene-
> CachedMeshDrawCommandStateBuckets[PassType];
                FCachedPassMeshDrawListContextCachedPassMeshDrawListContext(CommandInfo,
CachedMeshDrawCommandLock, SceneDrawList, CachedMeshDrawCommandStateBuckets, *Scene);

                //createPassofFMeshPassProcessor
                PassProcessorCreateFunction CreateFunction =
FPassProcessorManager::GetCreateFunction(ShadingPath, PassType);
                FMeshPassProcessor* PassMeshProcessor = CreateFunction(Scene, nullptr,
&CachedPassMeshDrawListContext);

                if(PassMeshProcessor != nullptr) {

                    for(constFMeshInfoAndIndex& MeshAndInfo : MeshBatches) {

                        FPrimitiveSceneInfo* SceneInfo =
SceneInfos[MeshAndInfo.InfoIndex];
                        FStaticMeshBatch& Mesh = SceneInfo-
> StaticMeshes[MeshAndInfo.MeshIndex];

                        CommandInfo = FCachedMeshDrawCommandInfo(PassType);
                        FStaticMeshBatchRelevance& MeshRelevance = SceneInfo-
> StaticMeshRelevances[MeshAndInfo.MeshIndex];

                        check(!MeshRelevance.CommandInfosMask.Get(PassType));

                        check(!Mesh.bRequiresPerElementVisibility); uint64
                        BatchElementMask = ~0ull;
                        //Add toMeshBatcharrivePassMeshProcessor, the internalFMeshBatchConvert to
FMeshDrawCommand.
                        PassMeshProcessor->AddMeshBatch(Mesh, BatchElementMask, SceneInfo-
> Proxy);

                        if(CommandInfo.CommandIndex !=-1 || CommandInfo.StateBucketId !=
- 1)
                        {
                            static_assert(sizeof(MeshRelevance.CommandInfosMask) *8>=
EMeshPass::Num,"CommandInfosMask is too small to contain all mesh passes.");
                            MeshRelevance.CommandInfosMask.Set(PassType);
                            MeshRelevance.CommandInfosBase++;

                            intCommandInfoIndex = MeshAndInfo.MeshIndex * EMeshPass::Num
+ PassType;
                            check(SceneInfo-
> StaticMeshCommandInfos[CommandInfoIndex].MeshPass == EMeshPass::Num);
                            //WillCommandInfoCache toPrimitiveSceneInfomiddle. SceneInfo-
                            >StaticMeshCommandInfos[CommandInfoIndex] =
```

```
CommandInfo;

                                        (......)
                                    }
                                }
                                // destroyFMeshPassProcessor
                                PassMeshProcessor->~FMeshPassProcessor();
                            }
                        }
                    }

                    (......)
            };

            // Parallel Mode
            if (FApp::ShouldUseThreadingForPerformance())
            {
                ParallelForTemplate(NumBatches, DoWorkLambda,
                EParallelForFlags::PumpRenderingThread);
            }
            //Single-threaded mode
            else
            {
                for(intIdx =0; Idx < NumBatches; Idx++) {

                    DoWorkLambda(Idx);
                }
            }

            FGraphicsMinimalPipelineStateId::InitializePersistentIds();

            (.....)
}
```

From the above code, we can see that when a static mesh is added to a scene, it will cache FMeshBatch and may cache the corresponding FMeshDrawCommand. The key interface for determining whether to support caching FMeshDrawCommand is SupportsCachingMeshDrawCommands, which is implemented as follows:

```
// Engine\Source\Runtime\Engine\Private\PrimitiveSceneProxy.cpp

bool SupportsCachingMeshDrawCommands(constFMeshBatch& MeshBatch) {

    return
        // FMeshBatchThere is only one element.
        (MeshBatch.Elements.Num() ==1) &&

        //Vertex Factory Support CacheFMeshDrawCommand MeshBatch.VertexFactory->GetType()-
        >SupportsCachingMeshDrawCommands();
}


// Engine\Source\Runtime\RenderCore\Public\VertexFactory.h

bool FVertexFactoryType::SupportsCachingMeshDrawCommands()const {
```

```
    return bSupportsCachingMeshDrawCommands;
}
```

It can be seen that the condition that determines whether FMeshDrawCommand can be cached is that FMeshBatch has only one element and the vertex factory it uses supports caching.

> Currently only FLocalVertexFactory (UStaticMeshComponent) supports this, other vertex factories need to rely on the view to set the shader binding.

If any of the conditions are not met, FMeshDrawCommand cannot be cached. In more detail, the following conditions need to be met:

- The Pass is an enumeration of EMeshPass::Type.
- The EMeshPassFlags::CachedMeshCommands flag is correctly passed when registering a custom mesh pass processor.
- The mesh pass processor can handle all shader binding data without relying on FSceneView, as FSceneView is null during caching.

It is important to note that if any data referenced by a cached command drawing changes, the command must be invalidated and regenerated.

> Calling FPrimitiveSceneInfo::BeginDeferredUpdateStaticMeshes can invalidate the specified drawing command.
> Setting Scene->bScenesPrimitivesNeedStaticMeshElementUpdate to true can invalidate all caches, which will seriously affect performance. It is recommended not to use it or use it less frequently.

Invalidating the cache will affect rendering performance. An alternative solution is to put the mutable data into the UniformBuffer of the Pass and execute different shader logic through the UniformBuffer to separate the dependency on view-based shader binding.

Unlike the dynamic drawing path, when collecting static mesh elements, the FPrimitiveSceneProxy::DrawStaticElements interface is called. This interface is implemented by a specific subclass. Let's take a look at the implementation process of its subclass FStaticMeshSceneProxy:

```
// Engine\Source\Runtime\Engine\Private\StaticMeshRender.cpp

void FStaticMeshSceneProxy::DrawStaticElements(FStaticPrimitiveDrawInterface* PDI) {

    checkSlow(IsInParallelRenderingThread());

    //  Is it enabled?bUseViewOwnerDepthPriorityGroup (!
    if  HasViewDependentDPG())
    {
        // Determine the DPG the primitive should be drawn in. uint8
        PrimitiveDPG = GetStaticDepthPriorityGroup(); int32 NumLODs =
        RenderData->LODResources.Num();
        //Never use the dynamic path in this path, because only unselected elements will
```

```
use DrawStaticElements
        boolbIsMeshElementSelected =false;
        constautoFeatureLevel = GetScene().GetFeatureLevel();
        const boolIsMobile = IsMobilePlatform(GetScene().GetShaderPlatform()); constint32
        NumRuntimeVirtualTextureTypes = RuntimeVirtualTextureMaterialTypes.Num();


        //check if a LOD is being forced if
        (ForcedLodModel >0) {

            //GetLODLevel (index)
            int32 LODIndex = FMath::Clamp(ForcedLodModel, ClampedMinLOD +1, NumLODs) -1; const
            FStaticMeshLODResources& LODModel = RenderData->LODResources[LODIndex];

            //Plot all submodels.
            for(int32 SectionIndex =0; SectionIndex < LODModel.Sections.Num();
SectionIndex++)
            {
                constint32 NumBatches = GetNumMeshBatches(); PDI-
                >ReserveMemoryForMeshes(NumBatches * (1+
NumRuntimeVirtualTextureTypes));

                //Add all batches of elements toPDIdraw.
                for(int32 BatchIndex =0; BatchIndex < NumBatches; BatchIndex++) {

                    FMeshBatch BaseMeshBatch;

                    if(GetMeshElement(LODIndex, BatchIndex, SectionIndex, PrimitiveDPG,
bIsMeshElementSelected,true, BaseMeshBatch))
                    {
                        (......)
                        {
                            //JoinPDIExecute drawing
                            PDI->DrawMesh(BaseMeshBatch, FLT_MAX);
                        }
                    }
                }
            }
        }

        (......)
    }
}
```

It can be seen that the DrawStaticElements interface will pass in an instance of FStaticPrimitiveDrawInterface to collect all static elements of the PrimitiveSceneProxy. Let's go into the declaration and implementation of FStaticPrimitiveDrawInterface and its subclass FBatchingSPDI to explore its true appearance:

```
// Engine\Source\Runtime\Engine\Public\SceneManagement.h

classFStaticPrimitiveDrawInterface {

public:
    virtualvoid SetHitProxy(HHitProxy* HitProxy) =0;
```

```
        virtualvoid ReserveMemoryForMeshes(int32 MeshNum) =0;

        //PDIDrawing interface
        virtualvoid DrawMesh(constFMeshBatch& Mesh,floatScreenSize) =0;
};


// Engine\Source\Runtime\Renderer\Private\PrimitiveSceneInfo.cpp

classFBatchingSPDI: public FStaticPrimitiveDrawInterface {

public:
        (……)

        //accomplishPDIDrawing interface
        virtualvoid DrawMesh(constFMeshBatch& Mesh,floatScreenSize) final override {

                if(Mesh.HasAnyDrawCalls()) {

                        FPrimitiveSceneProxy* PrimitiveSceneProxy = PrimitiveSceneInfo->Proxy; PrimitiveSceneProxy-
                        >VerifyUsedMaterial(Mesh.MaterialRenderProxy);

                        //Create NewFStaticMeshBatchInstance, and add toPrimitiveSceneInfoofStaticMeshelist.
                        FStaticMeshBatch* StaticMesh = new(PrimitiveSceneInfo->StaticMeshes)
FStaticMeshBatch(
                                PrimitiveSceneInfo,
                                Mesh,
                                CurrentHitProxy ? CurrentHitProxy->Id : FHitProxyId() ) ;



                        constERHIFeatureLevel::Type FeatureLevel = PrimitiveSceneInfo->Scene-
> GetFeatureLevel();
                        StaticMesh->PreparePrimitiveUniformBuffer(PrimitiveSceneProxy, FeatureLevel);

                        // Volumetric self shadow mesh commands need to be generated every frame, as
they depend on single frame uniform buffers with self shadow data.
                        const boolbSupportsCachingMeshDrawCommands =
SupportsCachingMeshDrawCommands(*StaticMesh, FeatureLevel) && !PrimitiveSceneProxy-
> CastsVolumetricTranslucentShadow();

                        //deal withRelevance
                        boolbUseSkyMaterial = Mesh.MaterialRenderProxy->GetMaterial(FeatureLevel)-
> IsSky();
                        boolbUseSingleLayerWaterMaterial = Mesh.MaterialRenderProxy-
> GetMaterial(FeatureLevel)->GetShadingModels().HasShadingModel(MSM_SingleLayerWater);
                        FStaticMeshBatchRelevance* StaticMeshRelevance = new(PrimitiveSceneInfo-
> StaticMeshRelevances) FStaticMeshBatchRelevance(
                                * StaticMesh,
                                ScreenSize,
                                bSupportsCachingMeshDrawCommands,
                                bUseSkyMaterial,
                                bUseSingleLayerWaterMaterial,
                                FeatureLevel

                        ) ;
                }
        }

private:
```

```
    FPrimitiveSceneInfo* PrimitiveSceneInfo;
    TRefCountPtr<HHitProxy> CurrentHitProxy;
};
```

The main function of FBatchingSPDI::DrawMesh is to convert PrimitiveSceneProxy into FStaticMeshBatch and then process the Relevance data of the mesh.

3.4 Rendering Mechanism Summary

3.4.1 Drawing pipeline optimization technology

The previous chapters have explained in detail how UE converts primitives from Components into final drawing instructions step by step. The purpose of doing this is mainly to improve rendering performance. In summary, the optimization technologies involved are mainly the following:

- **Draw Call Merging**

Since allFMeshDrawCommandsare captured in advance instead of being submitted to the GPU immediately, this provides a favorable basic guarantee for Draw Call merging. However, the current version of the merge is based on the characteristics of D3D11, and whether to merge into the same instance call is determined based on shader binding. Aggregate merging based on D3D12 has not

has yet to be implemented.

In addition to merging, sorting can also enable similar instructions to be drawn at adjacent times, improving CPU and GPU cache hits and reducing the number of calling instructions.

- **Dynamic Instantiation**

In order to merge two Draw Calls, they must have consistent shader bindings (FMeshDrawCommand::MatchesForDynamicInstancingreturn true).

Currently only cached mesh drawing commands are dynamically instantiated, and are subject to FLocalVertexFactorycaching support. In addition, there are some special cases that will prevent merging:

- Lightmap generates very small textures (adjustableDefaultEngine.iniMaxLightmapRadius **parameter**).
- Per-component vertex color.
- SpeedTree with wind node.

Use the console command**r.MeshDrawCommands.LogDynamicInstancingStats 1**to explore the benefits of dynamic instancing.

- **Parallel drawing**

Most mesh drawing tasks are not executed in the rendering thread, but are triggered in parallel by the TaskGraph system. The parallel part includes Pass Content settings, dynamic instruction generation/ sorting/merging, etc.
The number of parallel operations is determined by the number of CPU cores of the running device. After parallel operation is enabled, there is a Join phase to wait for all parallel threads to complete execution (FSceneRenderer::WaitForTasksClearSnapshotsAndDeleteSceneRendererenable parallel drawing waiting).

- **Cache drawing instructions**

In order to improve the proportion and efficiency of the cache, UE separates the drawing of dynamic and static objects, forming dynamic drawing paths and static drawing paths respectively. The static drawing path can cache FMeshBatch and FMeshDrawCommand when the primitives are added to the scene, thus achieving the high efficiency of generating multiple drawings once.

- **Improve cache hit rate**

The cache of CPU or GPU is based on the principles of temporal locality and spatial locality. Temporal locality means that if the recently accessed data is accessed again, the probability of cache hit is higher; spatial locality means that the adjacent data of the currently processed data is more likely to be cache hit, and also includes the prefetch hit rate.
UE improves cache hit rate by:

- Based on data-driven design rather than object-oriented design.
    - Such as the structural design of FMeshDrawCommand.
- Store data continuously.
    - Use TChunkedArray to store FMeshDrawCommand.
- Memory alignment.
    - Use a custom memory aligner and memory allocator.
- Lightweight data structure.
- Continuous access to data.
    - Continuously iterate over the drawing instructions.
- Drawing instruction sequencing.
    - Keep similar instructions together to take advantage of the temporal locality of the cache.

3.4.2 Debug console variables

The following are console commands related to parallel drawing, so as to dynamically set or debug its performance and behavior:

Control variables	Analysis
r.MeshDrawCommands.ParallelPassSetup	Switch mesh draw command to process passes in parallel.
r.MeshDrawCommands.UseCachedCommands	Toggle drawing command caching.
r.MeshDrawCommands.DynamicInstancing	Switch dynamic instantiation.
r.MeshDrawCommands.LogDynamicInstancingStats	Outputs dynamically instantiated data, often used to explore the benefits of dynamic instantiation.
r.RHICmdBasePassDeferredContexts	Toggle parallel drawing of the base pass.

3.4.3 Limitations

The current model drawing path of UE introduces many steps and concepts, the purpose of which is to improve rendering efficiency as much as possible. However, this is not only good but also bad. As the saying goes, there is no free lunch in the world. In general, this rendering mechanism has the following disadvantages:

- The system appears large and complex, increasing the learning cost for beginners. Increase the
- cost of reconstruction and expansion. For example, it is not possible to quickly implement multi-pass drawing or add a specified pass. You must deeply understand/be familiar with /modify the underlying source code of the engine to achieve it.
- UE's heavy drawing pipeline encapsulation has a certain basic consumption. For simple application scenarios, its performance may not be as good as those rendering engines that are not encapsulated.

Choosing the more important of two options is the result of UE's long-term trade-offs and improvements.
But this rendering pipeline is future-proof, catering to technologies such as virtualized textures and geometry, RGD, GPU Driven Rendering Pipeline, and real-time ray tracing.

3.4.4 This assignment

There were no homework assignments in the first two articles, but this article will start to assign some small homework assignments to help readers deepen their understanding and master UE's rendering system. The small homework assignments in this article are as follows:

- Briefly recap the process and design concepts of the model drawing pipeline and its purpose.
- Please explain what logic can be optimized in the current model drawing pipeline.
- Add a Mesh Component that can draw any number of materials.
- Add a dedicated Pass to draw the depth of translucent and masked objects.

The above are all open-ended questions with no standard answers. Students who have ideas are welcome to leave a comment in the comment section and I will try my best to reply.

-
-
-
-
-

References

- Unreal Engine 4 Sources Unreal
- Engine 4 Documentation
- Rendering and Graphics
- Materials
- Graphics Programming
- Unreal Engine 4 Rendering
- Mesh Drawing Pipeline
- Mesh Drawing Pipeline Conversion Guide for Unreal Engine 4.22
- Refactoring the Mesh Drawing Pipeline for Unreal Engine 4.22 | GDC 2019 | Unreal Engine
- Take a look at Mesh Drawing Pipeline in UE 4.22
- Rendering - Schematic Overview
- UE4 Render System Sheet

- From constructing FVisibleMeshDrawCommand to RHICommandList
- Analysis of UE4 rendering module
- Unreal 4 rendering programming (Shader) [Volume 12: MeshDrawPipline]
- UE4 StaticMesh rendering sorting UE4
- new rendering pipeline summary Hash
- Functions
- DirectX 12 Multi-Indirect Draw
- Indirect Drawing
- Indirect drawing and GPU culling
- GPU-Driven Rendering
- How to improve the cache hit rate of code

https://github.com/pe7yu