Analysis of Unreal Rendering System (08) - Shaders System

Table of contents

8.1 Overview

Shader is a logical instruction executed on the GPU side. Depending on the execution unit, it can be divided into vertex shader, pixel shader, compute shader, geometry shader, mesh shader, etc.

In order to be cross-platform and cross-graphics API, UE's Shader has done a lot of encapsulation and abstraction, which has explained many types and concepts. In addition, in order to optimize and improve code reuse, concepts and types such as arrangement , PSO, and DDC have been added.

Many previous chapters have involved the concepts, types and codes of Shader. This article will explain its system in more depth and extensive. It mainly explains the following contents of UE:

- The basic concept of Shader. The base type of
- Shader. Shader implementation level. How to
- use Shader and its use cases. The
- implementation and principle of Shader.
- Shader's cross-platform mechanism.
-

It should be noted that the Shaders involved in this article include both the concepts and types of the C++ layer and the concepts and types of the GPU layer.

8.2 Shader Basics

This chapter will analyze the basic concepts and types involved in Shader, and explain the basic relationship and usage between them.

8.2.1 FShader

FShaderis a type of compiled shader code and its parameter bindings. It is the most basic, core, and common type in our rendering code. Its definition is as follows:

```
// Engine\Source\Runtime\RenderCore\Public\Shader.h

classRENDERCORE_API FShader {

public:
    (……)

    //Modify compilation environment parameters before compilation is triggered, which can be overridden by subclasses.
    static void ModifyCompilationEnvironment(constFShaderPermutationParameters&,
FShaderCompilerEnvironment&) {}
    //Whether to compile the specified arrangement, can be overridden by subclasses.
    static bool ShouldCompilePermutation(constFShaderPermutationParameters&) {return true; }
```

```
//Checks whether the compilation result is valid, which can be overridden by subclasses.
static bool ValidateCompiledResult(EShaderPlatform InPlatform,const FShaderParameterMap&
InParameterMap, TArray<FString>& OutError) {returntrue; }

//Get all kinds of dataHashThe interface.
constFSHAHash&GetHash()const;
constFSHAHash&GetVertexFactoryHash()const; const
FSHAHash&GetOutputHash()const;

//Save and testshaderThe compilation result of the code.
void Finalize(constFShaderMapResourceCode* Code);

//Data acquisition interface.
inlineFShaderType*GetType(constFShaderMapPointerTable& InPointerTable)const{ return
Type.Get(InPointerTable.ShaderTypes); }
inlineFShaderType*GetType(constFPointerTableBase* InPointerTable)const{return Type.Get(InPointerTable); }

inlineFVertexFactoryType*GetVertexFactoryType(constFShaderMapPointerTable& InPointerTable)const{
returnVFType.Get(InPointerTable.VFTypes); }
inlineFVertexFactoryType*GetVertexFactoryType(constFPointerTableBase* InPointerTable)const{
returnVFType.Get(InPointerTable); }
inlineFShaderType*GetTypeUnfrozen()const{returnType.GetUnfrozen(); } inlineint32GetResourceIndex()const
{ checkSlow(ResourceIndex != INDEX_NONE);return ResourceIndex; }

inlineEShaderPlatformGetShaderPlatform()const{returnTarget.GetPlatform(); } inlineEShaderFrequency
GetFrequency()const{returnTarget.GetFrequency(); } inlineconstFShaderTargetGetTarget()const{return
Target; } inlinebool IsFrozen()const{returnType.IsFrozen(); } inlineuint32GetNumInstructions()const{return
NumInstructions; }


#if WITH_EDITORONLY_DATA
inlineuint32GetNumTextureSamplers()const{returnNumTextureSamplers; } inlineuint32GetCodeSize
()const{returnCodeSize; }
inlinevoid SetNumInstructions(uint32 Value) { NumInstructions = Value; }
#else
inlineuint32GetNumTextureSamplers()const{return0u; } inlineuint32
GetCodeSize()const{return0u; }
#endif

//Tries to return an auto-boundUniform Buffer,If not present, returns unbound.
template<typename UniformBufferStructType>
constTShaderUniformBufferParameter<UniformBufferStructType>&
GetUniformBufferParameter()const;
constFShaderUniformBufferParameter&          GetUniformBufferParameter(const const;
FShaderParametersMetadata* SearchStruct)
constFShaderUniformBufferParameter&GetUniformBufferParameter(constFHashedName SearchName)
          const;
const   FShaderParametersMetadata*FindAutomaticallyBoundUniformBufferStruct(int32 const;
BaseIndex)
staticinlineconstFShaderParametersMetadata*GetRootParametersMetadata();

(.....)

public:
//Shader parameter bindings.
LAYOUT_FIELD(FShaderParameterBindings, //Mapping          Bindings);
information for shader parameter bindings.
LAYOUT_FIELD(FShaderParameterMapInfo,          ParameterMapInfo);
```

```
protected:
    LAYOUT_FIELD(TMemoryImageArray<FHashedName>, UniformBufferParameterStructs);
    LAYOUT_FIELD(TMemoryImageArray<FShaderUniformBufferParameter>, UniformBufferParameters);


    //  under3The first one is the editor parameter.
    //  The shader's compiled output and the resulting parameter map hash,  Used to find matching resources.
    LAYOUT_FIELD_EDITORONLY(FSHAHash, //  OutputHash);
    Vertex factory resource hash
    LAYOUT_FIELD_EDITORONLY(FSHAHash, // shader  VFSourceHash);
    The resource hash value.
    LAYOUT_FIELD_EDITORONLY(FSHAHash,  SourceHash);

private:
    //The shader type.
    LAYOUT_FIELD(TIndexedPtr<FShaderType>, //The  Type);
    vertex factory type.
    LAYOUT_FIELD(TIndexedPtr<FVertexFactoryType>, //Target  VFType);
    platform and shading frequency (frequency).
    LAYOUT_FIELD(FShaderTarget, Target);

    //existFShaderMapResourceofshaderindex.
    LAYOUT_FIELD(int32,  ResourceIndex);
    // shaderNumber of instructions.
    LAYOUT_FIELD(uint32,  NumInstructions);
    //The number of texture samplers.
    LAYOUT_FIELD_EDITORONLY(uint32, //
    shaderCode size.  NumTextureSamplers);
    LAYOUT_FIELD_EDITORONLY(uint32,
                                  CodeSize);
};
```

As can be seen above, FShader stores Shader-associated binding parameters, vertex factories, compiled resources and other data, and provides compiler modification and detection interfaces, as well as various data acquisition interfaces.

FShader is actually a base parent class, and its subclasses are:

- **FGlobalShader**:Global shader, its subclass has only one instance in memory, often used for screen block drawing, post-processing, etc. Its definition is as follows:

```
// Engine\Source\Runtime\RenderCore\Public\GlobalShader.h

classFGlobalShader: public FShader {

public:
    (.....)

    FGlobalShader() : FShader() {}
    FGlobalShaderconstShaderMetaType::CompiledShaderInitializerType& Initializer);

    //Set view shader parameters.
    template<typename TViewUniformShaderParameters, typename ShaderRHIParamRef, typename
    TRHICmdList>
```

```
    inlinevoid SetParameters(TRHICmdList& RHICmdList, ...);
};
```

Compared with the parent class FShader, the SetParameters interface for setting the view unified buffer is added.

- **FMaterialShader**:Material shader, the shader referenced by the material specified by FMaterialShaderType, is a shader subset of the material blueprint after instantiation. It is defined as follows:

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

classRENDERER_API FMaterialShader: public FShader {

public:
    (.....)

    FMaterialShader() =default;
    FMaterialShaderconstFMaterialShaderType::CompiledShaderInitializerType& Initializer);


    //Setting up the viewUniform  Bufferparameter.
    template<typename  ShaderRHIParamRef>
    void SetViewParameters(FRHICommandList& RHICmdList, ...); //Sets material
    related but notFMeshBatchRelevant pixel shader parameters
    template<typename TRHIShader>
    void SetParameters(FRHICommandList& RHICmdList, ...); //Get shader
    parameter bindings.
    void GetShaderBindings(constFScene* Scene, ...)const;

private:
    //Is it allowed?UniformExpression caching.
    staticint32 bAllowCachedUniformExpressions; //
    bAllowCachedUniformExpressionsCorresponding console traversal.
    staticFAutoConsoleVariableRef CVarAllowCachedUniformExpressions;

#if!(UE_BUILD_TEST || UE_BUILD_SHIPPING || !WITH_EDITOR) //Validate
    expressions and shader graphs for validity.
    void VerifyExpressionAndShaderMaps(constFMaterialRenderProxy* MaterialRenderProxy,const
FMaterial& Material,constFUniformExpressionCache* UniformExpressionCache)
                                    const;
#endif
    //  Assigned ParametersUniform  Buffer.
    LAYOUT_FIELD(TMemoryImageArray<FShaderUniformBufferParameter>,
    ParameterCollectionUniformBuffers);
    //Material ShaderUniform Buffer. LAYOUT_FIELD(FShaderUniformBufferParameter,
    MaterialUniformBuffer);


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

The following are some subclasses under the FShader inheritance system:

FShader
- FGlobalShader
  - TMeshPaintVertexShader
  - TMeshPaintPixelShader
  - FDistanceFieldDownsamplingCS
  - FBaseGPUSkinCacheCS
    - TGPUSkinCacheCS
  - FBaseRecomputeTangentsPerTriangleShader
  - FBaseRecomputeTangentsPerVertexShader
  - FRadixSortUpsweepCS
  - FRadixSortDownsweepCS FParticleTileVS
  - FBuildMipTreeCS FScreenVS FScreenPS
  - FScreenPSInvertAlpha FSimpleElementVS
  - FSimpleElementPS FStereoLayerVS
  - FStereoLayerPS_Base
    - FStereoLayerPS
  - FUpdateTexture2DSubresouceCS
  - FUpdateTexture3DSubresouceCS
  - FCopyTexture2DCS
  - TCopyDataCS
  - FLandscapeLayersVS
  - FLandscapeLayersHeightmapPS
  - FGenerateMipsCS
  - FGenerateMipsVS
  - FGenerateMipsPS
  - FCopyTextureCS
  - FMediaShadersVS
  - FRGBConvertPS
  - FYUVConvertPS
  - FYUY2ConvertPS
  - FRGB10toYUVv210ConvertPS
  - FInvertAlphaPS
  - FSetAlphaOnePS
  - FReadTextureExternalPS
  - FOculusVertexShader
  - FRasterizeToRectsVS
  - FResolveVS
  - FResolveDepthPS
  - FResolveDepth2XPS
  - FAmbientOcclusionPS
  - FGTAOSpatialFilterCS
  - FGTAOTemporalFilterCS
  - FDeferredDecalVS
  - FDitheredTransitionStencilPS
  - FObjectCullVS
  - FObjectCullPS
  - FDeferredLightPS
  - TDeferredLightHairVS
  - FFXAAVS
  - FFXAAPS
  - FMotionBlurShader
  - FSubsurfaceShader

FTonemapVS FTonemapPS
FTonemapCS FUpscalePS
FTAAStandaloneCS
FSceneCapturePS FHZBTestPS
FOcclusionQueryVS
FOcclusionQueryPS FHZBBuildPS
FHZBBuildCS
FDownsampleDepthPS
FTiledDeferredLightingCS
FShader_VirtualTextureCompress
FShader_VirtualTextureCopy
FPageTableUpdateVS
FPageTableUpdatePS
FSlateElementVS FSlateElementPS
(.....)

FMaterialShader
FDeferredDecalPS
FLightHeightfieldsPS
FLightFunctionVS
FLightFunctionPS
FPostProcessMaterialShader
TTranslucentLightingInjectPS
FVolumetricFogLightFunctionPS
FMeshMaterialShader

FLightmapGBufferVS
FLightmapGBufferPS
FVLM Voxelization VS
FVLMVoxelizationGS
FVLMVoxelizationPS
FLandscapeGrassWeightVS
FLandscapeGrassWeightPS
FLandscapePhysicalMaterial
FAnisotropyVS
FAnisotropyPS
TBasePassVertexShaderPolicyParamType

TBasePassVertexShaderBaseType
TBasePassVS
TBasePassPixelShaderPolicyParamType
TBasePassPixelShaderBaseType
TBasePassPS
FMeshDecalsVS
FMeshDecalsPS
TDepthOnlyVS
TDepthOnlyPS
FDistortionMeshVS
FDistortionMeshPS
FHairMaterialVS
FHairMaterialPS
FHairVisibilityVS
FHairVisibilityPS
TLightMapDensityVS
TLightMapDensityPS
FShadowDepthVS

```
            FShadowDepthBasePS
                TShadowDepthPS
            FTranslucencyShadowDepthVS
            FTranslucencyShadowDepthPS  FVelocityVS
            FVelocityPS       FRenderVolumetricCloudVS
            FVolumetricCloudShadowPS
            FVoxelizeVolumeVS       FVoxelizeVolumePS
            FShader_VirtualTextureMaterialDraw (.....)



        FSlateMaterialShaderVS
        FSlateMaterialShaderPS
        (.....)
```

The above only lists part of the FShader inheritance system, including some Shader types that have been parsed before, such as FDeferredLightPS, FFXAAPS, FTonemapPS, FUpscalePS, TBasePassPS, TDepthOnlyPS, etc.

FGlobalShader includes Shader codes for post-processing, lighting, tools, visualization, terrain, virtual textures, etc. It can be VS, PS, CS, but CS must be a subclass of FGlobalShader; FMaterialShader mainly includes Shader codes for models, special passes , voxelization, etc. It can be VS, PS, GS, etc., but there will be no CS.

If you define a new subclass of FShader, you need to use the following macro declarations and implement the corresponding code (some common macros):

```
//------ ShaderDeclaring and implementing macros ------

//Declare a specified type (FShadersubclass)Shader,Can beGlobal, Material, MeshMaterial, ...
#defineDECLARE_SHADER_TYPE(ShaderClass,ShaderMetaTypeShortcut,...) //Implements
the specified typeShader,Can beGlobal, Material, MeshMaterial, ...
#define
IMPLEMENT_SHADER_TYPE(TemplatePrefix,ShaderClass,SourceFilename,FunctionName,Frequency)

//statementFGlobalShaderand its subclasses.
#defineDECLARE_GLOBAL_SHADER(ShaderClass) //
accomplishFGlobalShaderand its subclasses.
#defineIMPLEMENT_GLOBAL_SHADER(ShaderClass,SourceFilename,FunctionName,Frequency)

//accomplishMaterialShaders.
#define
IMPLEMENT_MATERIAL_SHADER_TYPE(TemplatePrefix,ShaderClass,SourceFilename,FunctionName,Freq uency)


//Other less common macros
(.....)

//------Example1 ------

classFDeferredLightPS: public FGlobalShader {

        //existFDeferredLightPSDeclare global shader in class
```

```
        DECLARE_SHADER_TYPE(FDeferredLightPS, Global) (.....)

};
//accomplishFDeferredLightPSThe shader, associates it with the code file, the main entry point and the shading frequency.
IMPLEMENT_GLOBAL_SHADER(FDeferredLightPS,"/Engine/Private/DeferredLightPixelShaders.usf", "DeferredLightPixelMain",
SF_Pixel);


//------Example2 ------

classFDeferredDecalPS: public FMaterialShader {

        //Declare material shader inside a class
        DECLARE_SHADER_TYPE(FDeferredDecalPS,Material); (.....)

};
//accomplishFDeferredDecalPSclass, associate it with the code file, the main entry point and the shading frequency.
IMPLEMENT_MATERIAL_SHADER_TYPE(,FDeferredDecalPS,TEXT("/Engine/Private/DeferredDecal.usf") , TEXT("MainPS"),SF_Pixel);
```

8.2.2 Shader Parameter

Shader parameters are a set of data passed from the CPU's C++ layer to the GPU Shader and stored in GPU registers or video memory. The following are definitions of common types of shader parameters:

```
// Engine\Source\Runtime\RenderCore\Public\ShaderParameters.h

//The register binding parameter of the shader, its type can befloat1/2/3/4, array,UAV
wait. classFShaderParameter
{
        (.....)
public:
        //Binds the parameter with the specified name.
        void Bind(constFShaderParameterMap& ParameterMap,constTCHAR* ParameterName,
EShaderParameterFlags Flags = SPF_Optional);
        //Whether it has been bound by the shader.
        bool IsBound()const; //
        Whether to initialize.
        inlinebool IsInitialized()const;


        //Data acquisition interface.
        uint32    GetBufferIndex()        const;
        uint32    GetBaseIndex()      const;
        uint32    GetNumBytes()   const;

        (.....)
};

//Shader resource binding (texture or sampler)
classFShaderResourceParameter {

        (.....)
public:
        //Binds the parameter with the specified name.
```

```
        void Bind(constFShaderParameterMap& ParameterMap,constTCHAR*
ParameterName,EShaderParameterFlags Flags = SPF_Optional);
        bool IsBound()const;
        inlinebool IsInitialized()const;

        uint32     GetBaseIndex()const;
        uint32     GetNumResources()const;

        (......)
};

//BoundUAVorSRVThe type of
resource. class
FRWShaderParameter {
        (......)
public:
        //Binds the parameter with the specified name.
        void Bind(constFShaderParameterMap& ParameterMap,constTCHAR* BaseName);

        bool    IsBound()const;
        bool IsUAVBound()const;
        uint32     GetUAVIndex()const;

        //Set the buffer data toRHI.
        template<typename TShaderRHIRef, typename TRHICmdList>
        inlinevoid SetBuffer(TRHICmdList& RHICmdList,constTShaderRHIRef& Shader,const FRWBuffer & RWBuffer)
const;
        template<typename TShaderRHIRef, typename TRHICmdList>
        inlinevoid SetBuffer(TRHICmdList& RHICmdList,constTShaderRHIRef& Shader,const FRWBufferStructured&
RWBuffer)const;

        //Set the texture data toRHI.
        template<typename TShaderRHIRef, typename TRHICmdList>
        inlinevoid SetTexture(TRHICmdList& RHICmdList,constTShaderRHIRef& Shader, FRHITexture* Texture,
FRHIUnorderedAccessView* UAV)const;

        //fromRHIUnsetUAV.
        template<typename TRHICmdList>
        inlinevoid UnsetUAV(TRHICmdList& RHICmdList, FRHIComputeShader* ComputeShader)const;

        (......)
};

//Create a specified platformUniform BufferShader code declaration of the structure.
externvoid CreateUniformBufferShaderDeclaration(constTCHAR* Name,const FShaderParametersMetadata&
UniformBufferStruct, EShaderPlatform Platform, FString& OutDeclaration);


//Shader uniform buffer parameters.
classFShaderUniformBufferParameter {

        (......)
public:
        //Modify the compilation environment variables.
        static void ModifyCompilationEnvironment(constTCHAR* ParameterName,const FShaderParametersMetadata&
Struct,EShaderPlatform Platform,FShaderCompilerEnvironment& OutEnvironment);
```

```
        //Bind shader parameters.
        void Bind(constFShaderParameterMap& ParameterMap,constTCHAR*
ParameterName,EShaderParameterFlags Flags = SPF_Optional);

        bool IsBound()const;
        inlinebool IsInitialized()const; uint32
        GetBaseIndex()const;

        (......)
};

//Specify the shader uniform buffer parameters of the structure
template<typename TBufferStruct>
classTShaderUniformBufferParameter: public FShaderUniformBufferParameter {

public:
        static void ModifyCompilationEnvironment(constTCHAR* ParameterName,EShaderPlatform Platform,
FShaderCompilerEnvironment& OutEnvironment);

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

It can be seen that shader parameters can be bound to any GPU type of resources or data, but different classes can only bind to specific shader types and cannot be mixed. For example, FRWShaderParameter can only bind to UAV or SRV. With the above types , you can declare specific
Shader parameters in the C++ layer Shader class with the related macros of LAYOUT_FIELD.

LAYOUT_FIELD is a macro that can declare the type, name, initial value, bit field, write function and other data of the specified shader parameter. Its relevant definition is as follows:

```
// Engine\Source\Runtime\Core\Public\Serialization\MemoryLayout.h

//Normal layout
#defineLAYOUT_FIELD(T, Name, ...) //With
initial value
#defineLAYOUT_FIELD_INITIALIZED(T, Name, Value, ...) //bringmutable
and initial value
#defineLAYOUT_MUTABLE_FIELD_INITIALIZED(T, Name, Value, ...) //Array Layout

#defineLAYOUT_ARRAY(T, Name, NumArray, ...)
#defineLAYOUT_MUTABLE_BITFIELD(T, Name, BitFieldSize, ...) //Bit Field

#defineLAYOUT_BITFIELD(T, Name, BitFieldSize, ...) //With write
  function
#defineLAYOUT_FIELD_WITH_WRITER(T, Name, Func)
#defineLAYOUT_MUTABLE_FIELD_WITH_WRITER(T, Name, Func)
#define     LAYOUT_WRITE_MEMORY_IMAGE(Func)
#define     LAYOUT_TOSTRING(Func)
```

With the help of macros such as LAYOUT_FIELD, you can declare shader parameters of a specified type in a C++ class. For example:

```
structFMyExampleParam
{
```

```
    //Declare a non-virtual class.
    DECLARE_TYPE_LAYOUT(FMyExampleParam, NonVirtual);

    //Bit Field
    LAYOUT_FIELD(FShaderParameter, ShaderParam);//is equivalent to:FShaderParameter ShaderParam;
    LAYOUT_FIELD(FShaderResourceParameter, TextureParam);//is equivalent to: FShaderResourceParameter
    TextureParam;
    LAYOUT_FIELD(FRWShaderParameter, OutputUAV);//is equivalent to:FRWShaderParameter OutputUAV;

    //Array,3The parameter is the maximum number.
    LAYOUT_ARRAY(FShaderResourceParameter, TextureArray,5);//is equivalent to:
FShaderResourceParameter TextureArray[5];
    LAYOUT_ARRAY(int32, Ids,64);//is equivalent to:int32 Ids[64];

    LAYOUT_FIELD_INITIALIZED(uint32, Size,0);//is equivalent to:int32 Size = 0;

    void WriteDataFunc(FMemoryImageWriter& Writer,const
TMemoryImagePtr<FOtherExampleParam>& InParameters)const;
    //With write function.
    LAYOUT_FIELD_WITH_WRITER(TMemoryImagePtr<FOtherExampleParam>,          Parameters,
    WriteDataFunc);
};
```

8.2.3 Uniform Buffer

UE's Uniform Buffer involves several core concepts. The bottom layer is the FRHIUniformBuffer of the RHI layer, which encapsulates the uniform buffer (also called Constant Buffer) of various graphics APIs. Its definition is as follows (the implementation and debugging code are removed):

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

classFRHIUniformBuffer: public FRHIResource {

public:
    //Constructor.
    FRHIUniformBuffer(constFRHIUniformBufferLayout&          InLayout);

    //Reference counting operations.
    uint32    AddRef()    const;
    uint32    Release()     const;

    //Data acquisition interface.
    uint32GetSize()const;
    constFRHIUniformBufferLayout&GetLayout()const; bool IsGlobal()
    const;

private:
    // RHI Uniform BufferLayout. const
    FRHIUniformBufferLayout* //Buffer size.   Layout;

    uint32 LayoutConstantBufferSize;
};
```

The next level up is TUniformBufferRef, which references the above FRHIUniformBuffer:

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

//definitionFRHIUniformBufferThe reference type.
typedefTRefCountPtr<FRHIUniformBuffer> FUniformBufferRHIRef;


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

//References the specified typeFRHIUniformBufferInstance resources of . Note that it inherits
FUniformBufferRHIRef. template<typename TBufferStruct>
classTUniformBufferRef: public FUniformBufferRHIRef {

public:
	TUniformBufferRef();

	//Created from the given valueUniform Buffer,And returns a structure reference. (Template)
	staticTUniformBufferRef<TBufferStruct>CreateUniformBufferImmediate(const TBufferStruct& Value,
EUniformBufferUsage Usage, EUniformBufferValidation Validation = EUniformBufferValidation::ValidateResources);

	//Creates a [local] with the given value.Uniform Buffer,And returns a reference to the structure.
	staticFLocalUniformBufferCreateLocalUniformBuffer(FRHICommandList& RHICmdList,const TBufferStruct& Value,
EUniformBufferUsage Usage);

	//Immediately flush the buffer data toRHI.
	void UpdateUniformBufferImmediate(constTBufferStruct& Value);

private:
	//Private structure, can only be givenTUniformBufferandTRDGUniformBuffer
	create. TUniformBufferRef(FRHIUniformBuffer* InRHIRef);

	template<typename TBufferStruct2> friend
	classTUniformBuffer;

	friendclassTRDGUniformBuffer<TBufferStruct>;
};
```

The next level up is TUniformBuffer and TRDGUniformBuffer which reference FUniformBufferRHIRef. They are defined as follows:

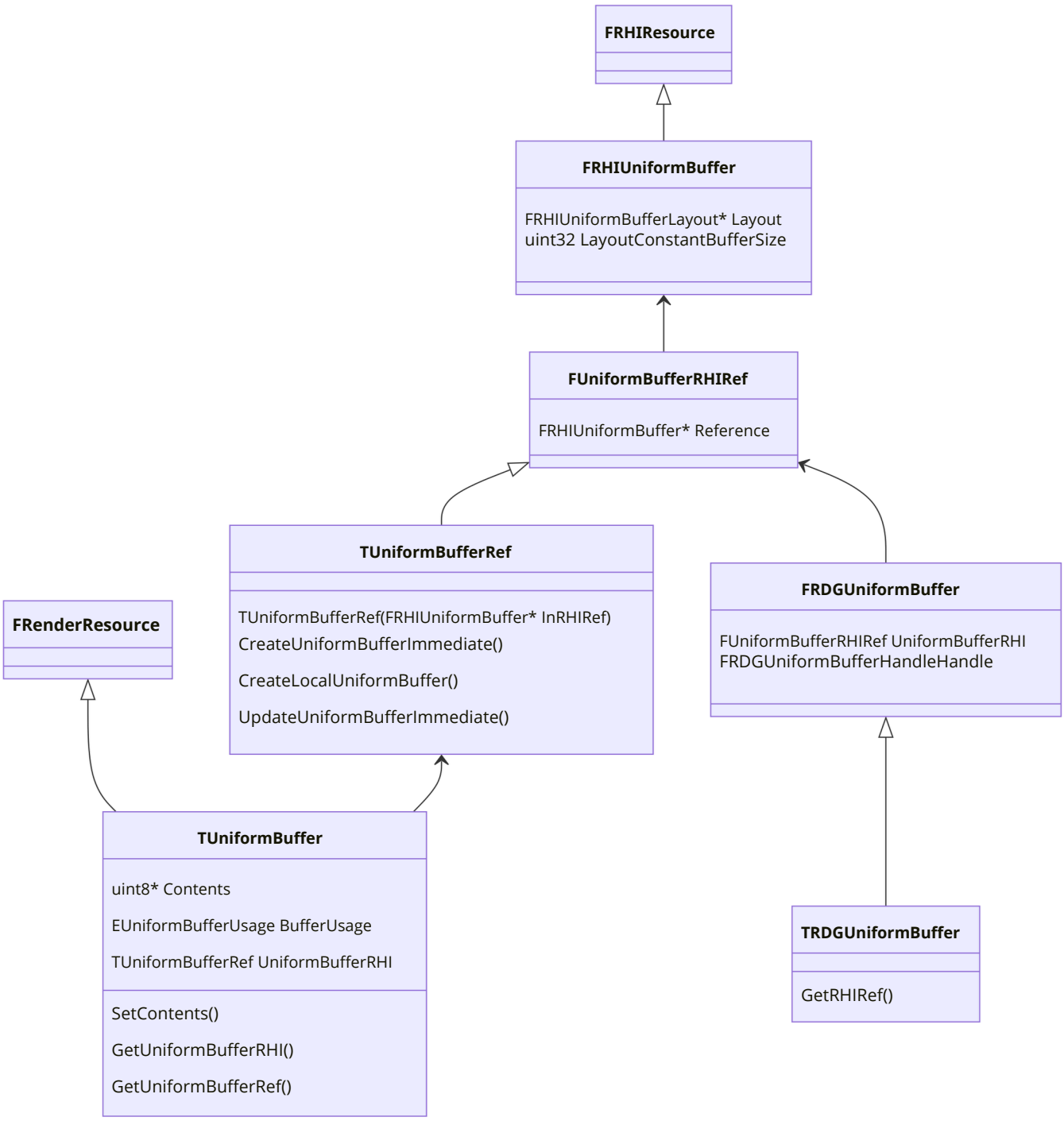
```
// Engine\Source\Runtime\RenderCore\Public\UniformBuffer.h

//CitedUniform BufferResources.
template<typename TBufferStruct>
classTUniformBuffer: public FRenderResource {

public:
	//Constructor.
	TUniformBuffer()
		: BufferUsage(UniformBuffer_MultiFrame) ,
		Contents(nullptr){}

	//Destructor.
	~TUniformBuffer()
	{
		if(Contents)
		{
```

```
            FMemory::Free(Contents);
        }
    }

    //set upUniform BufferContent data.
    void SetContents(constTBufferStruct& NewContents) {

        SetContentsNoUpdate(NewContents);
        UpdateRHI();
    }
    //ClearUniform BufferContent data. (If the content is empty, it will be
    created first) void SetContentsToZero() {

        if(!Contents)
        {
            Contents = (uint8*)FMemory::Malloc(sizeof(TBufferStruct),
SHADER_PARAMETER_STRUCT_ALIGNMENT);
        }
        FMemory::Memzero(Contents,sizeof(TBufferStruct)); UpdateRHI();

    }

    //Get the content.
    constuint8*GetContents()const {

        returnContents;
    }

    //----OverloadFRenderResourceInterface----

    //Initialization DynamicsRHIresource.
    virtualvoid InitDynamicRHI() override {

        check(IsInRenderingThread());
        UniformBufferRHI.SafeRelease(); if
        (Contents)
        {
            //Created from the content data of the binary streamRHIresource.
            UniformBufferRHI = CreateUniformBufferImmediate<TBufferStruct>(*((const
TBufferStruct*)Contents), BufferUsage);
        }
    }
    //Release DynamicsRHIresource.
    virtualvoid ReleaseDynamicRHI() override {

        UniformBufferRHI.SafeRelease();
    }

    //Data access interface.
    FRHIUniformBuffer*GetUniformBufferRHI() {           const

        returnUniformBufferRHI;
    }
    constTUniformBufferRef<TBufferStruct>&GetUniformBufferRef()const {

        returnUniformBufferRHI;
    }
```

```
        // Buffermark.
        EUniformBufferUsage BufferUsage;

protected:
        //set upUniform BufferContent data.
        void SetContentsNoUpdate(constTBufferStruct& NewContents) {

                if(!Contents)
                {
                        Contents = (uint8*)FMemory::Malloc(sizeof(TBufferStruct),
SHADER_PARAMETER_STRUCT_ALIGNMENT);
                }
                FMemory::Memcpy(Contents,&NewContents,sizeof(TBufferStruct));
        }

private:
        //TUniformBufferRefReferences.
        TUniformBufferRef<TBufferStruct> //CPU      UniformBufferRHI;
        Side content data.
        uint8* Contents;
};


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

classFRDGUniformBuffer: public FRDGResource {

public:
        bool IsGlobal()const;
        constFRDGParameterStruct&GetParameters()const;

        ////////////////////////////////////////////////// ////////////////////////// //GetRHI,Only available inPass
        Called during execution. FRHIUniformBuffer*GetRHI()const {


                returnstatic_cast<FRHIUniformBuffer*>(FRDGResource::GetRHI());
        }
        ////////////////////////////////////////////////// //////////////////////

protected:
        //Constructor.
        template <typename       TParameterStruct>
        explicitFRDGUniformBuffer(TParameterStruct* InParameters,constTCHAR* InName)
                :FRDGResource(InName)
                , ParameterStruct(InParameters)
                , bGlobal(ParameterStruct.HasStaticSlot())
        {}

private:
        const    FRDGParameterStruct ParameterStruct; Cited
        // FRHIUniformBufferResources.
        //    NoticeTUniformBufferRef<TBufferStruct>andFUniformBufferRHIRefTime equivalent.
        TRefCountPtr<FRHIUniformBuffer>            UniformBufferRHI;
        FRDGUniformBufferHandle Handle;

        //Is it global?ShaderStill localShaderBinding.
        uint8 bGlobal :1;
```

```
    friend    FRDGBuilder;
    friend    FRDGUniformBufferRegistry;
    friend    FRDGAllocator;
};

//FRDGUniformBufferTemplate version of . template <typename
ParameterStructType> classTRDGUniformBuffer: public
FRDGUniformBuffer {

public:
    //Data acquisition interface.
    constTRDGParameterStruct<ParameterStructType>&GetParameters()const;
    TUniformBufferRef<ParameterStructType>GetRHIRef()const; constParameterStructType*
    operator->()const;

private:
    explicitTRDGUniformBuffer(ParameterStructType* InParameters,constTCHAR* InName)
        :FRDGUniformBuffer(InParameters, InName)
    {}

    friend    FRDGBuilder;
    friend    FRDGUniformBufferRegistry;
    friend    FRDGAllocator;
};
```

After abstracting them into a UML inheritance diagram, it looks like this:



> Complaints: The text drawing syntax Mermaid cannot specify the layout, the automatically generated graphic layout is not beautiful enough, and after zooming in on the UI in the window, the text is not fully displayed. Just make do with it.

The above Uniform Buffer types can be defined by SHADER_PARAMETER related macros to define the structure and structure members. SHADER_PARAMETER related macros are defined as follows:

```
// Engine\Source\Runtime\RenderCore\Public\ShaderParameterMacros.h

// Shader Parameter Struct:Start/End.
#define    BEGIN_SHADER_PARAMETER_STRUCT(StructTypeName,          PrefixKeywords)
#define    END_SHADER_PARAMETER_STRUCT()

// Uniform Buffer Struct:Start/End/Achieve.
#define    BEGIN_UNIFORM_BUFFER_STRUCT(StructTypeName, PrefixKeywords)
#define BEGIN_UNIFORM_BUFFER_STRUCT_WITH_CONSTRUCTOR(StructTypeName,       PrefixKeywords)
```

```
#define END_UNIFORM_BUFFER_STRUCT()
#define IMPLEMENT_UNIFORM_BUFFER_STRUCT(StructTypeName,ShaderVariableName)
#define IMPLEMENT_STATIC_UNIFORM_BUFFER_STRUCT(StructTypeName,ShaderVariableName,StaticSlotName)
#define IMPLEMENT_UNIFORM_BUFFER_ALIAS_STRUCT(StructTypeName, UniformBufferAlias)
#define IMPLEMENT_STATIC_UNIFORM_BUFFER_SLOT(SlotName)



// Global Shader Parameter Struct:Start/End/Achieve.
#define    BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT
#define    BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT_WITH_CONSTRUCTOR
#define    END_GLOBAL_SHADER_PARAMETER_STRUCT
#define    IMPLEMENT_GLOBAL_SHADER_PARAMETER_STRUCT
#define    IMPLEMENT_GLOBAL_SHADER_PARAMETER_ALIAS_STRUCT

// Shader Parameters:Single, Array.
#define    SHADER_PARAMETER(MemberType, MemberName)
#define    SHADER_PARAMETER_EX(MemberType,MemberName,Precision)
#define    SHADER_PARAMETER_ARRAY(MemberType,MemberName,ArrayDecl)
#define SHADER_PARAMETER_ARRAY_EX(MemberType,MemberName,ArrayDecl,Precision)

// Shader Parameters:Texture,SRV, UAV,Samplers and their arrays
#define    SHADER_PARAMETER_TEXTURE(ShaderType,MemberName)
#define    SHADER_PARAMETER_TEXTURE_ARRAY(ShaderType,MemberName,    ArrayDecl)
#define    SHADER_PARAMETER_SRV(ShaderType,MemberName)
#define    SHADER_PARAMETER_UAV(ShaderType,MemberName)
#define    SHADER_PARAMETER_SAMPLER(ShaderType,MemberName)
#define    SHADER_PARAMETER_SAMPLER_ARRAY(ShaderType ,MemberName,    ArrayDecl)

// Shader Parameter StructInternalShader Parameter Structparameter.
#define    SHADER_PARAMETER_STRUCT(StructType,MemberName)
#define    SHADER_PARAMETER_STRUCT_ARRAY(StructType,MemberName,    ArrayDecl)
#define SHADER_PARAMETER_STRUCT_INCLUDE(StructType,MemberName)
//Reference to a [global] shader parameter structure.
#defineSHADER_PARAMETER_STRUCT_REF(StructType,MemberName)

//RDGPatternShader    Parameter.
#define    SHADER_PARAMETER_RDG_TEXTURE(ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_TEXTURE_SRV(ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_TEXTURE_UAV(ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_TEXTURE_UAV_ARRAY(ShaderType,MemberName,    ArrayDecl)
#define    SHADER_PARAMETER_RDG_BUFFER(ShaderType ,MemberName)
#define    SHADER_PARAMETER_RDG_BUFFER_SRV(ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_BUFFER_SRV_ARRAY(ShaderType,MemberName,    ArrayDecl)
#define    SHADER_PARAMETER_RDG_BUFFER_UAV(ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_BUFFER_UAV( ShaderType,MemberName)
#define    SHADER_PARAMETER_RDG_BUFFER_UAV_ARRAY(ShaderType,MemberName,    ArrayDecl)
#define    SHADER_PARAMETER_RDG_UNIFORM_BUFFER(StructType, MemberName)
```

Note that the local (ordinary) Shader Parameter Struct does not implement the (IMPLEMENT_SHADER_PARAMETER_STRUCT) macro, only the Global one has (IMPLEMENT_GLOBAL_SHADER_PARAMETER_STRUCT).

Here is an example showing how to use some of the above macros to declare various parameters of the shader:

```
//Defines a global shader parameter structure (available in .hor.cpp,But generally in.h)
BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT(FMyShaderParameterStruct,                    )
    //Regular single and array parameters.
    SHADER_PARAMETER(float, Intensity)
    SHADER_PARAMETER_ARRAY(FVector3, Vertexes, [8])

    //Samplers, Textures,SRV, UAV
    SHADER_PARAMETER_SAMPLER(SamplerState,        TextureSampler
    SHADER_PARAMETER_TEXTURE(Texture3D,       Texture3d)
    SHADER_PARAMETER_SRV(Buffer<float4>,         VertexColorBuffer)
    SHADER_PARAMETER_UAV(RWStructuredBuffer<float4>,              OutputTexture)

    //    Shader parameter structure
    //    Reference shader parameter structure (global only)
    SHADER_PARAMETER_STRUCT_REF(FViewUniformShaderParameters, //Contains      View)
    shader parameter structures (local or global)
    SHADER_PARAMETER_STRUCT_INCLUDE(FSceneTextureShaderParameters,              SceneTextures
    END_GLOBAL_SHADER_PARAMETER_STRUCT()

//Implements a global shader parameter structure (only available in .cpp)
IMPLEMENT_GLOBAL_SHADER_PARAMETER_STRUCT(FMyShaderParameterStruct,
"MyShaderParameterStruct");
```

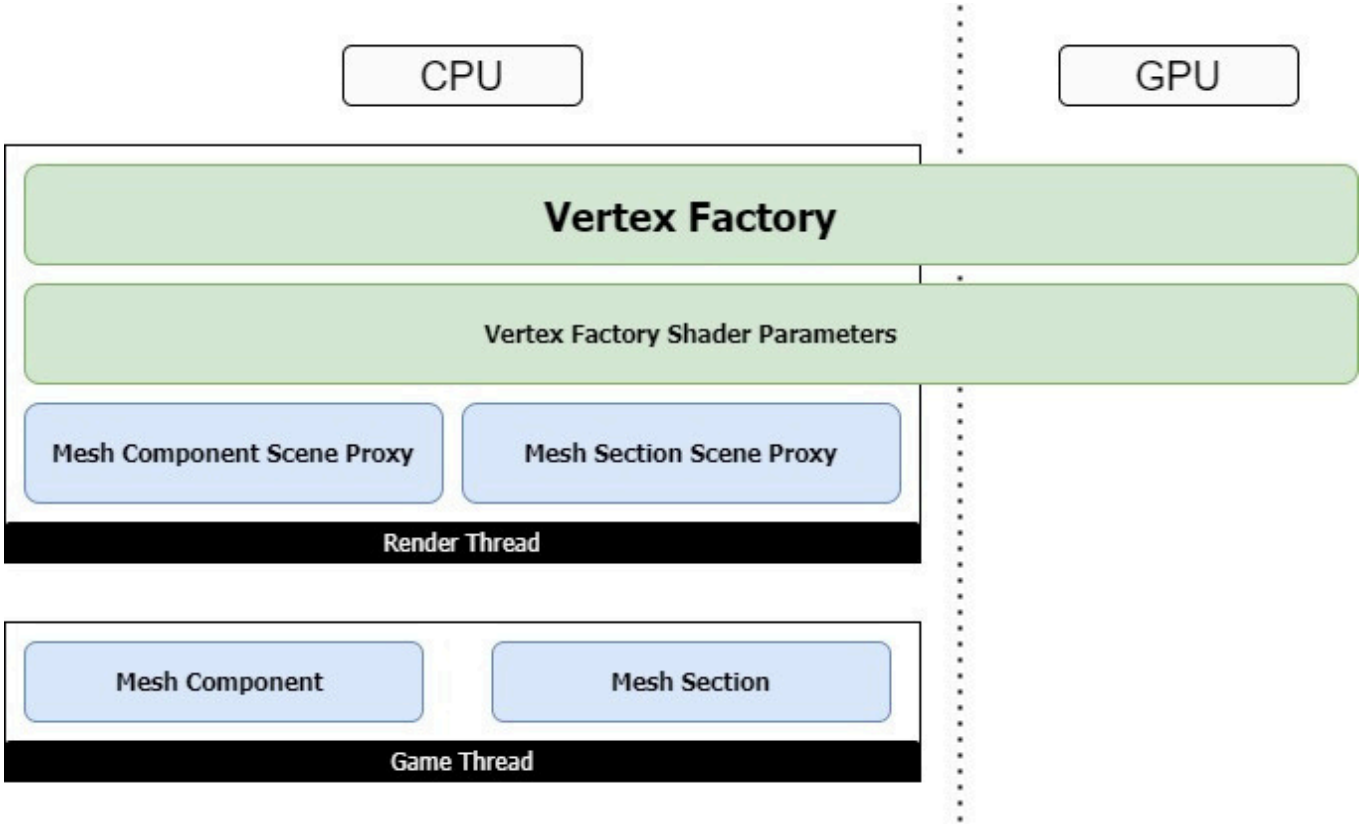
The above shader structure is declared and implemented on the C++ side. If it needs to be correctly passed into the Shader, additional C++ code is required to complete it:

```
//Declare the structure.
FMyShaderParameterStruct MyShaderParameterStruct;

//createRHIresource.
//Can be multiple frames (UniformBuffer_MultiFrame)So, just create1The pointer can be cached once, and there will be subsequent data update
calls UpdateUniformBufferImmediateThat's it.
//It can also be a single frame (UniformBuffer_SingleFrame),Then data needs to be created and updated for each frame.
autoMyShaderParameterStructRHI =
TUniformBufferRef<FMyShaderParameterStruct>::CreateUniformBufferImmediate(ShaderParameterS truct,
EUniformBufferUsage::UniformBuffer_MultiFrame);

//Update the shader parameter structure.
MyShaderParameterStruct.Intensity (.....)      =  1.0f;


//Update data toRHI.
MyShaderParameterStructRHI.UpdateUniformBufferImmediate(MyShaderParameterStruct);
```

8.2.4 Vertex Factory

We know that there are static meshes, skinned skeletons, procedural meshes, terrains and other types of meshes in the engine, and materials support these mesh types through the vertex factory FVertexFactory. In fact, the vertex factory involves various data and types, including but not limited to:

- Vertex shader. The input and output of the vertex shader require a vertex factory to indicate the layout of the data.
- Vertex factory parameters and RHI resources. These data will be passed from the C++ layer to the vertex shader for processing.
- Vertex buffer and vertex layout. Through vertex layout, we can customize and expand the input of vertex buffer to implement customized Shader code.
- Geometry preprocessing. Vertex buffers, mesh resources, material parameters, etc. can all be preprocessed before actual rendering.



The relationship between vertex factories in the rendering hierarchy. As can be seen from the figure, the vertex factory is an object of the rendering thread, spanning both the CPU and the GPU.

FVertexFactory encapsulates vertex data resources that can be linked to vertex shaders. It and related types are defined as follows:

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

//Vertex element.
structFVertexElement
{
    uint8    StreamIndex;            //  Stream Index
    uint8    Offset;                 // Offset
    TEnumAsByte<EVertexElementType> Type;//type uint8
             AttributeIndex;//        Attribute Index
    uint16    Stride;                //Step Length
    //Whether the instance index or vertex index is instanced, if0,The element is repeated for each
    instance. uint16 bUseInstanceIndex;

    FVertexElement();
    FVertexElement(uint8 InStreamIndex, ...);
```

```
        voidoperator=(constFVertexElement& Other);
        friend FArchive& operator<<(FArchive& Ar,FVertexElement& Element);

        FStringToString()const;
        void FromString(constFString& Src); void FromString(
        constFStringView& Src);
};

//The type of the list of vertex declaration elements.
typedefTArray<FVertexElement,TFixedAllocator<MaxVertexElementCount>                    >
FVertexDeclarationElementList;


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

//Vertex declarationRHIresource
classFRHIVertexDeclaration: public FRHIResource {

public:
        virtualbool GetInitializer(FVertexDeclarationElementList& Init) {returnfalse; }
};

//Vertex Buffer
classFRHIVertexBuffer: public FRHIResource {

public:
        FRHIVertexBuffer(uint32 InSize,uint32 InUsage);

        uint32    GetSize()      const;
        uint32    GetUsage()     const;

protected:
        FRHIVertexBuffer();

        void   Swap(FRHIVertexBuffer& Other);
        void   ReleaseUnderlyingResource();

private:
        //size.
        uint32 Size;
        //Buffer Marker,  likeBUF_UnorderedAccess
        uint32 Usage;
};


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

//Vertex input stream.
struct     FVertexInputStream
{
        //    Vertex Stream Index
        uint32   StreamIndex   :4;   //exist
        VertexBufferThe offset of. uint32
        Offset    :28;    //Vertex    Buffer
        FRHIVertexBuffer* VertexBuffer;
```

```
        FVertexInputStream();
        FVertexInputStream(uint32 InStreamIndex, uint32 InOffset, FRHIVertexBuffer* InVertexBuffer);


        inlinebooloperator==(constFVertexInputStream& rhs)const; inlinebooloperator!=(
        constFVertexInputStream& rhs)const;
};

//Array of vertex input streams.
typedefTArray<FVertexInputStream, TInlineAllocator<4>> FVertexInputStreamArray;

//Vertex Stream Tags
enum classEVertexStreamUsage: uint8 {

        Default                  =0<<0,//default =1<<0,//
        Instancing               Instantiation =1<<1,//cover
        Overridden               =1<<2//Manual acquisition
        ManualFetch
};

//Vertex input stream type.
enum classEVertexInputStreamType: uint8 {

        Default =0,          //  default
        PositionOnly,        //  Only location
        PositionAndNormalOnly        //Only position and normal
};

//Vertex stream component.
structFVertexStreamComponent {

        //Vertex buffer for streaming data, ifnull,No data will be read from this
        vertex stream. constFVertexBuffer* VertexBuffer = nullptr;

        // vertex bufferThe offset of.
        uint32 StreamOffset =0;
        //The offset of the data, relative to the beginning of each element in the vertex
        buffer. uint8 Offset =0; //The step size of the data.

        uint8 Stride =0; //The type of
        data read from the stream.
        TEnumAsByte<EVertexElementType> Type = VET_None; //Vertex
        stream tag.
        EVertexStreamUsage VertexStreamUsage = EVertexStreamUsage::Default;

        (.....)
};

//Parameter binding interface for vertex factories used by shaders.
classFVertexFactoryShaderParameters {

public:
        //Bind parameters toParameterMap.The specific logic is completed by the subclass.
        void Bind(constclass FShaderParameterMap& ParameterMap) {}

        //Get the shader binding and vertex stream of the vertex factory. The specific logic is
        completed by the subclass. void GetElementShaderBindings(
                constclass FSceneInterface* Scene,
```

```
            constclass FSceneView* View, constclass
            FMeshMaterialShader* Shader, const
            EVertexInputStreamType InputStreamType,
            ERHIFeatureLevel::Type FeatureLevel,
            constclass FVertexFactory* VertexFactory, conststruct
            FMeshBatchElement& BatchElement, class
            FMeshDrawSingleShaderBindings& ShaderBindings,
            FVertexInputStreamArray& VertexStreams)const{}

        (......)
};

//Class used to represent vertex factory types.
classFVertexFactoryType {

public:
        //Type Definition
        typedefFVertexFactoryShaderParameters* ShaderFreq(u*eCnocnys,tcrouncsttPcalarassm etersType)(EShaderFrequency
FShaderParameterMap& ParameterMap);
        typedefconstFTypeLayoutDesc* (*GetParameterTypeLayoutType)(EShaderFrequency ShaderFrequency);

        (......)

        //Get the number of vertex factory types.
        staticint32GetNumVertexFactoryTypes();

        //Get the global list of shader factories.
        staticRENDERCORE_API TLinkedList<FVertexFactoryType*>*&GetTypeList(); //Get a list of stored
        material types.
        staticRENDERCORE_APIconstTArray<FVertexFactoryType*>&GetSortedMaterialTypes(); //Search by name
        FVertexFactoryType
        staticRENDERCORE_API FVertexFactoryType*GetVFByName(constFHashedName& VFName);

        //initializationFVertexFactoryTypeStatic members must beVFCalled before a type is
        created. static void Initialize(constTMap<FString, TArray<constTCHAR*> >&
        ShaderFileToUniformBufferVariables);
        static void Uninitialize();

        //Constructor/destructor.
        RENDERCORE_APIFVertexFactoryType(...); virtual
        ~FVertexFactoryType();

        //Data acquisition interface.
        constTCHAR*GetName()const; FName
        GetFName()const;
        constFHashedName&GetHashedName()const; const
        TCHAR*GetShaderFilename()const;

        //Shader parameter interface.
        FVertexFactoryShaderParameters*CreateShaderParameters(...) const                const;
        FTypeLayoutDesc*GetShaderParameterLayout(...)const; void
        GetShaderParameterElementShaderBindings(...)const;

        //Tag access.
        bool   IsUsedWithMaterials()const;
        bool   SupportsStaticLighting()             const;
        bool   SupportsDynamicLighting()          const;
        bool SupportsPrecisePrevWorldPos()          const;
```

```
        bool   SupportsPositionOnly()const;
        bool   SupportsCachingMeshDrawCommands()        const;
        bool   SupportsPrimitiveIdStream()const;

        //Get the hash.
        friend uint32GetTypeHash(constFVertexFactoryType* Type); //Source code based on
        vertex factory type and contains calculated hashes.
        constFSHAHash&GetSourceHash(EShaderPlatform ShaderPlatform)const; //Whether to
        cache the shader type of the material.
        bool ShouldCache(constFVertexFactoryShaderPermutationParameters& Parameters)const;

        void ModifyCompilationEnvironment(...);
        void ValidateCompiledResult(EShaderPlatform Platform, ...);

        bool SupportsTessellationShaders()const;

        //Add citationsUniform     BufferInclude.
        void   AddReferencedUniformBufferIncludes(...);
        void FlushShaderFileCache(...);
        constTMap<constTCHAR*, FCachedUniformBufferDeclaration>&
GetReferencedUniformBufferStructsCache()const;

private:
        static      uint32 NumVertexFactories;
        static      boolbInitializedSerializationHistory;

        //Various data and tags of vertex factory types.
        const TCHAR* Name;
        const TCHAR* ShaderFilename;
        FName TypeName;
        FHashedName HashedName;
        uint32 bUsedWithMaterials :1; uint32 bSupportsStaticLighting:1; uint32 bSupportsDynamicLighting:1; uint32
        bSupportsPrecisePrevWorldPos:1; uint32 bSupportsPositionOnly :1; uint32
        bSupportsCachingMeshDrawCommands:1; uint32 bSupportsPrimitiveIdStream:1; ConstructParametersType
        ConstructParameters; GetParameterTypeLayoutType GetParameterTypeLayout;
        GetParameterTypeElementShaderBindingsType GetParameterTypeElementShaderBindings;
        ShouldCacheType ShouldCacheRef;




        ModifyCompilationEnvironmentType ModifyCompilationEnvironmentRef;
        ValidateCompiledResultType ValidateCompiledResultRef;
        SupportsTessellationShadersType SupportsTessellationShadersRef;

        //List of global vertex factory types.
        TLinkedList<FVertexFactoryType*> //Cache GlobalListLink;
        referencesUniform          BufferContains.
        TMap<constTCHAR*,       FCachedUniformBufferDeclaration>
ReferencedUniformBufferStructsCache;
        //trackReferencedUniformBufferStructsCacheWhich platforms' declarations are
        cached. boolbCachedUniformBufferStructDeclarations;
};


//------Vertex Factory Tool Macros ------
```

```
//Implementing Vertex Factory Parameter Types
#defineIMPLEMENT_VERTEX_FACTORY_PARAMETER_TYPE(FactoryClass,          ShaderFrequency,
  ParameterClass)

//Vertex Factory Type Declaration
#defineDECLARE_VERTEX_FACTORY_TYPE(FactoryClass) //Vertex factory
type implementation
#define
IMPLEMENT_VERTEX_FACTORY_TYPE(FactoryClass,ShaderFilename,bUsedWithMaterials,bSupportsStat
icLighting,bSupportsDynamicLighting,bPrecisePrevWorldPos,bSupportsPositionOnly)
//Vertex factory virtual function table implementation
#defineIMPLEMENT_VERTEX_FACTORY_VTABLE(FactoryClass



//Vertex Factory
classFVertexFactory: public FRenderResource {

public:
    FVertexFactory(ERHIFeatureLevel::Type InFeatureLevel);

    virtual FVertexFactoryType*GetType()const;

    //Get vertex data stream.
    void GetStreams(ERHIFeatureLevel::Type InFeatureLevel, EVertexInputStreamType VertexStreamType,
FVertexInputStreamArray& OutVertexStreams)const
    {
        //DefaultVertex Stream Type
        if(VertexStreamType == EVertexInputStreamType::Default) {

            boolbSupportsVertexFetch = SupportsManualVertexFetch(InFeatureLevel);

            //Construct the vertex factory data intoFVertexInputStreamand added to the output list
            for(int32 StreamIndex =0;StreamIndex < Streams.Num();StreamIndex++) {

                constFVertexStream& Stream = Streams[StreamIndex];

                if(!(EnumHasAnyFlags(EVertexStreamUsage::ManualFetch,
Stream.VertexStreamUsage) && bSupportsVertexFetch))
                {
                    if(!Stream.VertexBuffer) {

                        OutVertexStreams.Add(FVertexInputStream(StreamIndex,          0,    nullptr));
                    }
                    else
                    {
                        if    (EnumHasAnyFlags(EVertexStreamUsage::Overridden, && !
Stream.VertexStreamUsage)        Stream.VertexBuffer->IsInitialized())
                        {
                            OutVertexStreams.Add(FVertexInputStream(StreamIndex,          0,
nullptr));
                        }
                        else
                        {
                            OutVertexStreams.Add(FVertexInputStream(StreamIndex,
Stream.Offset, Stream.VertexBuffer->VertexBufferRHI));
                        }
                    }
                }
```

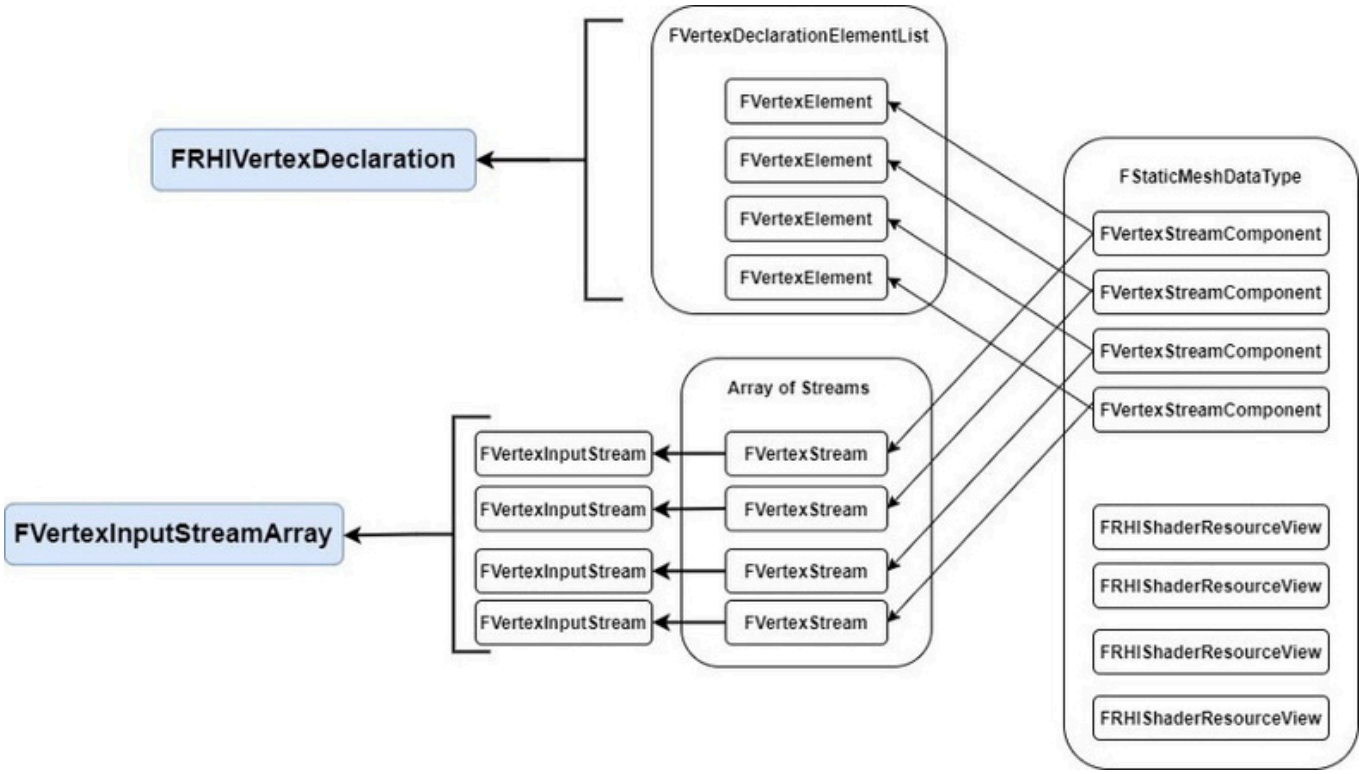
```
            }
        }
        //Vertex stream type with only positions and
        else if(VertexStreamType == EVertexInputStreamType::PositionOnly) {


            // Set the predefined vertex streams.
            for(int32 StreamIndex =0; StreamIndex < PositionStream.Num(); StreamIndex++) {

                constFVertexStream& Stream = PositionStream[StreamIndex];
                OutVertexStreams.Add(FVertexInputStream(StreamIndex, Stream.Offset,
Stream.VertexBuffer->VertexBufferRHI));
            }
        }
        //Vertex stream type with only position and normal
        else if(VertexStreamType == EVertexInputStreamType::PositionAndNormalOnly) {


            // Set the predefined vertex streams.
            for(int32 StreamIndex =0; StreamIndex < PositionAndNormalStream.Num();
StreamIndex++)
            {
                constFVertexStream& Stream = PositionAndNormalStream[StreamIndex];
                OutVertexStreams.Add(FVertexInputStream(StreamIndex, Stream.Offset,
Stream.VertexBuffer->VertexBufferRHI));
            }
        }
        else
        {
            // NOT_IMPLEMENTED
        }
    }

    //The data stream for the offset instance.
    voidOffsetInstanceStreams(uint32 InstanceOffset, EVertexInputStreamType VertexStreamType,
FVertexInputStreamArray& VertexStreams)const;

    static    void   ModifyCompilationEnvironment(...);
    static    void   ValidateCompiledResult(...);

    static bool SupportsTessellationShaders();

    //FRenderResourceInterface, ReleaseRHI
    resource. virtualvoid ReleaseRHI();

    //Set/get vertex declarationRHIReferences.
    FVertexDeclarationRHIRef&GetDeclaration(); void
    SetDeclaration(FVertexDeclarationRHIRef&                    NewDeclaration);

    //Get the vertex declaration by typeRHIReferences.
    constFVertexDeclarationRHIRef&GetDeclaration(EVertexInputStreamType InputStreamType) const

    {
        switch    (InputStreamType)
        {
        case   EVertexInputStreamType::Default:                    return    Declaration;
        case   EVertexInputStreamType::PositionOnly:                  return  PositionDeclaration;
        case   EVertexInputStreamType::PositionAndNormalOnly:         return
PositionAndNormalDeclaration;
        }
```

```
        returnDeclaration;
    }

    //Various markings.
    virtualbool            IsGPUSkinned()const;          virtualbool
    SupportsPositionOnlyStream()const;                   virtualbool
    SupportsPositionAndNormalOnlyStream()const;  virtualbool
    SupportsNullPixelShader()const;

    //Render primitives as camera-oriented sprites.
    virtualbool RendersPrimitivesAsCameraFacingSprites()const;

    //Whether vertex declarations are required.
    bool NeedsDeclaration() //Whether to  const;
    support manual vertex acquisition.
    inlinebool SupportsManualVertexFetch(constFStaticFeatureLevel InFeatureLevel)const; //Get the index based on the
    stream type.
    inlineint32GetPrimitiveIdStreamIndex(EVertexInputStreamType InputStreamType)const;

protected:
    inlinevoid SetPrimitiveIdStreamIndex(EVertexInputStreamType InputStreamType, int32 StreamIndex)

    {
        PrimitiveIdStreamIndex[static_cast<uint8>(InputStreamType)] = StreamIndex;
    }

    //Create vertex elements for a vertex stream component.
    FVertexElementAccessStreamComponent(constFVertexStreamComponent& Component,uint8 AttributeIndex);

    FVertexElementAccessStreamComponent(constFVertexStreamComponent& Component, uint8 AttributeIndex,
EVertexInputStreamType InputStreamType);
    //Initialize vertex declarations.
    void InitDeclaration(constFVertexDeclarationElementList& Elements,
EVertexInputStreamType StreamType = EVertexInputStreamType::Default)
    {
        if(StreamType == EVertexInputStreamType::PositionOnly) {

            PositionDeclaration =
PipelineStateCache::GetOrCreateVertexDeclaration(Elements);
        }
        else if(StreamType == EVertexInputStreamType::PositionAndNormalOnly) {

            PositionAndNormalDeclaration =
PipelineStateCache::GetOrCreateVertexDeclaration(Elements);
        }
        else// (StreamType == EVertexInputStreamType::Default) {

            // Create the vertex declaration for rendering the factory normally. Declaration =
            PipelineStateCache::GetOrCreateVertexDeclaration(Elements);
        }
    }

    //Vertex stream, you need to set the information body of the vertex stream.
    structFVertexStream
    {
        constFVertexBuffer* VertexBuffer = nullptr; uint32 Offset =0
        ; uint16 Stride =0;
```

```
        EVertexStreamUsage VertexStreamUsage = EVertexStreamUsage::Default; uint8 Padding =
        0;

        friendbooloperator==(constFVertexStream& A,constFVertexStream& B); FVertexStream();

    };

    //Vertex stream used for rendering vertex factories.
    TArray<FVertexStream,TInlineAllocator<8> > Streams;

    // VF(Vertex factories) can explicitly set this tofalse, to avoid errors when no declaration is made. Mainly used for getting data directly from
the bufferVF(likeNiagara).
    boolbNeedsDeclaration =true; bool
    bSupportsManualVertexFetch=false; int8
    PrimitiveIdStreamIndex[3] = {-1,-1,-1};

private:
    //A vertex stream with only positions, used for rendering depthPassThe
    Vertex Factory. TArray<FVertexStream,TInlineAllocator<2> > //Vertex          PositionStream;
    stream with just positions and normals.
    TArray<FVertexStream, TInlineAllocator<3>> PositionAndNormalStream;

    //For general rendering vertex factoriesRHIVertex
    declaration. FVertexDeclarationRHIRef Declaration;

    //PositionStreamandPositionAndNormalStreamCorrespondingRHI
    resource. FVertexDeclarationRHIRPeof sFiVtieornteDxeDcelacrlatriaotnio;nRHIRef
                                        PositionAndNormalDeclaration;
};
```

Many types of Vertex Factory are shown above, and there are several core classes, such as FVertexFactory, FVertexElement, FRHIVertexDeclaration, FRHIVertexBuffer, FVertexFactoryType, FVertexStreamComponent, FVertexInputStream, FVertexFactoryShaderParameters, etc. So what is the relationship between them?

To better illustrate the relationship between them, take the FStaticMeshDataType of the static model as an example:



FStaticMeshDataType will contain several FVertexStreamComponent instances, each of which contains an FVertexElement instance index in the FVertexDeclarationElementList and an FVertexStream instance index in the FVertexInputStreamArray list.

In addition, FVertexFactory is a base class, and the built-in subclasses are mainly:

- FGeometryCacheVertexVertexFactory: Vertex factory for geometry cache vertices, commonly used for pre-generated mesh types such as cloth and action.
- FGPUBaseSkinVertexFactory: The parent class of GPU skinned skeletal meshes. Its subclasses are:
    - TGPUSkinVertexFactory: Vertex factory for GPU skinning that can specify bone weights.
- FLocalVertexFactory: Local vertex factory, commonly used in static meshes, it has a large number of subclasses:
    - FInstancedStaticMeshVertexFactory: Instanced static mesh vertex factory.
    - FSplineMeshVertexFactory: Spline mesh vertex factory. FGeometryCollectionVertexFactory:
    - Geometry collection vertex factory. FGPUSkinPassthroughVertexFactory: Skinned bone vertex
    - factory with Skin Cache mode enabled.
    - FSingleTriangleMeshVertexFactory: Vertex factory for a single triangle mesh, used for volumetric cloud rendering. . . . . .
    -
- FParticleVertexFactoryBase: Vertex factory base class for particle rendering.
- FLandscapeVertexFactory: Vertex factory for rendering terrain.

In addition to the above types inherited from FVertexFactory, there are some types that do not inherit from FVertexFactory, such as:

- FGPUBaseSkinAPEXClothVertexFactory: cloth vertex factory.
    - TGPUSkinAPEXClothVertexFactory: Cloth vertex factory with bone weight mode.

In addition to FVertexFactory, other corresponding core classes also have inheritance systems. For example, the subclasses of FVertexFactoryShaderParameters are:

- FGeometryCacheVertexFactoryShaderParameters
- FGPUSkinVertexFactoryShaderParameters
- FMeshParticleVertexFactoryShaderParameters
- FParticleSpriteVertexFactoryShaderParameters
- FGPUSpriteVertexFactoryShaderParametersVS
- FGPUSpriteVertexFactoryShaderParametersPS
- FSplineMeshVertexFactoryShaderParameters
- FLocalVertexFactoryShaderParametersBase
- FLandscapeVertexFactoryVertexShaderParameters
- FLandscapeVertexFactoryPixelShaderParameters . . . . . .
-

In addition, some vertex factories will also derive the FStaticMeshDataType type internally to reuse static mesh related data members.

To better illustrate how to use the vertex factory, let's take the most common FLocalVertexFactory and CableComponent that uses FLocalVertexFactory as examples:

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

classENGINE_API FLocalVertexFactory: public FVertexFactory {

public:
	FLocalVertexFactory(ERHIFeatureLevel::Type InFeatureLevel,const char* InDebugName);

	//Derived fromFStaticMeshDataTypeThe data type of the .
	structFDataType: public FStaticMeshDataType {

		FRHIShaderResourceView* PreSkinPositionComponentSRV = nullptr;
	};

	//Environment variable changes and verification.
	static bool ShouldCompilePermutation(constFVertexFactoryShaderPermutationParameters& Parameters);

	static void ModifyCompilationEnvironment(const FVertexFactoryShaderPermutationParameters&
Parameters, FShaderCompilerEnvironment& OutEnvironment);

	static void ValidateCompiledResult(constFVertexFactoryType* Type, EShaderPlatform Platform,const
FShaderParameterMap& ParameterMap, TArray<FString>& OutErrors);

	//Depend onTSynchronizedResourceData updated from the game
	thread. void SetData(constFDataType& InData);
```

```
        //Copies data from other vertex factories.
        void Copy(constFLocalVertexFactory& Other);

        //FRenderResourceinterface.
        virtualvoid InitRHI() override; virtualvoid
        ReleaseRHI() override {

            UniformBuffer.SafeRelease();
            FVertexFactory::ReleaseRHI();
        }

        //Vertex color interface.
        void SetColorOverrideStream(FRHICommandList& RHICmdList,constFVertexBuffer* ColorVertexBuffer)
const;
        void GetColorOverrideStream(constFVertexBuffer* ColorVertexBuffer,
FVertexInputStreamArray& VertexStreams)const;

        //Interface for shader parameters and other data.
        inlineFRHIShaderResourceView*GetPositionsSRV()const; inline
        FRHIShaderResourceView*GetPreSkinPositionSRV()const; inline
        FRHIShaderResourceView*GetTangentsSRV()const; inlineFRHIShaderResourceView*
        GetTextureCoordinatesSRV()const; inlineFRHIShaderResourceView*
        GetColorComponentsSRV()const; inlineconstuint32GetColorIndexMask()const; inline
        const int GetLightMapCoordinateIndex()const; inlineconst int GetNumTexcoords()const
        ; FRHIUniformBuffer*GetUniformBuffer()const;




        (.....)

protected:
        //Data passed in from the game thread.      FDataTypeyesFStaticMeshDataTypeSubclass of .
        FDataType Data;
        //Shader parameters for local vertex factories.
        TUniformBufferRef<FLocalVertexFactoryUniformShaderParameters> //Vertex color UniformBuffer;
        stream index.
        int32 ColorStreamIndex;


        (.....)
};

// Engine\Source\Runtime\Engine\Public\LocalVertexFactory.cpp

void FLocalVertexFactory::InitRHI() {

        //Whether to useGPUScenario.
        const boolbCanUseGPUScene = UseGPUScene(GMaxRHIShaderPlatform, GMaxRHIFeatureLevel);

        //    Initializes position streams and position declarations.
        if    (Data.PositionComponent.VertexBuffer                !=
Data.TangentBasisComponents[0].VertexBuffer)
        {
            //Add vertex declaration.
            autoAddDeclaration = [this, bCanUseGPUScene](EVertexInputStreamType InputStreamType,
boolbAddNormal)
            {
                //Vertex stream element.
                FVertexDeclarationElementList StreamElements;
```

```
                StreamElements.Add(AccessStreamComponent(Data.PositionComponent,0,
InputStreamType));

                bAddNormal = bAddNormal && Data.TangentBasisComponents[1].VertexBuffer !=
NULL;
                if(bAddNormal)
                {
                    StreamElements.Add(AccessStreamComponent(Data.TangentBasisComponents[1],
2, InputStreamType));
                }

                constuint8 TypeIndex = static_cast<uint8>(InputStreamType);
                PrimitiveIdStreamIndex[TypeIndex] =-1;
                if(GetType()->SupportsPrimitiveIdStream() && bCanUseGPUScene) {

                    // When the VF is used for rendering in normal mesh passes, this vertex
buffer and offset will be overridden

StreamElements.Add(AccessStreamComponent(FVertexStreamComponent(&GPrimitiveIdDummy,0,0, sizeof(uint32),
VET_UInt, EVertexStreamUsage::Instancing),1, InputStreamType));
                    PrimitiveIdStreamIndex[TypeIndex] = StreamElements.Last().StreamIndex;
                }

                //Initialization statement.
                InitDeclaration(StreamElements, InputStreamType);
            };

            //IncreasePositionOnlyandPositionAndNormalOnlyTwo vertex declarations, the former does not require normals.
            AddDeclaration(EVertexInputStreamType::PositionOnly,                false);
            AddDeclaration(EVertexInputStreamType::PositionAndNormalOnly,                    true);
        }

        //List of vertex declaration elements.
        FVertexDeclarationElementList Elements;

        //Vertex Position
        if(Data.PositionComponent.VertexBuffer !=NULL) {

            Elements.Add(AccessStreamComponent(Data.PositionComponent,0));
        }

        //Graphicsid
        {
            constuint8 Index = static_cast<uint8>(EVertexInputStreamType::Default);
            PrimitiveIdStreamIndex[Index] =-1;
            if(GetType()->SupportsPrimitiveIdStream() && bCanUseGPUScene) {

                // When the VF is used for rendering in normal mesh passes, this vertex buffer
and offset will be overridden
                Elements.Add(AccessStreamComponent(FVertexStreamComponent(&GPrimitiveIdDummy,
0,0,sizeof(uint32), VET_UInt, EVertexStreamUsage::Instancing),13));
                PrimitiveIdStreamIndex[Index] = Elements.Last().StreamIndex;
            }
        }

        //Tangents and normals, tangent normals are used by vertex streams,
        binormals areshadergenerate. uint8 TangentBasisAttributes[2] = {1,2};
        for(int32 AxisIndex =0;AxisIndex <2;AxisIndex++)
```

```
	{
		if(Data.TangentBasisComponents[AxisIndex].VertexBuffer {		!=	NULL)

Elements.Add(AccessStreamComponent(Data.TangentBasisComponents[AxisIndex],TangentBasisAttr ibutes[AxisIndex]));

		}
	}

	if(Data.ColorComponentsSRV == nullptr) {

		Data.ColorComponentsSRV = GNUllColorVertexBuffer.VertexBufferSRV;
		Data.ColorIndexMask =0;
	}

	//Vertex Color
	ColorStreamIndex =-1;
	if(Data.ColorComponent.VertexBuffer) {

		Elements.Add(AccessStreamComponent(Data.ColorComponent,3));
		ColorStreamIndex = Elements.Last().StreamIndex;
	}
	else
	{
		FVertexStreamComponent NullColorComponent(&GNullColorVertexBuffer,0,0, VET_Color,
		EVertexStreamUsage::ManualFetch);
		Elements.Add(AccessStreamComponent(NullColorComponent,3));
		ColorStreamIndex = Elements.Last().StreamIndex;
	}

	//Texture Coordinates
	if(Data.TextureCoordinates.Num()) {

		constint32 BaseTexCoordAttribute =4;
		for(int32 CoordinateIndex =0;CoordinateIndex <
Data.TextureCoordinates.Num();CoordinateIndex++)
		{
			Elements.Add(AccessStreamComponent(
				Data.TextureCoordinates[CoordinateIndex],
				BaseTexCoordAttribute + CoordinateIndex ) ) ;

		}

		for(int32 CoordinateIndex = Data.TextureCoordinates.Num(); CoordinateIndex <
MAX_STATIC_TEXCOORDS /2; CoordinateIndex++)
		{
			Elements.Add(AccessStreamComponent(
				Data.TextureCoordinates[Data.TextureCoordinates.Num() -1],
				BaseTexCoordAttribute + CoordinateIndex
				) ) ;
		}
	}

	//Light Map
	if(Data.LightMapCoordinateComponent.VertexBuffer) {

		Elements.Add(AccessStreamComponent(Data.LightMapCoordinateComponent,15));
	}
```

```
        else if(Data.TextureCoordinates.Num()) {

            Elements.Add(AccessStreamComponent(Data.TextureCoordinates[0],15));
        }

        //Initialize Vertex Declaration
        InitDeclaration(Elements);

        constint32 DefaultBaseVertexIndex =0; constint32
        DefaultPreSkinBaseVertexIndex =0;
        if(RHISupportsManualVertexFetch(GMaxRHIShaderPlatform) || bCanUseGPUScene) {

            SCOPED_LOADTIMER(FLocalVertexFactory_InitRHI_CreateLocalVFUniformBuffer); UniformBuffer =
            CreateLocalVFUniformBuffer(this, Data.LODLightmapDataIndex, nullptr, DefaultBaseVertexIndex,
            DefaultPreSkinBaseVertexIndex);
        }
}

//accomplishFLocalVertexFactoryThe parameter type of .
IMPLEMENT_VERTEX_FACTORY_PARAMETER_TYPE(FLocalVertexFactory,                SF_Vertex,
FLocalVertexFactoryShaderParameters);

//accomplishFLocalVertexFactory.
IMPLEMENT_VERTEX_FACTORY_TYPE_EX(FLocalVertexFactory,"/Engine/Private/LocalVertexFactory.u sh",true,true,true,true,
true,true,true);
```

Let's go into the CableComponent related types about the use of FLocalVertexFactory:

```
// Engine\Plugins\Runtime\CableComponent\Source\CableComponent\Private\CableComponent.cpp

classFCableSceneProxyfinal: public FPrimitiveSceneProxy {

public:
        FCableSceneProxy(UCableComponent*Component)
            : FPrimitiveSceneProxy(Component),
            Material(NULL) //Constructs a vertex factory.

            , VertexFactory(GetScene().GetFeatureLevel(),"FCableSceneProxy") (......)

        {
            //Initialize the buffer using the vertex factory.
            VertexBuffers.InitWithDummyData(&VertexFactory, GetRequiredVertexCount()); (......)

        }

        virtual ~FCableSceneProxy() {

            //Releases the vertex factory.
            VertexFactory.ReleaseResource(); (......)

        }

        //BuildCableGrid.
        void BuildCableMesh(constTArray<FVector>& InPoints, TArray<FDynamicMeshVertex>& OutVertices,
TArray<int32>& OutIndices)
        {
            (......)
```

```
    }

    //Set dynamic data (rendering thread call)
    void SetDynamicData_RenderThread(FCableDynamicData* {            NewDynamicData)

        //Release old data.
        if(DynamicData)
        {
            delete DynamicData;
            DynamicData =NULL;
        }
        DynamicData = NewDynamicData;

        //fromCablePoints construct vertices.
        TArray<FDynamicMeshVertex>          Vertices;
        TArray<int32> Indices;
        BuildCableMesh(NewDynamicData->CablePoints,             Vertices,       Indices);

        //Fill vertex buffer data.
        for(inti =0; i < Vertices.Num(); i++) {

            constFDynamicMeshVertex& Vertex = Vertices[i];

            VertexBuffers.PositionVertexBuffer.VertexPosition(i) = Vertex.Position;
            VertexBuffers.StaticMeshVertexBuffer.SetVertexTangents(i,
Vertex.TangentX.ToFVector(), Vertex.GetTangentY(), Vertex.TangentZ.ToFVector());
            VertexBuffers.StaticMeshVertexBuffer.SetVertexUV(i,0,
Vertex.TextureCoordinate[0]);
            VertexBuffers.ColorVertexBuffer.VertexColor(i) = Vertex.Color;
        }

        //Update vertex buffer data toRHI.
        {
            auto& VertexBuffer = VertexBuffers.PositionVertexBuffer;
            void*VertexBufferData = RHILockVertexBuffer(VertexBuffer.VertexBufferRHI,0,
VertexBuffer.GetNumVertices() * VertexBuffer.GetStride(), RLM_WriteOnly);
            FMemory::Memcpy(VertexBufferData, VertexBuffer.GetVertexData(),
VertexBuffer.GetNumVertices() * VertexBuffer.GetStride());
            RHIUnlockVertexBuffer(VertexBuffer.VertexBufferRHI);
        }

        (……)
    }

    virtualvoid GetDynamicMeshElements(constTArray<constFSceneView*>& Views,const FSceneViewFamily&
ViewFamily, uint32 VisibilityMap, FMeshElementCollector& Collector) const
        override
    {
        (……)

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

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

                constFSceneView* View = Views[ViewIndex];

                //structureFMeshBatchExamples.
                FMeshBatch& Mesh = Collector.AllocateMesh();
```

```
                    //Pass the vertex factory instance toFMeshBatch
                    Examples. Mesh.VertexFactory = &VertexFactory;

                    (......)

                    Collector.AddMesh(ViewIndex, Mesh);
                }
            }
        }

        (......)

private:
        //Material
        UMaterialInterface*          Material;
        //Vertex and index buffers.
        FStaticMeshVertexBuffers
                                     VertexBuffers;
        FCableIndexBuffer IndexBuffer; //Vertex
        Factory.
        FLocalVertexFactory          VertexFactory;
        //Dynamic data.
        FCableDynamicData*           DynamicData;

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

As can be seen from the above code, the steps of using the**existing**vertex factory are not complicated, mainly including initialization, assignment, and passing to the FMeshBatch instance.

However, whether using an existing or custom vertex factory, the vertex declaration order, type, number of components and slots of the vertex factory must be consistent with the FVertexFactoryInput of the HLSL layer. For example, the vertex declaration order of FLocalVertexFactory::InitRHI is position, tangent, color, texture coordinates, and light map. Then we enter the HLSL file corresponding to FLocalVertexFactory (specified by macros such as IMPLEMENT_VERTEX_FACTORY_TYPE) and take a look:

```
// Engine\Shaders\Private\LocalVertexFactory.ush

//Input structure corresponding to the local vertex factory.
struct     FVertexFactoryInput
{
        //    Location
        float4          Position          :   ATTRIBUTE0;

        //Tangents and Colors
#if     !MANUAL_VERTEX_FETCH
        #if METAL_PROFILE
                float3 TangentX                  :   ATTRIBUTE1;
                // TangentZ.w contains sign of tangent basis determinant float4 TangentZ :
                ATTRIBUTE2;

                float4          Color                : ATTRIBUTE3;
        #else
                half3          TangentX          :   ATTRIBUTE1;
```

```
            // TangentZ.w contains sign of tangent basis determinant half4 TangentZ :
            ATTRIBUTE2;

            half4           Color                   : ATTRIBUTE3;
        #endif
    #endif

        //Texture Coordinates
    #if    NUM_MATERIAL_TEXCOORDS_VERTEX
        #if !MANUAL_VERTEX_FETCH
            #if    GPUSKIN_PASS_THROUGH
                // These must match GPUSkinVertexFactory.usf float2
                TexCoords[NUM_MATERIAL_TEXCOORDS_VERTEX]                    : ATTRIBUTE4;
                #ifNUM_MATERIAL_TEXCOORDS_VERTEX > 4
                    #errorToo many texture coordinate sets defined on GPUSkin vertex input.
    Max:   4.
                #endif
            #else
                #ifNUM_MATERIAL_TEXCOORDS_VERTEX > 1
                    float4 PackedTexCoords4[NUM_MATERIAL_TEXCOORDS_VERTEX/2] : ATTRIBUTE4;
                #endif
                #ifNUM_MATERIAL_TEXCOORDS_VERTEX == 1 float2
                    PackedTexCoords2 : ATTRIBUTE4;
                #elifNUM_MATERIAL_TEXCOORDS_VERTEX == 3 float2
                    PackedTexCoords2 : ATTRIBUTE5;
                #elifNUM_MATERIAL_TEXCOORDS_VERTEX == 5 float2
                    PackedTexCoords2 : ATTRIBUTE6;
                #elifNUM_MATERIAL_TEXCOORDS_VERTEX == 7 float2
                    PackedTexCoords2 : ATTRIBUTE7;
                #endif
            #endif
        #endif
    #elifUSE_PARTICLE_SUBUVS && !MANUAL_VERTEX_FETCH float2
        TexCoords[1] : ATTRIBUTE4;
    #endif

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

Therefore, it can be seen that the data order of the FVertexFactoryInput structure corresponds one to one with the vertex declaration of FLocalVertexFactory.

8.2.5 Shader Permutation

UE's Shader code is a sample of the Uber Shader design architecture, which requires adding many different macros to the same shader code file to distinguish between different passes, functions, feature levels, and quality levels. At the C++ level, in order to facilitate the expansion, setting of these macro definitions and different values, UE adopts the concept of**shader permutation**.

Each permutation contains a unique hash key value, and the values of this set of permutations are filled into HLSL to compile the corresponding shader code. The following is the definition of the core type of the shader permutation:

```
// Engine\Source\Runtime\RenderCore\Public\ShaderPermutation.h

// BoolShader arrangement
structFShaderPermutationBool {

	using Type =bool;

	//The number of dimensions.
	staticconstexpr int32 PermutationCount =2; //Whether it is a multi-
	dimensional arrangement.
	staticconstexprboolIsMultiDimensional =false; //Conversionbool
	arriveintvalue.
	staticint32ToDimensionValueId(Type E) {

		returnE ?1:0;
	}
	//Converts to the defined value.
	static bool ToDefineValue(Type E) {

		returnE;
	}
	//From the arrangementidConvertbool.
	staticTypeFromDimensionValueId(int32 PermutationId) {

		checkf(PermutationId ==0|| PermutationId ==1, TEXT("Invalid shader permutation dimension id %i."),
		PermutationId);
		returnPermutationId ==1;
	}
};

//Integer shader array
template <typename TType, int32 TDimensionSize, int32 TFirstValue=0> struct
TShaderPermutationInt {

	using Type = TType;
	staticconstexpr int32 PermutationCount = TDimensionSize; staticconstexpr
	boolIsMultiDimensional =false;

	//Maximum and minimum values.
	staticconstexpr Type MinValue = static_cast<Type>(TFirstValue);
	staticconstexpr Type MaxValue = static_cast<Type>(TFirstValue + TDimensionSize -1);

	staticint32ToDimensionValueId(Type E) staticint32
	ToDefineValue(Type E);
	staticTypeFromDimensionValueId(int32 PermutationId);
};

//Variable-dimensional integer shader array.
template <int32... Ts>
structTShaderPermutationSparseInt {

	using Type = int32;
	staticconstexpr int32 PermutationCount =0; staticconstexprbool
	IsMultiDimensional =false;

	staticint32ToDimensionValueId(Type E);
	staticTypeFromDimensionValueId(int32 PermutationId);
```

```
};

//Shader array domain, the number is variable
template <typename... Ts> struct
TShaderPermutationDomain {

	using Type = TShaderPermutationDomain<Ts...>;

	staticconstexprboolIsMultiDimensional =true; staticconstexpr
	int32 PermutationCount =1;

	//Constructor.
	TShaderPermutationDomain<Ts...>() {} explicit
	TShaderPermutationDomain<Ts...>(int32 {                    PermutationId)

		checkf(PermutationId ==0, TEXT("Invalid shader permutation id %i."), PermutationId);

	}

	//Set the value of a dimension.
	template<class DimensionToSet>
	void Set(typename DimensionToSet::Type) {

		static_assert(sizeof(typename DimensionToSet::Type) ==0,"Unknown shader permutation
dimension.");
	}
	//Get the value of a dimension.
	template<class DimensionToGet>
	consttypename DimensionToGet::TypeGet()const {


		static_assert(sizeof(typename DimensionToGet::Type) ==0,"Unknown shader permutation
dimension.");
		returnDimensionToGet::Type();
	}

	//Modify the compilation environment variables.
	void ModifyCompilationEnvironment(FShaderCompilerEnvironment& OutEnvironment)const{}

	//Data conversion.
	staticint32ToDimensionValueId(constType& PermutationVector) {

		return0;
	}
	int32ToDimensionValueId()const {

		returnToDimensionValueId(*this);
	}
	staticTypeFromDimensionValueId(constint32 PermutationId) {

		returnType(PermutationId);
	}

	booloperator==(constType & Other)const {

		returntrue;
	}
};
```

```
//The following macro is convenient for writingshaderofC++Implement and set up shader arrangements in code.

//Declare the specified nameboolType Shader Arrangement
#defineSHADER_PERMUTATION_BOOL(InDefineName) //Declare the
specified nameintType Shader Arrangement
#defineSHADER_PERMUTATION_INT(InDefineName, //Declare a              Count)
specified name and scopeintType Shader Arrangement
#defineSHADER_PERMUTATION_RANGE_INT(InDefineName, Start, Count) //Declare a
sparseintType Shader Arrangement
#defineSHADER_PERMUTATION_SPARSE_INT(InDefineName,...) //Declare a shader array of
the specified name enumeration type
#defineSHADER_PERMUTATION_ENUM_CLASS(InDefineName, EnumName)
```

Looking at the template and macro definition above, are you a little confused? It doesn't matter. Combining the use cases of FDeferredLightPS, you will find that the shader arrangement is actually very simple:

```
//Delayed light sourcePS.
classFDeferredLightPS: public FGlobalShader {

    DECLARE_SHADER_TYPE(FDeferredLightPS, Global)

    //Declare the shader arrangement of each dimension. Note that inheritance is used, and the parent class isSHADER_PERMUTATION_xxxThe type of definition. //
    Note that the parent class noun (such asLIGHT_SOURCE_SHAPE, USE_SOURCE_TEXTURE, USE_IES_PROFILE, ...)It is HLSLThe name of the macro in the code.

    classFSourceShapeDim                : SHADER_PERMUTATION_ENUM_CLASS("LIGHT_SOURCE_SHAPE",
ELightSourceShape);
    class    FSourceTextureDim              : SHADER_PERMUTATION_BOOL("USE_SOURCE_TEXTURE"
    class    FIESProfileDim             : SHADER_PERMUTATION_BOOL("USE_IES_PROFILE"
    class    FInverseSquaredDim         :   SHADER_PERMUTATION_BOOL("INVERSE_SQUARED_FALLOFF"); :
    class    FVisualizeCullingDim           SHADER_PERMUTATION_BOOL("VISUALIZE_LIGHT_CULLING"
    class    FLightingChannelsDim       : SHADER_PERMUTATION_BOOL("USE_LIGHTING_CHANNELS"
    class    FTransmissionDim           : SHADER_PERMUTATION_BOOL("USE_TRANSMISSION");
    class    FHairLighting              : SHADER_PERMUTATION_INT("USE_HAIR_LIGHTING",          2);
    class    FAtmosphereTransmittance   :
SHADER_PERMUTATION_BOOL("USE_ATMOSPHERE_TRANSMITTANCE");
    class    FCloudTransmittance            : SHADER_PERMUTATION_BOOL("USE_CLOUD_TRANSMITTANCE");
    class    FAnistropicMaterials            :
SHADER_PERMUTATION_BOOL("SUPPORTS_ANISOTROPIC_MATERIALS");

    //Declare the shader array domain, containing all dimensions defined above.
    using FPermutationDomain = TShaderPermutationDomain<
        FSourceShapeDim,
        FSourceTextureDim,
        FIESProfileDim,
        FInverseSquaredDim,
        FVisualizeCullingDim,
        FLightingChannelsDim,
        FTransmissionDim,
        FHairLighting,
        FAtmosphereTransmittance,
        FCloudTransmittance,
        FAnistropicMaterials>;

    //Whether to compile the specified shader array.
```

```
	static bool ShouldCompilePermutation(constFGlobalShaderPermutationParameters& Parameters)

	{
		//Get the value of a shader array.
		FPermutationDomainPermutationVector(Parameters.PermutationId);

		//If it is parallel light, thenIESLight and inverse attenuation will have no meaning and can be ignored.
		if(PermutationVector.Get< FSourceShapeDim >() == ELightSourceShape::Directional
&&(
			PermutationVector.Get< FIESProfileDim >() ||
			PermutationVector.Get< FInverseSquaredDim >() ) )
		{
			returnfalse;
		}

		//If it is not a parallel light, then the atmosphere and cloud body transmission will have no meaning and can be ignored.
		if(PermutationVector.Get< FSourceShapeDim >() != ELightSourceShape::Directional
&&(PermutationVector.Get<FAtmosphereTransmittance>() ||
PermutationVector.Get<FCloudTransmittance>()))
		{
			returnfalse;
		}

		(......)

		returnIsFeatureLevelSupported(Parameters.Platform, ERHIFeatureLevel::SM5);
	}

	(......)
};

//Rendering light sources.
void FDeferredShadingSceneRenderer::RenderLight(FRHICommandList& RHICmdList, ...) {

	(......)

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

		FViewInfo& View = Views[ViewIndex];

		(......)

		if(LightSceneInfo->Proxy->GetLightType() == LightType_Directional) {

			(......)

			//statementFDeferredLightPSAn example of a shader array.
			FDeferredLightPS::FPermutationDomain PermutationVector;

			//Fill the arrangement value according to the rendering state.
			PermutationVector.Set<		FDeferredLightPS::FSourceShapeDim		> (
ELightSourceShape::Directional );
			PermutationVector.Set< FDeferredLightPS::FIESProfileDim >(false); PermutationVector.Set<
			FDeferredLightPS::FInverseSquaredDim >(false); PermutationVector.Set<
			FDeferredLightPS::FVisualizeCullingDim >( View.Family-
> EngineShowFlags.VisualizeLightCulling );
			PermutationVector.Set< FDeferredLightPS::FLightingChannelsDim >(
View.bUsesLightingChannels );
```

```
                PermutationVector.Set< FDeferredLightPS::FAnistropicMaterials >
(ShouldRenderAnisotropyPass());
                PermutationVector.Set< FDeferredLightPS::FTransmissionDim >( bTransmission );
                PermutationVector.Set<          FDeferredLightPS::FHairLighting>(0);
                PermutationVector.Set<          FDeferredLightPS::FAtmosphereTransmittance >
(bAtmospherePerPixelTransmittance);
                PermutationVector.Set<          FDeferredLightPS::FCloudTransmittance > ||
(bLight0CloudPerPixelTransmittance              bLight1CloudPerPixelTransmittance);

                //Use the filled array from the viewShaderMapGet the correspondingPSExamples.
                TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );

                //fillingPSOther data of.
                GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GFilterVertexDeclaration.VertexDeclarationRHI;
                GraphicsPSOInit.BoundShaderState.VertexShaderRHI                =
VertexShader.GetVertexShader();
                GraphicsPSOInit.BoundShaderState.PixelShaderRHI                =
PixelShader.GetPixelShader();

                SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                >SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, LightingChannelsTexture, &RenderLightParams);

                  (......)
            }

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

From this, we can see that shader permutations are essentially just a set of key values with indefinite dimensions. During the shader compilation phase, the shader compiler will try to generate corresponding shader instance code for each different permutation. Of course, it can also exclude some meaningless permutations through ShouldCompilePermutation. All precompiled shaders are stored in the view's ShaderMap. The key values of each dimension can be dynamically generated at runtime, and then the permutation domains composed of them are used to obtain the corresponding compiled shader code from the view's ShaderMap , so as to perform subsequent shader data settings and rendering.

In addition, it is worth mentioning that the names of the parent classes of the dimension arrangement (such as LIGHT_SOURCE_SHAPE, USE_SOURCE_TEXTURE, USE_IES_PROFILE, ...) are the macro names in the HLSL code. For example, FSourceShapeDim controls the LIGHT_SOURCE_SHAPE of the HLSL code. Different fragments of code will be selected according to the value of FSourceShapeDim, thereby controlling different versions and branches of shader code.

8.3 Shader Mechanism

This chapter mainly some of the underlying mechanisms of Shader, such as the analysis storage mechanism of Shader Map, Shader compilation and caching strategies, etc.

8.3.1 Shader Map

ShaderMap stores compiled shader codes and is divided into three types: FGlobalShaderMap, FMaterialShaderMap, and FMeshMaterialShaderMap.

8.3.1.1 FShaderMapBase

This section first explains the basic types and concepts related to Shader Map, as follows:

```
// Engine\Source\Runtime\Core\Public\Serialization\MemoryImage.h

//Pointer table base class.
classFPointerTableBase {

public:
	virtual ~FPointerTableBase()		{}
	virtual int32AddIndexedPointer(constFTypeLayoutDesc& TypeDesc,void* Ptr) =0; virtualvoid*GetIndexedPointer(
	constFTypeLayoutDesc& TypeDesc, uint32 i)const=0;
};

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

//A dedicatedshaderClass. AFShaderTypeCan be managed across multiple dimensionsFShaderMultiple instances of
EShaderPlatform,orpermutation id. FShaderTypeThe number of permutations ofGetPermutationCount()Given.

classFShaderType
{
public:
	//Shader types, including global, material, mesh material,Niagara
	wait. enum classEShaderTypeForDynamicCast: uint32 {

		Global,
		Material,
		MeshMaterial,
		Niagara,
		OCIO,
		NumShaderTypes,
	};

	(.....)

	//Static data acquisition interface.
	staticTLinkedList<FShaderType*>*&GetTypeList(); staticFShaderType*
	GetShaderTypeByName(constTCHAR* Name);
	staticTArray<constFShaderType*>GetShaderTypesByFilename(constTCHAR* Filename); static
	TMap<FHashedName, FShaderType*>&GetNameToTypeMap();
	static constTArray<FShaderType*>&GetSortedTypes(EShaderTypeForDynamicCast Type);

	static void Initialize(constTMap<FString, TArray<constTCHAR*> >&
	ShaderFileToUniformBufferVariables);
	static void Uninitialize();

	//Constructor.
	FShaderType(...);
	virtual ~FShaderType();
```

```
        FShader*ConstructForDeserialization()const;
        FShader*ConstructCompiled(constFShader::CompiledShaderInitializerType& Initializer) const;


        bool  ShouldCompilePermutation(...)const;
        void  ModifyCompilationEnvironment(..)const;
        bool  ValidateCompiledResult(...)const;

        //based onshader typeThe source code and includes calculating hash values.
        constFSHAHash&GetSourceHash(EShaderPlatform ShaderPlatform)const; //GetFShaderType
        The hash value of the pointer.
        friend uint32GetTypeHash(FShaderType* Ref);

        //Access interface.
        (......)

        void AddReferencedUniformBufferIncludes(FShaderCompilerEnvironment& OutEnvironment, FString&
OutSourceFilePrefix, EShaderPlatform Platform);
        void FlushShaderFileCache(constTMap<FString, TArray<constTCHAR*> >&
        ShaderFileToUniformBufferVariables);
        void GetShaderStableKeyParts(structFStableShaderKeyAndValue& SaveKeyVal);

private:
        EShaderTypeForDynamicCast ShaderTypeForDynamicCast; const
        FTypeLayoutDesc* TypeLayout; //name.

        constTCHAR* Name;
        //Type name.
        FName TypeName;
        //Hash Name
        FHashedName HashedName; //The
        hashed source file name.
        FHashedName HashedSourceFilename; //The
        source file name.
        constTCHAR*     SourceFilename;
        //Entrance life.
        constTCHAR*
        //Coloring frequency.     FunctionName;

        uint32    Frequency;
        uint32    TypeSize;

        //The number of permutations.
        int32 TotalPermutationCount;

        (......)

        //Global list.
        TLinkedList<FShaderType*> GlobalListLink;

protected:
        boolbCachedUniformBufferStructDeclarations; //
        ReferencedUniform BufferContains cache.
        TMap<constTCHAR*, FCachedUniformBufferDeclaration>
        ReferencedUniformBufferStructsCache;
};

//Shader map pointer table
classFShaderMapPointerTable: public FPointerTableBase {
```

```
public:
    virtual int32AddIndexedPointer(constFTypeLayoutDesc& TypeDesc,void* Ptr) override; virtualvoid*
    GetIndexedPointer(constFTypeLayoutDesc& TypeDesc, uint32 i)const override;


    virtualvoid SaveToArchive(FArchive& Ar,void* FrozenContent,bool
bInlineShaderResources)const;
    virtualvoid LoadFromArchive(FArchive& Ar,void* FrozenContent,bool
bInlineShaderResources,boolbLoadedByCookedMaterial);

    //Shader Types
    TPtrTable<FShaderType>          ShaderTypes;
    //Vertex Factory Type
    TPtrTable<FVertexFactoryType>
                                    VFTypes;
};

//A shader pipeline instance containing compile-time state.
classFShaderPipeline
{
public:
    explicitFShaderPipeline(constFShaderPipelineType* InType); ~FShaderPipeline();


    //Add shaders.
    void AddShader(FShader* Shader, int32 PermutationId); //Get the
    number of shaders.
    inlineuint32GetNumShaders()const;

    //Findshader.
    template<typename ShaderType>
    ShaderType*GetShader(constFShaderMapPointerTable& InPtrTable); FShader*
    GetShader(EShaderFrequency Frequency);
    constFShader*GetShader(EShaderFrequency Frequency)const;
    inlineTArray<TShaderRef<FShader>> GetShaders(constFShaderMapBase & InShaderMap) const;


    //check.
    void Validate(constFShaderPipelineType* InPipelineType)const; //Processes compiled
    shader code.
    void Finalize(constFShaderMapResourceCode* Code);

    (.....)

    enumEFilter
    {
        EAll,                   // All pipelines
        EOnlyShared,            // Only pipelines with shared shaders // Only
        EOnlyUnique,            pipelines with unique shaders
    };

    //Hash value.
    LAYOUT_FIELD(FHashedName, //All shadTinypg efrNeaqmueen)c;ies
    FShaderExamples.
    LAYOUT_ARRAY(TMemoryImagePtr<FShader>, //              Shaders,     SF_NumGraphicsFrequencies);
    arrangementid.
    LAYOUT_ARRAY(int32, PermutationIds, SF_NumGraphicsFrequencies);
};
```

```
//Shader map table contents.
classFShaderMapContent {

public:
	structFProjectShaderPipelineToKey {

		inlineFHashedNameoperator()(constFShaderPipeline* InShaderPipeline) {return
		InShaderPipeline->TypeName; }
	};

	explicitFShaderMapContent(EShaderPlatform InPlatform);
	~FShaderMapContent();

	EShaderPlatformGetShaderPlatform()const;

	//check.
	void Validate(constFShaderMapBase& InShaderMap);

	//Findshader.
	template<typename ShaderType>
	ShaderType*GetShader(int32 PermutationId =0)const;
	template<typename ShaderType>
	ShaderType*GetShader(consttypename ShaderType::FPermutationDomain&
PermutationVector )const;
	FShader*GetShader(FShaderType* ShaderType, int32 PermutationId =0)const; FShader*GetShader(const
	FHashedName& TypeName, int32 PermutationId =0)const;

	//Check if there is a specifiedshader.
	bool HasShader(constFHashedName& TypeName, int32 PermutationId)const; bool HasShader(
	constFShaderType* Type, int32 PermutationId)const;

	inlineTArrayView<constTMemoryImagePtr<FShader>> GetShaders()const;
	inlineTArrayView<constTMemoryImagePtr<FShaderPipeline>> GetShaderPipelines()const;

	//Add, FindshaderorPipelineinterface.
	void AddShader(constFHashedName& TypeName, int32 PermutationId, FShader* Shader); FShader*
	FindOrAddShader(constFHashedName& TypeName, int32 PermutationId, FShader* Shader);

	void	AddShaderPipeline(FShaderPipeline* Pipeline);
	FShaderPipeline*		FindOrAddShaderPipeline(FShaderPipeline* Pipeline);

	//Delete an interface.
	void RemoveShaderTypePermutaion(constFHashedName& TypeName, int32 PermutationId); inlinevoid
	RemoveShaderTypePermutaion(constFShaderType* Type, int32 PermutationId); void RemoveShaderPipelineType(
	constFShaderPipelineType* ShaderPipelineType);

	//Get a list of shaders.
	void GetShaderList(constFShaderMapBase& InShaderMap,constFSHAHash& InMaterialShaderMapHash,
TMap<FShaderId, TShaderRef<FShader>>& OutShaders)const;
	void GetShaderList(constFShaderMapBase& InShaderMap, TMap<FHashedName,
TShaderRef<FShader>>& OutShaders)const;

	//Get a list of shader pipelines.
	void GetShaderPipelineList(constFShaderMapBase& InShaderMap, TArray<FShaderPipelineRef>&
OutShaderPipelines, FShaderPipeline::EFilter Filter)const;

	(.....)
```

```
        //Get the maximum number of instructions for a shader.
        uint32GetMaxNumInstructionsForShader(constFShaderMapBase& InShaderMap, FShaderType* ShaderType)const;

        //Save the compiledshaderCode.
        void Finalize(constFShaderMapResourceCode* Code); //Update the
        hash value.
        void UpdateHash(FSHA1& Hasher)const;

protected:
        using FMemoryImageHashTable = THashTable<FMemoryImageAllocator>;

        //Shader hash.
        LAYOUT_FIELD(FMemoryImageHashTable, //       ShaderHash);
        The shader type.
        LAYOUT_FIELD(TMemoryImageArray<FHashedName>, //List of       ShaderTypes);
        shader permutations.
        LAYOUT_FIELD(TMemoryImageArray<int32>, ShaderPermutations); //A list of
        shader instances.
        LAYOUT_FIELD(TMemoryImageArray<TMemoryImagePtr<FShader>>, Shaders); //List of
        shader pipelines.
        LAYOUT_FIELD(TMemoryImageArray<TMemoryImagePtr<FShaderPipeline>>, ShaderPipelines); //The platform the
        shader was compiled for.
        LAYOUT_FIELD(TEnumAsByte<EShaderPlatform>, Platform);
};

// FShaderMaThe base class of .
classFShaderMapBase
{
public:
        (……)

private:
        constFTypeLayoutDesc&       ContentTypeLayout;
        // ShaderMapresource.
        TRefCountPtr<FShaderMapResource> //       Resource;
        ShaderMapResource code.
        TRefCountPtr<FShaderMapResourceCode> //       Code;
        ShaderMapPointer table.
        FShaderMapPointerTable* PointerTable; //
        ShaderMapcontent.
        FShaderMapContent* Content; //
        Content size.
        uint32 FrozenContentSize; //
        Number of shaders.
        uint32 NumFrozenShaders;

};

//Shader map. Must be specifiedFShaderMapContentandFShaderMapPointerTable template<typename
ContentType, typename PointerTableType = FShaderMapPointerTable> class TShaderMap : public
FShaderMapBase
{
public:
        inlineconstPointerTableType& GetPointerTable(); inlineconst
        ContentType*GetContent()const; inlineContentType*
        GetMutableContent();

        void FinalizeContent() {
```

```
            ContentType* LocalContent = this->GetMutableContent(); LocalContent-
            >Finalize(this->GetResourceCode()); FShaderMapBase::FinalizeContent();

        }

protected:
        TShaderMap();
        virtual FShaderMapPointerTable*             CreatePointerTable();
};

//Shader pipeline reference.
classFShaderPipelineRef {

public:
        FShaderPipelineRef();
        FShaderPipelineRef(FShaderPipeline* InPipeline,constFShaderMapBase& InShaderMap);

        (.....)

        //Get the shader
        template<typename ShaderType>
        TShaderRef<ShaderType>GetShader()             const;
        TShaderRef<FShader>GetShader(EShaderFrequency Frequency)const; inline
        TArray<TShaderRef<FShader>> GetShaders()const;

        //Get the interface of shading pipeline, resources, etc.
        inlineFShaderPipeline*GetPipeline()const;
        FShaderMapResource*GetResource()const;
        constFShaderMapPointerTable&GetPointerTable()const;

        inlineFShaderPipeline* operator->()const;

private:
        FShaderPipeline* ShaderPipeline;//Shader pipeline. const
        FShaderMapBase* ShaderMap;//Shader map table.
};
```

Many of the above types are base classes, and the specific logic needs to be completed by subclasses.

8.3.1.2 FGlobalShaderMap

FGlobalShaderMap saves and manages all compiled FGlobalShader codes. Its definition and related types are as follows:

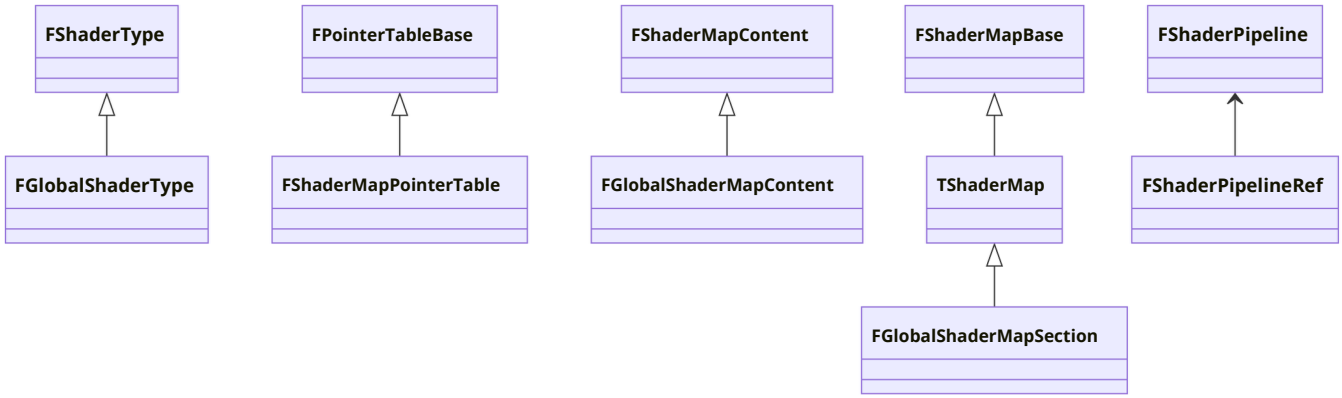
```
// Engine\Source\Runtime\RenderCore\Public\GlobalShader.h

//For processing the simplest shaders (without material and vertex factory links)shader meta type,Each simpleshaderThere should only be one instance of each.
classFGlobalShaderType: public FShaderType {

        friendclassFGlobalShaderTypeCompiler; public:


        typedefFShader::CompiledShaderInitializerType CompiledShaderInitializerType;
```

```
        FGlobalShaderType(...);

        bool ShouldCompilePermutation(EShaderPlatform Platform, int32 PermutationId)const; void
        SetupCompileEnvironment(EShaderPlatform Platform, int32 PermutationId, FShaderCompilerEnvironment&
        Environment);
};

//Global shader subtable.
classFGlobalShaderMapContent: public FShaderMapContent {

        (.....)
public:
        constFHashedName&GetHashedSourceFilename();

private:
        inlineFGlobalShaderMapContent(EShaderPlatform InPlatform,constFHashedName&
        InHashedSourceFilename);

        //The hashed source file name.
        LAYOUT_FIELD(FHashedName, HashedSourceFilename);
};

classFGlobalShaderMapSection: public TShaderMap<FGlobalShaderMapContent,
FShaderMapPointerTable>
{
        (.....)

private:
        inlineFGlobalShaderMapSection();
        inlineFGlobalShaderMapSection(EShaderPlatform InPlatform,constFHashedName&
        InHashedSourceFilename);

        TShaderRef<FShader>GetShader(FShaderType* ShaderType, int32 PermutationId =0)const; FShaderPipelineRef
        GetShaderPipeline(constFShaderPipelineType* PipelineType)const;
};

//GlobalShaderMap.
classFGlobalShaderMap
{
public:
        explicitFGlobalShaderMap(EShaderPlatform                InPlatform);
        ~FGlobalShaderMap();

        //By shader type and arrangementidGet the compiledshaderCode.
        TShaderRef<FShader>GetShader(FShaderType* ShaderType, int32 PermutationId =0)const; //According to the
        arrangementidGet the compiledshaderCode. template<typename ShaderType>

        TShaderRef<ShaderType>GetShader(int32 PermutationId =0)const {

                TShaderRef<FShader> Shader = GetShader(&ShaderType::StaticType, PermutationId); return
                TShaderRef<ShaderType>::Cast(Shader);
        }
        //Get the compiled one according to the arrangement in the shader type
        shaderCode. template<typename ShaderType>
        TShaderRef<ShaderType>GetShader(consttypename ShaderType::FPermutationDomain&
PermutationVector)const
        {
                returnGetShader<ShaderType>(PermutationVector.ToDimensionValueId());
```

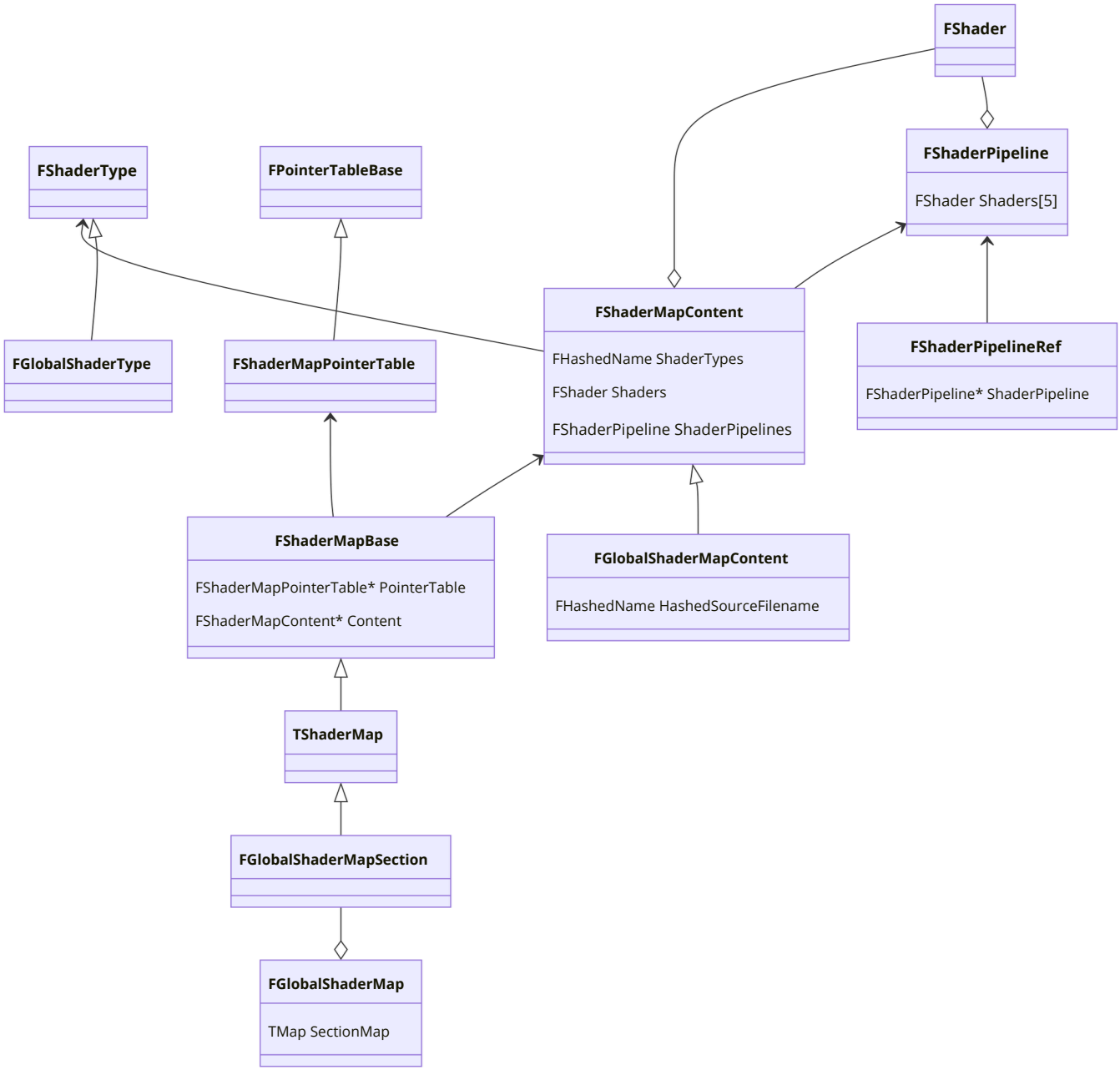
```
    }

    //Check if there is a specifiedshader.
    bool HasShader(FShaderType* Type, int32 PermutationId)const {

        returnGetShader(Type, PermutationId).IsValid();
    }

    //Get the shader pipeline
    FShaderPipelineRefGetShaderPipeline(constFShaderPipelineType* PipelineType)const;

    //Whether there is a shader pipeline.
    bool HasShaderPipeline(constFShaderPipelineType* ShaderPipelineType)const {

        returnGetShaderPipeline(ShaderPipelineType).IsValid();
    }

    bool   IsEmpty()const;
    void   Empty();
    void   ReleaseAllSections();

    //Find or addshader.
    FShader*FindOrAddShader(constFShaderType* ShaderType, int32 PermutationId, FShader* Shader);

    //Find or addshaderPipeline.
    FShaderPipeline*FindOrAddShaderPipeline(constFShaderPipelineType*
ShaderPipelineType, FShaderPipeline* ShaderPipeline);

    //Deleting an Interface
    void RemoveShaderTypePermutaion(constFShaderType* Type, int32 PermutationId); void
    RemoveShaderPipelineType(constFShaderPipelineType* ShaderPipelineType);

    // ShaderMapSectionoperate.
    void AddSection(FGlobalShaderMapSection* InSection); FGlobalShaderMapSection*FindSection(const
    FHashedName& HashedShaderFilename); FGlobalShaderMapSection*FindOrAddSection(const
    FShaderType* ShaderType);

    //IOinterface.
    void   LoadFromGlobalArchive(FArchive& Ar);
    void   SaveToGlobalArchive(FArchive& Ar);

    //Clean Allshader.
    void BeginCreateAllShaders();

    (......)

private:
    //StoredFGlobalShaderMapSectionThe mapping table of .
    TMap<FHashedName, FGlobalShaderMapSection*>           SectionMap;
    EShaderPlatform Platform;
};

//GlobalShaderMapA list ofSP_NumPlatformsyes49.
externRENDERCORE_API FGlobalShaderMap* GGlobalShaderMap[SP_NumPlatforms];
```

The above involves the types and concepts of ShaderMap Content, Section, PointerTable, ShaderType, etc. There are a lot of data and complex relationships, but it is much simpler and clearer

after being abstracted into a UML diagram:



For simplicity, the above class diagram only shows the inheritance relationship. If you add association, aggregation, composition and other relationships, it will look like the following:



The above explains the relationship between FGlobalShaderMap and its core classes. Now let's take a look at how it is applied to actual rendering. First, declare and define the instance and related interfaces of FGlobalShaderMap in GlobalShader.h and GlobalShader.cpp:

```
// Engine\Source\Runtime\RenderCore\Private\GlobalShader.h

//Declare externally accessibleFGlobalShaderMapList.
externRENDERCORE_API FGlobalShaderMap* GGlobalShaderMap[SP_NumPlatforms];

//Gets the specified shading platformFGlobalShaderMap.
externRENDERCORE_API FGlobalShaderMap*GetGlobalShaderMap(EShaderPlatform Platform);

//Get the specifiedFeatureLevelofFGlobalShaderMap.
inlineFGlobalShaderMap*GetGlobalShaderMap(ERHIFeatureLevel::Type FeatureLevel) {

    returnGetGlobalShaderMap(GShaderPlatformForFeatureLevel[FeatureLevel]);
}

// Engine\Source\Runtime\RenderCore\Private\GlobalShader.cpp

//Declare all shading platformsFGlobalShaderMap.
FGlobalShaderMap* GGlobalShaderMap[SP_NumPlatforms] = {};

//GetFGlobalShaderMap.
FGlobalShaderMap*GetGlobalShaderMap(EShaderPlatform {              Platform)

    returnGGlobalShaderMap[Platform];
}
```

However, the above only defines GGlobalShaderMap, and the array is just an empty list. The actual creation stack chain is as follows:

```
// Engine\Source\Runtime\Launch\Private\LaunchEngineLoop.cpp

//Engine pre-initialization.
int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

    (......)

    //Is it enabled?shaderCompile, which is usually enabled.
    boolbEnableShaderCompile = !FParse::Param(FCommandLine::Get(), TEXT(
"NoShaderCompile"));

    (......)

    if(bEnableShaderCompile && !IsRunningDedicatedServer() && !bIsCook) {

        (......)

        //CompileGlobalShaderMap
        CompileGlobalShaderMap(false);

        (......)
    }

    (......)
}

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

```
void CompileGlobalShaderMap(EShaderPlatform Platform,constITargetPlatform* TargetPlatform,bool
bRefreshShaderMap) {

    (……)

    //  If the corresponding platformGlobalShaderMapIf it is not created,
    if  create it. (!GGlobalShaderMap[Platform])
    {
        (……)

        //Create the corresponding platformFGlobalShaderMap. GGlobalShaderMap[Platform]
        = new FGlobalShaderMap(Platform);

        //  Cookedmodel.
        if  (FPlatformProperties::RequiresCookedData())
        {
            (……)
        }
        //  Uncookedmodel
        else
        {
            // FGlobalShaderMapofid. FGlobalShaderMapId
            ShaderMapId(Platform);

            constint32 ShaderFilenameNum =
ShaderMapId.GetShaderFilenameToDependeciesMap().Num();
            const floatProgressStep =25.0f/ShaderFilenameNum;

            TArray<uint32> AsyncDDCRequestHandles;
            AsyncDDCRequestHandles.SetNum(ShaderFilenameNum);

            int32 HandleIndex =0;

            //submitDDCask.
            for(constauto&ShaderFilenameDependencies:
ShaderMapId.GetShaderFilenameToDependeciesMap())
            {
                SlowTask.EnterProgressFrame(ProgressStep);

                constFString DataKey = GetGlobalShaderMapKeyString(ShaderMapId, Platform,
TargetPlatform,     ShaderFilenameDependencies.Value);

                AsyncDDCRequestHandles[HandleIndex]          =
GetDerivedDataCacheRef().GetAsynchronous(*DataKey,              TEXT("GlobalShaderMap"_SV));

                + + HandleIndex;
            }

            //Processing has been completed
            DDCask. TArray<uint8>CachedData;
            HandleIndex =      0;
            for(constauto&ShaderFilenameDependencies:
ShaderMapId.GetShaderFilenameToDependeciesMap())
            {
                SlowTask.EnterProgressFrame(ProgressStep);
                CachedData.Reset();
```

```
GetDerivedDataCacheRef().WaitAsynchronousCompletion(AsyncDDCRequestHandles[HandleIndex]);
                        if
(GetDerivedDataCacheRef().GetAsynchronousResults(AsyncDDCRequestHandles[HandleIndex], CachedData))

                        {
                                FMemoryReaderMemoryReader(CachedData);
                                GGlobalShaderMap[Platform]-
> AddSection(FGlobalShaderMapSection::CreateFromArchive(MemoryReader));
                        }
                        else
                        {
                                //Not inDDCFound in, ignore it.
                        }

                        + + HandleIndex;
                }
        }

        //If you haveshaderNot loaded, compile it.
        VerifyGlobalShaders(Platform, bLoadedFromCacheFile);

        //Create all shaders.
        if(GCreateShadersOnLoad && Platform == GMaxRHIShaderPlatform) {

                GGlobalShaderMap[Platform]->BeginCreateAllShaders();
        }
    }
}
```

As can be seen above, FGlobalShaderMap is created in the pre-initialization phase of the engine, and then tries to read the compiled shader data from DDC. After that, other modules can access and operate FGlobalShaderMap objects normally.

In addition, there is also an instance of FGlobalShaderMap inside FViewInfo, but it is also an instance obtained through GetGlobalShaderMap:

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

classFViewInfo: public FSceneView {

public:
    (......)

    FGlobalShaderMap* ShaderMap;

    (......)
};

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

void FViewInfo::Init() {

    (......)

    ShaderMap = GetGlobalShaderMap(FeatureLevel);
```

```
    (……)
}
```

In this way, most of the logic in the rendering module can easily obtain an instance of FViewInfo, and therefore can easily access an instance of FGlobalShaderMap (without specifying FeatureLevel).

8.3.1.3 FMaterialShaderMap

FMaterialShaderMap stores and manages a set of FMaterialShader instance objects. It and related types are defined as follows:

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

//MaterialShaderMapcontent.
classFMaterialShaderMapContent: public FShaderMapContent {

public:
    (……)

    inlineuint32GetNumShaders()const; inlineuint32
    GetNumShaderPipelines()const;

private:
    struct    FProjectMeshShaderMapToKey
    {
        inlineconstFHashedName&operator()(constFMeshMaterialShaderMap* InShaderMap) { returnInShaderMap-
        >GetVertexFactoryTypeName(); }
    };

    //Get/add/delete operations.
    FMeshMaterialShaderMap*GetMeshShaderMap(constFHashedName& VertexFactoryTypeName) const;

    void AddMeshShaderMap(constFVertexFactoryType* VertexFactoryType,
FMeshMaterialShaderMap* MeshShaderMap);
    void RemoveMeshShaderMap(constFVertexFactoryType* VertexFactoryType);

    //Ordered mesh shader mapping table, throughVFType->GetId()Index, for fast lookup at runtime.
    LAYOUT_FIELD(TMemoryImageArray<TMemoryImagePtr<FMeshMaterialShaderMap>>,
    OrderedMeshShaderMaps);
    //Material compilation output.
    LAYOUT_FIELD(FMaterialCompilationOutput, MaterialCompilationOutput); //Shader contents
    hash.
    LAYOUT_FIELD(FSHAHash, ShaderContentHash);

    LAYOUT_FIELD_EDITORONLY(TMemoryImageArray<FMaterialProcessedSource>,
    ShaderProcessedSource);    LAYOUT_FIELD_EDITORONLY(FMemoryImageString,
    LAYOUT_FIELD_EDITORONLY(FMemoryImageString,        FriendlyName);
    LAYOUT_FIELD_EDITORONLY(FMemoryImageString,        DebugDescription);
                                                       MaterialPath);
};

//Material shader mapping table, parent class isTShaderMap.
classFMaterialShaderMap: public TShaderMap<FMaterialShaderMapContent,
FShaderMapPointerTable>, public FDeferredCleanupInterface
{
public:
```

```
using Super = TShaderMap<FMaterialShaderMapContent, FShaderMapPointerTable>;

//Find Specifiedidand platformFMaterialShaderMapExamples.
staticTRefCountPtr<FMaterialShaderMap>FindId(constFMaterialShaderMapId& ShaderMapId,
EShaderPlatform Platform);

(.....)

//  ShaderMap interface
//  Get the shader instance.
TShaderRef<FShader>GetShader(FShaderType* ShaderType, int32 PermutationId =0)const; template<typename
ShaderType> TShaderRef<ShaderType>GetShader(int32 PermutationId = const;
0)
template<typename ShaderType> TShaderRef<ShaderType>GetShader(consttypename
ShaderType::FPermutationDomain& PermutationVector)const;

uint32GetMaxNumInstructionsForShader(FShaderType* ShaderType)const;

void FinalizeContent();

//Compile a material's shader and cache it toshader mapmiddle.
void Compile(FMaterial* Material,constFMaterialShaderMapId& ShaderMapId,
TRefCountPtr<FShaderCompilerEnvironment> MaterialEnvironment,const FMaterialCompilationOutput&
InMaterialCompilationOutput, EShaderPlatform Platform,bool bSynchronousCompile);


//Check if there isshaderLost.
bool IsComplete(constFMaterial* Material,boolbSilent); //Try adding an
existing compilation task.
bool TryToAddToExistingCompilationTask(FMaterial* Material);

//Built onshader mapofshaderList.
void GetShaderList(TMap<FShaderId, TShaderRef<FShader>>& OutShaders)const; void GetShaderList
(TMap<FHashedName, TShaderRef<FShader>>& OutShaders)const; void GetShaderPipelineList
(TArray<FShaderPipelineRef>& OutShaderPipelines)const;

uint32GetShaderNum()const;

//Registers a material shader map into the global table so it can be used
by materials. void Register(EShaderPlatform InShaderPlatform);

// Reference        counting.
void AddRef();
void Release();

//Delete Assignmentshader typeAll entries in the cache.
void FlushShadersByShaderType(constFShaderType* ShaderType);
void FlushShadersByShaderPipelineType(constFShaderPipelineType* ShaderPipelineType); void
FlushShadersByVertexFactoryType(constFVertexFactoryType* VertexFactoryType);

static void RemovePendingMaterial(FMaterial* Material);
static constFMaterialShaderMap*GetShaderMapBeingCompiled(constFMaterial* Material);

// Accessors.
FMeshMaterialShaderMap*GetMeshShaderMap(FVertexFactoryType* VertexFactoryType)const;
FMeshMaterialShaderMap*GetMeshShaderMap(constFHashedName& VertexFactoryTypeName) const;

constFMaterialShaderMapId&GetShaderMapId()const;
```

```
    (……)

private:
    //Global Materialsshader map.
    staticTMap<FMaterialShaderMapId,FMaterialShaderMap*>
    GIdToMaterialShaderMap[SP_NumPlatforms];
    staticFCriticalSection GIdToMaterialShaderMapCS; //The material
    being compiled.
    staticTMap<TRefCountPtr<FMaterialShaderMap>, TArray<FMaterial*> >
    ShaderMapsBeingCompiled;


    //Shader Mapping Tableid.
    FMaterialShaderMapId        ShaderMapId;
    //During compilationid.
    uint32 CompilingId;
    //The corresponding platform.
    constITargetPlatform*
                                    CompilingTargetPlatform;


    //The number of citations.
    mutable int32 NumRefs;

    //mark   boolbDeletedThroughDeferredCleanup;
    uint32              bRegistered:1;              uint32
    bCompilationFinalized           :1;           uint32
    bCompiledSuccessfully:1;   uint32   bIsPersistent
    :1;



    (……)
};
```

The difference between FMaterialShaderMap and FGlobalShaderMap is that it will additionally associate a material and a vertex factory. The internal data content of a single FMaterialShaderMap is as follows:

```
FMaterialShaderMap
    FLightFunctionPixelShader           -FMaterialShaderType
    FLocalVertexFactory         -FVertexFactoryType
        TDepthOnlyPS    -   FMeshMaterialShaderType
        TDepthOnlyVS    -   FMeshMaterialShaderType
        TBasePassPS     -   FMeshMaterialShaderType
        TBasePassVS     - FMeshMaterialShaderType
        (……)
    FGPUSkinVertexFactory       -FVertexFactoryType
        (……)
```

Since FMaterialShaderMap is bound to the material blueprint, it is a member of FMaterial:

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

classFMaterial
{
public:
    //Get the materialshaderExamples.
```

```
        TShaderRef<FShader>GetShader(class FMeshMaterialShaderType* ShaderType, FVertexFactoryType*
VertexFactoryType, int32 PermutationId,boolbFatalIfMissing =true) const;


        (……)

private:
        //Game Thread MaterialsShaderMap
        TRefCountPtr<FMaterialShaderMap> //Materials for    GameThreadShaderMap;
        the Rendering ThreadShaderMap
        TRefCountPtr<FMaterialShaderMap>            RenderingThreadShaderMap;

        (……)
};

// Engine\Source\Runtime\Engine\Private\Materials\MaterialShared.cpp

TShaderRef<FShader>FMaterial::GetShader(FMeshMaterialShaderType* ShaderType, FVertexFactoryType*
VertexFactoryType, int32 PermutationId,boolbFatalIfMissing)const {

        //fromRenderingThreadShaderMapGetshader.
        constFMeshMaterialShaderMap* MeshShaderMap = RenderingThreadShaderMap-
> GetMeshShaderMap(VertexFactoryType);
        FShader* Shader = MeshShaderMap ? MeshShaderMap->GetShader(ShaderType, PermutationId) : nullptr;


        (……)

        //returnFShaderReferences.
        returnTShaderRef<FShader>(Shader, *RenderingThreadShaderMap);
}
```

Therefore, it can be found that each FMaterial has an FMaterialShaderMap (one for the game thread and one for the rendering thread). If you want to get the Shader of a specified type of FMaterial, you need to get it from the FMaterialShaderMap instance of the FMaterial, thus completing the link between them.

8.3.1.4 FMeshMaterialShaderMap

The above section explains that FGlobalShaderMap stores and manages FGlobalShader, while FMaterialShaderMap stores and manages FMaterialShader. Correspondingly, FMeshMaterialShaderMap stores and manages FMeshMaterialShader. Its definition is as follows:

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

classFMeshMaterialShaderMap: public FShaderMapContent {

public:
        FMeshMaterialShaderMap(EShaderPlatform InPlatform, FVertexFactoryType* InVFType);

        //Starts compiling all materials of the specified material and vertex
        factory type. uint32BeginCompile(
                uint32 ShaderMapId,
                const    FMaterialShaderMapId& InShaderMapId,
                const    FMaterial* Material,
```

```
        const FMeshMaterialShaderMapLayout&    MeshLayout,
        FShaderCompilerEnvironment*    MaterialEnvironment,
        EShaderPlatform                                Platform,
        TArray<TSharedRef<FShaderCommonCompileJob,    ESPMode::ThreadSafe>>&        NewJobs,
        FString      DebugDescription,
        FString      DebugExtension
        ) ;

    void FlushShadersByShaderType(const FShaderType* ShaderType);
    void FlushShadersByShaderPipelineType(const FShaderPipelineType* ShaderPipelineType);

    (......)

private:
    //The vertex factory type name.
    LAYOUT_FIELD(FHashedName,       VertexFactoryTypeName);
};
```

FMeshMaterialShaderMap is usually not created independently, but attached to FMaterialShaderMapContent. It is created and destroyed together with FMaterialShaderMapContent. For details and applications, see the previous section.

8.3.2 Shader Compilation

This section describes how to compile the Global Shader usf file into the shader code for the target platform. To explain the compilation process of a single shader file, let's trace the RecompileShaders command processing (compiling the global shader):

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

bool RecompileShaders(const TCHAR* Cmd, FOutputDevice& Ar) {

    (......)

    FString FlagStr(FParse::Token(Cmd,0)); if( FlagStr.Len()
    >0) {

        //Flush the shader file cache.
        FlushShaderFileCache(); //Refresh
        rendering instructions.
        FlushRenderingCommands();

        //ProcessingRecompileShaders ChangedOrder
        if(FCString::Stricmp(*FlagStr,TEXT("Changed"))==0) {

            (......)
        }
        //ProcessingRecompileShaders Global`Order
        else if(FCString::Stricmp(*FlagStr,TEXT("Global"))==0) {

            (......)
        }
        //ProcessingRecompileShaders Material`Order
        else if(FCString::Stricmp(*FlagStr,TEXT("Material"))==0)
```

```
        {
            (......)
        }
        //  ProcessingRecompileShaders  All`Order
        else  if(FCString::Stricmp(*FlagStr,TEXT("All"))==0)
        {
            (......)
        }
        //ProcessingRecompileShaders <ShaderPath>`Order
        else
        {
            //Get by file nameFShaderType.
            TArray<constFShaderType*> ShaderTypes =
FShaderType::GetShaderTypesByFilename(*FlagStr);
            //according toFShaderTypeGetFShaderPipelineType.
            TArray<constFShaderPipelineType*> ShaderPipelineTypes =
FShaderPipelineType::GetShaderPipelineTypesByFilename(*FlagStr);
            if(ShaderTypes.Num() >0|| ShaderPipelineTypes.Num() >0) {

                FRecompileShadersTimerTestTimer(TEXT("RecompileShaders                SingleShader));

                TArray<constFVertexFactoryType*> FactoryTypes;

                //Iterate over all activeFeatureLevel,Compile them one by one.
                UMaterialInterface::IterateOverActiveFeatureLevels([&]
(ERHIFeatureLevel::Type InFeatureLevel) {
                    autoShaderPlatform = GShaderPlatformForFeatureLevel[InFeatureLevel]; //Start compiling
                    the specifiedShaderTypes,ShaderPipelineTypes,ShaderPlatformofshader.
                    BeginRecompileGlobalShaders(ShaderTypes, ShaderPipelineTypes,
ShaderPlatform);
                    //Finish compiling.
                    FinishRecompileGlobalShaders();
                });
            }
        }

        return1;
    }

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

The above code enters the key interface BeginRecompileGlobalShaders to start compiling the specified shader:

```
void BeginRecompileGlobalShaders(constTArray<constFShaderType*>& OutdatedShaderTypes, constTArray<const
FShaderPipelineType*>& OutdatedShaderPipelineTypes, EShaderPlatform ShaderPlatform,constITargetPlatform*
TargetPlatform) {

    if(!FPlatformProperties::RequiresCookedData()) {

        //Flushes pending accesses to existing global shaders.
        FlushRenderingCommands();

        //Compile globalShaderMap.
        CompileGlobalShaderMap(ShaderPlatform, TargetPlatform,false);
```

```
            //Test effectiveness.
            FGlobalShaderMap* GlobalShaderMap = GetGlobalShaderMap(ShaderPlatform); if
            (OutdatedShaderTypes.Num() >0|| OutdatedShaderPipelineTypes.Num() >0) {

                  VerifyGlobalShaders(ShaderPlatform,false, &OutdatedShaderTypes,
&OutdatedShaderPipelineTypes);
            }
      }
}

//Compiles a single global shader map.
void CompileGlobalShaderMap(EShaderPlatform Platform,constITargetPlatform* TargetPlatform,bool
bRefreshShaderMap) {

      (......)

      //Delete the old resource.
      if(bRefreshShaderMap || GGlobalShaderTargetPlatform[Platform] != TargetPlatform) {

            delete GGlobalShaderMap[Platform];
            GGlobalShaderMap[Platform] = nullptr;

            GGlobalShaderTargetPlatform[Platform] = TargetPlatform;

            //Make sure we look for updatedshaderSource
            files. FlushShaderFileCache();
      }

      //    Create and compileshader.
      if    (!GGlobalShaderMap[Platform])
      {
            (......)

            GGlobalShaderMap[Platform] = new FGlobalShaderMap(Platform);

            (......)

            //Check if there isshaderNot loaded, compile it if it is.
            VerifyGlobalShaders(Platform, bLoadedFromCacheFile);

            if(GCreateShadersOnLoad && Platform == GMaxRHIShaderPlatform) {

                  GGlobalShaderMap[Platform]->BeginCreateAllShaders();
            }
      }
}

//Check if there isshaderNot loaded, compile it if it is.
void VerifyGlobalShaders(EShaderPlatform Platform,boolbLoadedFromCacheFile,const TArray<constFShaderType*>*
OutdatedShaderTypes,constTArray<constFShaderPipelineType*>* OutdatedShaderPipelineTypes)

{
      (......)

      //GetFGlobalShaderMapExamples.
      FGlobalShaderMap* GlobalShaderMap = GetGlobalShaderMap(Platform);
```

```
(……)

//All assignments, includingsingleandpipeline.
TArray<TSharedRef<FShaderCommonCompileJob, ESPMode::ThreadSafe>> GlobalShaderJobs;

//Add firstsingle jobs.
TMap<TShaderTypePermutation<constFShaderType>, FShaderCompileJob*> SharedShaderJobs;

for(TLinkedList<FShaderType*>::TIterator ShaderTypeIt(FShaderType::GetTypeList()); ShaderTypeIt;
ShaderTypeIt.Next())
{
    FGlobalShaderType* GlobalShaderType = ShaderTypeIt->GetGlobalShaderType(); if(!
    GlobalShaderType)
    {
        continue;
    }

    int32 PermutationCountToCompile =0;
    for(int32 PermutationId =0; PermutationId < GlobalShaderType-
> GetPermutationCount(); PermutationId++) {

        if(GlobalShaderType->ShouldCompilePermutation(Platform, PermutationId)
            && (!GlobalShaderMap->HasShader(GlobalShaderType, PermutationId) ||
(OutdatedShaderTypes        && OutdatedShaderTypes->Contains(GlobalShaderType))))
        {
            //    If it is expiredshaderType, delete it.
            if    (OutdatedShaderTypes)
            {
                GlobalShaderMap->RemoveShaderTypePermutaion(GlobalShaderType,
PermutationId);
            }

            //Create Compilationglobal shader typeHomework
            auto* Job =
FGlobalShaderTypeCompiler::BeginCompileShader(GlobalShaderType, nullptr,        PermutationId,        Platform,
GlobalShaderJobs);
            TShaderTypePermutation<constFShaderType>
ShaderTypePermutation(GlobalShaderType, PermutationId);
            //Add to job list.
            SharedShaderJobs.Add(ShaderTypePermutation, Job);
            PermutationCountToCompile++;
        }
    }

    (……)
}

//deal withFShaderPipeline,If it is shareablepipeline,There is no need to add tasks
repeatedly. for(TLinkedList<FShaderPipelineType*>::TIterator
ShaderPipelineIt(FShaderPipelineType::GetTypeList()); ShaderPipelineIt.NeSxhta()d)erPipelineIt;

{
    constFShaderPipelineType* Pipeline = *ShaderPipelineIt; if(Pipeline-
    >IsGlobalTypePipeline()) {

        if(!GlobalShaderMap->HasShaderPipeline(Pipeline) ||
(OutdatedShaderPipelineTypes && OutdatedShaderPipelineTypes->Contains(Pipeline)))
        {
```

```
                auto& StageTypes = Pipeline->GetStages();
                TArray<FGlobalShaderType*> ShaderStages;
                for(int32 Index =0; Index < StageTypes.Num(); ++Index) {

                        FGlobalShaderType* GlobalShaderType = ((FShaderType*)
(StageTypes[Index]))->GetGlobalShaderType();
                        if(GlobalShaderType->ShouldCompilePermutation(Platform,
kUniqueShaderPermutationId))
                        {
                                ShaderStages.Add(GlobalShaderType);
                        }
                        else
                        {
                                break;
                        }
                }

                //   Delete expiredPipelineType
                if   (OutdatedShaderPipelineTypes)
                {
                        GlobalShaderMap->RemoveShaderPipelineType(Pipeline);
                }

                if(ShaderStages.Num() == StageTypes.Num()) {

                        (......)

                        if   (Pipeline->ShouldOptimizeUnusedOutputs(Platform))
                        {
                                // Make a pipeline job with all the stages
                                FGlobalShaderTypeCompiler::BeginCompileShaderPipeline(Platform,
Pipeline,       ShaderStages,       GlobalShaderJobs);
                        }
                        else
                        {
                                for(constFShaderType* ShaderType : StageTypes) {

                                        TShaderTypePermutation<constFShaderType>
ShaderTypePermutation(ShaderType, kUniqueShaderPermutationId);

                                        FShaderCompileJob** Job =
SharedShaderJobs.Find(ShaderTypePermutation);
                                        auto* SingleJob = (*Job)->GetSingleShaderJob(); auto&
                                        SharedPipelinesInJob = SingleJob-
> SharingPipelines.FindOrAdd(nullptr);
                                        //Add topipelineOperation.
                                        SharedPipelinesInJob.Add(Pipeline);
                                }
                        }
                }
        }
    }

    if(GlobalShaderJobs.Num() >0) {

        GetOnGlobalShaderCompilation().Broadcast(); //Add a
        compilation job toGShaderCompilingManagermiddle.
```

```
            GShaderCompilingManager->AddJobs(GlobalShaderJobs,true,false,"Globals");

            //Some platforms do not support asyncshaderCompile.
            const boolbAllowAsynchronousGlobalShaderCompiling =
                  !IsOpenGLPlatform(GMaxRHIShaderPlatform)              &&
!IsVulkanPlatform(GMaxRHIShaderPlatform) &&
                  !IsMetalPlatform(GMaxRHIShaderPlatform)               &&
!IsSwitchPlatform(GMaxRHIShaderPlatform) &&
                  GShaderCompilingManager->AllowAsynchronousShaderCompiling();

            if(!bAllowAsynchronousGlobalShaderCompiling) {

                  TArray<int32> ShaderMapIds;
                  ShaderMapIds.Add(GlobalShaderMapId);

                  GShaderCompilingManager->FinishCompilation(TEXT("Global"), ShaderMapIds);
            }
      }
}
```

From this we can see that the shader compilation job is completed by the global object GShaderCompilingManager. Now let's go into the type definition of FShaderCompilingManager:

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

classFShaderCompilingManager {

      (......)

private:
      ////////////////////////////////////////////////////// //// //Thread-shared properties:
      Only whenCompileQueueSectionOnly read and write when acquired. bool
      bCompilingDuringGame; //List of jobs currently being compiled.

      TArray<TSharedRef<FShaderCommonCompileJob,          ESPMode::ThreadSafe>>       CompileQueue;
      TMap<int32,     FShaderMapCompileResults> ShaderMapJobs;
      int32           NumOutstandingJobs;
      int32 NumExternalJobs;
      FCriticalSection          CompileQueueSection;

      ////////////////////////////////////////////////////// //// //Main thread state - only
      accessible to the main thread.
      TMap<int32, FShaderMapFinalizeResults> PendingFinalizeShaderMaps;
      TUniquePtr<FShaderCompileThreadRunnableBase> Thread;

      ////////////////////////////////////////////////////// //// //Configuration properties

      uint32    NumShaderCompilingThreads;
      uint32    NumShaderCompilingThreadsDuringGame;
      int32     MaxShaderJobBatchSize;
      int32   NumSingleThreadedRunsBeforeRetry;
      uint32 ProcessId;
      (......)

public:
      //Data access and setup interface.
```

```
    bool bool ShouldDisplayCompilingNotification()                const;
    int32 GetNumberRemainingJobs()  AllowAsynchronousShaderCompiling()          const;
            IsCompiling()const; HasShaderJobs()const;


                                                const;
    void    SetExternalJobs(int32           NumJobs);

    enum classEDumpShaderDebugInfo: int32 {

        Never                       = 0,
        Always                       = 1,
        OnError                       = 2,
        OnErrorOrWarning            = 3
    };

    (.....)

    //Add compilation job.
    ENGINE_APIvoid AddJobs(TArray<TSharedRef<FShaderCommonCompileJob, ESPMode::ThreadSafe>>&
NewJobs,boolbOptimizeForLowLatency,bool bRecreateComponentRenderStateOnCompletion,constFString
MaterialBasePath, FString PermutationString = FString(""),boolbSkipResultProcessing =false);



    //Delete the compilation job.
    ENGINE_APIvoid CancelCompilation(constTCHAR* MaterialName,constTArray<int32>&
    ShaderMapIdsToCancel);
    //Ends the compilation job, blocking the thread until the specified material compilation is completed.
    ENGINE_APIvoid FinishCompilation(constTCHAR* MaterialName,constTArray<int32>&
    ShaderMapIdsToFinishCompiling);
    //Block allshaderCompile,  Until it is completed.
    ENGINE_APIvoid      FinishAllCompilation();
    //Close the build manager.
    ENGINE_APIvoid      Shutdown();
    //Processing completed asynchronous results,  Attach them to the associated material.
    ENGINE_APIvoid ProcessAsyncResults(boolbLimitExecutionTime,bool
    bBlockOnGlobalShaderCompletion);

    static bool IsShaderCompilerWorkerRunning(FProcHandle & WorkerHandle);
};

// Engine\Source\Runtime\Engine\Private\ShaderCompiler\ShaderCompiler.cpp

void FShaderCompilingManager::AddJobs(TArray<TSharedRef<FShaderCommonCompileJob, ESPMode::ThreadSafe>>&
NewJobs,boolbOptimizeForLowLatency,bool bRecreateComponentRenderStateOnCompletion,constFString
MaterialBasePath,constFString PermutationString,boolbSkipResultProcessing) {



    (.....)

    //Register the job toGShaderCompilerStats.
    if(NewJobs.Num())
    {
        FShaderCompileJob* Job = NewJobs[0]->GetSingleShaderJob(); if(Job)//assume
        that all jobs are for the same platform {

            GShaderCompilerStats->RegisterCompiledShaders(NewJobs.Num(),          Job-
> Input.Target.GetPlatform(), MaterialBasePath, PermutationString); }
```

```
            else
            {
                GShaderCompilerStats->RegisterCompiledShaders(NewJobs.Num(), SP_NumPlatforms,
MaterialBasePath, PermutationString);
            }
        }

        // Enqueue the compiled list.
        if (bOptimizeForLowLatency)
        {
            int32 InsertIndex =0;

            for(; InsertIndex < CompileQueue.Num(); InsertIndex++) {

                if(!CompileQueue[InsertIndex]->bOptimizeForLowLatency) {

                    break;
                }
            }

            CompileQueue.InsertZeroed(InsertIndex, NewJobs.Num());

            for(int32 JobIndex =0; JobIndex < NewJobs.Num(); JobIndex++) {

                CompileQueue[InsertIndex + JobIndex] = NewJobs[JobIndex];
            }
        }
        else
        {
            CompileQueue.Append(NewJobs);
        }

        //Increase the number of jobs.
        FPlatformAtomics::InterlockedAdd(&NumOutstandingJobs, NewJobs.Num());

        //Increase the number of jobs for the shader map table.
        for(int32 JobIndex =0; JobIndex < NewJobs.Num(); JobIndex++) {

            NewJobs[JobIndex]->bOptimizeForLowLatency = bOptimizeForLowLatency;
            FShaderMapCompileResults& ShaderMapInfo =
ShaderMapJobs.FindOrAdd(NewJobs[JobIndex]->Id);
            ShaderMapInfo.bRecreateComponentRenderStateOnCompletion =
            bRecreateComponentRenderStateOnCompletion;
            ShaderMapInfo.bSkipResultProcessing = bSkipResultProcessing; auto* PipelineJob =
            NewJobs[JobIndex]->GetShaderPipelineJob(); if(PipelineJob)

            {
                ShaderMapInfo.NumJobsQueued += PipelineJob->StageJobs.Num();
            }
            else
            {
                ShaderMapInfo.NumJobsQueued++;
            }
        }
}

void FShaderCompilingManager::FinishCompilation(constTCHAR* MaterialName,const TArray<int32>&
ShaderMapIdsToFinishCompiling)
```

```
{
    (……)

    TMap<int32, FShaderMapFinalizeResults> CompiledShaderMaps;
    CompiledShaderMaps.Append( PendingFinalizeShaderMaps );
    PendingFinalizeShaderMaps.Empty();


    //Blocking compilation.
    BlockOnShaderMapCompletion(ShaderMapIdsToFinishCompiling, CompiledShaderMaps);

    //Retry and get the potential error.
    boolbRetry =false; do

    {
        bRetry = HandlePotentialRetryOnError(CompiledShaderMaps);
    }
    while(bRetry);

    //Processing compiledShaderMap.
    ProcessCompiledShaderMaps(CompiledShaderMaps, FLT_MAX);

    (……)
}
```

From the above, we can see that the final shader compilation job instance type is FShaderCommonCompileJob, and its instance pairs enter a global queue for multi-threaded asynchronous compilation. The following is the definition of FShaderCommonCompileJob and its related types:

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

//Stored for compilationshaderor      pipelineGeneral data of .
shader class
FShaderCommonCompileJob {
public:
    uint32 Id;
    //Whether the compilation is completed.
    boolbFinalized;
    //Success or not.
    bool    bSucceeded;
    bool    bOptimizeForLowLatency;

    FShaderCommonCompileJob(uint32 InId); virtual
    ~FShaderCommonCompileJob();

    //Data interface.
    virtual FShaderCompileJob*GetSingleShaderJob(); virtualconstFShaderCompileJob*
    GetSingleShaderJob()const; virtual FShaderPipelineCompileJob*GetShaderPipelineJob(); virtual
    constFShaderPipelineCompileJob*GetShaderPipelineJob()const;



    //Unshaded compiler jobs get a globalid.
    ENGINE_APIstaticuint32GetNextJobId();

private:
    //OperationidCounter.
```

```
    staticFThreadSafeCounter JobIdCounter;
};

//To compile a singleshaderAll input and output information.
classFShaderCompileJob: public FShaderCommonCompileJob {

public:
    //The vertex factory of the shader, which
    may benull. FVertexFactoryType* VFType; //
    The shader type.
    FShaderType* ShaderType; //
    arrangementid.
    int32 PermutationId;
    //Inputs and outputs of compilation.
    FShaderCompilerInput
    FShaderCompilerOutput      Input;
                               Output;

    //Share this assignment withPipelineList.
    TMap<constFVertexFactoryType*, TArray<constFShaderPipelineType*>> SharingPipelines;

    FShaderCompileJob(uint32 InId, FVertexFactoryType* InVFType, FShaderType* InShaderType, int32
    InPermutationId);

    virtual FShaderCompileJob*GetSingleShaderJob() override; virtualconstFShaderCompileJob*
    GetSingleShaderJob()constoverride;
};

//For compilationShaderPipelineInformation.
classFShaderPipelineCompileJob: public FShaderCommonCompileJob {

public:
    //Job list.
    TArray<TSharedRef<FShaderCommonCompileJob, ESPMode::ThreadSafe>> bool          StageJobs;
    bFailedRemovingUnused;

    //Belong toShaderPipeline
    constFShaderPipelineType* ShaderPipeline;

    FShaderPipelineCompileJob(uint32 InId,constFShaderPipelineType* InShaderPipeline, int32 NumStages);


    virtual FShaderPipelineCompileJob*GetShaderPipelineJob() override; virtualconst
    FShaderPipelineCompileJob*GetShaderPipelineJob()constoverride;
};
```

The above jobs are added to the FShaderCompilingManager::CompileQueue queue through the FShaderCompilingManager::AddJobs and other interfaces, and then the jobs are mainly pulled and executed by the FShaderCompileThreadRunnable::PullTasksFromQueue interface (multi-producer multi-consumer mode):

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

int32FShaderCompileThreadRunnable::PullTasksFromQueue() {

    int32 NumActiveThreads =0; {
```

```
            //Enter a critical section to access the input and output queues.
            FScopeLockLock(&Manager->CompileQueueSection);

            constint32 NumWorkersToFeed = Manager->bCompilingDuringGame? Manager-
> NumShaderCompilingThreadsDuringGame : WorkerInfos.Num(); //Count the
            number of jobs per worker thread.
            constautoNumJobsPerWorker = (Manager->CompileQueue.Num() / NumWorkersToFeed) +
1;

            //Traverse allWorkerInfos.
            for(int32 WorkerIndex =0; WorkerIndex < WorkerInfos.Num(); WorkerIndex++) {

                  FShaderCompileWorkerInfo& CurrentWorkerInfo = *WorkerInfos[WorkerIndex];

                  //If this worker thread does not have any queued jobs, search from other input queues.
                  if(CurrentWorkerInfo.QueuedJobs.Num() ==0&& WorkerIndex < NumWorkersToFeed) {

                        if(Manager->CompileQueue.Num() >0) {

                              boolbAddedLowLatencyTask =false;
                              constautoMaxNumJobs = FMath::Min3(NumJobsPerWorker, Manager-
> CompileQueue.Num(), Manager->MaxShaderJobBatchSize);

                              int32 JobIndex =0;
                              // Don't put more than one low latency task into a batch
                              for(; JobIndex < MaxNumJobs && !bAddedLowLatencyTask; JobIndex++) {

                                    bAddedLowLatencyTask |= Manager->CompileQueue[JobIndex]-
> bOptimizeForLowLatency;
                                    //From the managerCompileQueueAdded to this worker threadQueuedJobs.
                                    CurrentWorkerInfo.QueuedJobs.Add(Manager->CompileQueue[JobIndex]);
                              }

                              CurrentWorkerInfo.bIssuedTasksToWorker =false;
                              CurrentWorkerInfo.bLaunchedWorker =false;
                              CurrentWorkerInfo.StartTime = FPlatformTime::Seconds();
                              NumActiveThreads++;
                              //From the slave managerCompileQueueDelete the hijacked job.CompileQueueyesThreadSafe
PatternTArray.
                              Manager->CompileQueue.RemoveAt(0, JobIndex);
                        }
                  }
                  //    This worker thread has jobs.
                  else
                  {
                        if(CurrentWorkerInfo.QueuedJobs.Num() >0) {

                              NumActiveThreads++;
                        }

                        //    Add completed jobs to the output queue (
                        if    ShaderMapJobs) (CurrentWorkerInfo.bComplete)
                        {
                              for(int32 JobIndex =0; JobIndex <
CurrentWorkerInfo.QueuedJobs.Num(); JobIndex++)
                              {
                                    FShaderMapCompileResults& ShaderMapResults = Manager-
> ShaderMapJobs.FindChecked(CurrentWorkerInfo.QueuedJobs[JobIndex]->Id);
```

```
ShaderMapResults.FinishedJobs.Add(CurrentWorkerInfo.QueuedJobs[JobIndex]);
                                    ShaderMapResults.bAllJobsSucceeded =
ShaderMapResults.bAllJobsSucceeded && CurrentWorkerInfo.QueuedJobs[JobIndex]->bSucceeded;
                                }

                                (.....)

                                //renewNumOutstandingJobsquantity.
                                FPlatformAtomics::InterlockedAdd(&Manager->NumOutstandingJobs,                -
CurrentWorkerInfo.QueuedJobs.Num());

                                //Clear the job data.
                                CurrentWorkerInfo.bComplete =false;
                                CurrentWorkerInfo.QueuedJobs.Empty();
                            }
                        }
                    }
                }
                return    NumActiveThreads;
}
```

The type of the above working thread information CurrentWorkerInfo is FShaderCompileWorkerInfo:

```
//Shader compilation worker thread information.
struct    FShaderCompileWorkerInfo
{
    //    Work processhandle.May be illegal.
    FProcHandle    WorkerProcess;
    //Track whether there is any problem.
    boolbIssuedTasksToWorker; //Is it
    started?
    boolbLaunchedWorker;
    //Have all task questions been received?
    boolbComplete;
    //The time when the most recent task batch was started.
    doubleStartTime;


    //The worker process is responsible for compiling. (Note that it is thread-safe mode)
    TArray<TSharedRef<FShaderCommonCompileJob, ESPMode::ThreadSafe>> QueuedJobs;

    //Constructor.
    FShaderCompileWorkerInfo(); //
    Destructor, notVirtualof.
    ~FShaderCompileWorkerInfo() {

        if(WorkerProcess.IsValid()) {

            FPlatformProcess::TerminateProc(WorkerProcess);
            FPlatformProcess::CloseProc(WorkerProcess);
        }
    }
};
```

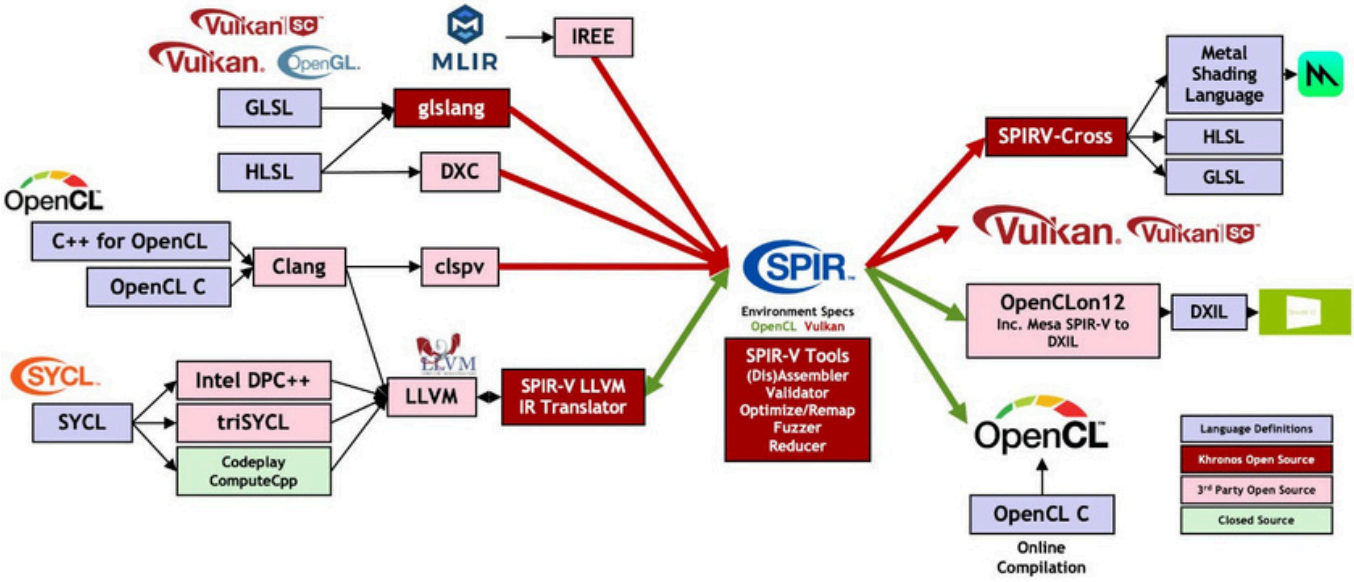
At this point, the compilation process and mechanism of Shader have been explained almost completely, and the remaining details and mechanisms can be studied by yourself.

8.3.3 Shader cross-platform

When we are developing, we will only write one UE Style HLSL. So how does UE compile them into Shader instructions for different graphics APIs (see the table below) and FeatureLevels?

Graphics API	Shading Language	Analysis
Direct3D	HLSL (High Level Shading Language	High-level shading language, only available on Windows platform
OpenGL	GLSL (OpenGL Shading Language	Cross-platform, but state machine-based design is incompatible with modern GPU architecture
OpenGL ES	ES GLSL	Dedicated to mobile platforms
Metal	MSL (Metal Shading Language	Only available for Apple systems
Vulkan	SPIR-V	SPIR-V is an intermediate language that can easily and completely translate shaders from other platforms.

SPIR-V is managed by Khronos (also the creator of OpenGL and Vulkan), and it is actually a huge ecosystem that includes shading languages, toolchains, and runtime libraries:



An overview of the SPIR-V ecosystem, where shader cross-platform is only part of it.

SPIR-V is also a shader cross-platform solution for many commercial engines or renderers. So does UE also use SPIR-V, or does it choose other solutions? This section will answer this question and explore the shader cross-platform solution used by U.E.

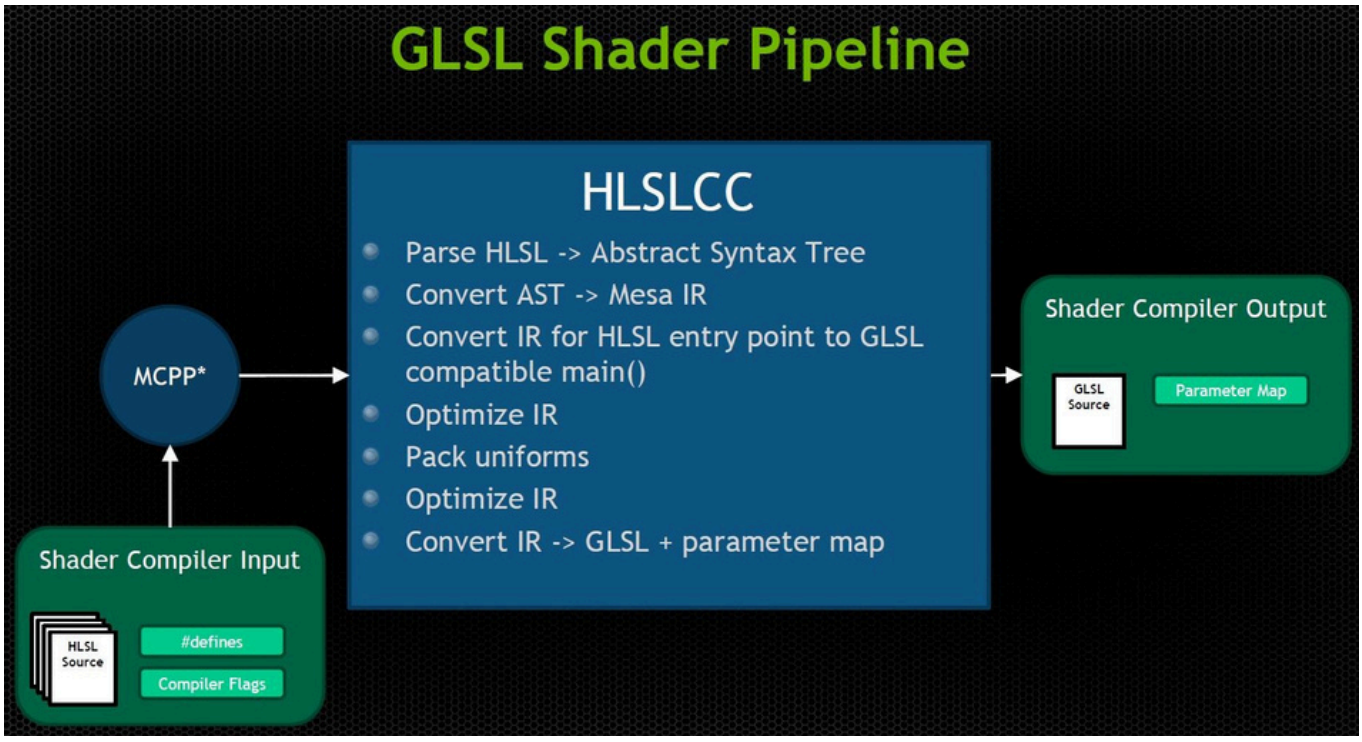
For Shader cross-platform, you usually need to consider the following points:

- **Single coding can be used on multiple platforms.**This is a basic requirement. If this feature cannot be achieved, there is no way to talk about cross-platform, which will also increase the workload of developers and reduce work efficiency.
- **Can compile offline.**Most shader compilers currently support this feature.
- **Reflection is needed to create metadata used by the renderer at runtime,**such as which index the texture is bound to, whether a uniform is used, etc.
- **Specific optimization measures,**such as offline verification, inlining, useless instruction and data detection and deletion, instruction merging and simplification, whether offline compilation is intermediate language or target machine code, etc.

UE considered several ideas for the Shader cross-platform solution in the early days:

- Using macros to encapsulate the differences between various shading languages is feasible for simple shading logic, but in reality, there are huge differences between various shading languages, which are almost impossible to abstract with macros. Therefore, it is not feasible.
- Use FXC to compile HLSL and then convert bytecode. Good results, but the fatal disadvantage is that it cannot support the Mac OS platform, so it is abandoned.
- Third-party cross-platform compiler. At that time (2014), there was no compiler that could support SM5.0 syntax and Coumte Shader.

Faced with the current situation at that time (around 2014), UE4.3 was inspired byglsl-optimizer and built its own wheel HLSLCC (HLSL Cross Compiler) based on Mesa GLSL parser and IR . HLSLCC uses the analyzer to analyze SM5.0 (instead of GLSL) and implements a Mesa IR to GLSL converter (similar to glsl-optimizer). In addition, Mesa naturally supports IR optimization, so HLSLCC also supports IR optimization.



Schematic diagram of the HLSLCC pipeline under GLSL. The input of the shader compiler is HLSL source code, which will first be converted into MCPP and then processed by HLSLCC into GLSL source code and parameter table.

The main working steps of HLSLCC are as follows:

- **Preprocessing**,preprocessing stage. Run through a C-style preprocessor. Before compiling, UE uses MCPP for preprocessing, so this step is skipped.
- **Parsing**,syntax analysis phase. Through Mesa's_**mesa_hlsl_parse**interface, HLSL will be parsed into an abstract syntax tree, and Lexer (syntax analysis) and Parser are generated by flex and bison respectively.
- **Compilation**phase. Use_**mesa_ast_to_hir**to compile the AST (abstract syntax tree) into Mesa IR. In this phase, the compiler performs implicit conversions, function overload resolution, and instructions for generating internal functions. It will also generate the GLSL main entry point, add global declarations of input and output variables to the IR, calculate the input of the HLSL entry point, call the HLSL entry point, and write the output to the global output variable.
- **Optimization**,optimization phase. Mainly through**do_optimization_pass**to perform multiple optimizations on IR, including direct function insertion, elimination of useless code, propagation of constants, elimination of common sub-expressions, etc.
- **Uniform packing**,global variable packing. Pack global uniform variables into arrays and keep mapping information so that the engine can bind parameters to the relevant parts of the uniform variable array.
- **Final optimization**,the final optimization stage. After packing uniform variables, a second optimization pass will be run on the IR to simplify the code generated when packing uniform variables.
- **Generate GLSL**.The final step is to convert the optimized IR into GLSL source code. In In addition to generating definitions of all constructs and uniform variable buffers and the source code itself, a mapping table is written in the comments at the beginning of the file.

The source code involved in the above description is in the Engine\Source\ThirdParty\hlslcc directory. The core files are:

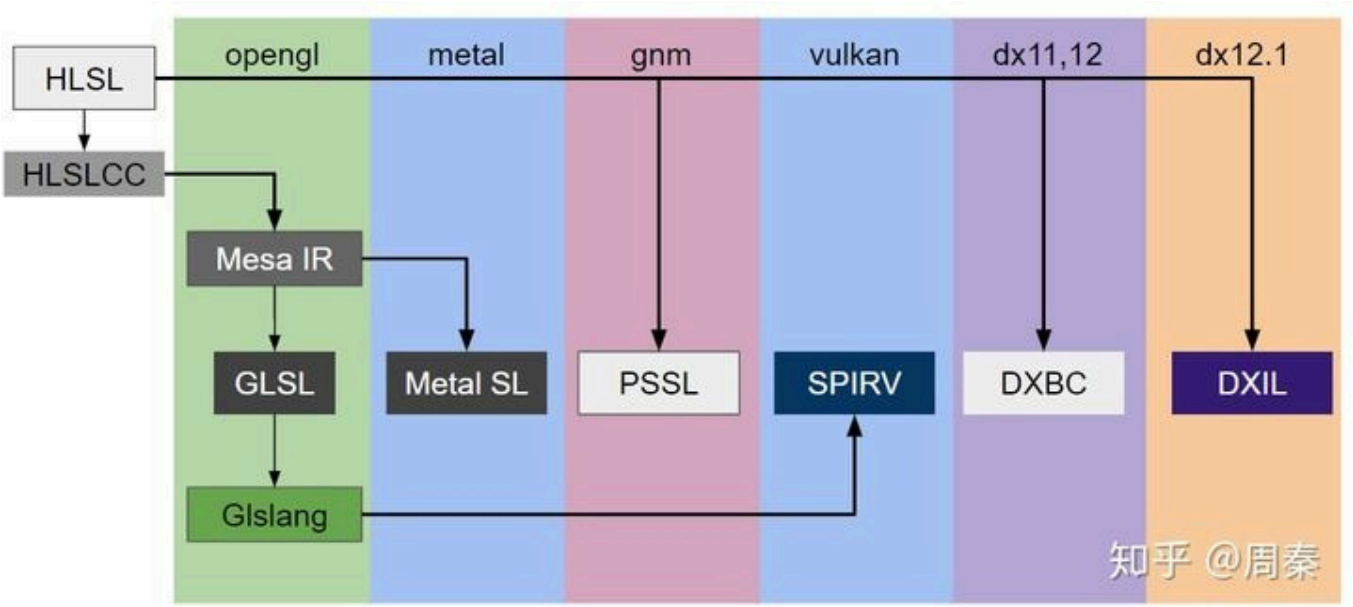
- ast.h
- glcpp-parse.h
- glsl_parser_extras.h
- hlsl_parser.h
- ir_optimization.h

The following are the core functions involved in the compilation phase:

Function Name	Analysis
apply_type_conversion	This function converts a value of one type to another type, if possible. Whether an implicit or explicit conversion is performed is controlled by a parameter.
arithmetic_result_type	This set of functions determines the result type of applying an operation to an input value.
validate_assignment	Determines whether an rvalue is assignable to an lvalue of a specific type. If necessary, allowed implicit conversions are applied.
do_assignment	Assigns an rvalue to an lvalue (if it can be done using validate_assignment).
ast_expression::hir	Convert an expression node in the AST to a set of IR instructions.
process_initializer	Applies an initialization expression to a variable.
ast_struct_specifier::hir	Construct an aggregate type to represent the declared structure.
ast_cbuffer_declaration::hir	Build a structure of constant buffer layout and store it as a uniform variable block.
process_mul	Special code to handle multiplication inside HLSL.
match_function_by_name	Look up a function signature based on the name and list of input arguments.
rank_parameter_lists	Compares two argument lists and assigns a numeric rank to indicate how well the two lists match. is a helper function that performs overload resolution: the lowest signature wins, and if any ranked signature has the same rank as the lowest signature, the function call is declared ranked ambiguous. A rank of zero indicates an exact match.
gen_texture_op	Handles method calls for built-in HLSL texture and sampler objects.

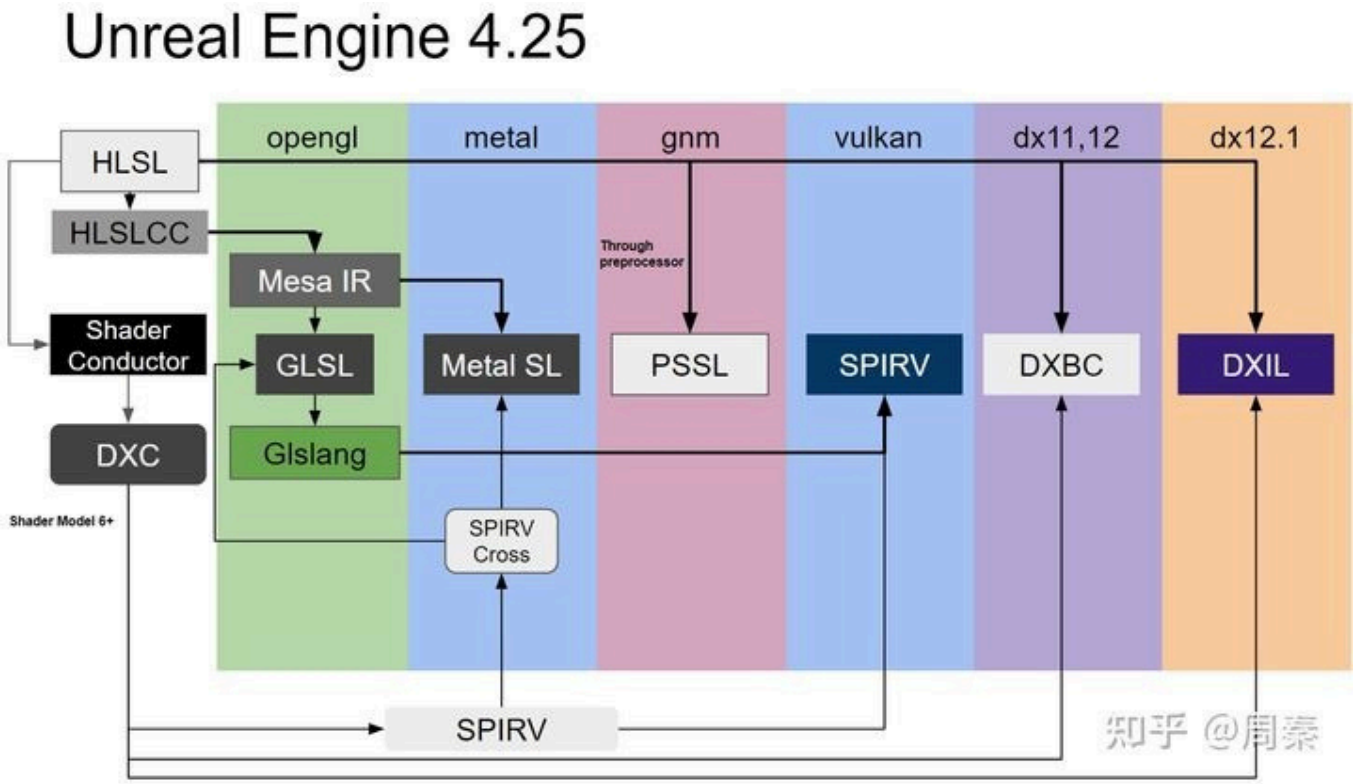
Function Name	Analysis
_mesa_glsl_initialize_fun Actions	Built-in functions that generate HLSL intrinsics. Most functions (such as sin and cos) generate IR code to perform the operation, but some functions (such as transpose and determinant) reserve function calls to defer the operation to be performed by the driver's GLSL compiler.

HLSLCC has gone through several iterations since the first version of UE4.3 to the current version 4.26. For example, in UE4.22, the cross-platform diagram of Shader is as follows:



UE4.22 shader cross-platform diagram, where Metal SL is translated from Mesa IR, and Vulkan is translated from Mesa IR-GLSL-GLSlang-SPIR-V multiple translations.

In UE4.25, the cross-platform diagram of Shader is as follows:



The biggest change in the shader cross-platform diagram of UE4.25 is the addition of Shader Conductor, which is then translated to Metal, Vulkan, DX and other platforms through DXC->SPIR-V.

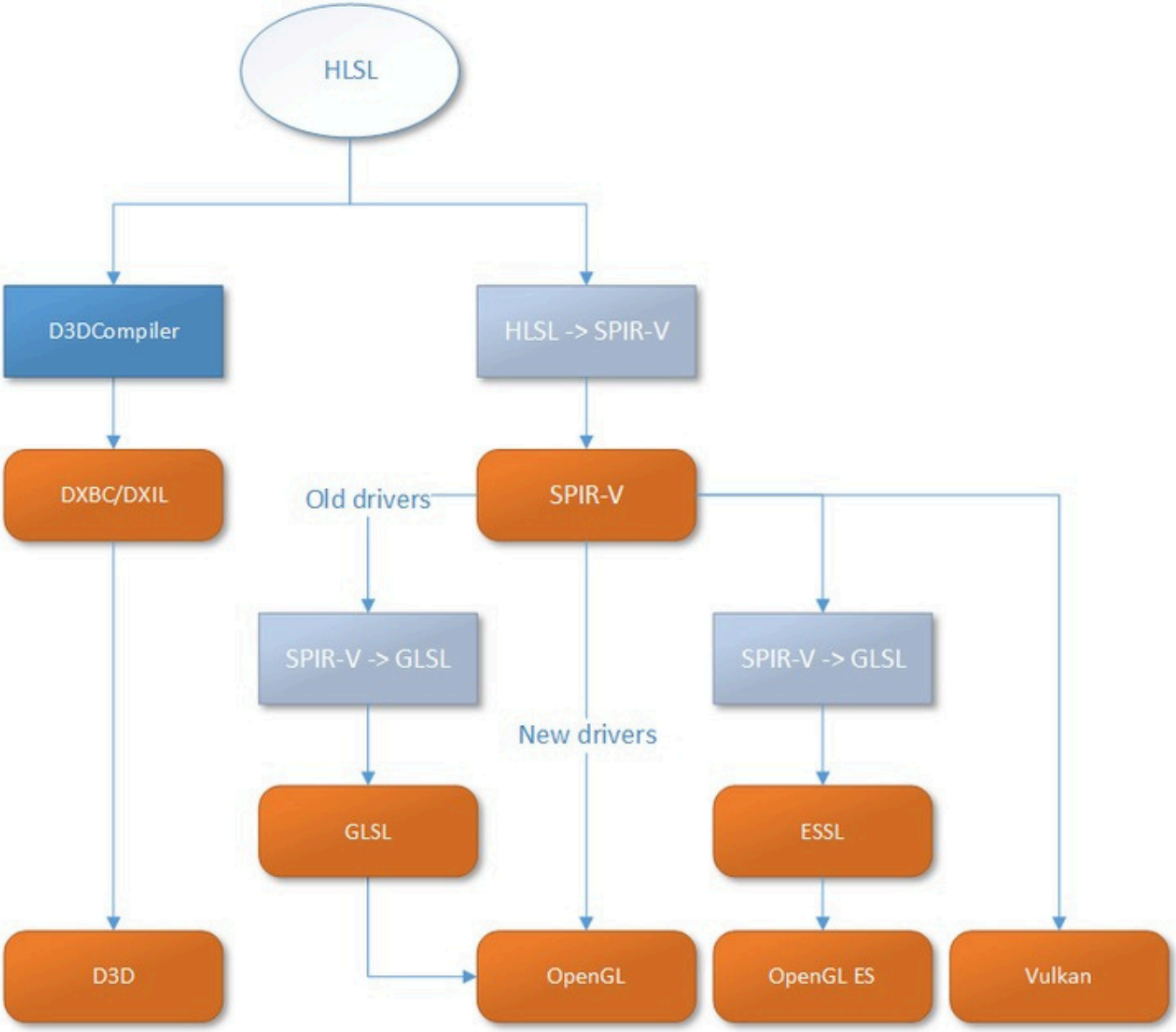
Therefore, the biggest change in UE4.25 is the addition of Shader Conductor, which is converted to SPIR-V to enable the transfer of platforms such as Metal and Vulkan.

Shader Conductor is also a third-party library, located in the Engine\Source\ThirdParty\ShaderConductor directory of the engine. Its core modules are:

- ShaderConductor.hpp
- ShaderConductor.cpp
- Native.h
- Native.cpp

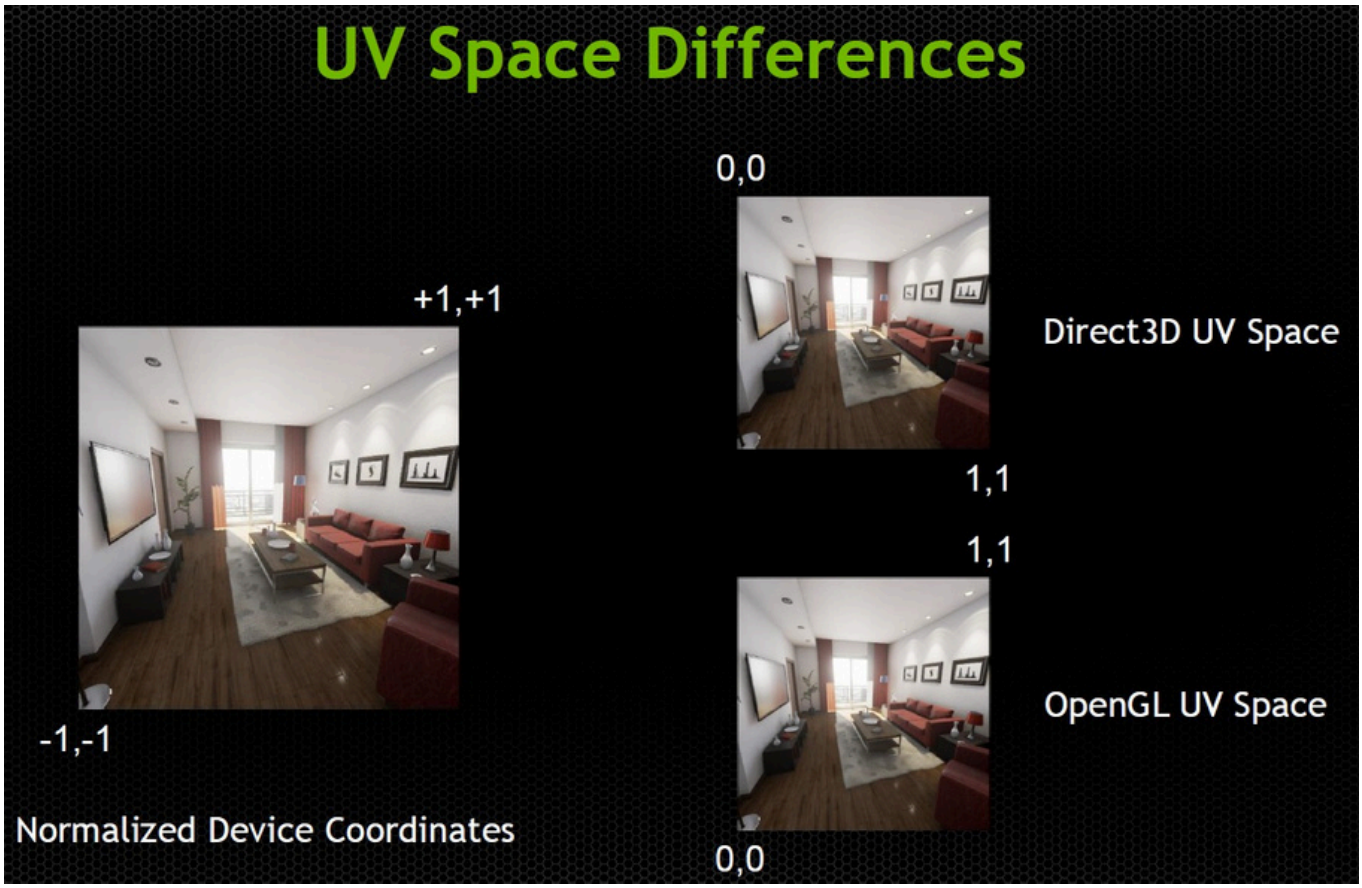
Shader Conductor also includes components such as DirectXShaderCompiler, SPIRV-Cross, SPIRV-Headers, and SPIRV-Tools.

The idea of UE4.25 is exactly the same as the Shader cross-platform solution of KlayGE by the rebel (Gong Minmin) :

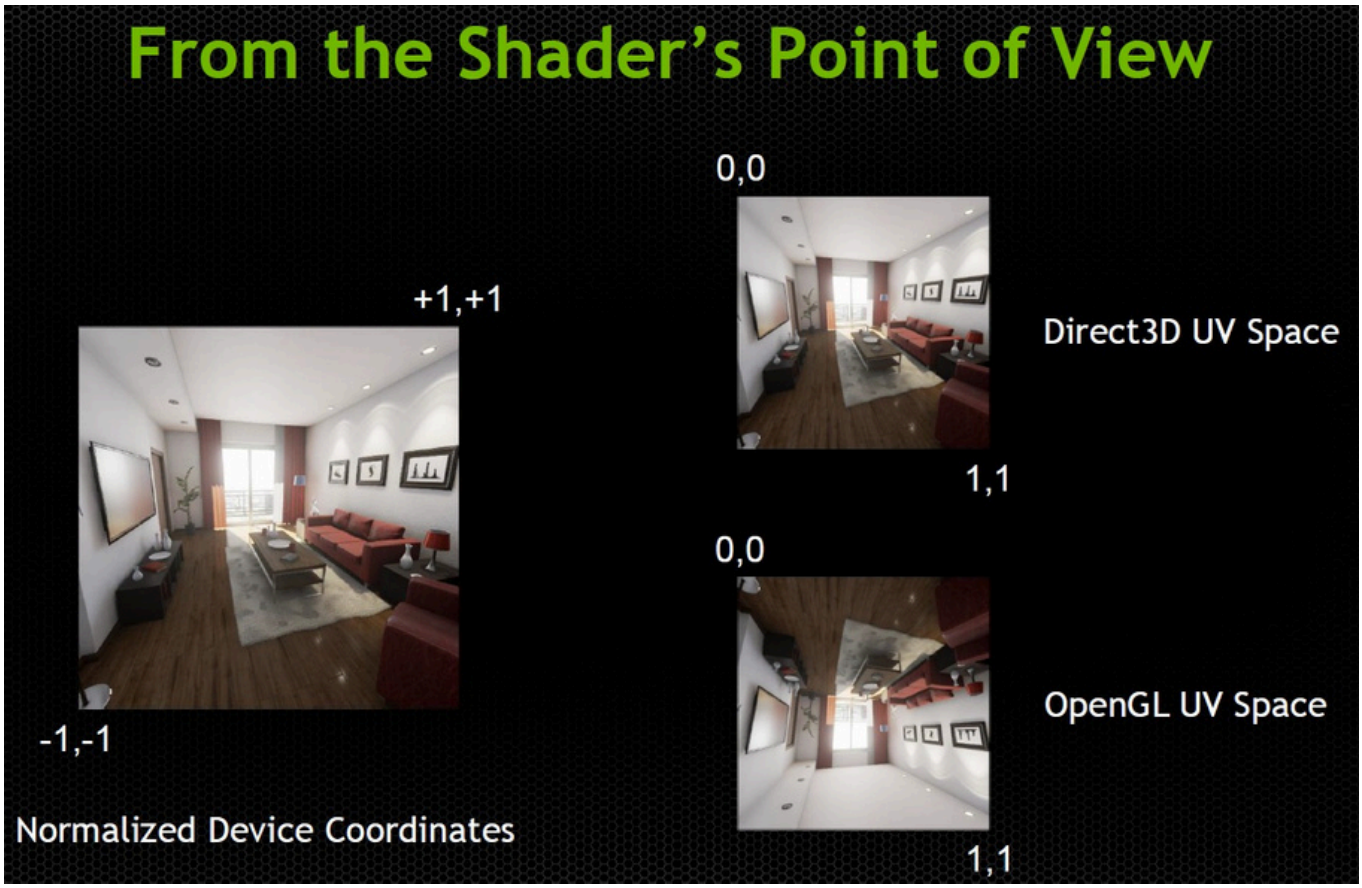


Vulkan not only has a new API, but also brings a new shader intermediate format SPIR-V. This is the most important step on the road to unified cross-platform shader compilation. Judging from the trend, more and more engines and renderers will use SPIR-V as the preferred cross-platform technology solution in the future.

Another small detail is that although Direct3D and OpenGL are consistent in standardized device coordinates, their coordinates in UV space are inconsistent:



In order to prevent shader developers from noticing this difference, UE uses a flipped image to force the UV coordinates to use a unified format:



The consequence of this is that the OpenGL texture is actually flipped vertically (as evidenced by the UE application on the OpenGL platform captured from RenderDoc), but it can be flipped again after

rendering. However, UE uses an upside down rendering method and integrates the upside down parameters into the projection matrix:

So it looks like both the normalized device coordinates and the texture under D3D are flipped vertically.

8.3.4 Shader Cache

There are two types of Shader caches. One is offline data stored in DDC, which is often used to speed up the efficiency of the editor and development stages. For details, see 8.3.1.2 FGlobalShaderMap. The other is the runtime Shader cache, which was taken over by FShaderCache in the early UE, but UE4.26 has canceled FShaderCache and replaced it with FShaderPipelineCache .

FShaderPipelineCache provides a new pipeline state object (PSO) logging, serialization, and precompilation mechanism. Caching pipeline state objects and serializing initializers to disk allows these states to be precompiled the next time the game runs, which can reduce lag. But FShaderPipelineCache relies on FShaderCodeLibrary, Share Material Shader Code, and PipelineFileCache on the RHI side.

Here is the definition of FShaderPipelineCache:

```
// Engine\Source\Runtime\RenderCore\Public\ShaderPipelineCache.h

classFShaderPipelineCache: public FTickableObjectRenderThread {

	//Compile the job structure.
	structCompileJob
	{
		FPipelineCacheFileFormatPSO		PSO;
		FShaderPipelineCacheArchive*		ReadRequests;
	};

public:
	//initializationFShaderPipelineCache.
	static void Initialize(EShaderPlatform Platform); //destroy
	FShaderPipelineCache static void Shutdown(); //Pause/resume
	package precompilation.

	static	void	PauseBatching();
	static	void	ResumeBatching();

	//Packaging Mode
	enum classBatchMode
	{
		Background,//The maximum package size isr.ShaderPipelineCache.BackgroundBatchSizeDecide. Fast,//
		The maximum package size isr.ShaderPipelineCache.BatchSizeDecide. Precompile//The maximum
		package size isr.ShaderPipelineCache.PrecompileBatchSizeDecide.
	};

	//Set and get data interface.
```

```
static void SetBatchMode(BatchMode Mode); static
        uint32    NumPrecompilesRemaining();
static    uint32 NumPrecompilesActive();

staticint32GetGameVersionForPSOFileCache();
static bool SetGameUsageMaskWithComparison(uint64 Mask, FPSOMaskComparisonFn
InComparisonFnPtr);
static bool IsBatchingPaused();

//OpenFShaderPipelineCache
static bool OpenPipelineFileCache(EShaderPlatform Platform);
static bool OpenPipelineFileCache(FStringconst& Name, EShaderPlatform Platform);

//Save/CloseFShaderPipelineCache
static bool SavePipelineFileCache(FPipelineFileCache::SaveMode Mode); static void
ClosePipelineFileCache();

//Constructor/destructor.
FShaderPipelineCache(EShaderPlatform virtual    Platform);
~FShaderPipelineCache();

// TickRelated interfaces.
bool IsTickable()          const;
//frameTick
void Tick(floatDeltaTime );
bool NeedsRenderingResumedForRenderingThreadTick()              const;

TStatIdGetStatId()const;

enumELibraryState
{
    Opened,
    Closed
};

//State change notification.
static void ShaderLibraryStateChanged(ELibraryState State, EShaderPlatform Platform, FStringconst& Name);


//Precompile context.
classFShaderCachePrecompileContext {

    boolbSlowPrecompileTask; public:

    FShaderCachePrecompileContext() : bSlowPrecompileTask(false) {} void
    SetPrecompilationIsSlowTask() { bSlowPrecompileTask =true; } bool IsPrecompilationSlowTask
    ()const{returnbSlowPrecompileTask; }
};

//Signal delegate function.
static    FShaderCachePreOpenDelegate&      GetCachePreOpenDelegate();
static    FShaderCacheOpenedDelegate&     GetCacheOpenedDelegate();
static    FShaderCacheClosedDelegate&     GetCacheClosedDelegate();
static    FShaderPrecompilationBeginDelegate&          GetPrecompilationBeginDelegate();
static    FShaderPrecompilationCompleteDelegate&          GetPrecompilationCompleteDelegate();

(.....)
```

```
private:
    //Package various precompiled data.
    staticFShaderPipelineCache*              ShaderPipelineCache;
    TArray<CompileJob>       ReadTasks;  TArray<CompileJob>
    TArray<FPipelineCachePSOHeader*>CompileTasks;
    TDoubleLinkedList<FPipelineCacheFileFormatPSORead*>
                                                    OrderedCompileTasks;
    TSet<uint32> CompiledHashes;                                    FetchTasks;


    FString FileName;
    EShaderPlatform      CurrentPlatform;
    FGuid CacheFileGuid;
    uint32    BatchSize;

    FShaderCachePrecompileContext ShaderCachePrecompileContext;

    FCriticalSection Mutex; TArray<FPipelineCachePSOHeader>
    PreFetchedTasks; TArray<CompileJob> ShutdownReadCompileTasks;
    TDoubleLinkedList<FPipelineCacheFileFormatPSORead*>
                                                                    ShutdownFetchTasks;

    TMap<FBlendStateInitializerRHI, FRHIBlendState*> BlendStateCache; TMap<FRasterizerStateInitializerRHI,
    FRHIRasterizerState*> RasterizerStateCache; TMap<FDepthStencilStateInitializerRHI, FRHIDepthStencilState*>
    DepthStencilStateCache;

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

The data obtained by pre-compiling the FShaderPipelineCache package is saved in the Saved directory of the project directory, with the suffix .upipelinecache:

```
// Engine\Source\Runtime\RHI\Private\PipelineFileCache.cpp

bool FPipelineFileCache::SavePipelineFileCache(FStringconst& Name, SaveMode Mode) {

    boolbOk =false;

    //Must be turned onPipelineFileCacheAnd recordPSOto the file cache.
    if(IsPipelineFileCacheEnabled() && LogPSOtoFileCache()) {

        if(FileCache)
        {
            //The name of the saved platform.
            FName PlatformName = FileCache->GetPlatformName(); //Saved
            directory
            FString Path = FPaths::ProjectSavedDir() /
FString::Printf(TEXT("%s_%s.upipelinecache"), *Name, *PlatformName.ToString());
            //Execute the save operation.
            bOk = FileCache->SavePipelineFileCache(Path, Mode, Stats, NewPSOs,
RequestedOrder, NewPSOUsage);

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

```
    returnbOk;
}
```

Since it is a runtime-effective shader cache, it must be integrated into the UE runtime module. In fact, its control is completed in FEngineLoop:

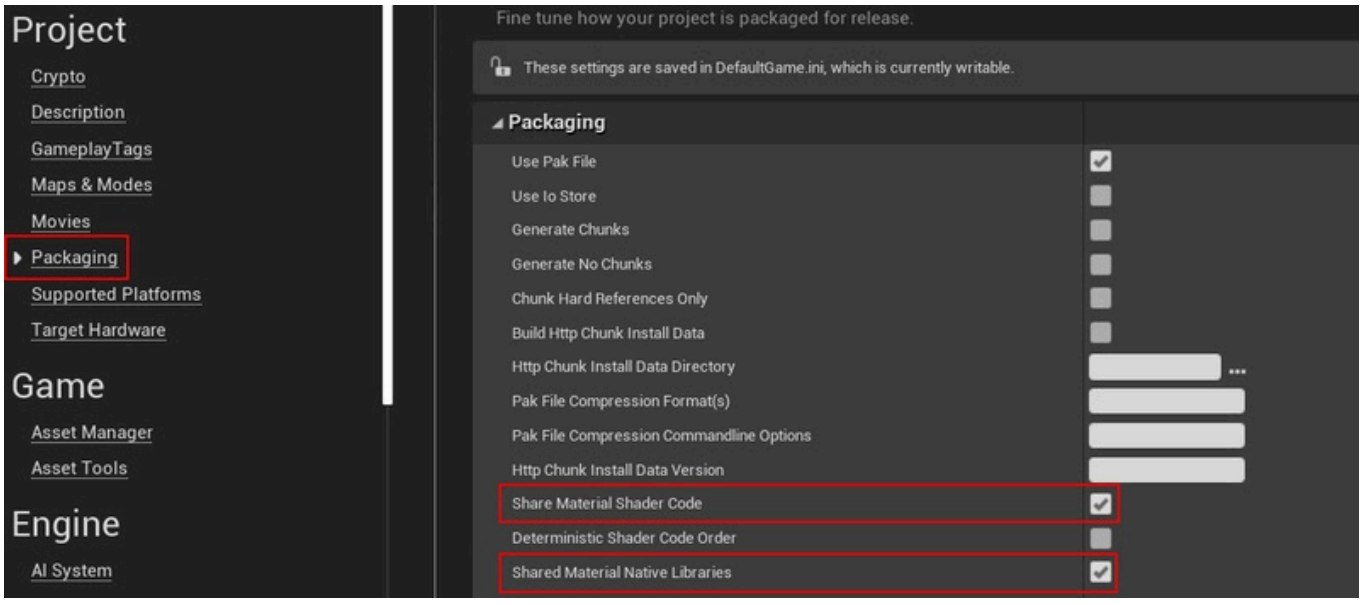
```
int32FEngineLoop::PreInitPreStartupScreen(constTCHAR* CmdLine) {

    (......)

    {
        boolbUseCodeLibrary = FPlatformProperties::RequiresCookedData() ||
        GAllowCookedDataInEditorBuilds;
        if(bUseCodeLibrary)
        {
            {
                FShaderCodeLibrary::InitForRuntime(GMaxRHIShaderPlatform);
            }

    #if!UE_EDITOR
            // Cooked data only - but also requires the code library - game only if
            (FPlatformProperties::RequiresCookedData()) {

                //initializationFShaderPipelineCache
                FShaderPipelineCache::Initialize(GMaxRHIShaderPlatform);
            }
    #endif//!UE_EDITOR
        }
    }

    (......)
}

int32FEngineLoop::PreInitPostStartupScreen(constTCHAR* CmdLine) {

    (......)

    IInstallBundleManager* BundleManager =
    IInstallBundleManager::GetPlatformInstallBundleManager();
    if(BundleManager == nullptr || BundleManager->IsNullInterface()) {

        (......)

        {
            //Open the game library containing the material shaders.
            FShaderCodeLibrary::OpenLibrary(FApp::GetProjectName(),
FPaths::ProjectContentDir());
            for(constFString& RootDir : FPlatformMisc::GetAdditionalRootDirectories()) {

                FShaderCodeLibrary::OpenLibrary(FApp::GetProjectName(),
FPaths::Combine(RootDir, FApp::GetProjectName(), TEXT("Content")));
            }

            //OpenFShaderPipelineCache.
            FShaderPipelineCache::OpenPipelineFileCache(GMaxRHIShaderPlatform);
        }
```

```
    }

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

In addition, GameEngine will also respond to the command line to continue and pause pre-compiled packaging at runtime. Once the actual FShaderPipelineCache is ready, the RHI layer can respond to its actual and signal, taking Vulkan's FVulkanPipelineStateCacheManager as an example:

```
// Engine\Source\Runtime\VulkanRHI\Private\VulkanPipeline.h

classFVulkanPipelineStateCacheManager {

    (......)

private:
    //trackShaderPipelineCacheThe precompiled delegate.
    void OnShaderPipelineCacheOpened(FStringconst& Name, EShaderPlatform Platform, uint32 Count,constFGuid&
VersionGuid, FShaderPipelineCache::FShaderCachePrecompileContext& ShaderCachePrecompileContext);

    void OnShaderPipelineCachePrecompilationComplete(uint32 Count,doubleSeconds,const
FShaderPipelineCache::FShaderCachePrecompileContext& ShaderCachePrecompileContext);

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

If you want to enable Shader Pipeline Cache, you need to check the following two items in the project configuration (enabled by default):



There are some command line variables that can set the properties of the Shader Pipeline Cache:

Command Line	effect
r.ShaderPipelineCache.Enabled	Enable Shader Pipeline Cache to load existing data from disk and precompile.

Command Line	effect
r.ShaderPipelineCache.BatchSize / BackgroundBatchSize	You can set the size of different batch modes.
r.ShaderPipelineCache.LogPSO	Enable PSO recording under Shader Pipeline Cache.
r.ShaderPipelineCache.SaveAfter PSOsLogged	Set the expected number of PSO records, and they will be saved automatically when this number is reached.

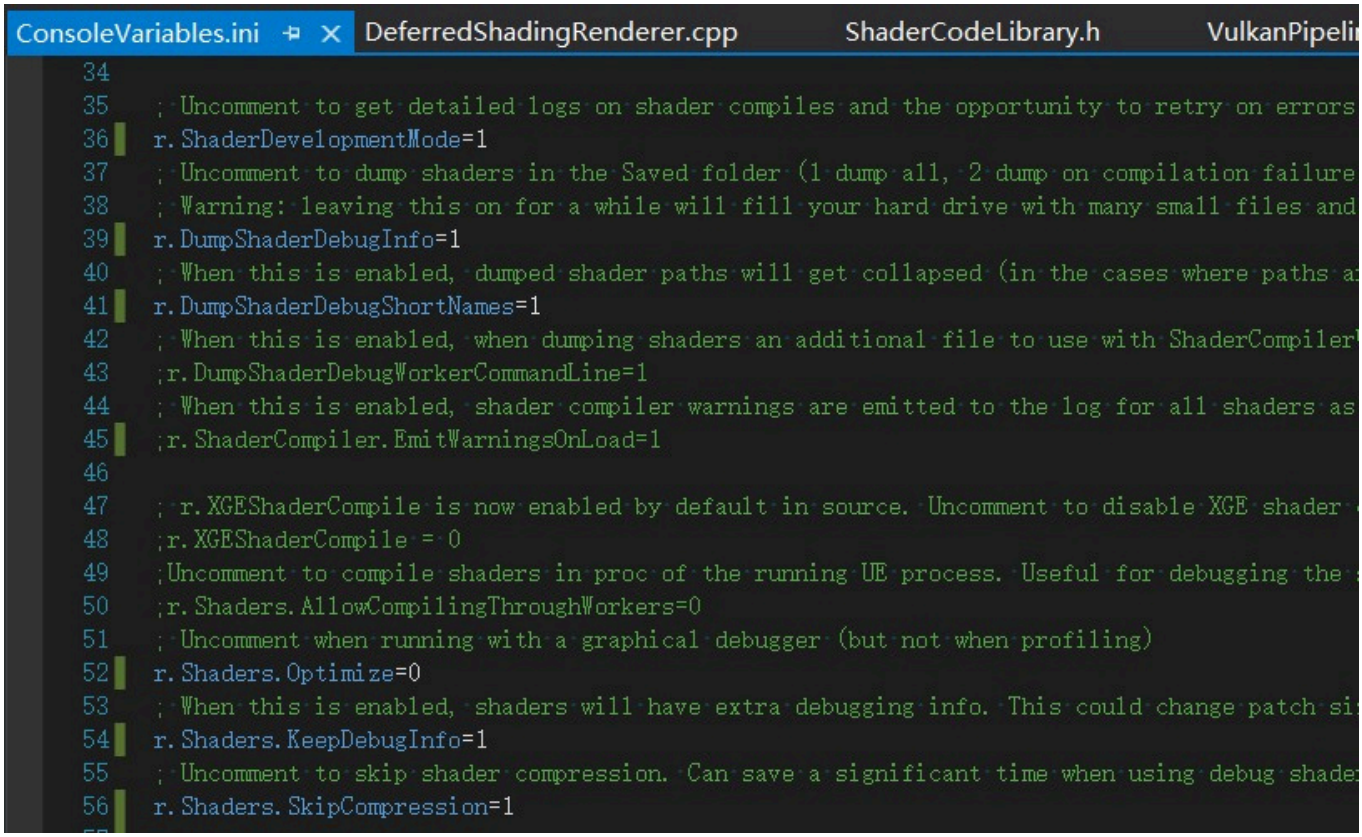
In addition, in GGameIni or GGameUserSettingsIni, the Shader Pipeline Cache uses the field [ShaderPipelineCache.CacheFile] to store information.

8.4 Shader Development

This chapter will describe Shader development cases, debugging techniques, and optimization techniques.

8.4.1 Shader Debugging

If the project is in the development stage, it is best to change the Shader compilation option to Development, which can be achieved by modifying the following configuration in Engine\Config\ConsoleVariables.ini:

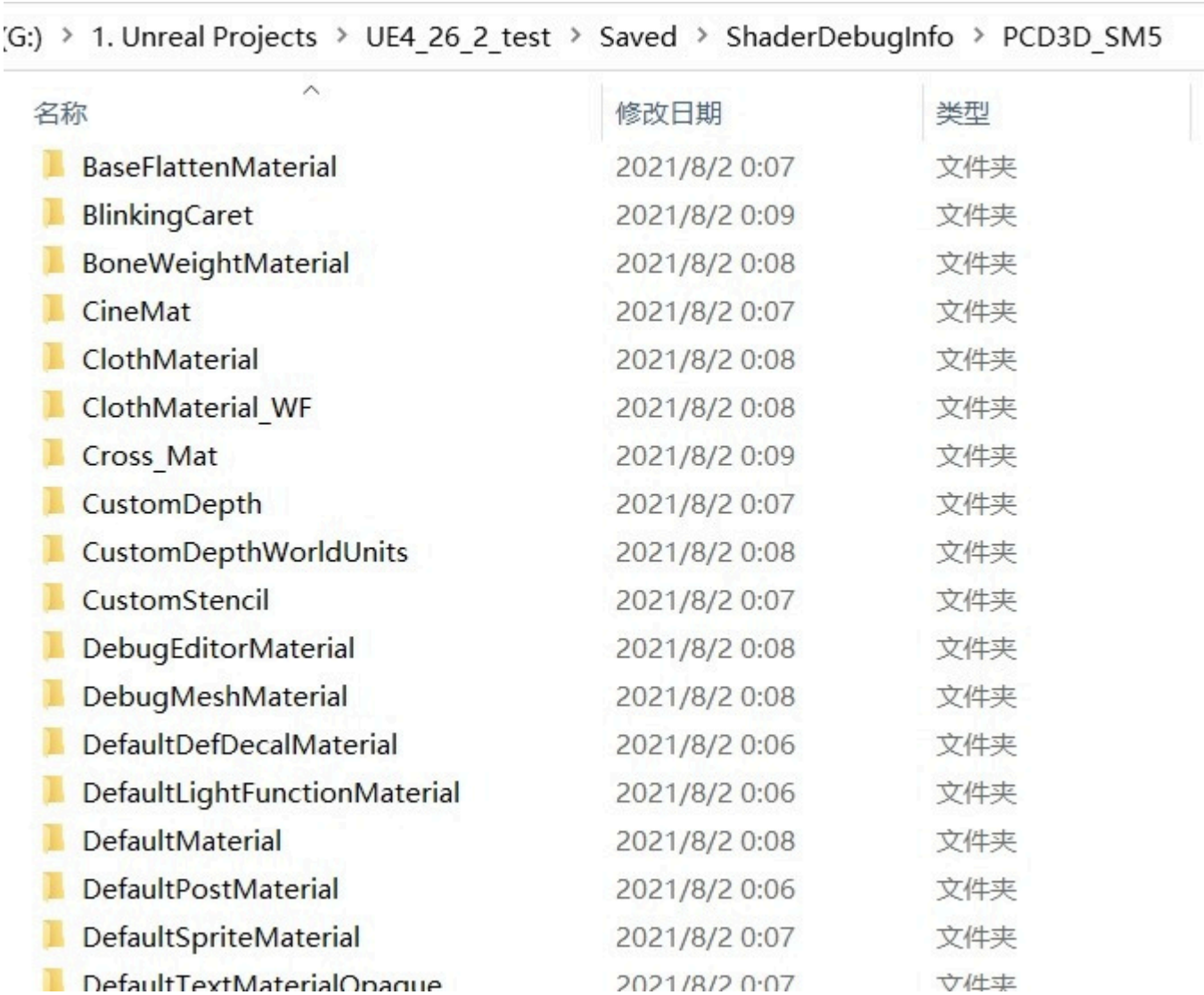


Just remove the semicolon in front of the command variable. Their meanings are as follows:

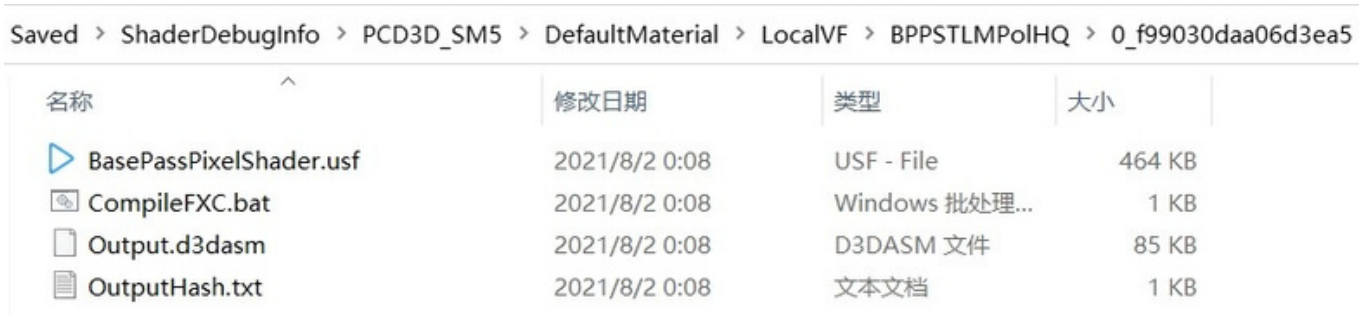
Command Line	Analysis
r.ShaderDevelopmentMode=1	Get detailed logs about shader compilation and the opportunity to retry on errors.
r.DumpShaderDebugInfo=1	Saves all compiled shader files to disk in the directory ProjectName/Saved/ShaderDebugInfo. Contains source files, preprocessed versions, and a batch file (for compiling preprocessed versions using the compiler's equivalent command line options).
r.DumpShaderDebugShortNames=1	The saved Shader path will be simplified.
r.Shaders.Optimize=0	Disable shader optimization so that shader debug information is preserved.
r.Shaders.KeepDebugInfo=1	Preserve debugging information, which is particularly useful when used with frame capture tools such as RenderDoc.
r.Shaders.SkipCompression=1	Ignoring shader compression can save time debugging shaders.

After enabling the above commands, you can use RenderDoc to capture frames to fully view the shader's variables and HLSL code (if not enabled, it will be assembly instructions), and you can also debug step by step. This can effectively improve the efficiency of shader development and debugging.

After r.DumpShaderDebugInfo is turned on, modify a line of code in UE's built-in shader at will (for example, add a space in Common.ush), restart the UE editor, the shader will be recompiled, and after completion, useful debugging information will be generated in the ProjectName/Saved/ShaderDebugInfo directory:



Open a specific material shader directory and you will find the source file, preprocessed version, batch file and hash value:



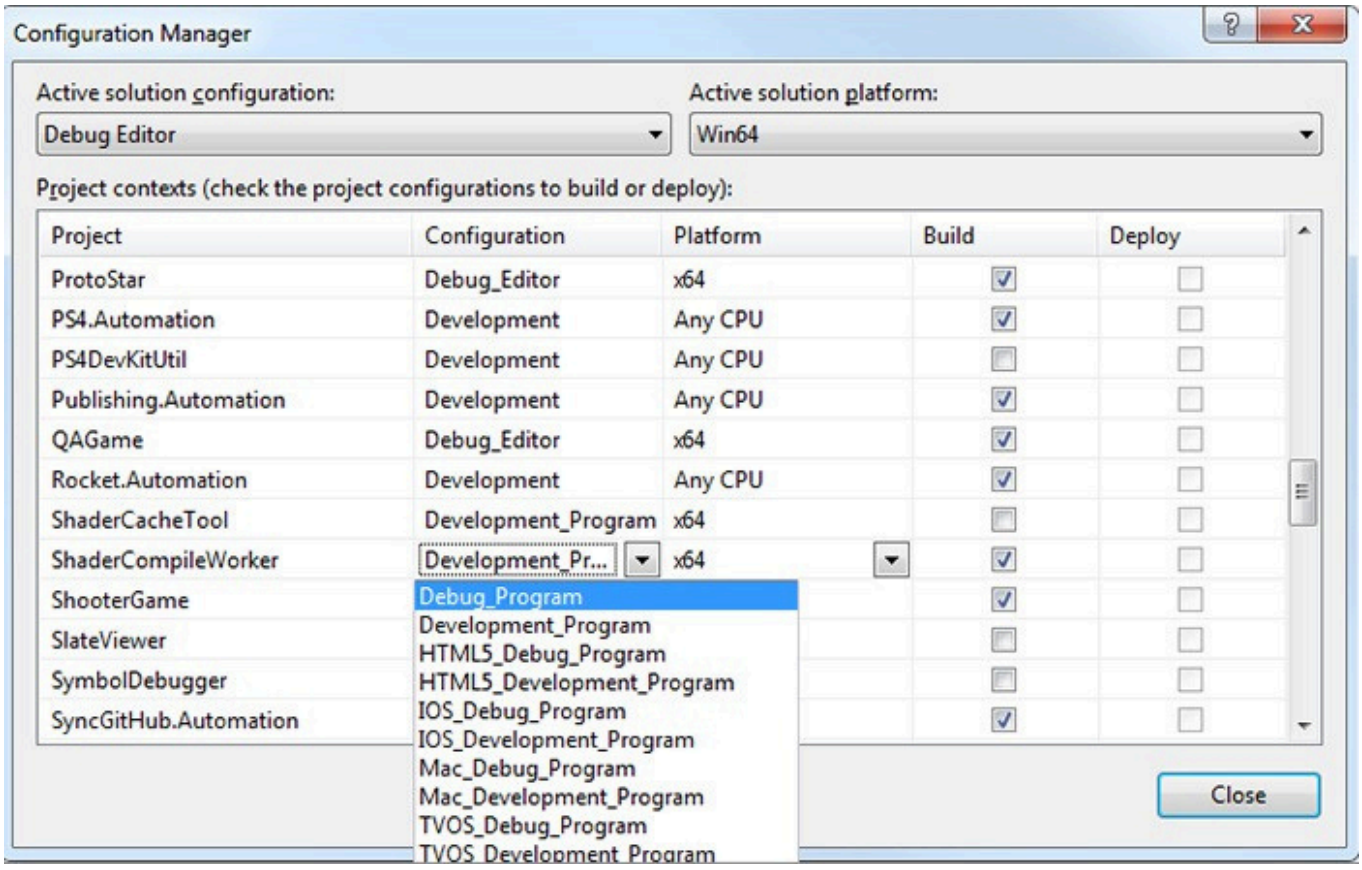
In addition, if you modify some Shader files (such as BasePassPixelShader.ush), you do not need to restart the UE editor. You can enterRecompileShaderscommands in the console to recompile the specified shader file.RecompileShadersThe specific meanings are as follows:

Order	Analysis
RecompileShaders all	Compile all shaders with modified source code, including global, material, and meshmaterial.

Order	Analysis
RecompileShaders changed	Compile the source code with the modified shader.
RecompileShaders global	Compile the source code with the modified global shader.
RecompileShaders Material	Compile the source code with the modified material shader.
RecompileShaders Material	Compiles the material with the specified name.
RecompileShaders	Compile the shader source file in the specified path.

Before executing the above commands, you must save the changes to the shader file.

Alternatively, to build the project in debug mode, you can set the Solution Properties (Visual Studio: Build -> Configuration Manager) of ShaderCompileWorker to Debug_Program:



This way you can use ShaderCompileWorker (SCW) to add the Shader debug command line:

```
PathToGeneratedUsfFile-directcompile-format=ShaderFormat -ShaderType -entry=EntryPoint
```

- *PathToGeneratedUsfFile* is the final usf file in the ShaderDebugInfo folder.

- *ShaderFormat*is the shader platform format you want to debug (in this case, this is PCD3D_SM5).
- *ShaderType*is an entry in vs/ps/gs/hs/ds/cs and corresponds to the "vertex", "pixel", "geometry", "hull", "domain" and "compute" shader types respectively. *EntryPoint*is
- the function name of the entry point for this shader in the usf file.

For example:

```
<ProjectPath>\Saved\ShaderDebugInfo\PCD3D_SM5\M_Egg\LocalVF\BPPSFNoLMPolicy\BasePassPixelS hader.usf-format
=PCD3D_SM5 -ps -entry=Main
```

You can set a breakpoint on the CompileD3DShader() function in D3D11ShaderCompiler.cpp and run SCW from the command line to understand how to call the platform compiler:

```
// Engine\Source\Developer\Windows\ShaderFormatD3D\Private\D3DShaderCompiler.cpp

void CompileD3DShader(constFShaderCompilerInput& Input, FShaderCompilerOutput& Output,
FShaderCompilerDefinitions& AdditionalDefines,constFString& WorkingDirectory, ELanguage Language)

{
	FString PreprocessedShaderSource;
	const boolbIsRayTracingShader = IsRayTracingShader(Input.Target); const boolbUseDXC
	= bIsRayTracingShader
		||Input.Environment.CompilerFlags.Contains(CFLAG_WaveOperations)
		|| Input.Environment.CompilerFlags.Contains(CFLAG_ForceDXC);
	constTCHAR* ShaderProfile = GetShaderProfileName(Input.Target, bUseDXC);

	if(!ShaderProfile)
	{
		Output.Errors.Add(FShaderCompilerError(TEXT("Unrecognized return;		shader	frequency")));

	}

	//Set additional definitions.
	AdditionalDefines.SetDefine(TEXT("COMPILER_HLSL"),1);

	if(bUseDXC)
	{
		AdditionalDefines.SetDefine(TEXT("PLATFORM_SUPPORTS_SM6_0_WAVE_OPERATIONS"),		1);
		AdditionalDefines.SetDefine(TEXT("PLATFORM_SUPPORTS_STATIC_SAMPLERS"),1);
	}

	if(Input.bSkipPreprocessedCache) {

		if(!FFileHelper::LoadFileToString(PreprocessedShaderSource,
* Input.VirtualSourceFilePath)) {

			return;
		}

		//Remove constants, since this is debug mode only.
		CrossCompiler::CreateEnvironmentFromResourceTable(PreprocessedShaderSource,
		(FShaderCompilerEnvironment&)Input.Environment);
	}
```

```
	else
	{
		if(!PreprocessShader(PreprocessedShaderSource, Output, Input, AdditionalDefines)) {

			return;
		}
	}

	GD3DAllowRemoveUnused =
Input.Environment.CompilerFlags.Contains(CFLAG_ForceRemoveUnusedInterpolators) ?1:0;

	FString EntryPointName = Input.EntryPointName;

	Output.bFailedRemovingUnused =false;
	if(GD3DAllowRemoveUnused ==1&& Input.Target.Frequency == SF_Vertex &&
	Input.bCompilingForShaderPipeline)
	{
		//Always increaseSV_Position
		TArray<FString> UsedOutputs = Input.UsedOutputs;
		UsedOutputs.AddUnique(TEXT("SV_POSITION"));

		//Any output-only system syntax cannot be deleted.
		TArray<FString>  Exceptions;  Exceptions.AddUnique(TEXT(
		"SV_ClipDistance"));          Exceptions.AddUnique(TEXT(
		"SV_ClipDistance0"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance1"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance2"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance3"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance4"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance5"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance6"));         Exceptions.AddUnique(TEXT(
		"SV_ClipDistance7"));


		Exceptions.AddUnique(TEXT("SV_CullDistance"));
		Exceptions.AddUnique(TEXT("SV_CullDistance0"));
		Exceptions.AddUnique(TEXT("SV_CullDistance1"));
		Exceptions.AddUnique(TEXT("SV_CullDistance2"));
		Exceptions.AddUnique(TEXT("SV_CullDistance3"));
		Exceptions.AddUnique(TEXT("SV_CullDistance4"));
		Exceptions.AddUnique(TEXT("SV_CullDistance5"));
		Exceptions.AddUnique(TEXT("SV_CullDistance6"));
		Exceptions.AddUnique(TEXT("SV_CullDistance7"));


		DumpDebugShaderUSF(PreprocessedShaderSource, Input);

		TArray<FString> Errors;
		if(!RemoveUnusedOutputs(PreprocessedShaderSource, UsedOutputs, Exceptions, EntryPointName,
		Errors))
		{
			DumpDebugShaderUSF(PreprocessedShaderSource, Input); UE_LOG(LogD3D11ShaderCompiler,
			Warning, TEXT("Failed to Remove unused outputs
[%s]!"), *Input.DumpDebugInfoPath);
			for(int32 Index =0; Index < Errors.Num(); ++Index) {

				FShaderCompilerError NewError;
				NewError.StrippedErrorMessage = Errors[Index];
				Output.Errors.Add(NewError);
			}
```

```
                Output.bFailedRemovingUnused =true;
            }
        }

        FShaderParameterParser ShaderParameterParser;
        if(!ShaderParameterParser.ParseAndMoveShaderParametersToRootConstantBuffer(
            Input, Output, PreprocessedShaderSource, IsRayTracingShader(Input.Target) ?
            TEXT("cbuffer") : nullptr))
        {
            return;
        }

        RemoveUniformBuffersFromSource(Input.Environment, PreprocessedShaderSource);

        uint32 CompileFlags = D3D10_SHADER_ENABLE_BACKWARDS_COMPATIBILITY
            //    UnzipuniforMatrix row-first (row-major),To matchCPUlayout.
            |    D3D10_SHADER_PACK_MATRIX_ROW_MAJOR;

        if(Input.Environment.CompilerFlags.Contains(CFLAG_Debug)) {

            //Add debug flags.
            CompileFlags |= D3D10_SHADER_DEBUG | D3D10_SHADER_SKIP_OPTIMIZATION;
        }
        else
        {
            if(Input.Environment.CompilerFlags.Contains(CFLAG_StandardOptimization)) {

                CompileFlags      |=  D3D10_SHADER_OPTIMIZATION_LEVEL1;
            }
            else
            {
                CompileFlags      |=  D3D10_SHADER_OPTIMIZATION_LEVEL3;
            }
        }

        for(int32 FlagIndex =0; FlagIndex < Input.Environment.CompilerFlags.Num(); FlagIndex++)

        {
            //The cumulative mark is set toshader.
            CompileFlags |=
TranslateCompilerFlagD3D11((ECompilerFlags)Input.Environment.CompilerFlags[FlagIndex]);
        }

        TArray<FString> FilteredErrors; if
        (bUseDXC)
        {
            if(!CompileAndProcessD3DShaderDXC(PreprocessedShaderSource, CompileFlags, Input, EntryPointName,
ShaderProfile, Language,false, FilteredErrors, Output))
            {
                if(!FilteredErrors.Num()) {

                    FilteredErrors.Add(TEXT("Compile Failed without errors!"));
                }
            }

CrossCompiler::FShaderConductorContext::ConvertCompileErrors(MoveTemp(FilteredErrors), Output.Errors);

        }
```

```
	else
	{
		//	Override the default compiler path to a newer onedll.
		FString CompilerPath = FPaths::EngineDir();

CompilerPath.Append(TEXT("Binaries/ThirdParty/Windows/DirectX/x64/d3dcompiler_47.dll"));

		if(!CompileAndProcessD3DShaderFXC(PreprocessedShaderSource, CompilerPath, CompileFlags,
Input, EntryPointName, ShaderProfile,false, FilteredErrors, Output))
		{
			if(!FilteredErrors.Num()) {

				FilteredErrors.Add(TEXT("Compile Failed without errors!"));
			}
		}

		//Handle errors.
		for(int32 ErrorIndex =0; ErrorIndex < FilteredErrors.Num(); ErrorIndex++) {

			constFString& CurrentError = FilteredErrors[ErrorIndex];
			FShaderCompilerError NewError;

			//Extract filename and line number from FXC output with format:
			// "d:\UE4\Binaries\BasePassPixelShader(30,7): error X3000: invalid target or string"
usage
			int32 FirstParenIndex = CurrentError.Find(TEXT("(")); int32
			LastParenIndex = CurrentError.Find(TEXT("):")); if(FirstParenIndex !=
			INDEX_NONE &&
				LastParenIndex != INDEX_NONE &&
				LastParenIndex > FirstParenIndex)
			{
				// Extract and store error message with source filename NewError.ErrorVirtualFilePath =
				CurrentError.Left(FirstParenIndex); NewError.ErrorLineString =
				CurrentError.Mid(FirstParenIndex +1,
LastParenIndex - FirstParenIndex - FCString::Strlen(TEXT("(")));
				NewError.StrippedErrorMessage = CurrentError.Right(CurrentError.Len() -
LastParenIndex - FCString::Strlen(TEXT("):")));
			}
			else
			{
				NewError.StrippedErrorMessage = CurrentError;
			}
			Output.Errors.Add(NewError);
		}
	}

	const boolbDirectCompile = FParse::Param(FCommandLine::Get(), TEXT("directcompile")); if(bDirectCompile)

	{
		for(constauto& Error : Output.Errors) {

			FPlatformMisc::LowLevelOutputDebugStringf(TEXT("%s\n"),
* Error.GetErrorStringWithLineMarker()); }

	}

	ShaderParameterParser.ValidateShaderParameterTypes(Input, Output);
```

```
    if(Input.ExtraSettings.bExtractShaderSource) {

        Output.OptionalFinalShaderSource = PreprocessedShaderSource;
    }
}
```

In addition, if you do not use tools such as RenderDoc, you can convert the data that needs to be debugged into a color value in a reasonable range to see whether its value is normal, for example:

```
//Divide the world coordinate by a value within a range and
output it to color. OutColor = frac(WorldPosition /1000);
```

Combined with the RecompileShaders directive, this technique is very useful and efficient.

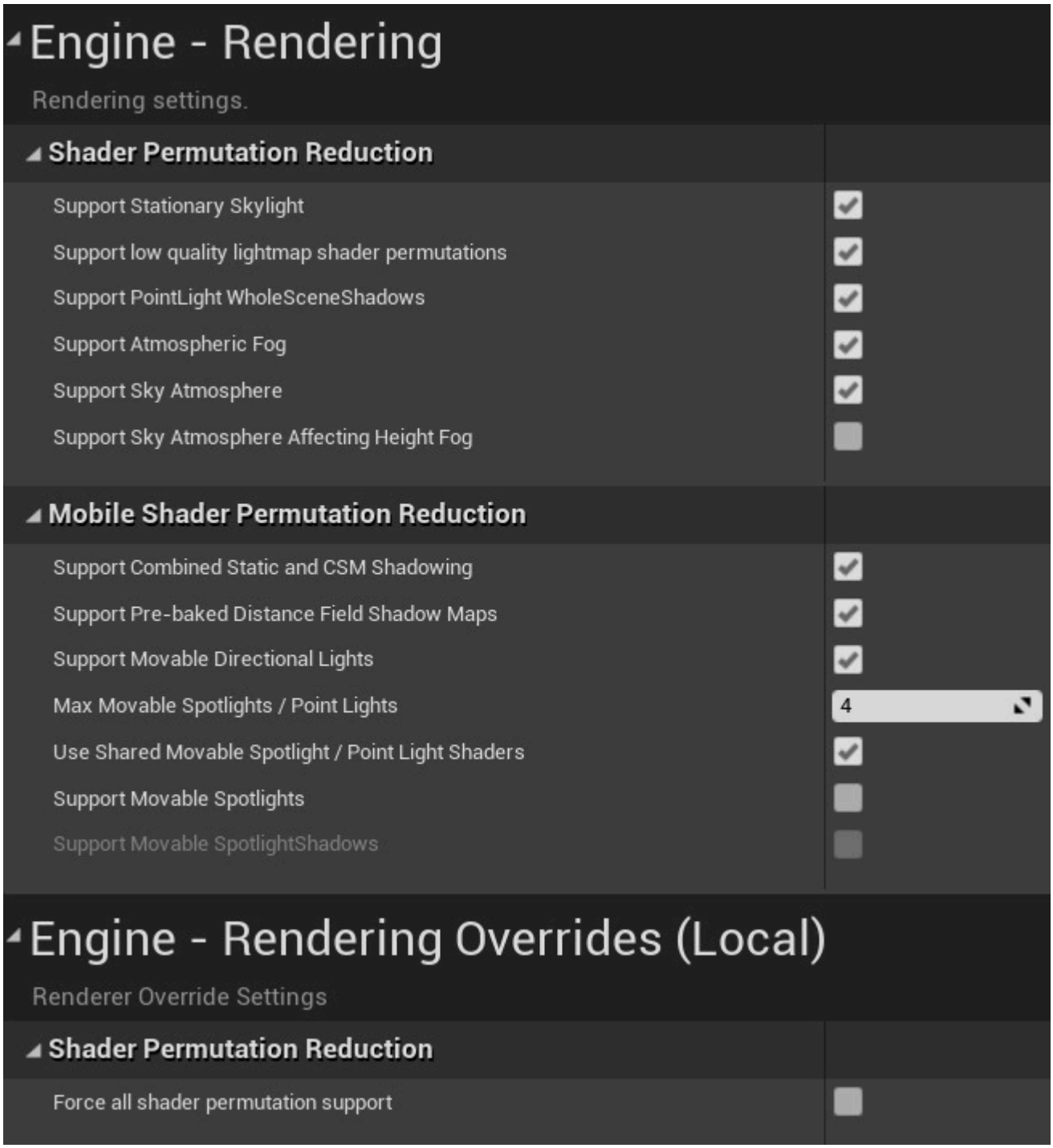
8.4.2 Shader Optimization

There are many different rendering optimization techniques, ranging from the system, architecture, and engineering levels to specific statements, but this section focuses on general Shader optimization techniques in the UE environment.

8.4.2.1 Optimizing Arrangement

Since UE adopts the Uber Shader design, the same shader source file contains a large number of macro definitions, which can be combined into a large number of target codes according to different values, and these macros are usually controlled by permutations. If we can effectively control the number of permutations, we can also reduce the number and time of shader compilation and improve runtime efficiency.

In the factory configuration there are some options that can be unchecked to reduce the number of permutations:



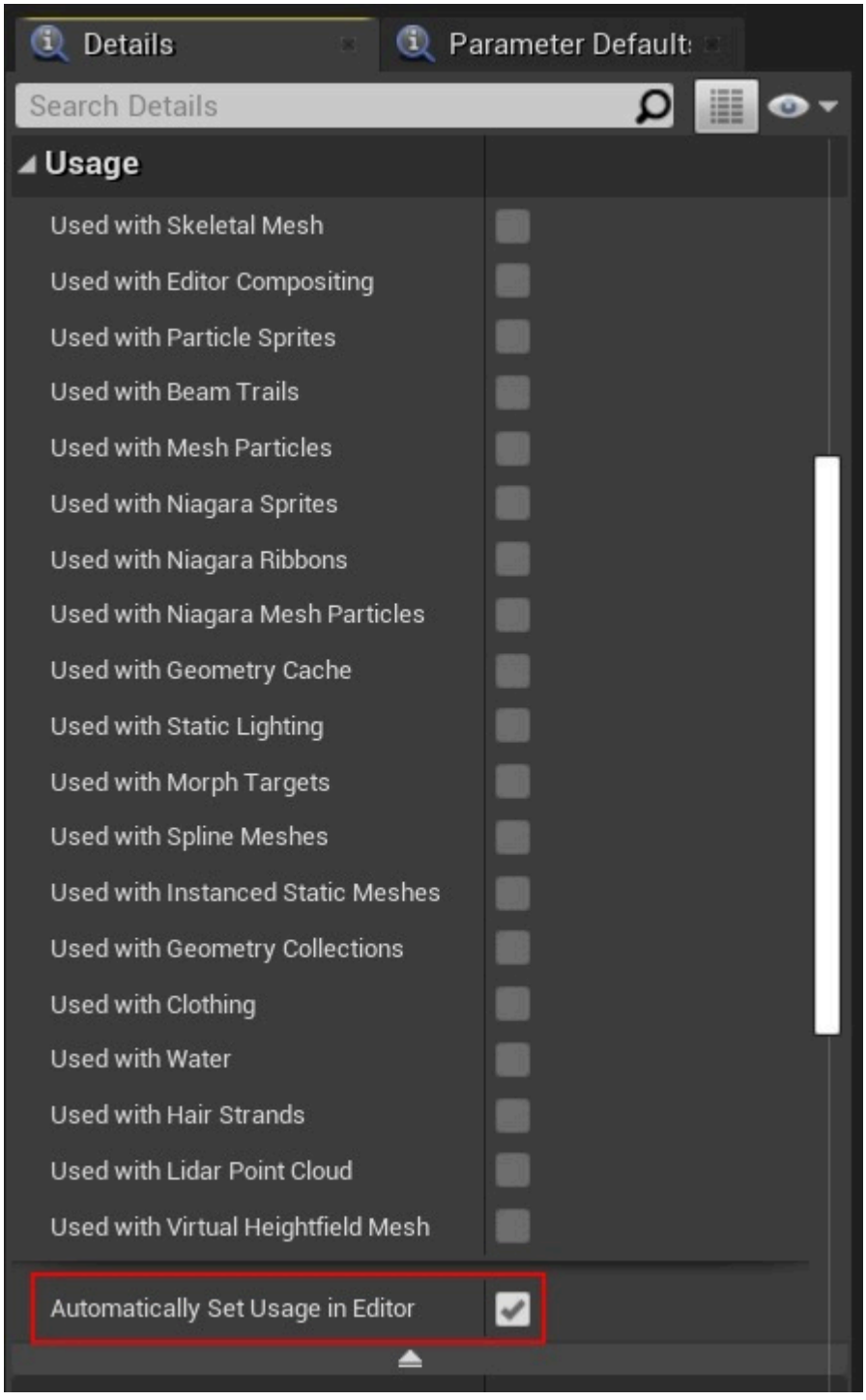
But please note that if you uncheck the box, it means that the engine will disable this function. You need to make trade-offs and choices based on the actual situation, and you should not optimize for
for the sake of optimization.

In addition, many built-in types in the engine rendering module provide the ShouldCompilePermutation interface so that the compiler can query the compiled object whether a certain arrangement needs to be compiled before formal compilation. If false is returned, the compiler will ignore the arrangement, thereby reducing the number of shaders. Types that support
ShouldCompilePermutation include but are not limited to:

- FShader
- FGlobalShader
- FMaterialShader

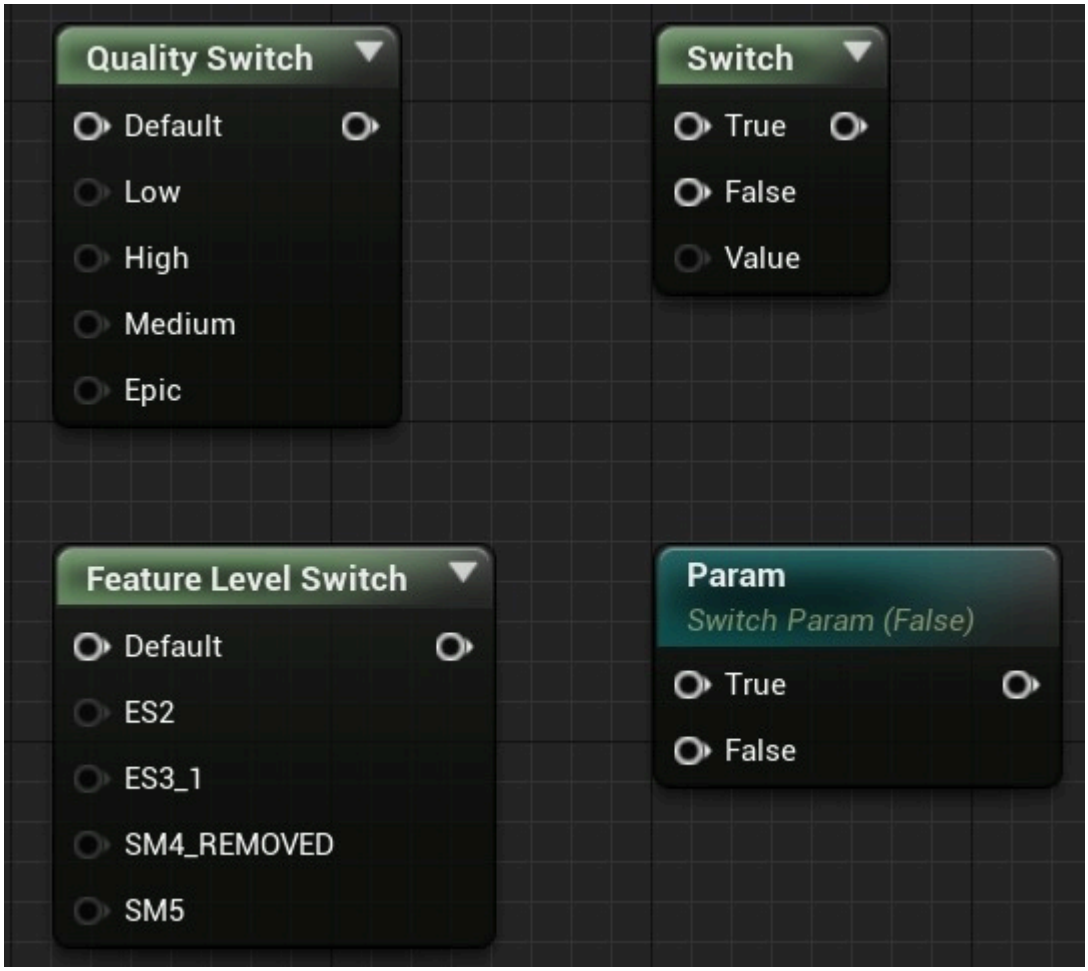
- FMeshMaterialShader
- FVertexFactory
- FLocalVertexFactory
- FShaderType
- FGlobalShaderType
- FMaterialShaderType
- Subclasses of the above types

Therefore, when we add new subclasses of the above types, it is necessary to take ShouldCompilePermutation seriously in order to remove some useless shader arrangements.
For materials, you can turn off the Automatically Set Usage in Editor option in the material property template to prevent additional markup during editing and increase shader alignment:



However, the benefits may not be obvious, and the material may not work properly due to missing certain tags (such as not supporting skinning bones, not supporting BS, etc.).

Also, be careful about adding Switch nodes, these will usually increase the number of permutations as well:



8.4.2.2 Instruction Optimization

- Avoid if and switch branch statements.
- Avoidforloop statements, especially those with a variable number of loops.
- Reduce the number of texture samples.
- Disableclipordiscardoperate.
- Reduce complex mathematical function calls.
- Use lower precision floating point numbers. OpenGL ES has three types of floating point precision: highp (32-bit floating point), mediump (16-bit floating point), and lowp (8-bit floating point). Many calculations do not require high precision and can be changed to lowprecision floating point.

- To avoid repeated calculations, all the same variables for all pixels can be calculated in advance, or passed in from the C++ layer:

```
precision mediumpfloat; floata =
0.9;
floatb =0.6;

varyingvec4vColor;

voidmain()
{
    gl_FragColor= vColor * a * b;// a * bEach pixel will be calculated, resulting in redundant consumption.a * bexist C++The layer is calculated
and then passed inshader.
}
```

- Vector delayed evaluation.

```
highpfloatf0, f1; highp
vec4v0, v1;


v0 = (v1 * f0) * f1;// v1andf0After calculation, a vector is returned, and thenf1Calculation, one more vector calculation is
required. //To:
v0 = v1 * (f0 * f1);//Calculate the two floating point numbers first, so that you only need to calculate the vector once.
```

- Take advantage of vector component masks.

```
highp   vec4   v0;
highp   vec4   v1;
highp   vec4   v2;
v2.xz = v0 * v1;// v2Only usedxzQuantity, ratiov2 = v0 * v1The writing method should be fast.
```

- Avoid or reduce temporary variables.
- Try to move pixel shader calculations to vertex shaders. For example, change pixel lights to vertex lights.
- Move all non-vertex or pixel-related calculations to the CPU and pass them in via uniforms.
- Grading strategy: Different platforms use algorithms of different complexity for different image qualities.
- Vertex input should be arranged in a structure-by-structure layout to avoid having one array per vertex attribute. Structure-by-structure layout helps improve GPU cache hit rate.
- Use Compute Shader to replace traditional VS and PS pipelines as much as possible. The CS pipeline is simpler and purer, which is conducive to parallel computing. Combined with the LDS mechanism, it can effectively improve efficiency.
- Render at reduced resolution. Some information does not need to be rendered at full resolution, such as blurred reflections, SSR, SSGI, etc.

8.4.3 Shader Development Case

Combining development cases will help consolidate the mastery and understanding of the UE Shader system.

8.4.3.1 Adding Global Shader

This section explains the Shader adding process and steps by adding a new and simplified Global Shader.

First, you need to add a new shader source file, named MyTest.ush here:

```
//VSMain entrance.
void MainVS(
	in float4 InPosition : ATTRIBUTE0, out float4
	Output : SV_POSITION)
{
	Output = InPosition;
}


//Color variables, given byC++Layer
input. float4 MyColor;

//PSMain entrance.
float4MainPS() : SV_Target0 {

	returnMyColor;
}
```

Add C++ related VS and PS:

```
#include"GlobalShader.h"

//VS,Inherited fromFGlobalShader classFMyVS:
public FGlobalShader {

	DECLARE_EXPORTED_SHADER_TYPE(FMyVS,	Global,	/*MYMODULE_API*/);

	FMyTestVS() {}
	FMyTestVS(constShaderMetaType::CompiledShaderInitializerType&	Initializer
		: FGlobalShader(Initializer)
	{
	}

	static bool ShouldCache(EShaderPlatform Platform) {

		returntrue;
	}
};

//accomplishVS.
IMPLEMENT_SHADER_TYPE(, FMyVS, TEXT("MyTest"), TEXT("MainVS"), SF_Vertex);


// PS,Inherited fromFGlobalShader
classFMyTestPS: public FGlobalShader
```

```
{
	DECLARE_EXPORTED_SHADER_TYPE(FMyPS, Global,/*MYMODULE_API*/);

	FShaderParameter MyColorParameter;

	FMyTestPS() {}
	FMyTestPS(const	ShaderMetaType::CompiledShaderInitializerType&	Initializer
		:	FGlobalShader(Initializer)
	{
		//	Bind shader parameters.
		MyColorParameter.Bind(Initializer.ParameterMap,	TEXT("MyColor"),	SPF_Mandatory);
	}

	static void ModifyCompilationEnvironment(EShaderPlatform Platform,
FShaderCompilerEnvironment& OutEnvironment)
	{
		FGlobalShader::ModifyCompilationEnvironment(Platform, OutEnvironment); //Added
		definition.
		OutEnvironment.SetDefine(TEXT("MY_DEFINE"),1);
	}

	static bool ShouldCache(EShaderPlatform Platform) {

		returntrue;
	}

	//Serialization.
	virtualbool Serialize(FArchive& Ar) override {

		boolbShaderHasOutdatedParameters = FGlobalShader::Serialize(Ar); Ar <<
			MyColorParameter;
		return bShaderHasOutdatedParameters;
	}

	void SetColor(FRHICommandList& RHICmdList,constFLinearColor& Color) {

		//Set the color toRHI.
		SetShaderValue(RHICmdList, RHICmdList.GetBoundPixelShader(), MyColorParameter,
Color);
	}
};

//accomplishPS.
IMPLEMENT_SHADER_TYPE(, FMyPS, TEXT("MyTest"), TEXT("MainPS"), SF_Pixel);
```

Finally, write the rendering code to call the above customized VS and PS:

```
void RenderMyTest(FRHICommandList& RHICmdList, ERHIFeatureLevel::Type FeatureLevel,const FLinearColor& Color)

{
	//Get the global shader map.
	autoShaderMap = GetGlobalShaderMap(FeatureLevel);

	//GetVSandPSExamples.
	TShaderMapRef<FMyVS>	MyVS(ShaderMap);
	TShaderMapRef<FMyPS>	MyPS(ShaderMap);
```

```
    //Rendering status.
    staticFGlobalBoundShaderState MyTestBoundShaderState; SetGlobalBoundShaderState(RHICmdList,
    FeatureLevel, MyTestBoundShaderState, GetVertexDeclarationFVector4(), *MyVS, *MyPS);


    //set upPSColor.
    MyPS->SetColor(RHICmdList, Color);

    //Set the rendering state.
    RHICmdList.SetRasterizerState(TStaticRasterizerState::GetRHI());
    RHICmdList.SetBlendState(TStaticBlendState<>::GetRHI());
    RHICmdList.SetDepthStencilState(TStaticDepthStencilState::GetRHI(),                    0);

    //Create the vertices of a full-screen cube.
    FVector4Vertices[4];
    Vertices0].Set(-1.0f,1.0f,0,1.0f);Vertices[1].Set(1.0f,1.0f
    ,0,1.0f);Vertices[2].Set(-1.0f,-1.0f,0,1.0f);Vertices[3
    ].Set(1.0f,-1.0f,0,1.0f);


    //Draw a block.
    DrawPrimitiveUP(RHICmdList, PT_TriangleStrip,2, Vertices,sizeof(Vertices[0]));
}
```

After RenderMyTest is implemented, it can be added to FDeferredShadingSceneRenderer::RenderFinish to access the main rendering process:

```
//Console variables to view the effect at runtime.
staticTAutoConsoleVariableCVarMyTest(
    TEXT("r.MyTest"),
    0,
    TEXT("Test My Global Shader, set it to 0 to disable, or to 1, 2 or 3 for fun!"), ECVF_RenderThreadSafe

) ;

void FDeferredShadingSceneRenderer::RenderFinish(FRHICommandListImmediate& RHICmdList) {

    (.....)

    //Add custom code to coverUEPrevious rendering.
    int32 MyTestValue = CVarMyTest.GetValueOnAnyThread(); if
    (MyTestValue !=0) {

        FLinearColorColor(MyTestValue ==1, MyTestValue ==2, MyTestValue ==3,1); RenderMyTest(RHICmdList,
        FeatureLevel, Color);
    }

    FSceneRenderer::RenderFinish(RHICmdList);

    (.....)
}
```

The final rendered color of the above logic is determined by r.MyTest: if it is 0, it is disabled; if it is 1, it displays red; if it is 2, it displays green; if it is 3, it displays blue.

8.4.3.2 Adding Vertex Factory

The process of adding a new FVertexFactory subclass is as follows:

```
// FMyVertexFactory.h

//Declare vertex factory shader parameters.
BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT(FMyVertexFactoryParameters,                )
        SHADER_PARAMETER(FVector4, Color)
        END_GLOBAL_SHADER_PARAMETER_STRUCT()

//Declare a type.
typedefTUniformBufferRef<FMyVertexFactoryParameters> FMyVertexFactoryBufferRef;

//Index buffer.
classFMyMeshIndexBuffer: public FIndexBuffer {

public:
        FMyMeshIndexBuffer(int32 InNumQuadsPerSide) : NumQuadsPerSide(InNumQuadsPerSide) {}

        void InitRHI() override {

                if(NumQuadsPerSide <256) {

                        IndexBufferRHI        =  CreateIndexBuffer<uint16>();
                }
                else
                {
                        IndexBufferRHI        =  CreateIndexBuffer<uint32>();
                }
        }

        int32GetIndexCount()const{returnNumIndices; };

private:
        template <typename        IndexType>
        FIndexBufferRHIRef        CreateIndexBuffer()
        {
                TResourceArray<IndexType,          INDEXBUFFER_ALIGNMENT>     Indices;

                //Allocate vertex index memory.
                Indices.Reserve(NumQuadsPerSide * NumQuadsPerSide *6);

                //useMortonBuild index buffers sequentially to better reuse vertices.
                for(int32 Morton =0; Morton < NumQuadsPerSide * NumQuadsPerSide; Morton++) {

                        int32 SquareX = FMath::ReverseMortonCode2(Morton); int32 SquareY =
                        FMath::ReverseMortonCode2(Morton >>1);

                        boolForwardDiagonal =false;

                        if(SquareX %2) {

                                ForwardDiagonal        =  !ForwardDiagonal;
                        }
                        if(SquareY %2) {
```

```
                ForwardDiagonal = !ForwardDiagonal;
            }

            int32 Index0 = SquareX + SquareY * (NumQuadsPerSide +1); int32 Index1 =
            Index0 +1;
            int32 Index2 = Index0 + (NumQuadsPerSide +1); int32 Index3 =
            Index2 +1;

            Indices.Add(Index3);        Indices.Add(Index1);
            Indices.Add(ForwardDiagonal ? Index2 : Index0);
            Indices.Add(Index0);        Indices.Add(Index2);
            Indices.Add(ForwardDiagonal ? Index1 : Index3);


        }

        NumIndices = Indices.Num();
        constuint32 Size = Indices.GetResourceDataSize(); constuint32
        Stride =sizeof(IndexType);

        // Create index buffer. Fill buffer with initial data upon creation FRHIResourceCreateInfo
        CreateInfo(&Indices);
        returnRHICreateIndexBuffer(Stride, Size, BUF_Static, CreateInfo);
    }

    int32 NumIndices =0; constint32
    NumQuadsPerSide =0;
};

//Vertex index.
classFMyMeshVertexBuffer: public FVertexBuffer {

public:
    FMyMeshVertexBuffer(int32 InNumQuadsPerSide) : NumQuadsPerSide(InNumQuadsPerSide) {}

    virtualvoid InitRHI() override {

        constuint32 NumVertsPerSide = NumQuadsPerSide +1;

        NumVerts = NumVertsPerSide * NumVertsPerSide;

        FRHIResourceCreateInfo CreateInfo; void*
        BufferData = nullptr;
        VertexBufferRHI = RHICreateAndLockVertexBuffer(sizeof(FVector4) * NumVerts, BUF_Static,
CreateInfo, BufferData);
        FVector4* DummyContents = (FVector4*)BufferData;

        for(uint32 VertY =0; VertY < NumVertsPerSide; VertY++) {

            FVector4 VertPos;
            VertPos.Y = (float)VertY / NumQuadsPerSide -0.5f;

            for(uint32 VertX =0; VertX < NumVertsPerSide; VertX++) {

                VertPos.X = (float)VertX / NumQuadsPerSide -0.5f;

                DummyContents[NumVertsPerSide * VertY + VertX] = VertPos;
            }
```

```
            }

            RHIUnlockVertexBuffer(VertexBufferRHI);
        }

        int32GetVertexCount()const{returnNumVerts; }

private:
        int32 NumVerts =0;
        constint32 NumQuadsPerSide =0;
};

//Vertex Factory.
classFMyVertexFactory: public FVertexFactory {

        DECLARE_VERTEX_FACTORY_TYPE(FMyVertexFactory);

public:
        using Super = FVertexFactory;

        FMyVertexFactory(ERHIFeatureLevel::Type InFeatureLevel);
        ~FMyVertexFactory();

        virtualvoid InitRHI() override; virtualvoid
        ReleaseRHI() override;

        static bool ShouldCompilePermutation(constFVertexFactoryShaderPermutationParameters& Parameters);

        static void ModifyCompilationEnvironment(const FVertexFactoryShaderPermutationParameters&
Parameters, FShaderCompilerEnvironment& OutEnvironment);

        static void ValidateCompiledResult(constFVertexFactoryType* Type, EShaderPlatform Platform,const
FShaderParameterMap& ParameterMap, TArray<FString>& OutErrors);

        inlineconstFUniformBufferRHIRefGetMyVertexFactoryUniformBuffer()const{return UniformBuffer; }


private:
        void    SetupUniformData();

        FMyMeshVertexBuffer* VertexBuffer = nullptr;
        FMyMeshIndexBuffer* IndexBuffer = nullptr;

        FMyVertexFactoryBufferRef UniformBuffer;
};


// FMyVertexFactory.cpp

#include"ShaderParameterUtils.h"

//accomplishFMyVertexFactoryParameters,NoteshaderThe name isMyVF.
IMPLEMENT_GLOBAL_SHADER_PARAMETER_STRUCT(FMyVertexFactoryParameters,"MyVF");


//Vertex factory shader parameters.
classFMyVertexFactoryShaderParameters: public FVertexFactoryShaderParameters {
```

```
	DECLARE_TYPE_LAYOUT(FMyVertexFactoryShaderParameters, NonVirtual);

public:

	void Bind(constFShaderParameterMap& ParameterMap) {

	}

	void GetElementShaderBindings(
		constclass FSceneInterface* Scene, constclass FSceneView*
		View, constclass FMeshMaterialShader* Shader, const
		EVertexInputStreamType InputStreamType,
		ERHIFeatureLevel::Type FeatureLevel,

		constclass FVertexFactory* InVertexFactory, conststruct
		FMeshBatchElement& BatchElement, class
		FMeshDrawSingleShaderBindings& ShaderBindings,
		FVertexInputStreamArray& VertexStreams)const
	{
		//Forced toFMyVertexFactory.
		FMyVertexFactory* VertexFactory = (FMyVertexFactory*)InVertexFactory;

		//IncreaseshaderHelp to define the
		form. ShaderBindings.Add(Shader-
> GetUniformBufferParameter<FMyVertexFactoryShaderParameters>(),		VertexFactory -
> GetMyVertexFactoryUniformBuffer());

		//Fills a vertex stream.
		if(VertexStreams.Num() >0) {

			//Process vertex stream indices.
			for(int32 i =0; i <2; ++i) {

				FVertexInputStream* InstanceInputStream		=
VertexStreams.FindByPredicate([i](constFVertexInputStream& InStream) {return InStream.StreamIndex
== i+1; });
				//Bind vertex stream index.
				InstanceInputStream->VertexBuffer = InstanceDataBuffers->GetBuffer(i);
			}

			//Handling offsets.
			if(InstanceOffsetValue >0) {

				VertexFactory->OffsetInstanceStreams(InstanceOffsetValue,		InputStreamType,
VertexStreams);
			}
		}
	}
};

//----------Implementing the Vertex Factory-----------

FMyVertexFactory::FMyVertexFactory(ERHIFeatureLevel::Type InFeatureLevel) {

	VertexBuffer = new FMyMeshVertexBuffer(16);
	IndexBuffer = new FMyMeshIndexBuffer(16);
}
```

```
FMyVertexFactory::~FMyVertexFactory() {

	delete	VertexBuffer;
	delete	IndexBuffer;
}

void FMyVertexFactory::InitRHI() {

	Super::InitRHI();

	//set upUniformdata.
	SetupUniformData();

	VertexBuffer->InitResource();
	IndexBuffer->InitResource();

	//Vertex Flow: Position
	FVertexStream PositionVertexStream;
	PositionVertexStream.VertexBuffer		=VertexBuffer;
	PositionVertexStream.Stride		= sizeof(FVector4);
	PositionVertexStream.Offset		=0;
	PositionVertexStream.VertexStreamUsage		= EVertexStreamUsage::Default;

	//Simple instantiation of vertex stream dataVertexBufferSet at
	bind time. FVertexStream InstanceDataVertexStream;
	InstanceDataVertexStream.VertexBuffer		=nullptr;
	InstanceDataVertexStream.Stride		= sizeof(FVector4);
	InstanceDataVertexStream.Offset		=0;
	InstanceDataVertexStream.VertexStreamUsage		= EVertexStreamUsage::Instancing;

	FVertexElementVertexPositionElement(Streams.Add(PositionVertexStream),0, VET_Float4, 0,
PositionVertexStream.Stride,false);

	//Vertex declaration.
	FVertexDeclarationElementList Elements;
	Elements.Add(VertexPositionElement);

	//Add indexed vertex stream.
	for(int32 StreamIdx =0; StreamIdx < NumAdditionalVertexStreams; ++StreamIdx) {

		FVertexElementInstanceElement(Streams.Add(InstanceDataVertexStream),0, VET_Float4,8+
StreamIdx, InstanceDataVertexStream.Stride,true);
		Elements.Add(InstanceElement);
	}

	//Initialization statement.
	InitDeclaration(Elements);
}

void FMyVertexFactory::ReleaseRHI() {

	UniformBuffer.SafeRelease();

	if(VertexBuffer)
	{
		VertexBuffer->ReleaseResource();
	}
```

```
	if(IndexBuffer)
	{
		IndexBuffer->ReleaseResource();
	}

	Super::ReleaseRHI();
}

void FMyVertexFactory::SetupUniformData() {

	FMyVertexFactoryParameters UniformParams;
	UniformParams.Color = FVector4(1,0,0,1);

	UniformBuffer = FMyVertexFactoryBufferRef::CreateUniformBufferImmediate(UniformParams,
	UniformBuffer_MultiFrame);
}

void FMyVertexFactory::ShouldCompilePermutation(const
FVertexFactoryShaderPermutationParameters& Parameters) {

	returntrue;
}

void FMyVertexFactory::ModifyCompilationEnvironment(const
FVertexFactoryShaderPermutationParameters& Parameters,		FShaderCompilerEnvironment&
OutEnvironment)
{
	OutEnvironment.SetDefine(TEXT("MY_MESH_FACTORY"),		1);
}

void FMyVertexFactory::ValidateCompiledResult(constFVertexFactoryType* Type, EShaderPlatform Platform,
constFShaderParameterMap& ParameterMap, TArray<FString>& OutErrors)

{
}
```

The logic of the C++ layer has been completed, but the HLSL layer also needs to write corresponding code:

```
#include"/Engine/Private/VertexFactoryCommon.ush"

//VSInterpolate toPSThe structure of.
structFVertexFactoryInterpolantsVSToPS {

#ifNUM_TEX_COORD_INTERPOLATORS
	float4 TexCoords[(NUM_TEX_COORD_INTERPOLATORS+1)/2]		:	TEXCOORD0;
#endif

#if	VF_USE_PRIMITIVE_SCENE_DATA
	nointerpolation uint PrimitiveId : PRIMITIVE_ID;
#endif

#if	INSTANCED_STEREO
	nointerpolation uint EyeIndex : PACKED_EYE_INDEX;
#endif
};
```

```
structFVertexFactoryInput {

	float4		Position		: ATTRIBUTE0;

	float4 InstanceData0 : ATTRIBUTE8; float4
	InstanceData1 : ATTRIBUTE9;

#if	VF_USE_PRIMITIVE_SCENE_DATA uint
	PrimitiveId : ATTRIBUTE13;
#endif
};

structFPositionOnlyVertexFactoryInput {

	float4		Position		: ATTRIBUTE0;

	float4 InstanceData0 : ATTRIBUTE8; float4
	InstanceData1 : ATTRIBUTE9;

#if	VF_USE_PRIMITIVE_SCENE_DATA uint
	PrimitiveId : ATTRIBUTE1;
#endif
};

structFPositionAndNormalOnlyVertexFactoryInput {

	float4		Position		: ATTRIBUTE0;
	float4		Normal			: ATTRIBUTE2;

	float4 InstanceData0 : ATTRIBUTE8; float4
	InstanceData1 : ATTRIBUTE9;

#if	VF_USE_PRIMITIVE_SCENE_DATA uint
	PrimitiveId : ATTRIBUTE1;
#endif
};

structFVertexFactoryIntermediates {

	float3 OriginalWorldPos;

	uint PrimitiveId;
};

uint	GetPrimitiveId(FVertexFactoryInterpolantsVSToPS			Interpolants
{
#if	VF_USE_PRIMITIVE_SCENE_DATA
	return Interpolants.PrimitiveId;
#else
	return	0;
#endif
}

void SetPrimitiveId(inout FVertexFactoryInterpolantsVSToPS Interpolants, uint PrimitiveId) {

#ifVF_USE_PRIMITIVE_SCENE_DATA
	Interpolants.PrimitiveId = PrimitiveId;
```

```
#endif
}

#ifNUM_TEX_COORD_INTERPOLATORS
float2GetUV(FVertexFactoryInterpolantsVSToPS Interpolants,intUVIndex) {

	float4 UVVector = Interpolants.TexCoords[UVIndex /2]; returnUVIndex %2
	? UVVector.zw : UVVector.xy;
}

void SetUV(inout FVertexFactoryInterpolantsVSToPS Interpolants,intUVIndex, float2 InValue)

{
	FLATTEN
	if(UVIndex %2) {

		Interpolants.TexCoords[UVIndex/2].zw = InValue;
	}
	else
	{
		Interpolants.TexCoords[UVIndex/2].xy = InValue;
	}
}
#endif

FMaterialPixelParametersGetMaterialPixelParameters(FVertexFactoryInterpolantsVSToPS Interpolants, float4
SvPosition)
{
	// GetMaterialPixelParameters is responsible for fully initializing the result FMaterialPixelParameters
	Result = MakeInitializedMaterialPixelParameters();

#if	NUM_TEX_COORD_INTERPOLATORS
	UNROLL
	for(intCoordinateIndex =0; CoordinateIndex < NUM_TEX_COORD_INTERPOLATORS; CoordinateIndex++)

	{
		Result.TexCoords[CoordinateIndex] = GetUV(Interpolants, CoordinateIndex);
	}
#endif		//NUM_MATERIAL_TEXCOORDS

	Result.TwoSidedSign		= 1;
	Result.PrimitiveId		= GetPrimitiveId(Interpolants);

	returnResult;
}

FMaterialVertexParametersGetMaterialVertexParameters(FVertexFactoryInput Input, FVertexFactoryIntermediates
Intermediates, float3 WorldPosition, half3x3 TangentToLocal) {

	FMaterialVertexParameters Result = (FMaterialVertexParameters)0;

	Result.WorldPosition		= WorldPosition;
	Result.TangentToWorld		= float3x3(1,0,0,0,1,0,0,0,1);
	Result.PreSkinnedPosition	= Input.Position.xyz;
	Result.PreSkinnedNormal =	float3(0,0,1);

#if	NUM_MATERIAL_TEXCOORDS_VERTEX
	UNROLL
```

```
	for(intCoordinateIndex =0; CoordinateIndex < NUM_MATERIAL_TEXCOORDS_VERTEX; CoordinateIndex+
	+)
	{
		Result.TexCoords[CoordinateIndex] = Intermediates.MorphedWorldPosRaw.xy;
	}
#endif//NUM_MATERIAL_TEXCOORDS_VERTEX

	returnResult;
}

FVertexFactoryIntermediatesGetVertexFactoryIntermediates(FVertexFactoryInput Input) {

	FVertexFactoryIntermediates Intermediates;

	// Get the packed instance data float4 Data0 =
	Input.InstanceData0; float4 Data1 =
	Input.InstanceData1;

	constfloat3 Translation = Data0.xyz; constfloat3 Scale =
	float3(Data1.zw,1.0f); constuint PackedDataChannel =
	asuint(Data1.x);

	// Lod level is in first 8 bits and ShouldMorph bit is in the 9th bit const floatLODLevel = (float)
	(PackedDataChannel &0xFF); constuint ShouldMorph = ((PackedDataChannel >>8) &0x1);


	// Calculate the world pos
	Intermediates.OriginalWorldPos = float3(Input.Position.xy,0.0f) * Scale + Translation;


#if	VF_USE_PRIMITIVE_SCENE_DATA
	Intermediates.PrimitiveId		=	Input.PrimitiveId;
#else
	Intermediates.PrimitiveId		=	0;
#endif

	returnIntermediates;
}

half3x3VertexFactoryGetTangentToLocal(FVertexFactoryInput Input,
FVertexFactoryIntermediates Intermediates)
{
	returnhalf3x3(1,0,0,0,1,0,0,0,1);
}

float4VertexFactoryGetRasterizedWorldPosition(FVertexFactoryInput 			Input,
FVertexFactoryIntermediates Intermediates, float4 InWorldPosition) {

	returnInWorldPosition;
}

float3VertexFactoryGetPositionForVertexLighting(FVertexFactoryInput Input,
FVertexFactoryIntermediates Intermediates, float3 TranslatedWorldPosition) {

	returnTranslatedWorldPosition;
}

FVertexFactoryInterpolantsVSToPSVertexFactoryGetInterpolantsVSToPS(FVertexFactoryInput
```

```
Input, FVertexFactoryIntermediates Intermediates, FMaterialVertexParameters VertexParameters)

{
	FVertexFactoryInterpolantsVSToPS Interpolants;

	Interpolants = (FVertexFactoryInterpolantsVSToPS)0;

#if	NUM_TEX_COORD_INTERPOLATORS
	float2 CustomizedUVs[NUM_TEX_COORD_INTERPOLATORS];
	GetMaterialCustomizedUVs(VertexParameters, CustomizedUVs);
	GetCustomInterpolators(VertexParameters, CustomizedUVs);

	UNROLL
	for(intCoordinateIndex =0; CoordinateIndex < NUM_TEX_COORD_INTERPOLATORS; CoordinateIndex++)

	{
		SetUV(Interpolants, CoordinateIndex, CustomizedUVs[CoordinateIndex]);
	}
#endif

#if	INSTANCED_STEREO
	Interpolants.EyeIndex		=	0;
#endif

	SetPrimitiveId(Interpolants, Intermediates.PrimitiveId);

	returnInterpolants;
}

float4VertexFactoryGetWorldPosition(FPositionOnlyVertexFactoryInput Input) {

	returnInput.Position;
}

float4VertexFactoryGetPreviousWorldPosition(FVertexFactoryInput 			Input,
FVertexFactoryIntermediates Intermediates)
{
	float4x4 PreviousLocalToWorldTranslated =
	GetPrimitiveData(Intermediates.PrimitiveId).PreviousLocalToWorld;
	PreviousLocalToWorldTranslated[3][0]		+
	PreviousLocalToWorldTranslated[3][1]		=ResolvedView.PrevPreViewTranslation.x;
	PreviousLocalToWorldTranslated[3][2]		+ =ResolvedView.PrevPreViewTranslation.y;
							+ =ResolvedView.PrevPreViewTranslation.z;
	returnmul(Input.Position, PreviousLocalToWorldTranslated);
}

float4VertexFactoryGetTranslatedPrimitiveVolumeBounds(FVertexFactoryInterpolantsVSToPS Interpolants)

{
	float4 ObjectWorldPositionAndRadius =
	GetPrimitiveData(GetPrimitiveId(Interpolants)).ObjectWorldPositionAndRadius;
	returnfloat4(ObjectWorldPositionAndRadius.xyz + ResolvedView.PreViewTranslation.xyz,
	ObjectWorldPositionAndRadius.w);
}

uintVertexFactoryGetPrimitiveId(FVertexFactoryInterpolantsVSToPS Interpolants) {

	returnGetPrimitiveId(Interpolants);
```

```
}

float3VertexFactoryGetWorldNormal(FPositionAndNormalOnlyVertexFactoryInput Input) {

    returnInput.Normal.xyz;
}

float3VertexFactoryGetWorldNormal(FVertexFactoryInput Input, FVertexFactoryIntermediates Intermediates)

{
    returnfloat3(0.0f,0.0f,1.0f);
}
```

It can be seen that if a new custom type of FVertexFactory is added, the following interface needs to be implemented in HLSL:

Function	describe
FVertexFactoryInput	Define the data layout input to VS, which needs to match the type of FVertexFactory on the C++ side.
FVertexFactoryIntermediates	Used to store cached intermediate data that will be used in multiple vertex factory functions, such as TangentToLocal.
FVertexFactoryInterpolantsVSToPS	Vertex factory data passed from VS to PS.
VertexFactoryGetWorldPosition	Called from the vertex shader to get the world-space vertex position.
VertexFactoryGetInterpolants VS ToPS	Convert FVertexFactoryInput to FVertexFactoryInterpolants to calculate the data that needs to be interpolated or passed to the PS before the hardware rasterizer interpolates.
GetMaterialPixelParameters	Called by PS to calculate and fill the FMaterialPixelParameters structure based on FVertexFactoryInterpolants.

8.5 Summary

This article mainly explains the basic concepts, types, and mechanisms of UE's shader system. I hope that after studying this article, you will no longer be unfamiliar with UE's shader and be able to apply it in actual project practices.

8.5.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the mastery and understanding of the UE Shader system:

- What are the important subclasses in the FShader inheritance system? What are their functions? What are the similarities and differences?
- How to declare, implement, apply and update Shader Parameter and Uniform Buffer to GPU? What
- is the storage and compilation mechanism of Shader Map?
- What solution does UE adopt in Shader cross-platform? Why do you do that? Is there a better way?

- How to better debug or optimize Shader?

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming
- Shader Development
- Debugging the Shader Compile Process
- Creating a Custom Mesh Component in UE4 | Part 0: Intro
- Creating a Custom Mesh Component in UE4 | Part 1: An In-depth Explanation of Vertex Factories
- Creating a Custom Mesh Component in UE4 | Part 2: Implementing the Vertex Factory
- Unreal Engine 4 Rendering Part 1: Introduction
- Unreal Engine 4 Rendering Part 5: Shader Permutations
- [UE4 Renderer]<03> PipelineBase
- UE4 material system source code analysis: material compiled into HLSL CODE

- UE4 HLSL and Shader Development Guide and Tips
- Uniform Buffer, FVertexFactory, FVertexFactoryType
- Game Engine Essay 0x02: The Road to Shader Cross-Platform Compilation
- UE4 Shader compilation and variant implementation
- Unreal 4 rendering programming (Shader) [Volume 4: Simple data communication between Unreal 4 C++ layer and Shader layer]
- UE4 Rendering Part 2: Shaders and Vertex Data Unreal
- Engine 4 Rendering Part 5: Shader Permutations
- HLSL Cross Compiler
- AsyncCompute
- In-depth understanding of GPU hardware architecture and operation mechanism
- General techniques for optimizing mobile game performance
- Adding Global Shaders to Unreal Engine
- Create a New Global Shader as a Plugin
- The Industry Open Standard Intermediate Language for Parallel Compute and Graphics
- Analysis of cross-platform engine Shader compilation process
- Considerations on Shader's cross-platform solution
- UE4's shader cross-platform solution
- The past, present and future of cross-platform shader compilation
- BRINGING UNREAL ENGINE 4 TO OPENGL
- FShaderCache

https://github.com/pe7yu