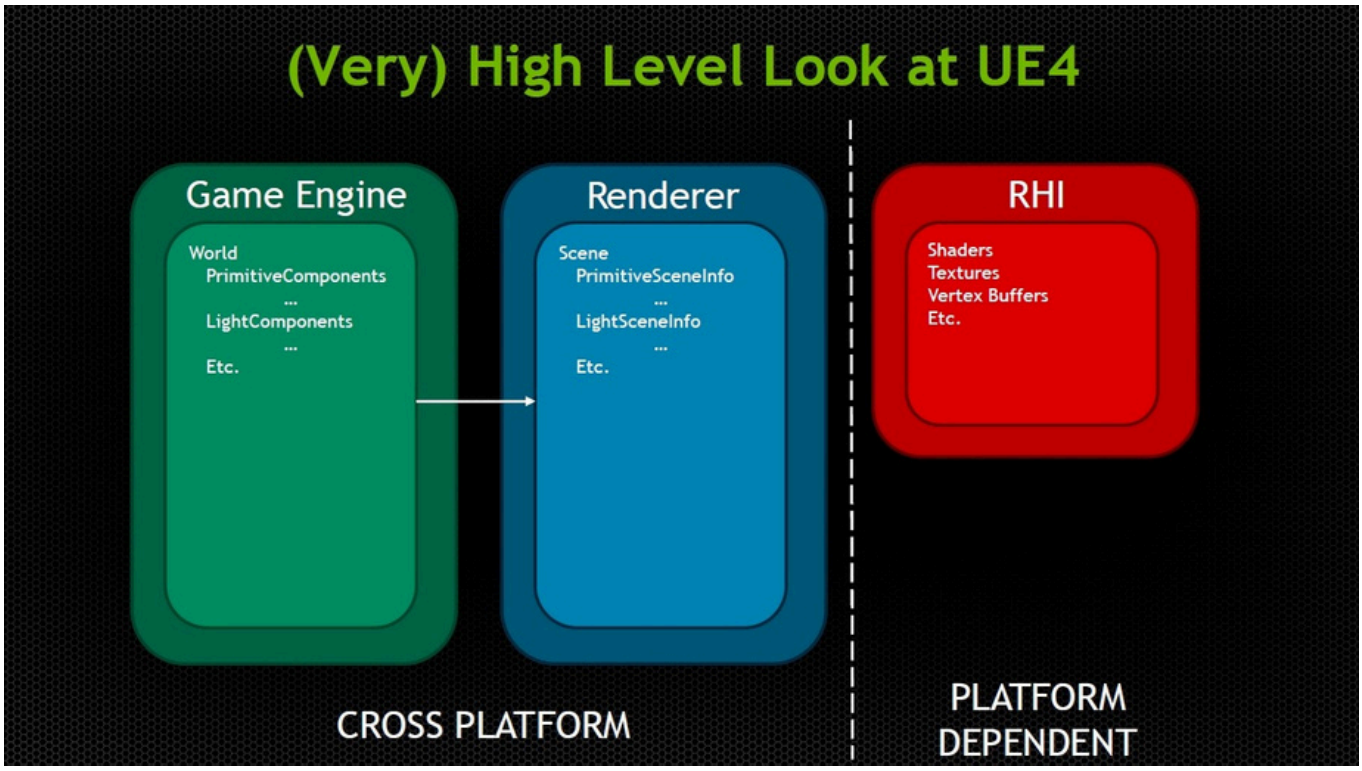
Analysis of Unreal Rendering System (10) - RHI

Table of contents

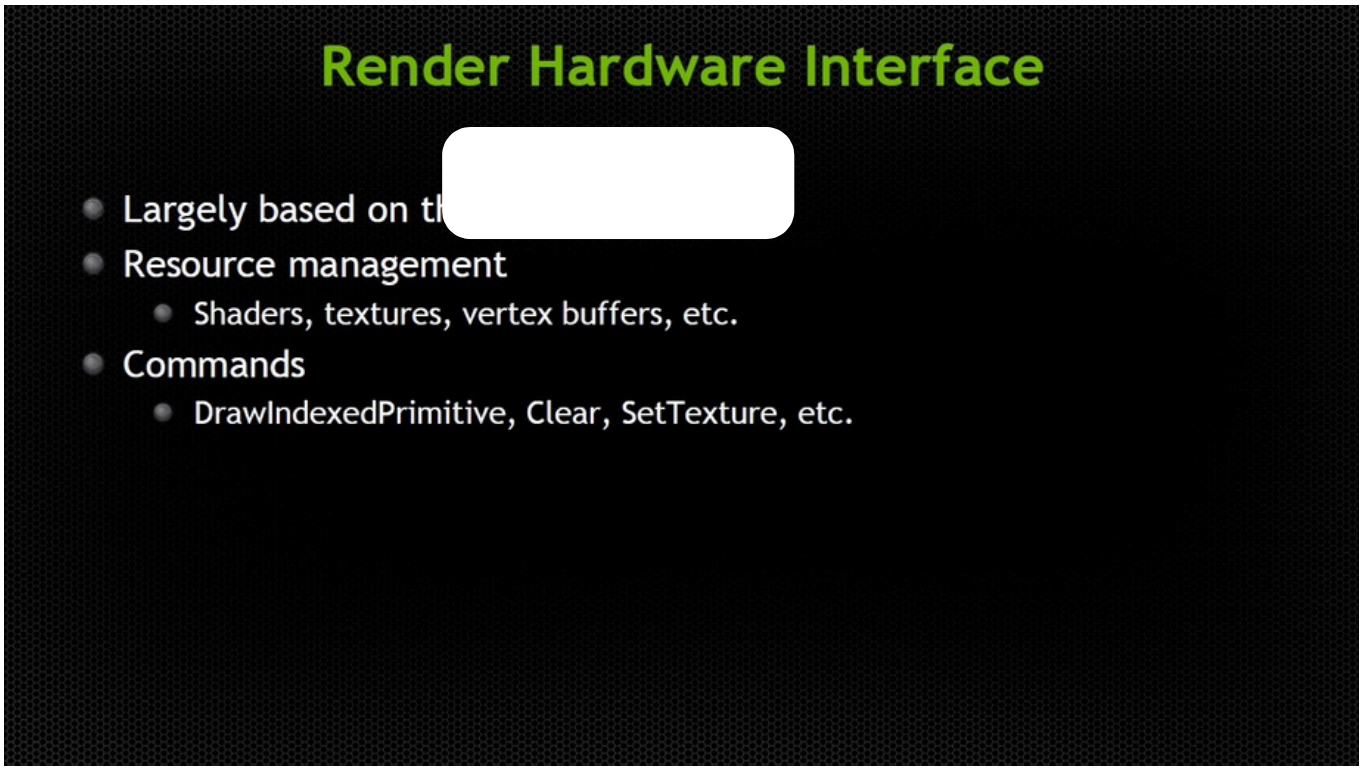
10.1 Overview

RHIstands for**Render Hardware Interface**,which is a very basic and important module in the UE rendering system. It encapsulates the differences between many graphics APIs (DirectX, OpenGL, Vulkan, Metal), and provides simple and consistent concepts, data, resources and interfaces for Game and Renderer modules, achieving the goal of running one rendering code on multiple platforms .

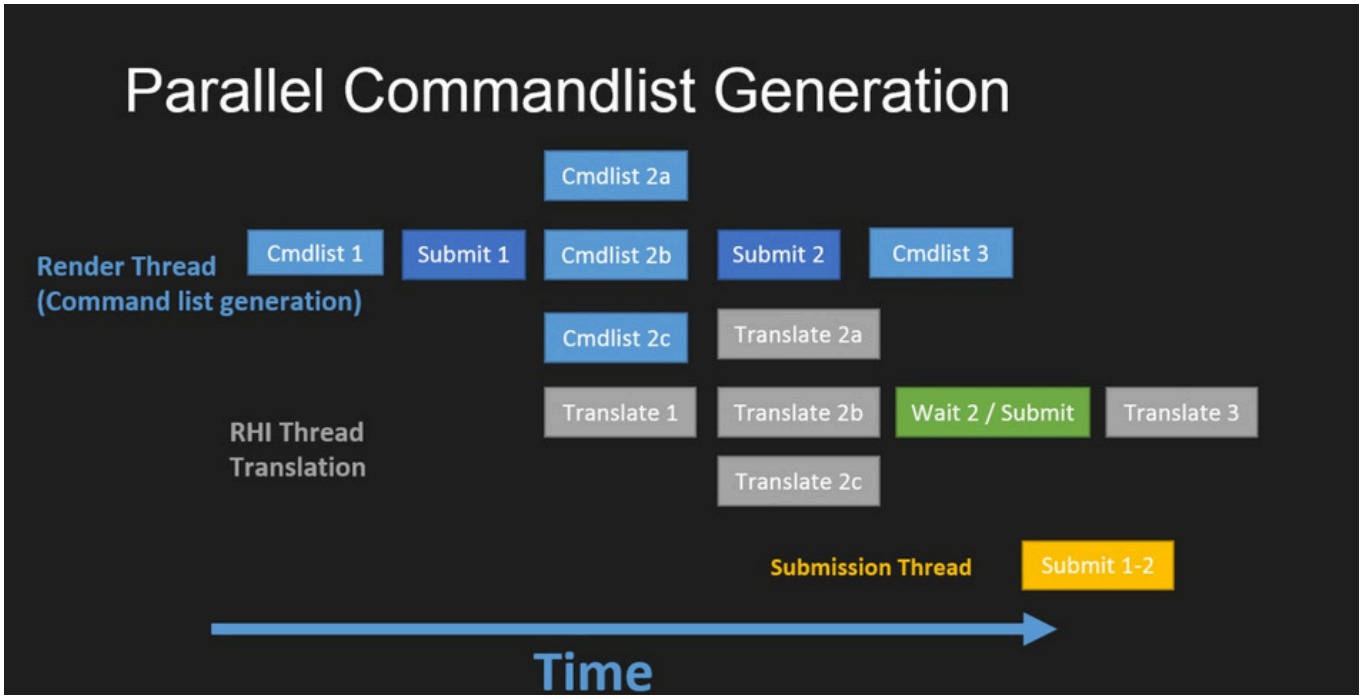


A diagram showing the hierarchical structure of Game, Renderer, and RHI, where RHI is platformdependent.

The original RHI was designed based on the D3D11 API, including resource management and command interfaces:



When the RHI thread is turned on, the RHI is accompanied by**the RHI thread**,which is responsible for translating the RHI intermediate instructions pushed by the rendering thread into the GPU instructions of the corresponding graphics platform. When some graphics APIs (DX12, Vulkan, host) support parallelism, if the rendering thread generates RHI intermediate instructions in parallel, the RHI thread will also translate them in parallel.



UE4's rendering thread generates intermediate instructions in parallel and the RHI thread translates and submits rendering instructions in parallel.

This article will focus on the basic concepts, types, interfaces of RHI, the relationship between them, the principles and mechanisms involved, and will also cover a small amount of implementation details of specific graphics APIs.

10.2 RHI Basics

This chapter will analyze the basic concepts and types involved in RHI and explain the relationship and principles between them.

10.2.1 FRenderResource

FRenderResource is the rendering resource representative of the rendering thread, which is managed and transmitted by the rendering thread. It is the intermediate data between the game thread and the RHI thread. Since the previous chapter has touched upon its concept but not elaborated in detail, it is included in this chapter. The definition of FRenderResource is as follows:

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

classRENDERCORE_APIFRenderResource {

public:
    //Traverse all resources and execute callback interface.
    template<typename FunctionType>
    static void ForAllResources(constFunctionType& Function); static
            void    InitRHIForAllResources();
    static      void ReleaseRHIForAllResources();
    static void ChangeFeatureLevel(ERHIFeatureLevel::Type NewFeatureLevel);

    FRenderResource();
    FRenderResource(ERHIFeatureLevel::Type virtual      InFeatureLevel);
    ~FRenderResource();


    //The following interfaces can only be called by the rendering thread.

    //Initialize the dynamic state of this resourceRHIResources and/orRHIThe
    render target texture. virtualvoid InitDynamicRHI() {}
    //Release this resource dynamicallyRHIResources and/orRHIThe
    render target texture. virtualvoid ReleaseDynamicRHI() {}

    //Initialize this resource usingRHI
    resource. virtualvoid InitRHI() {} //
    Release the resource used by thisRHI
    resource. virtualvoid ReleaseRHI() {}

    //Initialize resources.
    virtualvoid         InitResource();
    //Release resources.
    virtualvoid
                        ReleaseResource();

    //ifRHIThe resource has been initialized, will be released and
    reinitialized. void UpdateRHI();

    virtual FStringGetFriendlyName()const{returnTEXT("undefined"FORCEINLINEbool IsInitialized()const{
    returnListIndex != INDEX_NONE; }
```

```
    static void InitPreRHIResources();

private:
    //Global resource list (static).
    static    TArray<FRenderResource*>& FThreGaedtSRaefesoCuorucnetLeisr t();
    static    ResourceListIterationActive;

    int32 ListIndex;
    TEnumAsByte<ERHIFeatureLevel::Type> FeatureLevel;

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

The following is the interface for the game thread to send operations to the rendering thread FRenderResource:

```
//Initialize/update/release resources.
externRENDERCORE_APIvoid BeginInitResource(FRenderResource* Resource); externRENDERCORE_API
void BeginUpdateResourceRHI(FRenderResource* Resource); externRENDERCORE_APIvoid
BeginReleaseResource(FRenderResource* Resource); externRENDERCORE_APIvoid StartBatchedRelease
(); externRENDERCORE_APIvoid EndBatchedRelease();

externRENDERCORE_APIvoid ReleaseResourceAndFlush(FRenderResource* Resource);
```

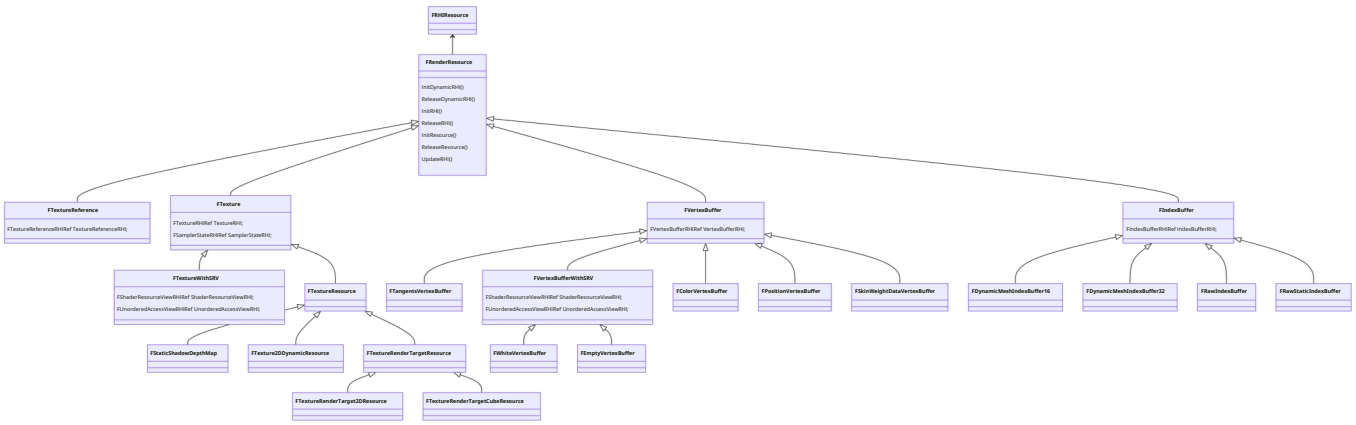
FRenderResource is just a basic parent class that defines a set of rendering resource behaviors. The actual data and logic are implemented by subclasses. The subclasses and hierarchies involved are many and complex. The following are the definitions of some important subclasses:

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

//Texture resources.
classFTexture: public FRenderResource {

public:
    FTextureRHIRef              TextureRHI;              // TexturedRHIresource.
    FSamplerStateRHIRef SamplerStateRHI;//Texture SamplersRHIresource. FSamplerStateRHIRef
    DeferredPassSamplerStateRHI;//Delayed Channel SamplerRHIresource.

    mutabledouble               LastRenderTime;     //The time of the last render.
    FMipBiasFade                MipBiasFade;        //Fade in/outMipOffset value.
    bool bool bool              bGreyScaleFormat;   //Grayscale image.
                                bIgnoreGammaConversions;    //  IgnoreGammaConvert.
                                bSRGB;                  //whethersRGBThe color of space.

    virtual uint32GetSizeX()const; virtual uint32
    GetSizeY()const; virtual uint32GetSizeZ()
    const;

    //Release resources.
    virtualvoid ReleaseRHI() override {

        TextureRHI.SafeRelease();
        SamplerStateRHI.SafeRelease();
        DeferredPassSamplerStateRHI.SafeRelease();
    }
```

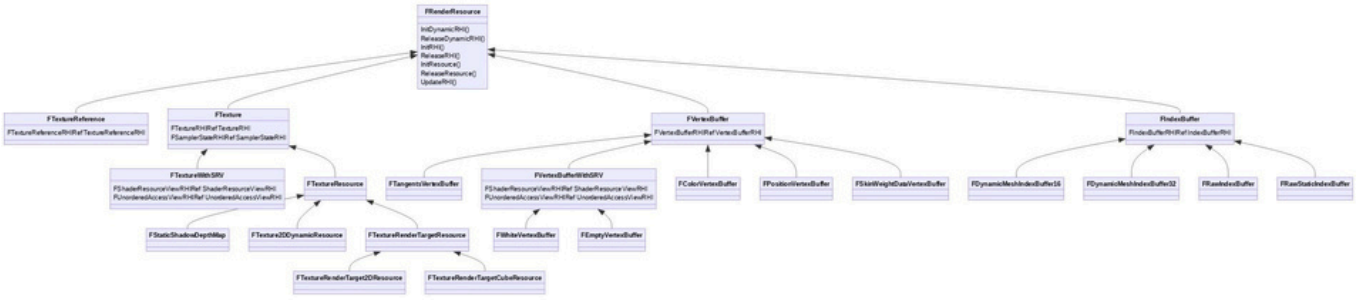
```
    virtual FStringGetFriendlyName()constoverride {returnTEXT("FTexture"); }

    (……)

protected:
    RENDERCORE_APIstaticFRHISamplerState*GetOrCreateSamplerState(const
FSamplerStateInitializerRHI& Initializer); };


//IncludedSRV/UAVTexture resources.
classFTextureWithSRV: public FTexture {

public:
    //Access the entire textureSRV.
    FShaderResourceViewRHIRef //        ShaderResourceViewRHI;
    Access the entire textureUAV.
    FUnorderedAccessViewRHIRef         UnorderedAccessViewRHI;

    virtualvoid ReleaseRHI() override;
};

//holdRHIThe rendering resource referenced by the texture resource.
classRENDERCORE_API FTextureReference: public FRenderResource {

public:
    //TexturedRHIResource references.
    FTextureReferenceRHIRef                TextureReferenceRHI;

    //FRenderResource        interface.
    virtual     void InitRHI();
    virtual     void ReleaseRHI();

    (……)
};

classRENDERCORE_API FVertexBuffer: public FRenderResource {

public:
    //Vertex BufferRHIResource references.
    FVertexBufferRHIRef VertexBufferRHI;

    virtualvoid ReleaseRHI() override;

    (……);
};

classRENDERCORE_API FVertexBufferWithSRV: public FVertexBuffer {

public:
    //Access to the entire bufferSRV/UAV.
    FShaderResourceViewRHIRef        ShaderResourceViewRHI;
    FUnorderedAccessViewRHIRef         UnorderedAccessViewRHI;

    (……)
};

//Index buffer.
classFIndexBuffer: public FRenderResource
```

```
{
public:
    //  Index buffer corresponding toRHIresource.
    FIndexBufferRHIRef        IndexBufferRHI;

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

From the above, we can see that the subclass of FRenderResource encapsulates the subclass resources of RHI accordingly, so that the rendering thread can pass the data and operations of the game thread to the RHI thread (or module). The following UML diagram intuitively presents part of the inheritance system of FRenderResource:



If you can't see clearly please click on the image below:



Once again, the above is only part of the inheritance system of FRenderResource, which cannot be drawn in full. It can be seen that FRenderResource has a huge subclass hierarchy to adapt to and meet the complex and changing needs of the UE rendering system in terms of resources.

10.2.2 FRHIResource

FRHIResource abstracts the resources on the GPU side and is also the parent class of many RHI resource types. The definition is as follows:

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

classRHI_API FRHIResource {

public:
	FRHIResource(boolInbDoNotDeferDelete =false); virtual
	~FRHIResource();

	//The reference count of the resource.
	uint32	AddRef()	const;
	uint32	Release()	const
	{
		int32 NewValue = NumRefs.Decrement(); if
		(NewValue ==0) {

			if(!DeferDelete())
			{
				delete this;
			}
			else
			{
				//Add to the list to be deleted.
				if(FPlatformAtomics::InterlockedCompareExchange(&MarkedForDelete,1,0)
==	0)
				{
					PendingDeletes.Push(const_cast<FRHIResource*>(this));
				}
			}
		}
		returnuint32(NewValue);
	}
	uint32GetRefCount()const;

	//Static interface.
	static void FlushPendingDeletes(boolbFlushDeferredDeletes =false); static
		bool	PlatformNeedsExtraDeletionLatency(); Bypass();
	static	bool

	void DoNoDeferDelete(); //Instant
	resource tracking.
	void	SetCommitted(bool	bInCommitted);
	bool	IsCommitted()const; IsValid
	bool	()const;

private:
	//Runtime flags and data.
	mutable FThreadSafeCounter		NumRefs;
	mutable	int32 MarkedForDelete;
	bool bDoNotDeferDelete;
	bool bCommitted;

	//The resource to be deleted.
	staticTLockFreePointerListUnordered<FRHIResource,		PLATFORM_CACHE_LINE_SIZE>
	PendingDeletes;
	//The resource being deleted.
	staticFRHIResource* CurrentlyDeleting;
```

```
    bool DeferDelete()const;

    // someAPINo internal reference counting is done, so you must wait a few extra frames before deleting the resource to ensure thatGPUComplete them completely. Avoid expensive Fences, etc.
    struct ResourcesToDelete
    {
        TArray<FRHIResource*> Resources; // The resource to be deleted.
        uint32 FrameDeleted; // The number of frames to wait.

        (.....)
    };

    //Delayed deletion of resource queues.
    static TArray<ResourcesToDelete> DeferredDeletionQueue;
    static uint32 CurrentFrame;
};
```

As can be seen above, FRHIResource provides several functions: reference counting, delayed deletion and tracking, runtime data and marking. It has a large number of subclasses, mainly:

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

// Status Block(State blocksresource

classFRHISamplerState: public FRHIResource {

public:
    virtualbool IsImmutable()const{returnfalse; }
};
classFRHIRasterizerState: public FRHIResource {

public:
    virtualbool GetInitializer(structFRasterizerStateInitializerRHI& Init) {return false; }

};
classFRHIDepthStencilState: public FRHIResource {

public:
    virtualbool GetInitializer(structFDepthStencilStateInitializerRHI& Init) {return false; }

};
classFRHIBlendState: public FRHIResource {

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

//Shader binding resources.

typedefTArray<structFVertexElement,TFixedAllocator<MaxVertexElementCount> >
FVertexDeclarationElementList;
classFRHIVertexDeclaration: public FRHIResource {

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

```
classFRHIBoundShaderState: public FRHIResource {};

//Shaders

classFRHIShader: public FRHIResource {

public:
	void SetHash(FSHAHash	InHash);
	FSHAHash	GetHash()const; FRHIShader
	explicit	(EShaderFrequency			InFrequency);
	inlineEShaderFrequencyGetFrequency()const;

private:
	FSHAHash Hash;
	EShaderFrequency	Frequency;
};

classFRHIGraphicsShader: public FRHIShader {

public:
	explicitFRHIGraphicsShader(EShaderFrequency InFrequency):FRHIShader(InFrequency) {}
};

classFRHIVertexShader: public FRHIGraphicsShader {

public:
	FRHIVertexShader() : FRHIGraphicsShader(SF_Vertex) {}
};

classFRHIHullShader: public FRHIGraphicsShader {

public:
	FRHIHullShader() : FRHIGraphicsShader(SF_Hull) {}
};

classFRHIDomainShader: public FRHIGraphicsShader {

public:
	FRHIDomainShader() : FRHIGraphicsShader(SF_Domain) {}
};

classFRHIPixelShader: public FRHIGraphicsShader {

public:
	FRHIPixelShader() : FRHIGraphicsShader(SF_Pixel) {}
};

classFRHIGeometryShader: public FRHIGraphicsShader {

public:
	FRHIGeometryShader() : FRHIGraphicsShader(SF_Geometry) {}
};

classRHI_API FRHIComputeShader: public FRHIShader {

public:
	FRHIComputeShader() : FRHIShader(SF_Compute), Stats(nullptr) {}
```

```
        inlinevoid SetStats(structFPipelineStateStats* Ptr) { Stats = Ptr; } void UpdateStats();


private:
        struct      FPipelineStateStats*            Stats;
};

//Pipeline Status

classFRHIGraphicsPipelineState: public FRHIResource {}; class
FRHIComputePipelineState: public FRHIResource {}; class
FRHIRayTracingPipelineState: public FRHIResource {};

//Buffer.

classFRHIUniformBuffer: public FRHIResource {

public:
        FRHIUniformBuffer(constFRHIUniformBufferLayout&                 InLayout);

        FORCEINLINE_DEBUGGABLE uint32AddRef()const;
        FORCEINLINE_DEBUGGABLE uint32Release()const; uint32
        GetSize()const;
        constFRHIUniformBufferLayout&GetLayout()const; bool
        HasStaticSlot()const;

private:
        const    FRHIUniformBufferLayout* Layout;
        uint32 LayoutConstantBufferSize;
};

classFRHIIndexBuffer: public FRHIResource {

public:
        FRHIIndexBuffer(uint32 InStride,uint32 InSize,uint32 InUsage);

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

protected:
        FRHIIndexBuffer();

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

private:
        uint32    Stride;
        uint32    Size;
        uint32    Usage;
};

classFRHIVertexBuffer: public FRHIResource {

public:
        FRHIVertexBuffer(uint32 InSize,uint32 InUsage) uint32GetSize
        ()const;
```

```
    uint32GetUsage()const;

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

private:
    uint32    Size;
    // eg        BUF_UnorderedAccess
    uint32    Usage;
};

classFRHIStructuredBuffer: public FRHIResource {

public:
    FRHIStructuredBuffer(uint32 InStride,uint32 InSize,uint32 InUsage)

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

private:
    uint32    Stride;
    uint32    Size;
    uint32    Usage;
};

//Texture

classFRHITexture: public FRHIResource {

public:
    FRHITexture(uint32 InNumMips, uint32 InNumSamples, EPixelFormat InFormat, uint32 InFlags,
FLastRenderTimeContainer* InLastRenderTime,constFClearValueBinding& InClearValue);


    //Dynamic type conversion interface.
    virtual class FRHITexture2D*GetTexture2D(); virtual class FRHITexture2DArray*
    GetTexture2DArray(); virtual class FRHITexture3D*GetTexture3D(); virtual class
    FRHITextureCube*GetTextureCube();  virtual  class  FRHITextureReference*
    GetTextureReference();


    virtual FIntVectorGetSizeXYZ()const=0; //Get the
    platform-specific native resource pointer.
    virtualvoid*GetNativeResource()const; virtualvoid*
    GetNativeShaderResourceView()const //Get platform-relatedRHI
    The base texture class. virtualvoid*GetTextureBaseRHI();


    //Data interface.
    uint32GetNumMips()        const;
    EPixelFormat      GetFormat();
    uint32    GetFlags()const;
    uint32 GetNumSamples()        const;
    bool IsMultisampled()        const;
    bool HasClearValue()       const;
```

```
        FLinearColorGetClearColor()const;
        void GetDepthStencilClearValue(float& OutDepth, uint32& OutStencil)const; float
                GetDepthClearValue()const;
        uint32     GetStencilClearValue()const;
        constFClearValueBindingGetClearBinding()const;
        virtualvoid GetWriteMaskProperties(void*& OutData, uint32& OutSize);

        (……)

        //RHIResource information.
        FRHIResourceInfo ResourceInfo;

private:
        //Texture data.
        FClearValueBinding          ClearValue;
        uint32 NumMips;
        uint32 NumSamples;
        EPixelFormat       Format;
        uint32 Flags;
        FLastRenderTimeContainer&          LastRenderTime;
        FLastRenderTimeContainer          DefaultLastRenderTime;
        FName TextureName;
};

// 2D RHITexture.
classFRHITexture2D: public FRHITexture {

public:
        FRHITexture2D(uint32 InSizeX,uint32 InSizeY,uint32 InNumMips,uint32
InNumSamples,EPixelFormat InFormat,uint32 InFlags,constFClearValueBinding& InClearValue);


        virtual FRHITexture2D*GetTexture2D() {returnthis; }

        uint32GetSizeX()const{returnSizeX; } uint32GetSizeY()const{
        returnSizeY; } inlineFIntPointGetSizeXY()const; virtual FIntVector
        GetSizeXYZ()constoverride;


private:
        uint32     SizeX;
        uint32     SizeY;
};

// 2D RHITexture array.
classFRHITexture2DArray: public FRHITexture2D {

public:
        FRHITexture2DArray(uint32 InSizeX,uint32 InSizeY,uint32 InSizeZ,uint32 InNumMips,uint32
NumSamples, EPixelFormat InFormat,uint32 InFlags,const FClearValueBinding& InClearValue);


        virtual FRHITexture2DArray*GetTexture2DArray() {returnthis; } virtual FRHITexture2D*
        GetTexture2D() {returnNULL; }

        uint32GetSizeZ()const{returnSizeZ; } virtual FIntVectorGetSizeXYZ()const
        final override;
```

```
private:
	uint32	SizeZ;
};

// 2D RHITexture.
classFRHITexture3D: public FRHITexture {

public:
	FRHITexture3D(uint32 InSizeX,uint32 InSizeY,uint32 InSizeZ,uint32 InNumMips,EPixelFormat InFormat,uint32
InFlags,constFClearValueBinding& InClearValue);

	virtual FRHITexture3D*GetTexture3D() {returnthis; } uint32GetSizeX()
	const{returnSizeX; } uint32GetSizeY()const{returnSizeY; } uint32GetSizeZ()
	const{returnSizeZ; } virtual FIntVectorGetSizeXYZ()constfinal override;



private:
	uint32	SizeX;
	uint32	SizeY;
	uint32	SizeZ;
};

//cubeRHITexture.
classFRHITextureCube: public FRHITexture {

public:
	FRHITextureCube(uint32 InSize,uint32 InNumMips,EPixelFormat InFormat,uint32 InFlags, const
FClearValueBinding& InClearValue);

	virtual	FRHITextureCube*GetTextureCube(); GetSize()
	uint32	const;
	virtual FIntVectorGetSizeXYZ()constfinal override;

private:
	uint32	Size;
};

//Texture references.
classFRHITextureReference: public FRHITexture {

public:
	explicitFRHITextureReference(FLastRenderTimeContainer*			InLastRenderTime);

	virtual FRHITextureReference*GetTextureReference() override {returnthis; } inlineFRHITexture*
	GetReferencedTexture()const; //Set the referenced texture

	void SetReferencedTexture(FRHITexture* InTexture); virtual FIntVector
	GetSizeXYZ()constfinal override;

private:
	//The texture resource being referenced.
	TRefCountPtr<FRHITexture>		ReferencedTexture;
};

classFRHITextureReferenceNullImpl: public FRHITextureReference {

public:
```

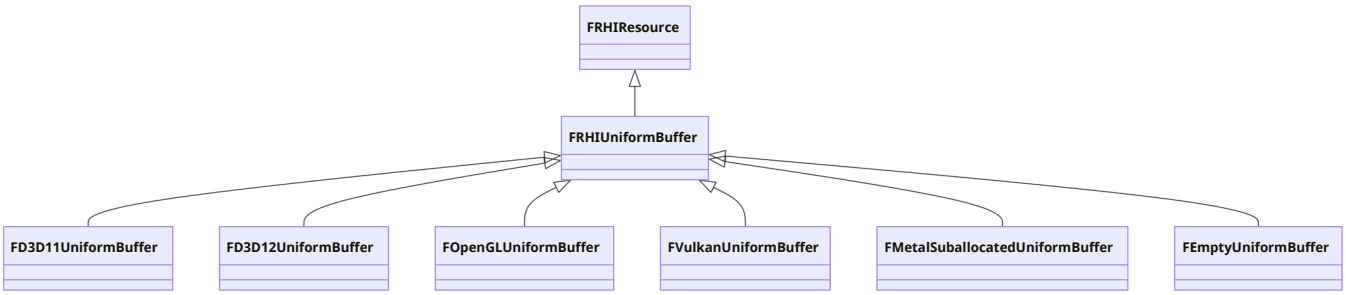
```
        FRHITextureReferenceNullImpl();

        void SetReferencedTexture(FRHITexture* InTexture) {

                FRHITextureReference::SetReferencedTexture(InTexture);
        }
};

//Miscellaneous resources.

//Timestamp calibration query.
classFRHITimestampCalibrationQuery: public FRHIResource {

public:
        uint64 GPUMicroseconds =0; uint64
        CPUMicroseconds =0;
};

//GPUFence class. GranularityRHI, i.e. it may only represent command buffer granularity.RHIThe special fence is derived from this, realizing the realGPU-
> CPUFence.

//The default implementation is always polling (Pollreturnfalse, until the next frame where the fence is inserted, because not allAPIBothGPU/CPUSynchronization object, you need to fake
it.
classFRHIGPUFence: public FRHIResource {

public:
        FRHIGPUFence(FName InName) : FenceName(InName) {} virtual
        ~FRHIGPUFence() {}

        virtualvoid Clear() =0;
        //Poll the fence and seeGPUWhether the signal has been sent. If so,
        returntrue. virtualbool Poll()const=0; //pollingGPUA subset of .

        virtualbool Poll(FRHIGPUMask GPUMask)const{returnPoll(); } //The number of
        pending write commands.
        FThreadSafeCounter NumPendingWriteCommands;

protected:
        FName   FenceName;
};

//GeneralFRHIGPUFenceaccomplish.
classRHI_API FGenericRHIGPUFence: public FRHIGPUFence {

public:
        FGenericRHIGPUFence(FName InName);

        virtualvoid Clear() final override; virtualbool Poll()const
        final override; void WriteInternal();


private:
        uint32     InsertedFrameNumber;
};

//Rendering query.
classFRHIRenderQuery: public FRHIResource {

};
```

```
//Pooled rendering queries.
classRHI_API FRHIPooledRenderQuery {

	TRefCountPtr<FRHIRenderQuery> Query;
	FRHIRenderQueryPool* QueryPool = nullptr;

public:
	bool IsValid()const;
	FRHIRenderQuery*GetQuery() void	const;
	ReleaseQuery();

	(.....)
};

//Rendering query pool.
classFRHIRenderQueryPool: public FRHIResource {

public:
	virtual ~FRHIRenderQueryPool() {};
	virtual FRHIPooledRenderQueryAllocateQuery() =0;

private:
	friendclassFRHIPooledRenderQuery;
	virtualvoid ReleaseQuery(TRefCountPtr<FRHIRenderQuery>&& Query) =0;
};

//Compute fences.
classFRHIComputeFence: public FRHIResource {

public:
	FRHIComputeFence(FName InName);

	FORCEINLINEbool GetWriteEnqueued()const; virtual
		void Reset();
	virtual	void WriteFence();

private:
	//Marks whether the tag has been written to since it was created.	When the command is created, the queue is waiting for captureCPUonGPUWhen suspending, check this flag.
	boolbWriteEnqueued;
};

//Viewport.
classFRHIViewport: public FRHIResource {

public:
	//Get the platform-specific native swap chain.
	virtualvoid*GetNativeSwapChain()const{returnnullptr; } //Get the native
	BackBufferTexture.
	virtualvoid*GetNativeBackBufferTexture()const{returnnullptr; } //Get the nativeBackBuffer
	Rendering textures.
	virtualvoid*GetNativeBackBufferRT()const{returnnullptr; } //Get the native window.

	virtualvoid*GetNativeWindow(void**AddParam = nullptr)const{returnnullptr; }

	//Set on the viewportFRHICustomPresentofhandler.
	virtualvoid SetCustomPresent(class FRHICustomPresent*) {}
	virtual class FRHICustomPresent*GetCustomPresent()const{returnnullptr; }
```

```
    //Update the viewport on the game thread frame.
    virtualvoid Tick(floatDeltaTime) {}
};

//view:UAV/SRV

classFRHIUnorderedAccessView: public FRHIResource {}; class
FRHIShaderResourceView: public FRHIResource {};

//VariousRHIResource reference type
definition. typedefTRefCountPtr<FRHISamplerState> FSamplerStateRHIRef;
typedef    TRefCountPtr<FRHIRasterizerState> FRasterizerStateRHIRef;
typedef    TRefCountPtr<FRHIDepthStencilState> FDepthStencilStateRHIRef;
typedef
typedef    TRefCountPtr<FRHIVertexDeclaration> FVertexDeclarationRHIRef;
typedef    TRefCountPtr<FRHIVertexShader> FVertexShaderRHIRef;
typedef    TRefCountPtr<FRHIHullShader> FHullShaderRHIRef;
typedef    TRefCountPtr<FRHIDomainShader> TRefCoFDunotmPtari<nFSRhHadIGeerRoHmIeRterfy;Shader>
typedef    TRefCountPtr<FRHIComputeShader> TReFfPCioxuenlSthPatrd<eFrRHIIRaeyf;TracingShader>
typedef    TRefCountPtr<FRHIComputeFence> TRefCounFGtPetor<mFeRtHryIBShoaudnedrSRhHaIdReerfS;tate>
typedef    TRefCountPtr<FRHIUniformBuffer> TRefCouFnCtoPmtrp<FuRteHSIhInadexrRBHuflfReerf>; TR
typedef    efCountPtr<FRHIVertexBuffer> TRefCountPtr<FRFHRaISytTrruacctiunrgeSdhBaudfeferRr>H IRef;
typedef    TRefCountPtr<FRHITexture> FTextureRHIRef; TRFeCfoCmoupnuttPetFre<nFcReHRIHTeIRxteuf;re2D>
typedef    FTexture2DRHIRef; TRefCountPtr<FRHITexture2FDBAorurnayd>S haderStateRHIRef;
typedef    FTexture2DArrayRHIRef; TRefCountPtr<FRHFITUenxiftourme3BDu>f fFeTreRxHtIuRref3;DRHIRef;
typedef    TRefCountPtr<FRHITextureCube> FTextuFreInCduebxeBRuHffIeRreRf;H IRef;
typedef    TRefCountPtr<FRHITextureReference> FTeFxVtuerteeRxeBfuerffeenrcReHRIHRIeRf;ef;
typedef    TRefCountPtr<FRHIRenderQuery> FRenderQuerFySRtHruIcRteufr; edBufferRHIRef;
typedef    TRefCountPtr<FRHIRenderQueryPool> FRenderQueryPoolRHIRef;
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef                                                    FTimestampCalibrationQueryRHIRef;
typedef    TRefCountPtr<FRHIGPUFence>          FGPUFenceRHIRef;
typedef    TRefCountPtr<FRHIViewport>       FViewportRHIRef;
typedef    TRefCountPtr<FRHIUnorderedAccessView>       FUnorderedAccessViewRHIRef;
typedef    TRefCountPtr<FRHIShaderResourceView>     FShaderResourceViewRHIRef;
           TRefCountPtr<FRHIGraphicsPipelineState>        FGraphicsPipelineStateRHIRef;
           TRefCountPtr<FRHIRayTracingPipelineState>          FRayTracingPipelineStateRHIRef;


// FRHIGPUMemoryReadbackGeneric segment buffer class to
use. classFRHIStagingBuffer: public FRHIResource {

public:
    FRHIStagingBuffer();
    virtual ~FRHIStagingBuffer();
    virtualvoid*Lock(uint32 Offset, uint32 NumBytes) =0; virtualvoid Unlock() =
    0; protected:

    boolbIsLocked;
};
```

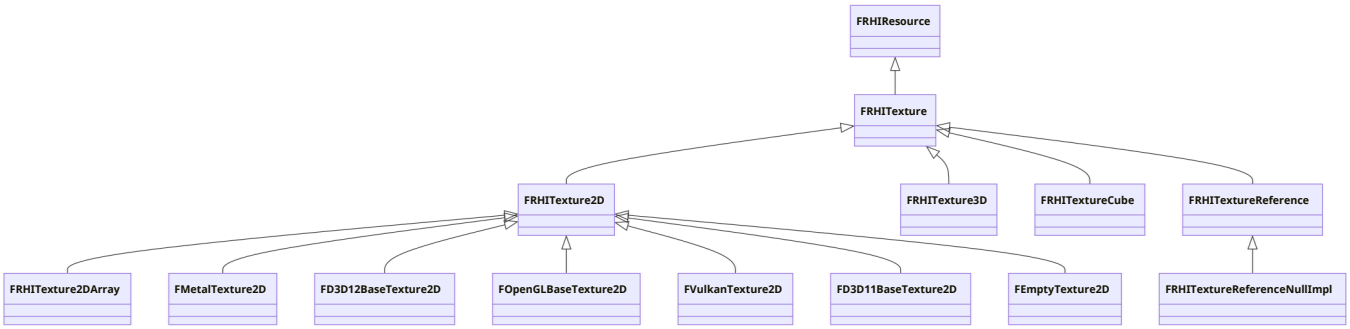
```
classFGenericRHIStagingBuffer: public FRHIStagingBuffer {

public:
        FGenericRHIStagingBuffer();
        ~FGenericRHIStagingBuffer();
        virtualvoid*Lock(uint32 Offset, uint32 NumBytes) final override; virtualvoid Unlock() final
        override;

        FVertexBufferRHIRef ShadowBuffer; uint32
        Offset;
};

//Custom rendering.
classFRHICustomPresent: public FRHIResource {

public:
        FRHICustomPresent() {}
        virtual ~FRHICustomPresent() {}

        //Called when the viewport size changes.
        virtualvoid OnBackBufferResize() =0; //Called from the
        rendering thread to see if a native render should be
        requested. virtualbool NeedsNativePresent() =0; //RHIThread
        call to perform custom rendering.
        virtualbool Present(int32& InOutSyncInterval) =0; //RHIThread call, in
        PresentThen call . virtualvoid PostPresent() {};


        //Called when the rendering thread is captured.
        virtualvoid OnAcquireThreadOwnership() {} //Called when the
        rendering thread is released.
        virtualvoid OnReleaseThreadOwnership() {}
};
```

As can be seen above, there are many types and subclasses of FRHIResource, which can be divided into state blocks, shader bindings, shaders, pipeline states, buffers, textures, views, and other miscellaneous items. It should be noted that the above only shows the platform-independent basic types. In fact, in different graphics APIs, the above types will be inherited. Taking FRHIUniformBuffer as an example, its inheritance system is as follows:

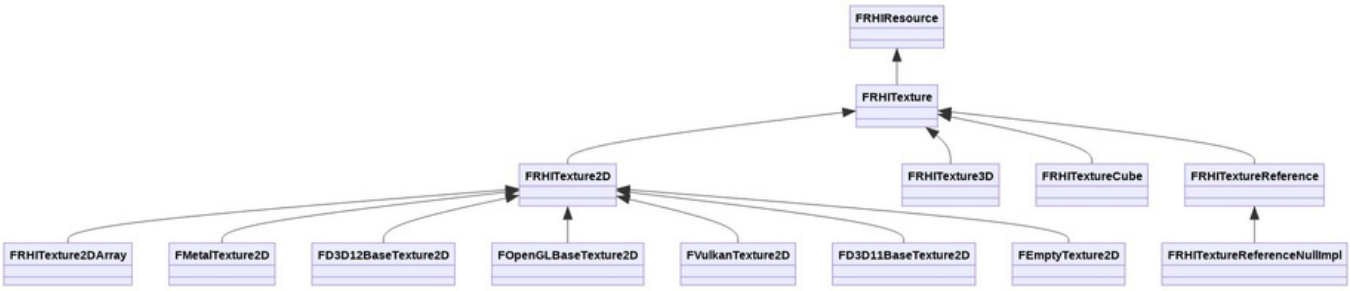


The above shows the subclasses of FRHIUniformBuffer in graphics APIs such as D3D11, D3D12, OpenGL, Vulkan, Metal, etc., in order to implement platform-dependent resources and operation interfaces for uniform buffers, and there is also a special empty implementation FEmptyUniformBuffer.

Similar to FRHIUniformBuffer, other direct or indirect subclasses of FRHIResource also need to be implemented by specific graphics API or operating system subclasses to support rendering on the platform. The following is a UML diagram of the most complex texture resource class inheritance system:



If you can't see it clearly, please click to enlarge the image below:



Note that the above diagram is simplified. In addition to FRHITexture2D being inherited by each graphics API, other texture types (such as FRHITexture2DArray, FRHITexture3D, FRHITextureCube, FRHITextureReference) will also be inherited and implemented by each platform.

10.2.3 FRHICommand

FRHICommand is the base class of rendering commands of the RHI module. These commands are usually pushed from the rendering thread to the RHI thread through the command queue and executed by the RHI thread at the right time. FRHICommand also inherits from FRHICommandBase, and their definitions are as follows:

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

//RHICommand base class.
struct      FRHICommandBase
{
        //    Next command. (node in command list)
        FRHICommandBase* Next = nullptr;

        //Destroyed after executing the command.
        virtualvoid ExecuteAndDestruct(FRHICommandListBase& CmdList,
FRHICommandListDebugContext& DebugContext) =0; };


emplate<typename TCmd, typename NameType = FUnnamedRhiCommand> struct
FRHICommand : public FRHICommandBase {
```

```
    //Destroyed after executing the command.
    voidExecuteAndDestruct(FRHICommandListBase& CmdList, FRHICommandListDebugContext& Context)
override final
    {
        TCmd *ThisCmd = static_cast<TCmd*>(this);
        ThisCmd->Execute(CmdList);
        ThisCmd->~TCmd();
    }
};
```

It is worth mentioning that FRHICommandBase has a Next variable pointing to the next node, which means that FRHICommandBase is a node in the command list. FRHICommand has a large number of subclasses, which are quickly declared through special macros:

```
//definitionRHIMacros for Command Subclasses
#defineFRHICOMMAND_MACRO(CommandName) struct                                    \
PREPROCESSOR_JOIN(CommandName##String, {                __LINE__)                \
                                                                                \
    static const TCHAR* TStr() { return TEXT(#CommandName); }                   \
};                                                                              \
//Command inheritedFRHICommand.
structCommandNamefinal: public FRHICommand<CommandName,
PREPROCESSOR_JOIN(CommandName##String, __LINE__)>
```

With the above macros, you can quickly define subclasses of FRHICommand (that is, specific RHI commands), for example:

```
FRHICOMMAND_MACRO(FRHICommandSetStencilRef) {

    uint32 StencilRef;
    FORCEINLINE_DEBUGGABLEFRHICommandSetStencilRef(uint32          InStencilRef)
        :StencilRef(InStencilRef)
    {
    }
    RHI_APIvoid Execute(FRHICommandListBase& CmdList);
};
```

After expanding the macro definition, the code is as follows:

```
structFRHICommandSetStencilRefString853 {

    static constTCHAR*TStr() {returnTEXT("FRHICommandSetStencilRef"); }
};

//FRHICommandSetStencilRefInheritedFRHICommand.
structFRHICommandSetStencilReffinal: public FRHICommand<FRHICommandSetStencilRef,
FRHICommandSetStencilRefString853>
{
    uint32 StencilRef;
    FRHICommandSetStencilRef(uint32 InStencilRef)
        : StencilRef(InStencilRef)
    {
    }
```

```
    RHI_APIvoid Execute(FRHICommandListBase& CmdList);
};
```

There are many RHI commands declared using FRHICOMMAND_MACRO, some of which are listed below:

```
FRHICOMMAND_MACRO(FRHISyncFrameCommand)
FRHICOMMAND_MACRO(FRHICommandStat)
FRHICOMMAND_MACRO(FRHICommandRHIThreadFence)
FRHICOMMAND_MACRO(FRHIAsyncComputeSubmitList)
FRHICOMMAND_MACRO(FRHICommandSubmitSubList)

FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitSubListParallel)
FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitSubList)
FRHICOMMAND_MACRO(FRHICommandWaitForAndSubmitRTSubList)
FRHICOMMAND_MACRO(FRHICommandWaitForTemporalEffect)
FRHICOMMAND_MACRO(FRHICommandWaitForTemp oralEffect)
FRHICOMMAND_MACRO(FRHICommandBroadcastTemporalEffect)

FRHICOMMAND_MACRO(FRHICommandBeginUpdateMultiFrameResource)
FRHICOMMAND_MACRO(FRHICommandEndUpdateMultiFrameResource)
FRHICOMMAND_MACRO(FRHICommandBeginUpdateMultiFrameUAV)
FRHICOMMAND_MACRO(FRHICommandEndUpdateMultiFrameUAV)
FRHICOMMAND_MACRO(FRHICommandSetGPUMask)

FRHICOMMAND_MACRO(FRHICommandSetStencilRef)
FRHICOMMAND_MACRO(FRHICommandSetBlendFactor)
FRHICOMMAND_MACRO(FRHICommandSetStreamSource)
FRHICOMMAND_MACRO(FRHICommandSetStreamSource)
FRHICOMMAND_MACRO(FRHICommandSetViewport)
FRHICOMMAND_MACRO(FRHICommandSetScissorRect)

FRHICOMMAND_MACRO(FRHICommandBeginRenderPass)
FRHICOMMAND_MACRO(FRHICommandEndRenderPass)
FRHICOMMAND_MACRO(FRHICommandNextSubpass)
FRHICOMMAND_MACRO(FRHICommandBeginParallelRenderPass)
FRHICOMMAND_MACRO(FRHICommandEndParallelRenderPass)
FRHICOMMAND_MACRO(FRHICommandBe ginRenderSubPass)
FRHICOMMAND_MACRO(FRHICommandEndRenderSubPass)

FRHICOMMAND_MACRO(FRHICommandDrawPrimitive)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedPrimitive)
FRHICOMMAND_MACRO(FRHICommandDrawPrimitiveIndirect)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedIndirect)
FRHICOMMAND_MACRO(FRHICommandDrawIndexedPrimitiveIndirect)

FRHICOMMAND_MACRO(FRHICommandSetGraphicsPipelineState)
FRHICOMMAND_MACRO(FRHICommandBeginUAVOverlap)
FRHICOMMAND_MACRO(FRHICommandEndUAVOverlap)

FRHICOMMAND_MACRO(FRHICommandSetDepthBounds)
FRHICOMMAND_MACRO(FRHICommandSetShadingRate)
FRHICOMMAND_MACRO(FRHICommandSetShadingRateImage)
FRHICOMMAND_MACRO(FRHICommandClearUAVFloat)
FRHICOMMAND_MACRO(FRHICommandCopyToResolveTarget)
FRHICOMMAND_MACRO(FR HICommandCopyTexture)
```

```
FRHICOMMAND_MACRO(FRHICommandBeginTransitions)
FRHICOMMAND_MACRO(FRHICommandEndTransitions)
FRHICOMMAND_MACRO(FRHICommandResourceTransition)
FRHICOMMAND_MACRO(FRHICommandClearColorTexture)
FRHICOMMAND_MACRO(FRHICommandClearDepthStencilTexture)
FRHICOMMAND_MACRO(FRHICommandClearColorTextures)

FRHICOMMAND_MACRO(FRHICommandSetGlobalUniformBuffers)
FRHICOMMAND_MACRO(FRHICommandBuildLocalUniformBuffer)

FRHICOMMAND_MACRO(FRHICommandBeginRenderQuery)
FRHICOMMAND_MACRO(FRHICommandEndRenderQuery)
FRHICOMMAND_MACRO(FRHICommandPollOcclusionQueries)

FRHICOMMAND_MACRO(FRHICommandBeginScene)
FRHICOMMAND_MACRO(FRHICommandEndScene)
FRHICOMMAND_MACRO(FRHICommandBeginFrame)
FRHICOMMAND_MACRO(FRHICommandEndFrame)
FRHICOMMAND_MACRO(FRHICommandBeginDrawingViewport)
FRHICOMMAND_MACRO(FRHICommandEndDrawingViewport)

FRHICOMMAND_MACRO(FRHICommandInvalidateCachedState)
FRHICOMMAND_MACRO(FRHICommandDiscardRenderTargets)

FRHICOMMAND_MACRO(FRHICommandUpdateTextureReference)
FRHICOMMAND_MACRO(FRHICommandUpdateRHIResources)
FRHICOMMAND_MACRO(FRHICommandBackBufferWaitTrackingBeginFrame)
FRHICOMMAND_MACRO(FRHICommandFlushTextureCacheBOP)
FRHICOMMAND_MACRO(FRHICommandCopyBufferRegion)
FRHICOMMAND_MACRO(FRHICommandCopyBufferRegions)

FRHICOMMAND_MACRO(FClearCachedRenderingDataCommand)
FRHICOMMAND_MACRO(FClearCachedElementDataCommand)

FRHICOMMAND_MACRO(FRHICommandRayTraceOcclusion)
FRHICOMMAND_MACRO(FRHICommandRayTraceIntersection)
FRHICOMMAND_MACRO(FRHICommandRayTraceDispatch)
FRHICOMMAND_MACRO(FRHICommandSetRayTracingBindings)
FRHICOMMAND_MACRO(FRHICommandClearRayTracingBindings)
```

In addition to the above subclasses declared with FRHICOMMAND_MACRO, the following subclasses of FRHICommand also have direct derivations:

- FRHICommandSetShaderParameter
- FRHICommandSetShaderUniformBuffer
- FRHICommandSetShaderTexture
- FRHICommandSetShaderResourceViewParameter
- FRHICommandSetUAVParameter
- FRHICommandSetShaderSampler
- FRHICommandSetComputeShader
- FRHICommandSetComputePipelineState
- FRHICommandDispatchComputeShader
- FRHICommandDispatchIndirectComputeShader

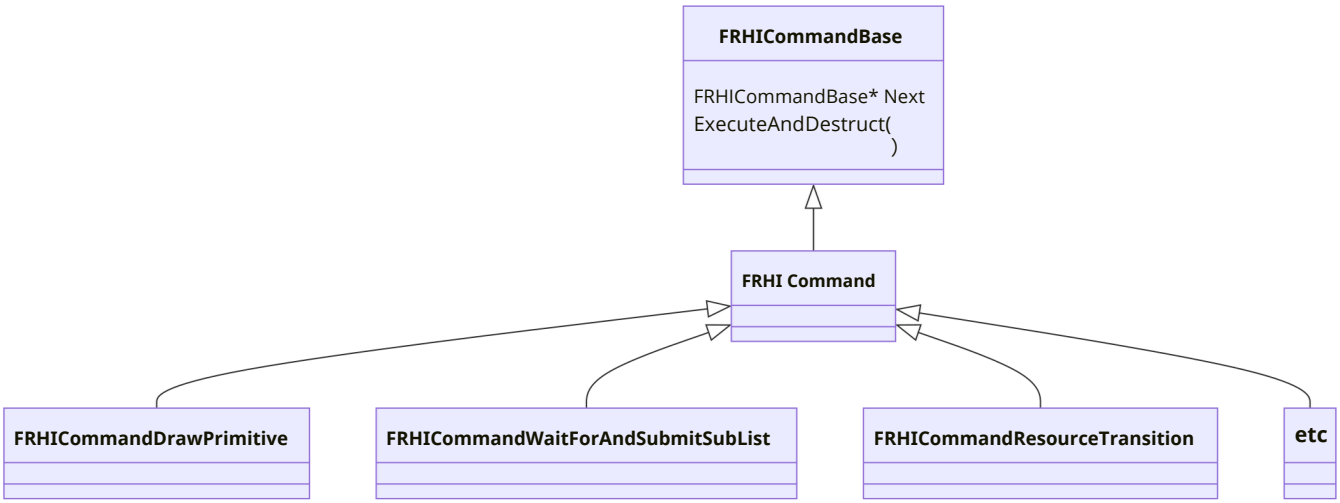
- FRHICommandSetAsyncComputeBudget
- FRHICommandCopyToStagingBuffer
- FRHICommandWriteGPUFence
- FRHICommandSetLocalUniformBuffer
- FRHICommandSubmitCommandsHint
- FRHICommandPushEvent
- FRHICommandPopEvent
- FRHICommandBuildAccelerationStructure
- FRHICommandBuildAccelerationStructures . . . . . .
-

There is no essential difference whether you derive directly or use FRHICOMMAND_MACRO. They are all subclasses of FRHICommand and can provide RHI layer intermediate rendering commands for rendering thread operations. It's just that using FRHICOMMAND_MACRO is simpler and requires less repetitive code.

Therefore, we can see that there are many types of RHI commands, mainly including the following categories:

- Setting, updating, cleaning, converting, copying, and reading back data and resources.
- Primitive drawing.
- Start and end events for Pass, SubPass, Scene, ViewPort, etc.
- Fence, wait, broadcast interface.
- Ray tracing.
- Slate, debugging related commands.

The core inheritance system of FRHICommand is drawn below:



10.2.4 FRHICommandList

FRHICommandList is the command queue of RHI, which is used to manage and execute a group of FRHICommand objects. Its definition and that of the parent class are as follows:

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

//RHICommand list base class.
classFRHICommandListBase: public FNoncopyable {

public:
	~FRHICommandListBase();

	//Comes with custom recyclenew/deleteoperate.
	void* operatornew(size_tSize); voidoperatordelete(
	void*RawMemory);

	//Flush the command queue.
	inlinevoid Flush(); //Whether to
	use immediate mode.
	inlinebool IsImmediate(); //Whether to perform
	asynchronous calculation immediately.
	inlinebool IsImmediateAsyncCompute();

	//Get the occupied memory.
	constint32GetUsedMemory()const;

	//Enqueue a submission to an asynchronous command queue.
	void QueueAsyncCommandListSubmit(FGraphEventRef& AnyThreadCompletionEvent, class
FRHICommandList* CmdList);
	//Enqueues submissions to a parallel asynchronous command queue.
	void QueueParallelAsyncCommandListSubmit(FGraphEventRef* AnyThreadCompletionEvents, boolbIsPrepass,
class FRHICommandList** CmdLists, int32* NumDrawsIfKnown, int32 Num, int32 MinDrawsPerTranslate,bool
bSpewMerge);
	//Enqueues a submission to the render thread command queue.
	void QueueRenderThreadCommandListSubmit(FGraphEventRef& RenderThreadCompletionEvent, class
FRHICommandList* CmdList);
	//Enqueue a submission to a command queue.
	void QueueCommandListSubmit(class FRHICommandList* CmdList); //Added
	dispatching of preceding tasks.
	void AddDispatchPrerequisite(constFGraphEventRef& Prereq);

	//Waiting for the interface.
	void WaitForTasks(boolbKnownToBeComplete =false); void
		WaitForDispatch();
	void WaitForRHIThreadTasks();
	void HandleRTThreadTaskCompletion(constFGraphEventRef& MyCompletionGraphEvent);

	//Assign interfaces.
	void*Alloc(int32 AllocSize, int32 Alignment); template
	<typename T>
	void*Alloc(); template
	<typename T>
	constTArrayView<T>AllocArray(constTArrayView<T> InArray); TCHAR*AllocString(
	constTCHAR* Name); //Assignment instructions.

	void*AllocCommand(int32 AllocSize, int32 Alignment); template
	<typename TCmd>
	void*AllocCommand();

	bool HasCommands() const;
	bool IsExecuting() const;
```

```
    bool IsBottomOfPipe()       const;
    bool IsTopOfPipe()      const;
    bool IsGraphics()       const;
    bool IsAsyncCompute()       const;
    //RHIPipeline,  ERHIPipeline::GraphicsorERHIPipeline::AsyncCompute.
    ERHIPipeline      GetPipeline()const;

    //IgnoreRHIThreads are executed directly
    synchronously. bool Bypass()const;

    //Swap command queues.
    void ExchangeCmdList(FRHICommandListBase& //set     Other);
    upContext.
    void SetContext(IRHICommandContext* InContext);
    IRHICommandContext&GetContext();
    void SetComputeContext(IRHIComputeContext* InComputeContext);
    IRHIComputeContext&GetComputeContext();
    void CopyContext(FRHICommandListBase& ParentCommandList);

    void  MaybeDispatchToRHIThread();
    void  MaybeDispatchToRHIThreadInner();

    (.....)

private:
    //The head of the command list.
    FRHICommandBase* Root;
    //PointingRootPointer to .
    FRHICommandBase**  CommandLink;

    boolbExecuting;
    uint32   NumCommands;
    uint32   UID;

    //The device context.
    IRHICommandContext*     Context;
    //Computation context.
    IRHIComputeContext*     ComputeContext;

    FMemStackBase MemManager;
    FGraphEventArray RTTasks;

    //Reset.
    void Reset();

public:
    enum class    ERenderThreadContext
    {
        SceneRenderTargets,
        Num
    };

    //The rendering thread context.
    void*RenderThreadContexts[(int32)ERenderThreadContext::Num];

protected:
    //the values of this struct must be copied when the commandlist is split structFPSOContext
```

```
    {
        uint32 CachedNumSimultanousRenderTargets =0;
        TStaticArray<FRHIRenderTargetView, MaxSimultaneousRenderTargets>
        CachedRenderTargets;
        FRHIDepthRenderTargetView CachedDepthStencilTarget;

        ESubpassHint SubpassHint = ESubpassHint::None; uint8
        SubpassIndex =0; uint8 MultiViewCount =0;

        boolHasFragmentDensityAttachment =false; PSOContext;
    }

    //The bound shader inputs.
    FBoundShaderStateInput BoundShaderInput; //Bound
    compute shadersRHIresource. FRHIComputeShader*
    BoundComputeShaderRHI;

    //Make the bound shader effective.
    void   ValidateBoundShader(FRHIVertexShader*          ShaderRHI);
    void   ValidateBoundShader(FRHIPixelShader*           ShaderRHI);
    (.....)

    void CacheActiveRenderTargets(...);
    void CacheActiveRenderTargets(constFRHIRenderPassInfo& Info); void
        IncrementSubpass();
    void   ResetSubpass(ESubpassHint SubpassHint);

public:
    void CopyRenderThreadContexts(constFRHICommandListBase& ParentCommandList); void
    SetRenderThreadContext(void* InContext, ERenderThreadContext Slot); void*GetRenderThreadContext
    (ERenderThreadContext Slot);

    //General data.
    structFCommonData
    {
        classFRHICommandListBase*Parent= nullptr;

        enum classECmdListType {

            Immediate =1,
            Regular,
        };
        ECmdListType Type = ECmdListType::Regular; bool
        bInsideRenderPass =false; boolbInsideComputePass =
        false;
    };

    bool DoValidation()const;
    inlinebool IsOutsideRenderPass()const; inlinebool
    IsInsideRenderPass()const; inlinebool
    IsInsideComputePass()const;

    FCommonData Data;
};

//Compute command queue.
classFRHIComputeCommandList: public FRHICommandListBase {
```

```
public:
    FRHIComputeCommandList(FRHIGPUMask GPUMask) : FRHICommandListBase(GPUMask) {}

    void* operatornew(size_tSize); voidoperatordelete(
    void*RawMemory);

    //Setting and getting shader parameters.
    inlineFRHIComputeShader*GetBoundComputeShader()const;
    void SetGlobalUniformBuffers(constFUniformBufferStaticBindings& UniformBuffers); void
    SetShaderUniformBuffer(FRHIComputeShader* Shader, uint32 BaseIndex, FRHIUniformBuffer*
    UniformBuffer);
    void SetShaderUniformBuffer(constFComputeShaderRHIRef& Shader, uint32 BaseIndex, FRHIUniformBuffer*
    UniformBuffer);
    void SetShaderParameter(FRHIComputeShader* Shader, uint32 BufferIndex, uint32 BaseIndex, uint32
NumBytes,const void* NewValue);
    void SetShaderParameter(FComputeShaderRHIRef& Shader, uint32 BufferIndex, uint32 BaseIndex, uint32
NumBytes,const void* NewValue);
    void SetShaderTexture(FRHIComputeShader* Shader, uint32 TextureIndex, FRHITexture* Texture);

    void SetShaderResourceViewParameter(FRHIComputeShader* Shader, uint32 SamplerIndex,
FRHIShaderResourceView* SRV);
    void SetShaderSampler(FRHIComputeShader* Shader, uint32 SamplerIndex,
FRHISamplerState* State);
    void SetUAVParameter(FRHIComputeShader* Shader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV);
    void SetUAVParameter(FRHIComputeShader* Shader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV, uint32 InitialCount);
    void   SetComputeShader(FRHIComputeShader* ComputeShader); SetComputePipelineState
    void   (FComputePipelineState* ComputePipelineState,
FRHIComputeShader*     ComputeShader);

    void SetAsyncComputeBudget(EAsyncComputeBudget Budget); //Dispatching a
    compute shader.
    void DispatchComputeShader(uint32 ThreadGroupCountX, uint32 ThreadGroupCountY, uint32
    ThreadGroupCountZ);
    void DispatchIndirectComputeShader(FRHIVertexBuffer* ArgumentBuffer, uint32 ArgumentOffset);


    //Clean up.
    void ClearUAVFloat(FRHIUnorderedAccessView* UnorderedAccessViewRHI,constFVector4& Values);

    void ClearUAVUint(FRHIUnorderedAccessView* UnorderedAccessViewRHI,constFUintVector4& Values);


    //Resource conversion.
    void BeginTransitions(TArrayView<constFRHITransition*> Transitions); void EndTransitions
    (TArrayView<constFRHITransition*> Transitions); inlinevoid Transition(TArrayView<const
    FRHITransitionInfo>Infos); void BeginTransition(constFRHITransition* Transition); void
    EndTransition(constFRHITransition* Transition); void Transition(constFRHITransitionInfo&
    Info)


    //----OldAPI ----

    void TransitionResource(ERHIAccess TransitionType,constFTextureRHIRef& InTexture); void TransitionResource
    (ERHIAccess TransitionType, FRHITexture* InTexture); inlinevoid TransitionResources(ERHIAccess TransitionType,
    FRHITexture*const* InTextures, int32 NumTextures);
```

```
    void TransitionResourceArrayNoCopy(ERHIAccess TransitionType, TArray<FRHITexture*>& InTextures);

    inlinevoid TransitionResources(ERHIAccess TransitionType, EResourceTransitionPipeline /* ignored
TransitionPipeline */, FRHIUnorderedAccessView*const* InUAVs, int32 NumUAVs, FRHIComputeFence* WriteFence);

    void TransitionResource(ERHIAccess TransitionType, EResourceTransitionPipeline TransitionPipeline,
FRHIUnorderedAccessView* InUAV, FRHIComputeFence* WriteFence);
    void TransitionResource(ERHIAccess TransitionType, EResourceTransitionPipeline TransitionPipeline,
FRHIUnorderedAccessView* InUAV);
    void TransitionResources(ERHIAccess TransitionType, EResourceTransitionPipeline TransitionPipeline,
FRHIUnorderedAccessView*const* InUAVs, int32 NumUAVs);
    void WaitComputeFence(FRHIComputeFence* WaitFence);

    void BeginUAVOverlap();
    void EndUAVOverlap();
    void   BeginUAVOverlap(FRHIUnorderedAccessView*            UAV);
    void   EndUAVOverlap(FRHIUnorderedAccessView* UAV);
    void BeginUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs); void
    EndUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs);

    void PushEvent(constTCHAR* Name, FColor Color); void
        PopEvent();
    void BreakPoint();

    void SubmitCommandsHint();
    void CopyToStagingBuffer(FRHIVertexBuffer* SourceBuffer, FRHIStagingBuffer*
DestinationStagingBuffer, uint32 Offset, uint32 NumBytes);

    void   WriteGPUFence(FRHIGPUFence* Fence);
    void   SetGPUMask(FRHIGPUMask InGPUMask);

    (.....)
};

//RHICommand queue.
classFRHICommandList: public FRHIComputeCommandList {

public:
    FRHICommandList(FRHIGPUMask GPUMask) : FRHIComputeCommandList(GPUMask) {}

    bool AsyncPSOCompileAllowed()const;

    void* operatornew(size_tSize); voidoperatordelete(
    void*RawMemory);

    //Get the bound shader.
    inlineFRHIVertexShader*GetBoundVertexShader()const; inlineFRHIHullShader*
    GetBoundHullShader()const; inlineFRHIDomainShader*
    GetBoundDomainShader()const; inlineFRHIPixelShader*GetBoundPixelShader
    ()const; inlineFRHIGeometryShader*GetBoundGeometryShader()const;



    //Updates a multi-frame resource.
    void   BeginUpdateMultiFrameResource(FRHITexture* Texture);
    void   EndUpdateMultiFrameResource(FRHITexture* Texture);
    void   BeginUpdateMultiFrameResource(FRHIUnorderedAccessView*                UAV);
    void   EndUpdateMultiFrameResource(FRHIUnorderedAccessView* UAV);
```

```
// Uniform Bufferinterface.
FLocalUniformBufferBuildLocalUniformBuffer(const void* Contents, uint32 ContentsSize, const
FRHIUniformBufferLayout& Layout);
template <typename TRHIShader>
void SetLocalShaderUniformBuffer(TRHIShader* Shader, uint32 BaseIndex,const
FLocalUniformBuffer& UniformBuffer);
template <typename TShaderRHI>
void SetLocalShaderUniformBuffer(constTRefCountPtr<TShaderRHI>& Shader, uint32 BaseIndex,const
FLocalUniformBuffer& UniformBuffer);
void SetShaderUniformBuffer(FRHIGraphicsShader* Shader, uint32 BaseIndex,
FRHIUniformBuffer* UniformBuffer);
template <typename TShaderRHI>
FORCEINLINEvoid SetShaderUniformBuffer(constTRefCountPtr<TShaderRHI>& Shader, uint32 BaseIndex,
FRHIUniformBuffer* UniformBuffer);

//Shader parameters.
void SetShaderParameter(FRHIGraphicsShader* Shader, uint32 BufferIndex, uint32 BaseIndex, uint32
NumBytes,const void* NewValue);
template <typename TShaderRHI>
void SetShaderParameter(constTRefCountPtr<TShaderRHI>& Shader, uint32 BufferIndex, uint32 BaseIndex,
uint32 NumBytes,const void* NewValue);
void SetShaderTexture(FRHIGraphicsShader* Shader, uint32 TextureIndex, FRHITexture* Texture);

template <typename TShaderRHI>
void SetShaderTexture(constTRefCountPtr<TShaderRHI>& Shader, uint32 TextureIndex, FRHITexture* Texture);

void SetShaderResourceViewParameter(FRHIGraphicsShader* Shader, uint32 SamplerIndex,
FRHIShaderResourceView*SRV);
template <typename TShaderRHI>
void SetShaderResourceViewParameter(constTRefCountPtr<TShaderRHI>& Shader, uint32 SamplerIndex,
FRHIShaderResourceView* SRV);
void SetShaderSampler(FRHIGraphicsShader* Shader, uint32 SamplerIndex,
FRHISamplerState* State);
template <typename TShaderRHI>
void SetShaderSampler(constTRefCountPtr<TShaderRHI>& Shader, uint32 SamplerIndex, FRHISamplerState*
State);
void SetUAVParameter(FRHIPixelShader* Shader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV);
void SetUAVParameter(constTRefCountPtr<FRHIPixelShader>& Shader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV);
void SetBlendFactor(constFLinearColor& BlendFactor = FLinearColor::White);

//Primitive drawing.
void DrawPrimitive(uint32 BaseVertexIndex, uint32 NumPrimitives, uint32 NumInstances); void
DrawIndexedPrimitive(FRHIIndexBuffer* IndexBuffer, int32 BaseVertexIndex, uint32 FirstInstance, uint32
NumVertices, uint32 StartIndex, uint32 NumPrimitives, uint32 NumInstances);

void DrawPrimitiveIndirect(FRHIVertexBuffer* ArgumentBuffer, uint32 ArgumentOffset); void
DrawIndexedIndirect(FRHIIndexBuffer* IndexBufferRHI, FRHIStructuredBuffer* ArgumentsBufferRHI, uint32
DrawArgumentsIndex, uint32 NumInstances);
void DrawIndexedPrimitiveIndirect(FRHIIndexBuffer* IndexBuffer, FRHIVertexBuffer* ArgumentsBuffer,
uint32 ArgumentOffset);

//Set up the data.
void SetStreamSource(uint32 StreamIndex, FRHIVertexBuffer* VertexBuffer, uint32 Offset);

void  SetStencilRef(uint32 StencilRef);
void SetViewport(floatMinX,floatMinY,floatMinZ,floatMaxX,floatMaxY,float
```

```
MaxZ);
        void SetStereoViewport(floatLeftMinX,floatRightMinX,floatLeftMinY,float RightMinY,floatMinZ,floatLeftMaxX,float
RightMaxX,floatLeftMaxY,floatRightMaxY, floatMaxZ);

        void SetScissorRect(boolbEnable, uint32 MinX, uint32 MinY, uint32 MaxX, uint32 MaxY); void
        ApplyCachedRenderTargets(FGraphicsPipelineStateInitializer& GraphicsPSOInit); void SetGraphicsPipelineState(class
        FGraphicsPipelineState* GraphicsPipelineState, constFBoundShaderStateInput& ShaderInput,bool
        bApplyAdditionalState);
        void SetDepthBounds(floatMinDepth,floatMaxDepth);
        void SetShadingRate(EVRSShadingRate ShadingRate, EVRSRateCombiner Combiner); void SetShadingRateImage
        (FRHITexture* RateImageTexture, EVRSRateCombiner Combiner);


        //Copy the texture.
        void CopyToResolveTarget(FRHITexture* SourceTextureRHI, FRHITexture* DestTextureRHI, const
FResolveParams& ResolveParams);
        void CopyTexture(FRHITexture* SourceTextureRHI, FRHITexture* DestTextureRHI,const FRHICopyTextureInfo&
        CopyInfo);

        void ResummarizeHTile(FRHITexture2D* DepthTexture);

        //Rendering query.
        void    BeginRenderQuery(FRHIRenderQuery* RenderQuery)
        void    EndRenderQuery(FRHIRenderQuery* RenderQuery)
        void    CalibrateTimers(FRHITimestampCalibrationQuery*                CalibrationQuery);
        void    PollOcclusionQueries()

        /* LEGACY API */
        void TransitionResource(FExclusiveDepthStencil DepthStencilMode, FRHITexture* DepthTexture);

        void BeginRenderPass(constFRHIRenderPassInfo& InInfo,constTCHAR* Name); void
                EndRenderPass();
        void NextSubpass();

        //The following interface needs to be called in the command queue of immediate mode.
        void BeginScene();
        void EndScene();
        void BeginDrawingViewport(FRHIViewport* Viewport, FRHITexture* RenderTargetRHI); void
        EndDrawingViewport(FRHIViewport* Viewport,boolbPresent,boolbLockToVsync); void
                BeginFrame();
        void EndFrame();

        void RHIInvalidateCachedState();
        void DiscardRenderTargets(boolDepth,boolStencil, uint32 ColorBitMask);

        void CopyBufferRegion(FRHIVertexBuffer* DestBuffer, uint64 DstOffset, FRHIVertexBuffer*
SourceBuffer, uint64 SrcOffset, uint64 NumBytes);

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

FRHICommandListBase defines the basic data (command list, device context) and interfaces (refresh, wait, queue, dispatch, memory allocation, etc.) required by the command queue. FRHIComputeCommandList defines interfaces related to compute shaders, GPU resource state transitions, and settings of some shader parameters. FRHICommandList defines the interface of the

general rendering pipeline, including VS, PS, GS binding, primitive drawing, more shader parameter settings and resource state transitions, resource creation, update, and wait, etc.

FRHICommandList also has several subclasses, defined as follows:

```
//The immediate mode command queue.
classFRHICommandListImmediate: public FRHICommandList {


    //Command anonymous function.
    template <typename LAMBDA>
    structTRHILambdaCommandfinal: public FRHICommandBase {

        LAMBDA Lambda;

        void ExecuteAndDestruct(FRHICommandListBase& CmdList,
FRHICommandListDebugContext&) override final;
    };

    FRHICommandListImmediate();
    ~FRHICommandListImmediate();

public:
    //Refresh command immediately.
    void ImmediateFlush(EImmediateFlushType::Type //blockRHIFlushType);
    Threads.
    bool StallRHIThread(); //
    UnblockRHIThreads.
    void UnStallRHIThread(); //Is it
    blocked?
    static bool IsStalled();


    void SetCurrentStat(TStatId Stat);

    static    FGraphEventRefRenderThreadTaskFence();
    static    FGraphEventArray&GetRenderThreadTaskArray();
    static void WaitOnRenderThreadTaskFence(FGraphEventRef& Fence); static bool
    AnyRenderThreadTasksOutstanding(); FGraphEventRefRHIThreadFence(bool
    bSetLockFence =false);


    //Sort the given asynchronous compute command list into the order of the current immediate command list.
    void QueueAsyncCompute(FRHIComputeCommandList& RHIComputeCmdList);

    bool IsBottomOfPipe();
    bool IsTopOfPipe();
    template    <typename LAMBDA>
    void EnqueueLambda(LAMBDA&&    Lambda);

    //Resource creation.
    FSamplerStateRHIRefCreateSamplerState(constFSamplerStateInitializerRHI& Initializer) FRasterizerStateRHIRef
    CreateRasterizerState(constFRasterizerStateInitializerRHI& Initializer)

    FDepthStencilStateRHIRefCreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer)
    FBlendStateRHIRefCreateBlendState(constFBlendStateInitializerRHI& Initializer) FPixelShaderRHIRef
    CreatePixelShader(TArrayView<constuint8> Code,constFSHAHash& Hash)

    FVertexShaderRHIRefCreateVertexShader(TArrayView<constuint8> Code,constFSHAHash&
```

```
Hash)
    FHullShaderRHIRefCreateHullShader(TArrayView<constuint8> Code,constFSHAHash& Hash) FDomainShaderRHIRef
    CreateDomainShader(TArrayView<constuint8> Code,constFSHAHash& Hash)

    FGeometryShaderRHIRefCreateGeometryShader(TArrayView<constuint8> Code,const FSHAHash& Hash)

    FComputeShaderRHIRefCreateComputeShader(TArrayView<constuint8> Code,constFSHAHash& Hash)

    FComputeFenceRHIRefCreateComputeFence(constFName& Name)
    FGPUFenceRHIRefCreateGPUFence(constFName& Name) FStagingBufferRHIRef
    CreateStagingBuffer() FBoundShaderStateRHIRefCreateBoundShaderState(...)
    FGraphicsPipelineStateRHIRefCreateGraphicsPipelineState(const
    FGraphicsPipelineStateInitializer& Initializer)

    TRefCountPtr<FRHIComputePipelineState>CreateComputePipelineState(FRHIComputeShader* ComputeShader)

    FUniformBufferRHIRefCreateUniformBuffer(...)
    FIndexBufferRHIRefCreateAndLockIndexBuffer(uint32 Stride, uint32 Size, EBufferUsageFlags InUsage, ERHIAccess
InResourceState, FRHIResourceCreateInfo& CreateInfo, void*& OutDataBuffer)

    FIndexBufferRHIRefCreateAndLockIndexBuffer(uint32 Stride, uint32 Size, uint32 InUsage,
FRHIResourceCreateInfo& CreateInfo,void*& OutDataBuffer)

    //Vertex/index interface.
    void*LockIndexBuffer(FRHIIndexBuffer* IndexBuffer, uint32 Offset, uint32 SizeRHI, EResourceLockMode
    LockMode);
    void UnlockIndexBuffer(FRHIIndexBuffer* IndexBuffer);
    void*LockStagingBuffer(FRHIStagingBuffer* StagingBuffer, FRHIGPUFence* Fence, uint32 Offset, uint32 SizeRHI);

    void UnlockStagingBuffer(FRHIStagingBuffer* StagingBuffer);
    FVertexBufferRHIRefCreateAndLockVertexBuffer(uint32 Size, EBufferUsageFlags InUsage,
. . . ) ;
    FVertexBufferRHIRefCreateAndLockVertexBuffer(uint32 Size, uint32 InUsage,
FRHIResourceCreateInfo& CreateInfo,void*& OutDataBuffer);
    void*LockVertexBuffer(FRHIVertexBuffer* VertexBuffer, uint32 Offset, uint32 SizeRHI, EResourceLockMode
    LockMode);
    void UnlockVertexBuffer(FRHIVertexBuffer* VertexBuffer);
    void CopyVertexBuffer(FRHIVertexBuffer* SourceBuffer, FRHIVertexBuffer* DestBuffer); void*
    LockStructuredBuffer(FRHIStructuredBuffer* StructuredBuffer, uint32 Offset, uint32 SizeRHI, EResourceLockMode
    LockMode);
    void UnlockStructuredBuffer(FRHIStructuredBuffer* StructuredBuffer);

    // UAV/SRVcreate.
    FUnorderedAccessViewRHIRefCreateUnorderedAccessView(FRHIStructuredBuffer* StructuredBuffer,
boolbUseUAVCounter,boolbAppendBuffer)
    FUnorderedAccessViewRHIRefCreateUnorderedAccessView(FRHITexture* Texture, uint32 MipLevel)

    FUnorderedAccessViewRHIRefCreateUnorderedAccessView(FRHITexture* Texture, uint32 MipLevel, uint8
    Format)
    FUnorderedAccessViewRHIRef      CreateUnorderedAccessView(FRHIVertexBuffer*      VertexBuffer,
uint8 Format)
    FUnorderedAccessViewRHIRef      CreateUnorderedAccessView(FRHIIndexBuffer*      IndexBuffer,
uint8 Format)
    FShaderResourceViewRHIRef      CreateShaderResourceView(FRHIStructuredBuffer*
    StructuredBuffer)
    FShaderResourceViewRHIRef      CreateShaderResourceView(FRHIVertexBuffer* VertexBuffer,
uint32 Stride, uint8 Format)
    FShaderResourceViewRHIRef      CreateShaderResourceView(const
```

```
FShaderResourceViewInitializer& Initializer)
        FShaderResourceViewRHIRefCreateShaderResourceView(FRHIIndexBuffer* Buffer)

        uint64     CalcTexture2DPlatformSize(...);
        uint64     CalcTexture3DPlatformSize(...);
        uint64     CalcTextureCubePlatformSize(...);

        //Texture operations.
        void    GetTextureMemoryStats(FTextureMemoryStats&              OutStats);
        bool GetTextureMemoryVisualizeData(...);
        void CopySharedMips(FRHITexture2D* DestTexture2D, FRHITexture2D* SrcTexture2D); void TransferTexture
        (FRHITexture2D* Texture, FIntRect Rect, uint32 SrcGPUIndex, uint32 DestGPUIndex,boolPullData);

        void TransferTextures(constTArrayView<constFTransferTextureParams> Params); void GetResourceInfo
        (FRHITexture* Ref, FRHIResourceInfo& OutInfo); FShaderResourceViewRHIRefCreateShaderResourceView
        (FRHITexture* Texture,const FRHITextureSRVCreateInfo& CreateInfo);

        FShaderResourceViewRHIRefCreateShaderResourceView(FRHITexture* Texture, uint8 MipLevel);

        FShaderResourceViewRHIRefCreateShaderResourceView(FRHITexture* Texture, uint8 MipLevel, uint8
NumMipLevels, uint8 Format);
        FShaderResourceViewRHIRef         CreateShaderResourceViewWriteMask(FRHITexture2D*
        Texture2DRHI);
        FShaderResourceViewRHIRef uint32CCroematepSuhteaMdermReosroySuirzceeViewFMask(FRHITexture2D*
                                                                                      Texture2DRHI);
        (FRHITexture* FTexture2DRHIRefAsyncReallocateTeTxetuxrtue2reDR(.H..I)); ;
        ETextureReallocationStatus
                                              FinalizeAsyncReallocateTexture2D(FRHITexture2D* Texture2D,
bool bBlockUntilCompleted);
        ETextureReallocationStatus            CancelAsyncReallocateTexture2D(FRHITexture2D* Texture2D,
bool bBlockUntilCompleted);
        void*LockTexture2D(...);
        void UnlockTexture2D(FRHITexture2D* Texture, uint32 MipIndex,boolbLockWithinMiptail, boolbFlushRHIThread =
true);
        void*LockTexture2DArray(...);
        void UnlockTexture2DArray(FRHITexture2DArray* Texture, uint32 TextureIndex, uint32 MipIndex,
                  boolbLockWithinMiptail);
        void    UpdateTexture2D(...);
        void    UpdateFromBufferTexture2D(...);
        FUpdateTexture3DData       BeginUpdateTexture3D(...);
        void    EndUpdateTexture3D(FUpdateTexture3DData& UpdateData); EndMultiUpdateTexture3D
        void    (TArray<FUpdateTexture3DData>& UpdateDataArray); UpdateTexture3D(...);
        void
        void*     LockTextureCubeFace(...);
        void UnlockTextureCubeFace(FRHITextureCube* Texture, ...);

        //Read texture surface data.
        void ReadSurfaceData(FRHITexture* Texture, ...); void
        ReadSurfaceData(FRHITexture* Texture, ...);
        void MapStagingSurface(FRHITexture* Texture,void*& OutData, int32& OutWidth, int32& OutHeight);

        void  MapStagingSurface(FRHITexture*  Texture,  ...);  void
        UnmapStagingSurface(FRHITexture*      Texture);      void
        ReadSurfaceFloatData(FRHITexture*   Texture,   ...);   void
        ReadSurfaceFloatData(FRHITexture*   Texture,   ...);   void
        Read3DSurfaceFloatData(FRHITexture* Texture,...);

        //Resource state transitions for the rendering thread.
        void AcquireTransientResource_RenderThread(FRHITexture* Texture);
```

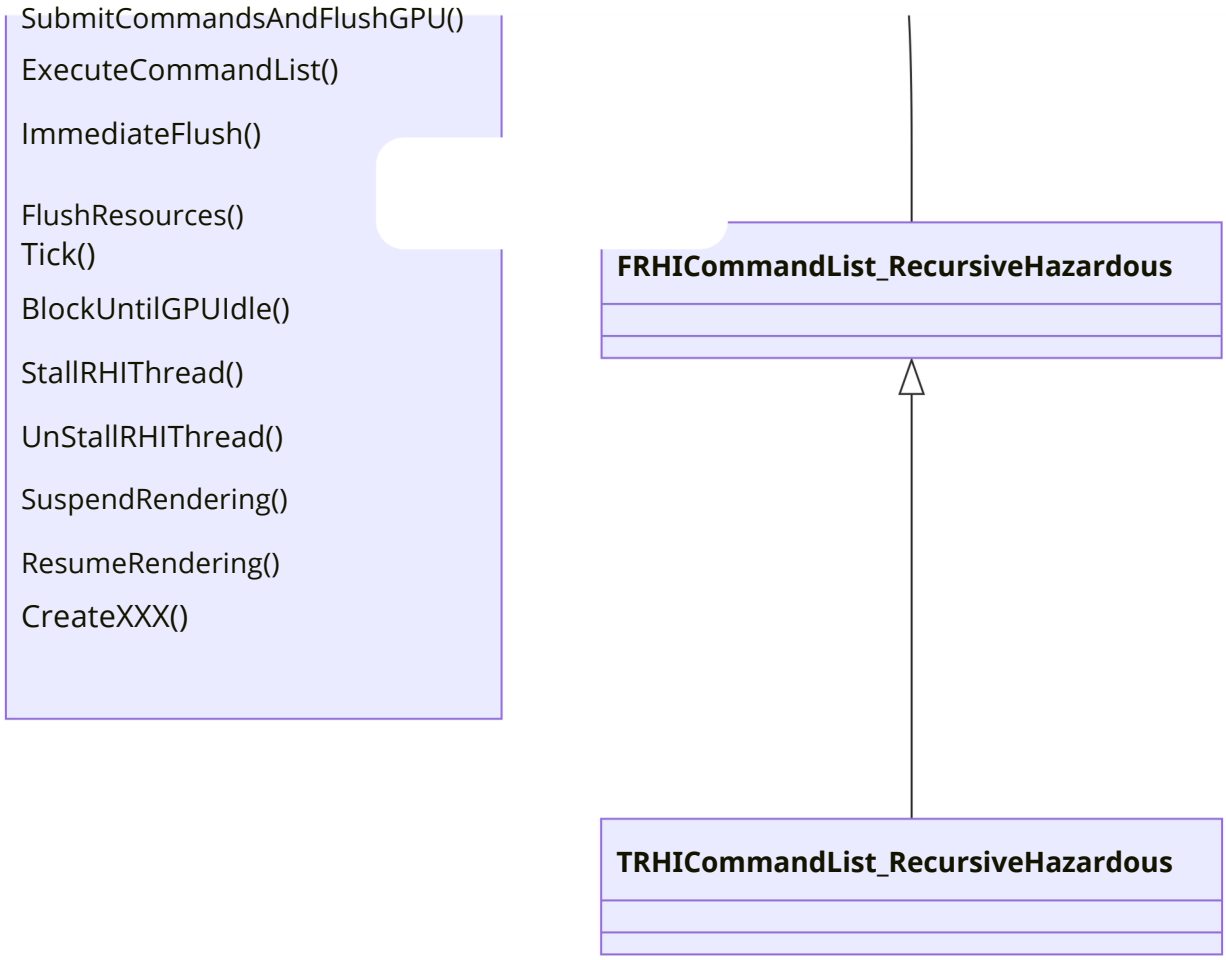
```
void DiscardTransientResource_RenderThread(FRHITexture* Texture);
void AcquireTransientResource_RenderThread(FRHIVertexBuffer* Buffer);
void DiscardTransientResource_RenderThread(FRHIVertexBuffer* Buffer);
void AcquireTransientResource_RenderThread(FRHIStructuredBuffer* Buffer);
void DiscardTransientResource_RenderThread(FRHIStructuredBuffer* Buffer);

//Get rendering query results.
bool GetRenderQueryResult(FRHIRenderQuery* RenderQuery, ...); void
PollRenderQueryResults();

//Viewport
FViewportRHIRefCreateViewport(void* WindowHandle, ...); uint32
GetViewportNextPresentGPUIndex(FRHIViewport* FTexture2DRHIRefViewport);
GetViewportBackBuffer(FRHIViewport* void Viewport);
AdvanceFrameForGetViewportBackBuffer(FRHIViewport* void Viewport);
ResizeViewport(FRHIViewport* Viewport, ...);

void AcquireThreadOwnership();
void ReleaseThreadOwnership();

//Submit the command and refresh toGPU.
void SubmitCommandsAndFlushGPU(); //Execute
the command queue.
void ExecuteCommandList(FRHICommandList* CmdList);

//Update resources.
void UpdateTextureReference(FRHITextureReference* TextureRef, FRHITexture* NewTexture);

void UpdateRHIResources(FRHIResourceUpdateInfo* UpdateInfos, int32 Num,bool bNeedReleaseRefs);

//Refresh resources.
void FlushResources();

//Frame update.
void Tick(floatDeltaTime); //Block
untilGPUidle.
void BlockUntilGPUIdle();

//Pause/start rendering.
void SuspendRendering();
void ResumeRendering();
bool IsRenderingSuspended();

//Compress/decompress data.
bool EnqueueDecompress(uint8_t* SrcBuffer,uint8_t* DestBuffer,intCompressedSize, void* ErrorCodeBuffer);

bool EnqueueCompress(uint8_t* SrcBuffer,uint8_t* DestBuffer,intUnCompressedSize, void* ErrorCodeBuffer);


//Other interfaces.
bool GetAvailableResolutions(FScreenResolutionArray& Resolutions,bool bIgnoreRefreshRate);

void GetSupportedResolution(uint32& Width, uint32& Height);
void VirtualTextureSetFirstMipInMemory(FRHITexture2D* Texture, uint32 FirstMip); void
VirtualTextureSetFirstMipVisible(FRHITexture2D* Texture, uint32 FirstMip);

//Get the original data.
void*GetNativeDevice();
```

```
        void*GetNativeInstance(); //Get the
        immediate mode command context.
        IRHICommandContext*      GetDefaultContext();
        //Get the command context container.
        IRHICommandContextContainer*GetCommandContextContainer(int32 Index, int32 Num);


        uint32GetGPUFrameCycles();
};

//existRHIType definitions for recursive use of tag command lists in implementations.
classFRHICommandList_RecursiveHazardous: public FRHICommandList {

public:
        FRHICommandList_RecursiveHazardous(IRHICommandContext *Context, FRHIGPUMask InGPUMask =
        FRHIGPUMask::All());
};

//RHIInternally used utility class for safer useFRHICommandList_RecursiveHazardous
template <typename ContextType>
classTRHICommandList_RecursiveHazardous: public FRHICommandList_RecursiveHazardous {

        template <typename LAMBDA>
        structTRHILambdaCommandfinal: public FRHICommandBase {

                LAMBDA Lambda;

                TRHILambdaCommand(LAMBDA&& InLambda); void
                ExecuteAndDestruct(FRHICommandListBase&                    CmdList,
FRHICommandListDebugContext&) override final;
        };

public:
        TRHICommandList_RecursiveHazardous(ContextType *Context, FRHIGPUMask GPUMask =
        FRHIGPUMask::All());

        template  <typename  LAMBDA>
        void RunOnContext(LAMBDA&&      Lambda);
};
```

FRHICommandListImmediate encapsulates the immediate mode graphics API interface, which is widely used in the UE rendering system. It additionally defines resource operation, creation, update, reading and state conversion interfaces, and also adds thread synchronization and GPU synchronization interfaces.

The following is a UML diagram to summarize the core inheritance system of FRHICommandList:


FNoncopyable
FRHICommandListBase
FRHICommandBase* Root
FRHICommandBase** CommandLink
IRHICommandContext* Context
IRHIComputeContext* ComputeContext
AllocCommand()
Flush()
WaitForXXX()
QueueCommandListXXX()
FRHIComputeCommandList
DispatchComputeShader()
DispatchIndirectComputeShader()
SetShaderXXX()
FRHICommandList
SetShaderXXX()
GetBoundXXXShader()
DrawPrimitive()
DrawXXX()
FRHICommandListImmediate



10.3 RHIContext, DynamicRHI

This chapter will explain the concepts, types, and associations of RHI Context and DynamicRHI.

10.3.1 IRHICommandContext

IRHICommandContext is the command context interface class of RHI, which defines a set of graphics API related operations. On platforms that can process command lists in parallel, it is a separate object. It and related inherited types are defined as follows:

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

//A context in which computational work can be performed.gfxPerform asynchronous or computation on
a pipeline. classIRHIComputeContext {

public:
	virtual ~IRHIComputeContext();

	//Set up/dispatch a compute shader.
	virtualvoid RHISetComputeShader(FRHIComputeShader* ComputeShader) =0; virtualvoid
	RHISetComputePipelineState(FRHIComputePipelineState* ComputePipelineState);

	virtualvoid RHIDispatchComputeShader(uint32 ThreadGroupCountX, uint32
ThreadGroupCountY, uint32 ThreadGroupCountZ) =0;
	virtualvoid RHIDispatchIndirectComputeShader(FRHIVertexBuffer* ArgumentBuffer, uint32
```

```
ArgumentOffset) =0;
        virtualvoid RHISetAsyncComputeBudget(EAsyncComputeBudget Budget) {}

        //Convert resources.
        virtualvoid RHIBeginTransitions(TArrayView<constFRHITransition*> Transitions) =0; virtualvoid RHIEndTransitions
        (TArrayView<constFRHITransition*> Transitions) =0;

        // UAV
        virtualvoid RHIClearUAVFloat(FRHIUnorderedAccessView* UnorderedAccessViewRHI,const FVector4& Values) =0;

        virtualvoid RHIClearUAVUint(FRHIUnorderedAccessView* UnorderedAccessViewRHI,const FUintVector4& Values)
=0;
        virtualvoid RHIBeginUAVOverlap() {} virtualvoid
        RHIEndUAVOverlap() {}
        virtualvoid RHIBeginUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs) {} virtualvoid
        RHIEndUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs) {}

        //Shader parameters.
        virtualvoid RHISetShaderTexture(FRHIComputeShader* PixelShader, uint32 TextureIndex, FRHITexture*
NewTexture) =0;
        virtualvoid RHISetShaderSampler(FRHIComputeShader* ComputeShader, uint32 SamplerIndex,
FRHISamplerState* NewState) =0;
        virtualvoid RHISetUAVParameter(FRHIComputeShader* ComputeShader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV) =0;
        virtualvoid RHISetUAVParameter(FRHIComputeShader* ComputeShader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV, uint32 InitialCount) =0;
        virtualvoid RHISetShaderResourceViewParameter(FRHIComputeShader* ComputeShader, uint32
SamplerIndex, FRHIShaderResourceView* SRV) =0;
        virtualvoid RHISetShaderUniformBuffer(FRHIComputeShader* ComputeShader, uint32 BufferIndex,
FRHIUniformBuffer* Buffer) =0;
        virtualvoid RHISetShaderParameter(FRHIComputeShader* ComputeShader, uint32 BufferIndex,
uint32 BaseIndex, uint32 NumBytes,const void* NewValue) =0;
        virtualvoid RHISetGlobalUniformBuffers(constFUniformBufferStaticBindings& InUniformBuffers);


        //Push/pop events.
        virtualvoid RHIPushEvent(constTCHAR* Name, FColor Color) =0; virtualvoid
        RHIPopEvent() =0;

        //Other interfaces.
        virtualvoid RHISubmitCommandsHint() =0; virtualvoid
        RHIInvalidateCachedState() {}
        virtualvoid RHICopyToStagingBuffer(FRHIVertexBuffer* SourceBufferRHI, FRHIStagingBuffer*
DestinationStagingBufferRHI, uint32 InOffset, uint32 InNumBytes);
        virtualvoid RHIWriteGPUFence(FRHIGPUFence* FenceRHI); virtualvoid
        RHISetGPUMask(FRHIGPUMask GPUMask);

        //Acceleration structure.
        virtualvoid RHIBuildAccelerationStructure(FRHIRayTracingGeometry* Geometry); virtualvoid
        RHIBuildAccelerationStructures(constTArrayView<const FAccelerationStructureBuildParams> Params);

        virtualvoid RHIBuildAccelerationStructure(FRHIRayTracingScene* Scene);

        //Get the computation context.
        inlineIRHIComputeContext&GetLowestLevelContext() {return*this; } inline
        IRHIComputeContext&GetHighestLevelContext() {return*this; }
};
```

```
//Command context.
classIRHICommandContext: public IRHIComputeContext {

public:
	virtual ~IRHICommandContext();

	//Dispatching calculations.
	virtualvoid RHIDispatchComputeShader(uint32 ThreadGroupCountX, uint32
ThreadGroupCountY, uint32 ThreadGroupCountZ) =0;
	virtualvoid RHIDispatchIndirectComputeShader(FRHIVertexBuffer* ArgumentBuffer, uint32 ArgumentOffset) =0;


	//Rendering query.
	virtualvoid RHIBeginRenderQuery(FRHIRenderQuery* RenderQuery) =0; virtualvoid
	RHIEndRenderQuery(FRHIRenderQuery* RenderQuery) =0; virtualvoid
	RHIPollOcclusionQueries();

	//Start/stop the interface.
	virtualvoid RHIBeginDrawingViewport(FRHIViewport* Viewport, FRHITexture* RenderTargetRHI) =0
	;
	virtualvoid RHIEndDrawingViewport(FRHIViewport* Viewport,boolbPresent,bool bLockToVsync) =0;

	virtualvoid RHIBeginFrame() =0; virtualvoid
	RHIEndFrame() =0; virtualvoid
	RHIBeginScene() =0; virtualvoid RHIEndScene
	() =0;
	virtualvoid RHIBeginUpdateMultiFrameResource(FRHITexture* Texture); virtualvoid
	RHIEndUpdateMultiFrameResource(FRHITexture* Texture); virtualvoid
	RHIBeginUpdateMultiFrameResource(FRHIUnorderedAccessView* UAV); virtualvoid
	RHIEndUpdateMultiFrameResource(FRHIUnorderedAccessView* UAV);

	//Set up the data.
	virtualvoid RHISetStreamSource(uint32 StreamIndex, FRHIVertexBuffer* VertexBuffer, uint32 Offset) =0;

	virtualvoid RHISetViewport(floatMinX,floatMinY,floatMinZ,floatMaxX,float MaxY,floatMaxZ) =0;

	virtualvoid RHISetStereoViewport(...);
	virtualvoid RHISetScissorRect(boolbEnable, uint32 MinX, uint32 MinY, uint32 MaxX, uint32 MaxY) =0;

	virtualvoid RHISetGraphicsPipelineState(FRHIGraphicsPipelineState* GraphicsState, boolbApplyAdditionalState)
=0;

	//Set shader parameters.
	virtualvoid RHISetShaderTexture(FRHIGraphicsShader* Shader, uint32 TextureIndex, FRHITexture*
NewTexture) =0;
	virtualvoid RHISetShaderTexture(FRHIComputeShader* PixelShader, uint32 TextureIndex, FRHITexture*
NewTexture) =0;
	virtualvoid RHISetShaderSampler(FRHIComputeShader* ComputeShader, uint32 SamplerIndex,
FRHISamplerState* NewState) =0;
	virtualvoid RHISetShaderSampler(FRHIGraphicsShader* Shader, uint32 SamplerIndex, FRHISamplerState*
NewState) =0;
	virtualvoid RHISetUAVParameter(FRHIPixelShader* PixelShader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV) =0;
	virtualvoid RHISetUAVParameter(FRHIComputeShader* ComputeShader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV) =0;
	virtualvoid RHISetUAVParameter(FRHIComputeShader* ComputeShader, uint32 UAVIndex,
FRHIUnorderedAccessView* UAV, uint32 InitialCount) =0;
	virtualvoid RHISetShaderResourceViewParameter(FRHIComputeShader* ComputeShader,
```

```
uint32 SamplerIndex, FRHIShaderResourceView* SRV) =0;
        virtualvoid RHISetShaderResourceViewParameter(FRHIGraphicsShader* Shader, uint32 SamplerIndex,
FRHIShaderResourceView* SRV) =0;
        virtualvoid RHISetShaderUniformBuffer(FRHIGraphicsShader* Shader, uint32 BufferIndex, FRHIUniformBuffer*
Buffer) =0;
        virtualvoid RHISetShaderUniformBuffer(FRHIComputeShader* ComputeShader, uint32 BufferIndex,
FRHIUniformBuffer* Buffer) =0;
        virtualvoid RHISetShaderParameter(FRHIGraphicsShader* Shader, uint32 BufferIndex, uint32 BaseIndex,
uint32 NumBytes,const void* NewValue) =0;
        virtualvoid RHISetShaderParameter(FRHIComputeShader* ComputeShader, uint32 BufferIndex,
uint32 BaseIndex, uint32 NumBytes,const void* NewValue) =0;
        virtualvoid RHISetStencilRef(uint32 StencilRef) {}
        virtualvoid RHISetBlendFactor(constFLinearColor& BlendFactor) {}


        //Draw the primitives.
        virtualvoid RHIDrawPrimitive(uint32 BaseVertexIndex, uint32 NumPrimitives, uint32 NumInstances) =0;

        virtualvoid RHIDrawPrimitiveIndirect(FRHIVertexBuffer* ArgumentBuffer, uint32 ArgumentOffset) =0;

        virtualvoid RHIDrawIndexedIndirect(FRHIIndexBuffer* IndexBufferRHI, FRHIStructuredBuffer* ArgumentsBufferRHI,
int32 DrawArgumentsIndex, uint32 NumInstances) = 0;

        virtualvoid RHIDrawIndexedPrimitive(FRHIIndexBuffer* IndexBuffer, int32 BaseVertexIndex, uint32
FirstInstance, uint32 NumVertices, uint32 StartIndex, uint32 NumPrimitives, uint32 NumInstances) =0;

        virtualvoid RHIDrawIndexedPrimitiveIndirect(FRHIIndexBuffer* IndexBuffer, FRHIVertexBuffer*
ArgumentBuffer, uint32 ArgumentOffset) =0;


        //Other interfaces
        virtualvoid RHISetDepthBounds(floatMinDepth,floatMaxDepth) =0; virtualvoid RHISetShadingRate
        (EVRSShadingRate ShadingRate, EVRSRateCombiner Combiner);

        virtualvoid RHISetShadingRateImage(FRHITexture* RateImageTexture, EVRSRateCombiner Combiner);

        virtualvoid RHISetMultipleViewports(uint32 Count,constFViewportBounds* Data) =0; virtualvoid
        RHICopyToResolveTarget(FRHITexture* SourceTexture, FRHITexture* DestTexture,constFResolveParams&
ResolveParams) =0;
        virtualvoid RHIResummarizeHTile(FRHITexture2D* DepthTexture); virtualvoid
        RHICalibrateTimers();
        virtualvoid RHICalibrateTimers(FRHITimestampCalibrationQuery* CalibrationQuery); virtualvoid
        RHIDiscardRenderTargets(boolDepth,boolStencil, uint32 ColorBitMask) {}


        //Texture
        virtualvoid RHIUpdateTextureReference(FRHITextureReference* TextureRef, FRHITexture* NewTexture) =0;

        virtualvoid RHICopyTexture(FRHITexture* SourceTexture, FRHITexture* DestTexture, const
FRHICopyTextureInfo& CopyInfo);
        virtualvoid RHICopyBufferRegion(FRHIVertexBuffer* DestBuffer, ...);

        // PassRelated.
        virtualvoid RHIBeginRenderPass(constFRHIRenderPassInfo& InInfo,constTCHAR* InName)
=  0;
        virtualvoid RHIEndRenderPass() =0; virtualvoid
        RHINextSubpass();

        //Ray tracing.
        virtualvoid RHIClearRayTracingBindings(FRHIRayTracingScene* Scene); virtualvoid
        RHIBuildAccelerationStructures(constTArrayView<const
```

```
FAccelerationStructureBuildParams> Params);
        virtualvoid RHIBuildAccelerationStructure(FRHIRayTracingGeometry* Geometry) final override;

        virtualvoid RHIBuildAccelerationStructure(FRHIRayTracingScene* Scene); virtualvoid
        RHIRayTraceOcclusion(FRHIRayTracingScene* Scene, ...); virtualvoid RHIRayTraceIntersection
        (FRHIRayTracingScene* Scene, ...);
        virtualvoid RHIRayTraceDispatch(FRHIRayTracingPipelineState* RayTracingPipelineState,
. . . ) ;
        virtualvoid RHISetRayTracingHitGroups(FRHIRayTracingScene* Scene, ...); virtualvoid
        RHISetRayTracingHitGroup(FRHIRayTracingScene* Scene, ...); virtualvoid
        RHISetRayTracingCallableShader(FRHIRayTracingScene* Scene, ...); virtualvoid
        RHISetRayTracingMissShader(FRHIRayTracingScene* Scene, ...);

        (.....)

protected:
        //RenderingPassinformation.
        FRHIRenderPassInfo        RenderPassInfo;
};
```

As can be seen above, the interface of IRHICommandContext is highly similar to and overlaps with the interface of FRHICommandList. IRHICommandContext also has many subclasses:

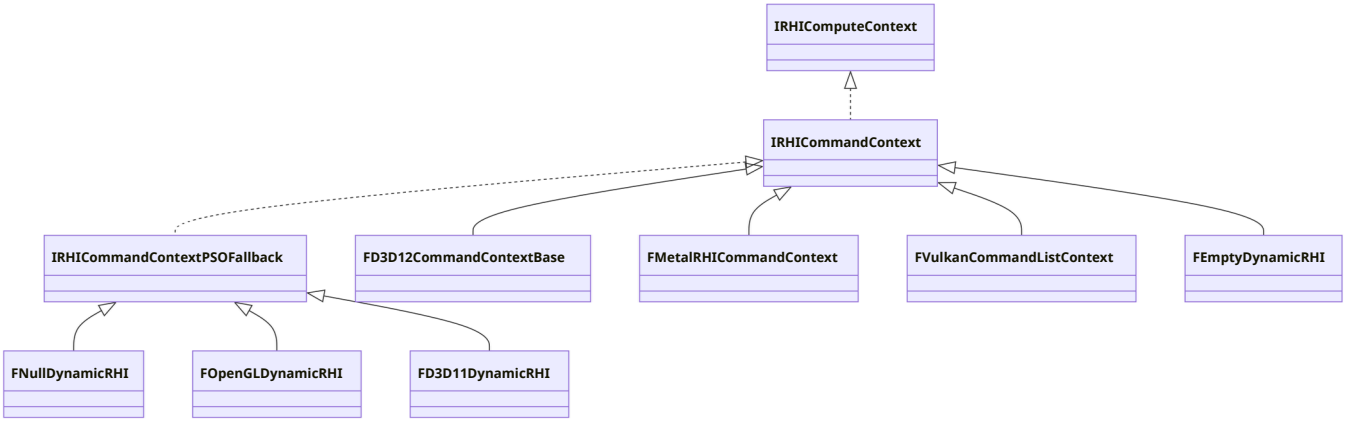
- IRHICommandContextPSOFallback: RHI command context that does not support real graphics pipeline.
    - FNullDynamicRHI: Dynamically bound RHI with null implementation.
    - FOpenGLDynamicRHI: Dynamic RHI of OpenGL.
    - FD3D11DynamicRHI: Dynamic RHI for D3D11.
- FMetalRHICommandContext: The command context of the Metal platform.
- FD3D12CommandContextBase: Command context for D3D12.
- FVulkanCommandListContext: The command queue context for the Vulkan platform.
- FEmptyDynamicRHI: Interface for dynamically bound RHI implementations.
- FValidationContext: validation context.

Among the above subclasses, some platform-related subclasses also inherit FDynamicRHI. IRHICommandContextPSOFallback is special. Its subclasses are all graphics APIs that do not support parallel drawing (OpenGL, D3D11). IRHICommandContextPSOFallback is defined as follows:

```
classIRHICommandContextPSOFallback: public IRHICommandContext {

public:
        //Set the rendering state.
        virtualvoid RHISetBoundShaderState(FRHIBoundShaderState* BoundShaderState) =0; virtualvoid
        RHISetDepthStencilState(FRHIDepthStencilState* NewState, uint32 StencilRef) =0;

        virtualvoid RHISetRasterizerState(FRHIRasterizerState* NewState) =0; virtualvoid RHISetBlendState
        (FRHIBlendState* NewState,constFLinearColor& BlendFactor) =0;
```

```
    virtualvoid RHIEnableDepthBoundsTest(boolbEnable) =0; //Pipeline status.

    virtualvoid RHISetGraphicsPipelineState(FRHIGraphicsPipelineState* GraphicsState, bApplyAdditionalState)
bool override;
};
```

The core inheritance UML diagram of IRHICommandContext is as follows:



10.3.2 IRHICommandContextContainer

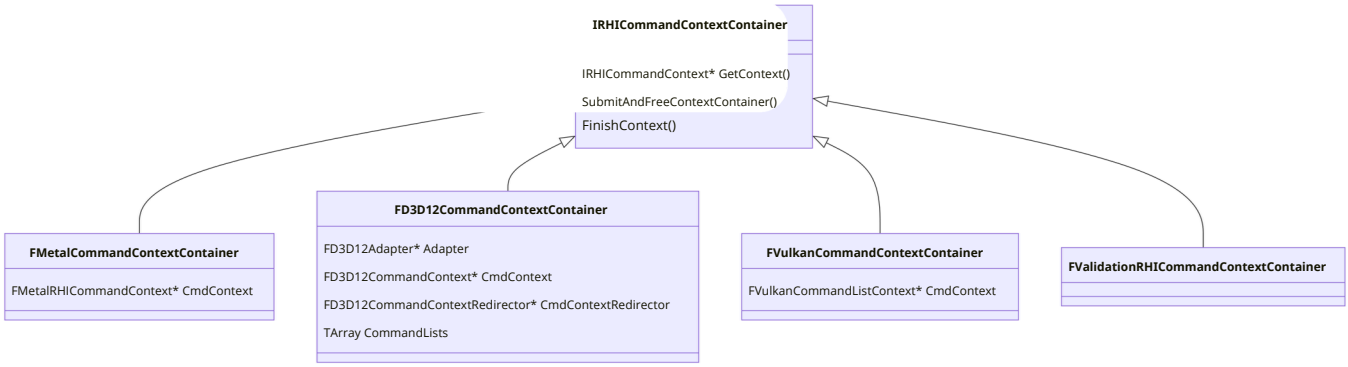
IRHICommandContextContainer is the type that contains the IRHICommandContext object. It and the core inherited subclasses are defined as follows:

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

classIRHICommandContextContainer {

public:
    virtual ~IRHICommandContextContainer();

    //GetIRHICommandContextExamples.
    virtual IRHICommandContext*GetContext();
    virtualvoid SubmitAndFreeContextContainer(int32 Index, int32 Num); virtualvoid
    FinishContext();
};

// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalContext.cpp

classFMetalCommandContextContainer: public IRHICommandContextContainer {

    // FMetalRHICommandContextNext in the list.
    FMetalRHICommandContext* int3C2mdContext;
            Index;
    int32   Num;

public:
    void* operatornew(size_tSize); voidoperatordelete(
    void*RawMemory);

    FMetalCommandContextContainer(int32 InIndex, int32 InNum); virtual
    ~FMetalCommandContextContainer() override final;

    virtual IRHICommandContext*GetContext() override final;
```

```
        virtualvoid FinishContext() override final; //Submit and free
        yourself.
        virtualvoid SubmitAndFreeContextContainer(int32 NewIndex, int32 NewNum) override final;

};

// FMetalCommandContextContainerAllocator.
staticTLockFreeFixedSizeAllocator<sizeof(FMetalCommandContextContainer), PLATFORM_CACHE_LINE_SIZE,
FThreadSafeCounter> FMetalCommandContextContainerAllocator;

// Engine\Source\Runtime\D3D12RHI\Private\D3D12CommandContext.cpp

classFD3D12CommandContextContainer: public IRHICommandContextContainer {

        //adapter.
        FD3D12Adapter* Adapter; //
        Command context.
        FD3D12CommandContext* CmdContext; //
        Contextual redirector.
        FD3D12CommandContextRedirector* CmdContextRedirector;
        FRHIGPUMask GPUMask;


        //Command queue list.
        TArray<FD3D12CommandListHandle> CommandLists;

public:
        void* operatornew(size_tSize); voidoperatordelete(
        void* RawMemory);

        FD3D12CommandContextContainer(FD3D12Adapter* InAdapter, FRHIGPUMask InGPUMask); virtual
        ~FD3D12CommandContextContainer() override

        virtual IRHICommandContext*GetContext() override; virtualvoid
        FinishContext() override;
        virtualvoid SubmitAndFreeContextContainer(int32 Index, int32 Num) override;
};

// Engine\Source\Runtime\VulkanRHI\Private\VulkanContext.h

structFVulkanCommandContextContainer: public IRHICommandContextContainer, public
VulkanRHI::FDeviceChild
{
        //The command queue context.
        FVulkanCommandListContext* CmdContext;

        FVulkanCommandContextContainer(FVulkanDevice* InDevice);

        virtual IRHICommandContext*GetContext() override final; virtualvoid
        FinishContext() override final;
        virtualvoid SubmitAndFreeContextContainer(int32 Index, int32 Num) override final;

        void* operatornew(size_tSize); voidoperatordelete(
        void* RawMemory);
};
```

IRHICommandContextContainer is equivalent to a container that stores one or a group of command contexts to support parallel submission of command queues. It is only implemented in modern

graphics APIs such as D3D12, Metal, and Vulkan. The complete inheritance UML diagram is as follows:



10.3.3 FDynamicRHI

FDynamicRHI is an interface implemented by dynamically bound RHI. The interface it defines is similar to CommandList and CommandContext. Some of them are as follows:

```
classRHI_API FDynamicRHI {

public:
	virtual ~FDynamicRHI() {}

	virtualvoid Init() =0; virtualvoid
	PostInit() {} virtualvoid Shutdown() =0;


	void InitPixelFormatInfo(constTArray<uint32>& PixelFormatBlockBytesIn);

	//----RHIinterface----

	//The following interface requirementsFlushType: Thread safe
	virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
		=0;
	virtual	FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) =0;
	virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) =0;
	virtual FBlendStateRHIRefRHICreateBlendState(constFBlendStateInitializerRHI& Initializer) =0;


	//The following interface requirementsFlushType: Wait RHI Thread
	virtual FVertexDeclarationRHIRefRHICreateVertexDeclaration(const
FVertexDeclarationElementList& Elements) =0;
	virtual FPixelShaderRHIRefRHICreatePixelShader(TArrayView<constuint8> Code,const FSHAHash& Hash) =0;

	virtual FVertexShaderRHIRefRHICreateVertexShader(TArrayView<constuint8> Code,const FSHAHash& Hash) =0;

	virtual FHullShaderRHIRefRHICreateHullShader(TArrayView<constuint8> Code,const FSHAHash& Hash) =0;

	virtual FDomainShaderRHIRefRHICreateDomainShader(TArrayView<constuint8> Code,const FSHAHash& Hash) =0;

	virtual FGeometryShaderRHIRefRHICreateGeometryShader(TArrayView<constuint8> Code, constFSHAHash&
Hash) =0;
```

```
        virtual FComputeShaderRHIRefRHICreateComputeShader(TArrayView<constuint8> Code, constFSHAHash&
Hash) =0;

         // FlushType: Must be Thread-Safe.
        virtualFRenderQueryPoolRHIRefRHICreateRenderQueryPool(ERenderQueryType QueryType, uint32 NumQueries
= UINT32_MAX);
        inlineFComputeFenceRHIRefRHICreateComputeFence(constFName& Name);

        virtual FGPUFenceRHIRefRHICreateGPUFence(constFName &Name); virtualvoid RHICreateTransition
        (FRHITransition* Transition, ERHIPipeline SrcPipelines, ERHIPipeline DstPipelines,
ERHICreateTransitionFlags CreateFlags, TArrayView<constFRHITransitionInfo>Infos);

        virtualvoid RHIReleaseTransition(FRHITransition* Transition);

        // FlushType: Thread safe.
        virtual FStagingBufferRHIRefRHICreateStagingBuffer();
        virtualvoid*RHILockStagingBuffer(FRHIStagingBuffer* StagingBuffer, FRHIGPUFence* Fence, uint32 Offset,
uint32 SizeRHI);
        virtualvoid RHIUnlockStagingBuffer(FRHIStagingBuffer* StagingBuffer);

        // FlushType: Thread safe, but varies depending on the RHI
        virtualFBoundShaderStateRHIRefRHICreateBoundShaderState(FRHIVertexDeclaration* VertexDeclaration,
FRHIVertexShader* VertexShader, FRHIHullShader* HullShader, FRHIDomainShader* DomainShader,
FRHIPixelShader* PixelShader, FRHIGeometryShader* GeometryShader) =0;

        // FlushType: Thread safe
        virtual FGraphicsPipelineStateRHIRefRHICreateGraphicsPipelineState(const
FGraphicsPipelineStateInitializer& Initializer);

        // FlushType: Thread safe, but varies depending on the RHI
        virtual FUniformBufferRHIRefRHICreateUniformBuffer(const void* Contents,const
FRHIUniformBufferLayout& Layout, EUniformBufferUsage Usage, EUniformBufferValidation Validation) =0;

        virtualvoid RHIUpdateUniformBuffer(FRHIUniformBuffer* UniformBufferRHI,const void* Contents) =0;


        // FlushType: Wait RHI Thread
        virtual FIndexBufferRHIRefRHICreateIndexBuffer(uint32 Stride, uint32 Size, uint32 InUsage, ERHIAccess
InResourceState, FRHIResourceCreateInfo& CreateInfo) =0;
        virtualvoid*RHILockIndexBuffer(FRHICommandListImmediate& RHICmdList, FRHIIndexBuffer* IndexBuffer,
uint32 Offset, uint32 Size, EResourceLockMode LockMode);
        virtualvoid RHIUnlockIndexBuffer(FRHICommandListImmediate& RHICmdList, FRHIIndexBuffer*
                        IndexBuffer);
        virtualvoid         RHITransferIndexBufferUnderlyingResource(FRHIIndexBuffer*
DestIndexBuffer,        FRHIIndexBuffer* SrcIndexBuffer);

        // FlushType: Wait RHI Thread
        virtual FVertexBufferRHIRefRHICreateVertexBuffer(uint32 Size, uint32 InUsage, ERHIAccess
InResourceState, FRHIResourceCreateInfo& CreateInfo) =0;
        // FlushType: Flush RHI Thread
        virtualvoid*RHILockVertexBuffer(FRHICommandListImmediate& RHICmdList, FRHIVertexBuffer*
VertexBuffer, uint32 Offset, uint32 SizeRHI, EResourceLockMode LockMode);

        virtualvoid RHIUnlockVertexBuffer(FRHICommandListImmediate& RHICmdList,
FRHIVertexBuffer* VertexBuffer);
        // FlushType: Flush Immediate (seems dangerous)
        virtualvoid RHICopyVertexBuffer(FRHIVertexBuffer* SourceBuffer, FRHIVertexBuffer* DestBuffer) =0;
```

```
virtualvoid RHITransferVertexBufferUnderlyingResource(FRHIVertexBuffer* DestVertexBuffer, FRHIVertexBuffer* SrcVertexBuffer);

// FlushType: Wait RHI Thread
virtualFStructuredBufferRHIRefRHICreateStructuredBuffer(uint32 Stride, uint32 Size, uint32 InUsage, ERHIAccess InResourceState, FRHIResourceCreateInfo& CreateInfo) =0;
// FlushType: Flush RHI Thread
virtualvoid*RHILockStructuredBuffer(FRHICommandListImmediate& RHICmdList, FRHIStructuredBuffer* StructuredBuffer, uint32 Offset, uint32 SizeRHI, EResourceLockMode LockMode);

virtualvoid RHIUnlockStructuredBuffer(FRHICommandListImmediate& RHICmdList, FRHIStructuredBuffer* StructuredBuffer);

// FlushType: Wait RHI Thread
virtual FUnorderedAccessViewRHIRefRHICreateUnorderedAccessView(FRHIStructuredBuffer* StructuredBuffer,bool bUseUAVCounter,boolbAppendBuffer) =0;
// FlushType: Wait RHI Thread
virtual FUnorderedAccessViewRHIRefRHICreateUnorderedAccessView(FRHITexture* Texture, uint32 MipLevel) =0;

// FlushType: Wait RHI Thread
virtual FUnorderedAccessViewRHIRefRHICreateUnorderedAccessView(FRHITexture* Texture, uint32 MipLevel, uint8 Format);

(.....)

//RHIFrame update must be called from the main thread.
FlushType: Thread safe virtualvoid RHITick(floatDeltaTime) =0;
//blockCPUuntilGPUExecution completes and becomes idle.FlushType: Flush Immediate (seems wrong) virtualvoid RHIBlockUntilGPUIdle() =0;
//Start the current frame and make sureGPUActively workingFlushType: Flush Immediate (copied from RHIBlockUntilGPUIdle)
virtualvoid RHISubmitCommandsAndFlushGPU() {};

//notifyRHIPrepare to pause it.
virtualvoid RHIBeginSuspendRendering() {};
//pauseRHIRendering and handing control over to the system's operations,
FlushType: Thread safe virtualvoid RHISuspendRendering() {}; //continueRHI Rendering,FlushType: Thread safe virtualvoid RHIResumeRendering() {}; //
FlushType: Flush Immediate

virtualbool RHIIsRenderingSuspended() {returnfalse; };

// FlushType: called from render thread when RHI thread is flushed //Only in FRHIResource::FlushPendingDeletesCalled every frame before the delay is removed.
virtualvoid RHIPerFrameRHIFlushComplete();

//Execute the command queue,FlushType: Wait RHI Thread
virtualvoid RHIExecuteCommandList(FRHICommandList* CmdList) =0;

// FlushType: Flush RHI Thread virtualvoid*
RHIGetNativeDevice() =0; // FlushType: Flush RHI Thread
virtualvoid*RHIGetNativeInstance() =0;


//Get the command context.FlushType: Thread safe
virtual IRHICommandContext*RHIGetDefaultContext() =0; //Get the computation context.FlushType: Thread safe
virtual IRHIComputeContext*RHIGetDefaultAsyncComputeContext();
```

```
    // FlushType: Thread safe
    virtual class IRHICommandContextContainer*RHIGetCommandContextContainer(int32 Index, int32 Num) =0;


    //API directly called by the rendering thread to optimizeRHICall.
    virtual FVertexBufferRHIRefCreateAndLockVertexBuffer_RenderThread(class
FRHICommandListImmediate& RHICmdList, uint32 Size, uint32 InUsage, ERHIAccess InResourceState,
FRHIResourceCreateInfo& CreateInfo,void*& OutDataBuffer);
    virtual FIndexBufferRHIRefCreateAndLockIndexBuffer_RenderThread(class FRHICommandListImmediate&
RHICmdList, uint32 Stride, uint32 Size, uint32 InUsage, ERHIAccess InResourceState, FRHIResourceCreateInfo&
CreateInfo,void*& OutDataBuffer);

    (.....)

    // Buffer Lock/Unlock
    virtualvoid*LockVertexBuffer_BottomOfPipe(class FRHICommandListImmediate& RHICmdList, ...);

    virtualvoid*LockIndexBuffer_BottomOfPipe(class FRHICommandListImmediate& RHICmdList,
. . . ) ;

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

Only some of the interfaces are shown above, some of which require to be called from the rendering thread, and some must be called from the game thread. Most interfaces require to refresh the specified type of command before being called, such as:

```
classRHI_API FDynamicRHI {

    // FlushType: Wait RHI Thread
    void RHIExecuteCommandList(FRHICommandList*          CmdList);

    // FlushType: Flush Immediate void
    RHIBlockUntilGPUIdle();

    // FlushType: Thread safe void RHITick(
    floatDeltaTime);
};
```

Then the code to call the above interface is as follows:

```
classRHI_API FRHICommandListImmediate: public FRHICommandList {

    void ExecuteCommandList(FRHICommandList* CmdList) {

        //waitRHIThreads.
        FScopedRHIThreadStallerStallRHIThread(*this); GDynamicRHI-
        >RHIExecuteCommandList(CmdList);
    }

    void BlockUntilGPUIdle() {

        //CallFDynamicRHI::RHIBlockUntilGPUIdleNeed to refreshRHI.
        ImmediateFlush(EImmediateFlushType::FlushRHIThread);
```

```
            GDynamicRHI->RHIBlockUntilGPUIdle();
        }

        void Tick(floatDeltaTime) {

            //becauseFDynamicRHI::RHITickyesThread Safethread-safe), so there is no need to callImmediateFlushOr waiting for something
Item.
            GDynamicRHI->RHITick(DeltaTime);
        }
};
```

Let's continue to look at the subclass definition of FDynamicRHI:

```
// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalDynamicRHI.h

classFMetalDynamicRHI: public FDynamicRHI {

public:
        FMetalDynamicRHI(ERHIFeatureLevel::Type              RequestedFeatureLevel);
        ~FMetalDynamicRHI();

        //Set up the necessary internal resources
        void SetupRecursiveResources();

        // FDynamicRHI interface. virtual
        void Init(); virtualvoid Shutdown() {}

        virtualconstTCHAR*GetName() override {returnTEXT("Metal"); }

        virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
                    final override;
        virtual      FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) final override;
        virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(...) final override;

        (......)

private:
        //An immediate mode context.
        FMetalRHIImmediateCommandContext //An                ImmediateContext;
        asynchronous computation context.
        FMetalRHICommandContext* AsyncComputeContext; //Vertex
        declaration cache.
        TMap<uint32, FVertexDeclarationRHIRef> VertexDeclarationCache;
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12RHIPrivate.h

classFD3D12DynamicRHI: public FDynamicRHI {

        staticFD3D12DynamicRHI* SingleD3DRHI;

public:
        staticD3D12RHI_API FD3D12DynamicRHI*GetD3DRHI() {returnSingleD3DRHI; }

        FD3D12DynamicRHI(constTArray<TSharedPtr<FD3D12Adapter>>& ChosenAdaptersIn,bool
        bInPixEventEnabled);
```

```
    virtual ~FD3D12DynamicRHI();

    // FDynamicRHI interface. virtualvoid Init()
    override; virtualvoid PostInit() override;
    virtualvoid Shutdown() override;

    virtualconstTCHAR*GetName() override {returnTEXT("D3D12"); }

    virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
                final override;
    virtual     FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) final override;
    virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) final override;

    (.....)

protected:
    //The selected adapter.
    TArray<TSharedPtr<FD3D12Adapter>> //       ChosenAdapters;
    AMD AGSTool library context. AGSContext*
    AmdAgsContext;

    // D3D12equipment.
    inlineFD3D12Device*GetRHIDevice(uint32 GPUIndex) {

        returnGetAdapter().GetDevice(GPUIndex);
    }

    (.....)
};

// Engine\Source\Runtime\EmptyRHI\Public\EmptyRHI.h

classFEmptyDynamicRHI: public FDynamicRHI, public IRHICommandContext {

    (.....)
};

// Engine\Source\Runtime\NullDrv\Public\NullRHI.h

classFNullDynamicRHI: public FDynamicRHI, public IRHICommandContextPSOFallback {

    (.....)
};


classOPENGLDRV_API FOpenGLDynamicRHIfinal: public FDynamicRHI, public
IRHICommandContextPSOFallback
{
public:
    FOpenGLDynamicRHI();
    ~FOpenGLDynamicRHI();

    //FDynamicRHI       interface.
    virtual     void    Init();
    virtual     void    PostInit();
```

```
        virtualvoid Shutdown();
        virtualconstTCHAR*GetName() override {returnTEXT("OpenGL"); }

        virtualFRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) final override;
        virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) final override;
        virtual FBlendStateRHIRefRHICreateBlendState(constFBlendStateInitializerRHI& Initializer) final override;


        (.....)

private:
        //counter.
        uint32 SceneFrameCounter;
        uint32 ResourceTableFrameCounter;

        //RHIDevice state, independent of the underlying layer usedOpenGL
        Context. FOpenGLRHIState                        PendingState;
        FOpenGLStreamedVertexBufferArray            DynamicVertexBuffers;
        FOpenGLStreamedIndexBufferArray                 DynamicIndexBuffers;
        FSamplerStateRHIRef                             PointSamplerState;

        //The viewport that was created.
        TArray<FOpenGLViewport*> Viewports;
        TRefCountPtr<FOpenGLViewport> bool                DrawingViewport;
                                                        bRevertToSharedContextAfterDrawingViewport;

        //History of bound shader states.
        TGlobalResource<TBoundShaderStateHistory<10000> > BoundShaderStateHistory;

        //Per-context state caching.
        FOpenGLContextState      InvalidContextState;
        FOpenGLContextState          SharedContextState;
        FOpenGLContextState          RenderingContextState;

        //Uniform buffers.
        TArray<FRHIUniformBuffer*> GlobalUniformBuffers; TMap<GLuint,
        TPair<GLenum, GLenum>> TextureMipLimits;

        //Data related to the underlying platform.
        FPlatformOpenGLDevice* PlatformDevice;

        //Query related.
        TArray<FOpenGLRenderQuery*> Queries; FCriticalSection
        QueriesListCriticalSection;

        //Configure and present data.
        FOpenGLGPUProfiler GPUProfilingData;
        FCriticalSection CustomPresentSection;
        TRefCountPtr<classFRHICustomPresent>            CustomPresent;

        (.....)
};

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

classFValidationRHI: public FDynamicRHI
```

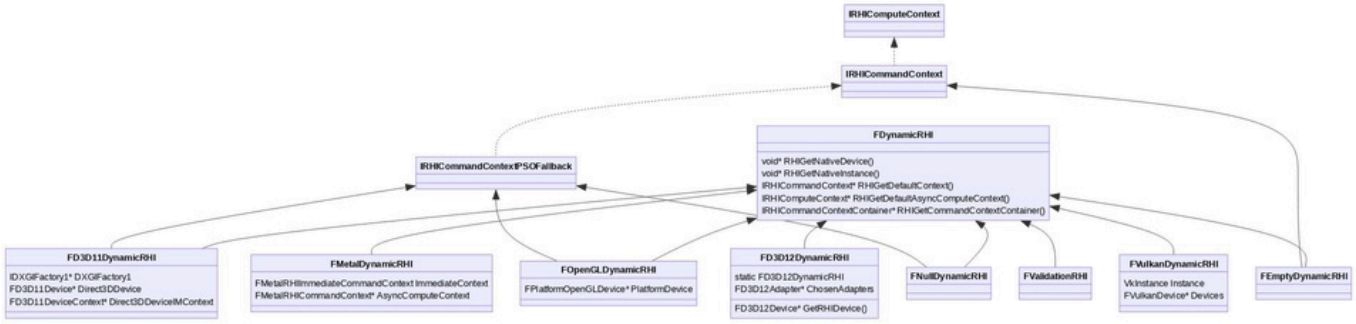
```
{
public:
	RHI_API	FValidationRHI(FDynamicRHI* virtual InRHI);
	RHI_API	~FValidationRHI();

	virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
		override final;
	virtual	FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) override final;
	virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) override final;

	(.....)

	//RHIExamples.
	FDynamicRHI*		RHI;
	//The context to which it belongs.
	TIndirectArray<IRHIComputeContext> //
	Depth-stencil state list.				OwnedContexts;
	TMap<FRHIDepthStencilState*, FDepthStencilStateInitializerRHI>
							DepthStencilStates;
};

// Engine\Source\Runtime\VulkanRHI\Public\VulkanDynamicRHI.h

classFVulkanDynamicRHI: public FDynamicRHI {

public:
	FVulkanDynamicRHI();
	~FVulkanDynamicRHI();

	// FDynamicRHI interface. virtualvoid Init() final
	override; virtualvoid PostInit() final override; virtual
	void Shutdown() final override;;

	virtualconstTCHAR*GetName() final override {returnTEXT("Vulkan"); }

	void InitInstance();

	virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
		final override;
	virtual	FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) final override;
	virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) final override;

	(.....)

protected:
	//Examples.
	VkInstance	Instance;
	TArray<const	ANSICHAR*>	InstanceExtensions;
	TArray<const	ANSICHAR*>	InstanceLayers;

	//equipment.
	TArray<FVulkanDevice*>		Devices;
	FVulkanDevice* Device;

	//Viewport.
```

```
    TArray<FVulkanViewport*> Viewports;
    TRefCountPtr<FVulkanViewport> DrawingViewport;

    //cache.
    IConsoleObject* SavePipelineCacheCmd = nullptr; IConsoleObject*
    RebuildPipelineCacheCmd = nullptr;

    //Critical area.
    FCriticalSection LockBufferCS;

    //Internal interface.
    void   CreateInstance();
    void   SelectAndInitDevice(); InitGPU
    void   (FVulkanDevice* InitDevice      Device);
    void   (FVulkanDevice*                 Device);

    (......)
};

// Engine\Source\Runtime\Windows\D3D11RHI\Private\D3D11RHIPrivate.h

classD3D11RHI_API FD3D11DynamicRHI: public FDynamicRHI, public
IRHICommandContextPSOFallback
{
public:
    FD3D11DynamicRHI(IDXGIFactory1* InDXGIFactory1,D3D_FEATURE_LEVEL          InFeatureLevel,int32
InChosenAdapter,constDXGI_ADAPTER_DESC& ChosenDescription);
    virtual ~FD3D11DynamicRHI();

    virtualvoid InitD3DDevice();

    // FDynamicRHI interface. virtualvoid Init()
    override; virtualvoid PostInit() override;
    virtualvoid Shutdown() override;

    virtualconstTCHAR*GetName() override {returnTEXT("D3D11"); }

    // HDR display output virtual
            void EnableHDR();
    virtual void ShutdownHDR();

    virtual FSamplerStateRHIRefRHICreateSamplerState(constFSamplerStateInitializerRHI& Initializer)
              final override;
    virtual FRasterizerStateRHIRefRHICreateRasterizerState(const
FRasterizerStateInitializerRHI& Initializer) final override;
    virtual FDepthStencilStateRHIRefRHICreateDepthStencilState(const
FDepthStencilStateInitializerRHI& Initializer) final override;

    (......)

    ID3D11Device*GetDevice()const {

        returnDirect3DDevice;
    }
    FD3D11DeviceContext*GetDeviceContext() {          const

        returnDirect3DDeviceIMContext;
    }
```

```
    IDXGIFactory1*GetFactory()const {

        returnDXGIFactory1;
    }

protected:
    // D3DFactory (interface).
    TRefCountPtr<IDXGIFactory1>           DXGIFactory1;
      // D3Dequipment.
    TRefCountPtr<FD3D11Device> // D3DThe    Direct3DDevice;
    immediate context of the device.
    TRefCountPtr<FD3D11DeviceContext>          Direct3DDeviceIMContext;

    //Thread lock.
    FD3D11LockTracker      LockTracker;
    FCriticalSection           LockTrackerCS;

    //Viewport.
    TArray<FD3D11Viewport*> Viewports;
    TRefCountPtr<FD3D11Viewport> DrawingViewport;

    // AMD AGSTool library context.
    AGSContext* AmdAgsContext;

    // RT, UAV,Shaders and other resources.
    TRefCountPtr<ID3D11RenderTargetView>
    CurrentRenderTargets[D3D11_SIMULTANEOUS_RENDER_TARGET_COUNT];
    TRefCountPtr<FD3D11UnorderedAccessView> CurrentUAVs[D3D11_PS_CS_UAV_REGISTER_COUNT];
    ID3D11UnorderedAccessView* UAVBound[D3D11_PS_CS_UAV_REGISTER_COUNT];

    TRefCountPtr<FD3D11TextureBase> CurrentDepthTexture; FD3D11BaseShaderResource*
    CurrentResourcesBoundAsSRVs[SF_NumStandardFrequencies]
    [D3D11_COMMONSHADER_INPUT_RESOURCE_SLOT_COUNT];
    FD3D11BaseShaderResource*
    CurrentResourcesBoundAsVBs[D3D11_IA_VERTEX_INPUT_RESOURCE_SLOT_COUNT];
    FD3D11BaseShaderResource* CurrentResourceBoundAsIB;
    int32 MaxBoundShaderResourcesIndex[SF_NumStandardFrequencies];
    FUniformBufferRHIRef BoundUniformBuffers[SF_NumStandardFrequencies]
    [MAX_UNIFORM_BUFFERS_PER_SHADER_STAGE];
    uint16 DirtyUniformBuffers[SF_NumStandardFrequencies];
    TArray<FRHIUniformBuffer*> GlobalUniformBuffers;


    //The constant buffer that was created.
    TArray<TRefCountPtr<FD3D11ConstantBuffer>          > VSConstantBuffers;
    TArray<TRefCountPtr<FD3D11ConstantBuffer>          > HSConstantBuffers;
    TArray<TRefCountPtr<FD3D11ConstantBuffer>          > DSConstantBuffers;
    TArray<TRefCountPtr<FD3D11ConstantBuffer>          > PSConstantBuffers;
    TArray<TRefCountPtr<FD3D11Constant Buffer>         > GSConstantBuffers;
                                                       > CSConstantBuffers;

    //History of bound shader states.
    TGlobalResource<TBoundShaderStateHistory<10000> > BoundShaderStateHistory;
    FComputeShaderRHIRef CurrentComputeShader;

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

Their core inheritance UML diagram is as follows:

Click on the image below to enlarge it:



It should be noted that in addition to inheriting FDynamicRHI, traditional graphics APIs (D3D11, OpenGL) also need to inherit IRHICommandContextPSOFallback, because they need to use the latter's interface to process PSO data and behavior to ensure consistent processing of PSO by traditional and modern APIs. For this reason, modern graphics APIs (D3D12, Vulkan, Metal) do not need to inherit any inheritance system type of IRHICommandContext, and can directly inherit FDynamicRHI to handle all data and operations of the RHI layer.

Since the DynamicRHI of modern graphics APIs (D3D12, Vulkan, Metal) does not inherit any inheritance system type of IRHICommandContext, how do they implement the FDynamicRHI::RHIGetDefaultContext interface? Let's take FD3D12DynamicRHI as an example:

```
IRHICommandContext*FD3D12DynamicRHI::RHIGetDefaultContext() {

	FD3D12Adapter& Adapter = GetAdapter();

	IRHICommandContext* DefaultCommandContext = nullptr; if
	(GNumExplicitGPUsForRendering >1)//manyGPU {

		DefaultCommandContext = static_cast<IRHICommandContext*>
		(&Adapter.GetDefaultContextRedirector());
	}
	else//oneGPU
	{
		FD3D12Device* Device = Adapter.GetDevice(0
		);DefaultCommandContext	= static_cast<IRHICommandContext*>(&Device-
> GetDefaultCommandContext()); }


	returnDefaultCommandContext;
}
```

Whether it is single GPU or multi-GPU, it is forced to be converted from FD3D12CommandContext, and FD3D12CommandContext is a sub-sub-subclass of IRHICommandContext, so static type conversion is no problem at all.

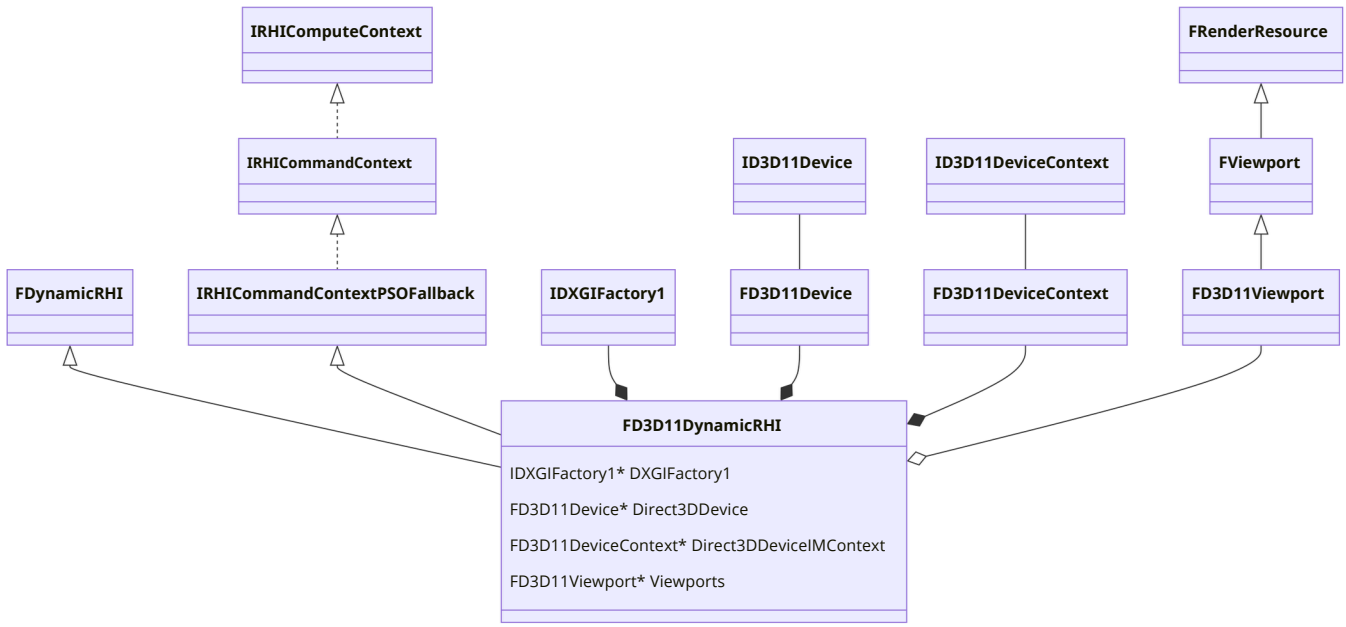
10.3.3.1 FD3D11DynamicRHI

FD3D11DynamicRHI contains or references several D3D11 platform-related core types, which are defined as follows:

```
// Engine\Source\Runtime\Windows\D3D11RHI\Private\D3D11RHIPrivate.h

classD3D11RHI_API FD3D11DynamicRHI: public FDynamicRHI, public
IRHICommandContextPSOFallback
{
	(......)

protected:
	// D3DFactory (interface).
	TRefCountPtr<IDXGIFactory1>		DXGIFactory1;
	  // D3Dequipment.
	TRefCountPtr<FD3D11Device> // D3DThe	Direct3DDevice;
	immediate context of the device.
	TRefCountPtr<FD3D11DeviceContext>		Direct3DDeviceIMContext;

	//Viewport.
	TArray<FD3D11Viewport*> Viewports;
	TRefCountPtr<FD3D11Viewport> DrawingViewport;

	// AMD AGSTool library context.
	AGSContext* AmdAgsContext;

	(......)
};

// Engine\Source\Runtime\Windows\D3D11RHI\Private\Windows\D3D11RHIBasePrivate.h

typedef	ID3D11DeviceContext FD3D11DeviceContext;
typedef	ID3D11Device FD3D11Device;

// Engine\Source\Runtime\Windows\D3D11RHI\Public\D3D11Viewport.h

classFD3D11Viewport: public FRHIViewport {

public:
	FD3D11Viewport(class FD3D11DynamicRHI* InD3DRHI) : D3DRHI(InD3DRHI),
PresentFailCount(0), ValidState (0),FrameSyncEvent(InD3DRHI);
	FD3D11Viewport(class FD3D11DynamicRHI* InD3DRHI, HWND InWindowHandle, uint32 InSizeX, uint32 InSizeY,
boolbInIsFullscreen, EPixelFormat InPreferredPixelFormat);
	~FD3D11Viewport();

	virtualvoid Resize(uint32 InSizeX, uint32 InSizeY,boolbInIsFullscreen, EPixelFormat PreferredPixelFormat);

	void	ConditionalResetSwapChain(boolbIgnoreFocus);
	void	CheckHDRMonitorStatus();

	//Presents the swap chain.
```

```
        bool Present(boolbLockToVsync);

        // Accessors.
        FIntPointGetSizeXY()const; FD3D11Texture2D*
        GetBackBuffer() EColorSpaceAndEOTF        const;
        GetPixelColorSpace()                                  const;

        void   WaitForFrameEventCompletion();
        void   IssueFrameEvent()

        IDXGISwapChain*GetSwapChain()const;
        virtualvoid*GetNativeSwapChain()constoverride; virtualvoid*
        GetNativeBackBufferTexture()constoverride; virtualvoid*
        GetNativeBackBufferRT()constoverrid;

        virtualvoid SetCustomPresent(FRHICustomPresent* InCustomPresent) override virtual
        FRHICustomPresent*GetCustomPresent()const;

        virtualvoid*GetNativeWindow(void**AddParam = nullptr)constoverride; staticFD3D11Texture2D*
        GetSwapChainSurface(FD3D11DynamicRHI* D3DRHI, EPixelFormat PixelFormat, uint32 SizeX, uint32 SizeY,
IDXGISwapChain* SwapChain);

protected:
        //dynamicRHI.
        FD3D11DynamicRHI*     D3DRHI;
        //Swap chain.
        TRefCountPtr<IDXGISwapChain> //Post-      SwapChain;
        render buffer.
        TRefCountPtr<FD3D11Texture2D>                BackBuffer;

        FD3D11EventQuery FrameSyncEvent;
        FCustomPresentRHIRef CustomPresent;

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

FD3D11DynamicRHI is drawn as a UML diagram as follows:



10.3.3.2 FOpenGLDynamicRHI

The core types related to FOpenGLDynamicRHI are defined as follows:

```
classOPENGLDRV_API FOpenGLDynamicRHIfinal: public FDynamicRHI, public
IRHICommandContextPSOFallback
{
	(.....)

private:
	//The viewport that was created.
	TArray<FOpenGLViewport*> //Data related Viewports;
	to the underlying platform.
	FPlatformOpenGLDevice* PlatformDevice;
};

// Engine\Source\Runtime\OpenGLDrv\Public\OpenGLResources.h

classFOpenGLViewport: public FRHIViewport {

public:
	FOpenGLViewport(class FOpenGLDynamicRHI* InOpenGLRHI,void* InWindowHandle,uint32 InSizeX,uint32
InSizeY,boolbInIsFullscreen,EPixelFormat PreferredPixelFormat);
	~FOpenGLViewport();

	void Resize(uint32 InSizeX,uint32 InSizeY,boolbInIsFullscreen);

	// Accessors.
	FIntPointGetSizeXY()const;
	FOpenGLTexture2D *GetBackBuffer() bool	const;
	IsFullscreen(void)const;

	void	WaitForFrameEventCompletion();
	void IssueFrameEvent();
	virtualvoid*GetNativeWindow(void** AddParam)constoverride;

	structFPlatformOpenGLContext*GetGLContext()const;
	FOpenGLDynamicRHI*GetOpenGLRHI()const;

	virtualvoid SetCustomPresent(FRHICustomPresent* InCustomPresent) override; FRHICustomPresent*
	GetCustomPresent()const;

private:
	FOpenGLDynamicRHI* OpenGLRHI;
	struct	FPlatformOpenGLContext*	OpenGLContext;
	uint32	SizeX; SizeY;
	uint32
	boolbIsFullscreen;
	EPixelFormat PixelFormat; bool
	bIsValid;
	TRefCountPtr<FOpenGLTexture2D>	BackBuffer;
	FOpenGLEventQuery FrameSyncEvent;
	FCustomPresentRHIRef CustomPresent;
};

// Engine\Source\Runtime\OpenGLDrv\Private\Android\AndroidOpenGL.cpp

//AndroidOpenGLequipment. struct
FPlatformOpenGLDevice
```

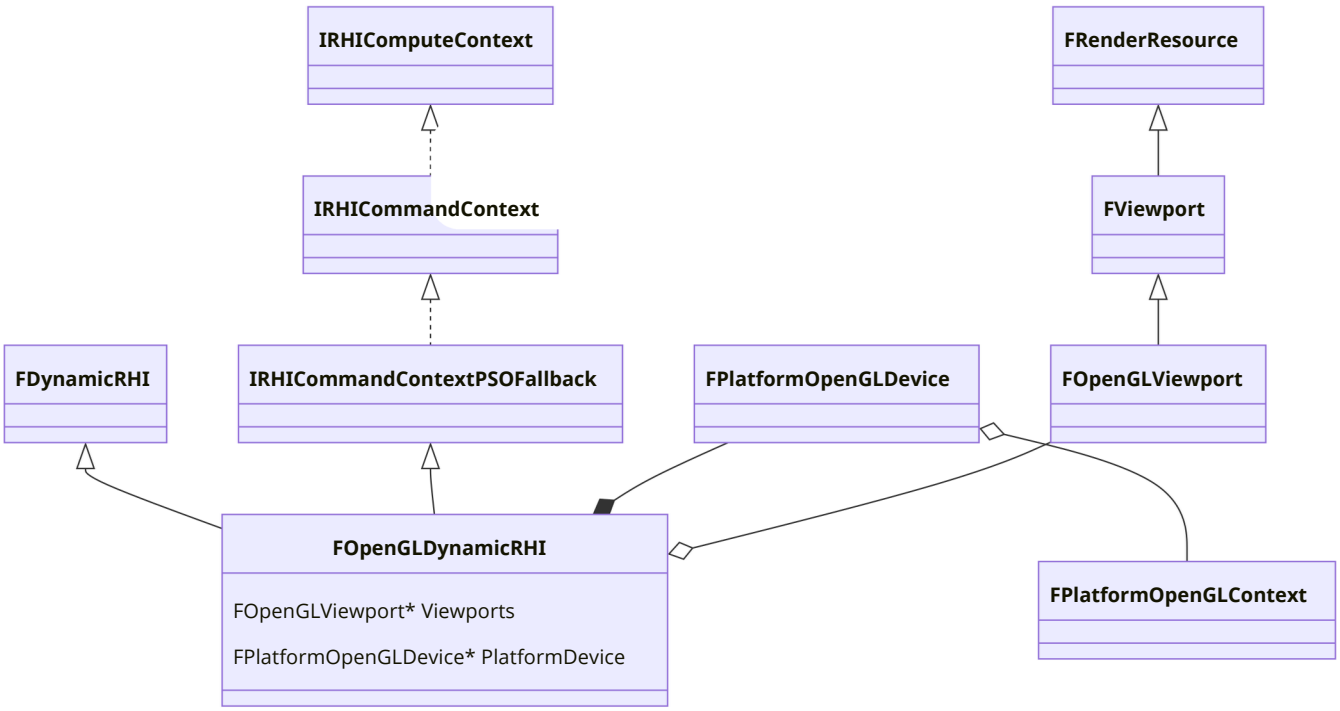
```
{
	boolTargetDirty;

	void	SetCurrentSharedContext();
	void	SetCurrentRenderingContext();
	void	SetupCurrentContext();
	void	SetCurrentNULLContext();

	FPlatformOpenGLDevice();
	~FPlatformOpenGLDevice();

	void	Init();
	void	LoadEXT();
	void	Terminate();
	void	ReInit();
};

// Engine\Source\Runtime\OpenGLDrv\Private\Windows\OpenGLWindows.cpp

// WindowsSystematicOpenGL
equipment. struct
FPlatformOpenGLDevice {
	FPlatformOpenGLContext		SharedContext;
	FPlatformOpenGLContext		RenderingContext;
	TArray<FPlatformOpenGLContext*> bool	ViewportContexts;
					TargetDirty;

	/** Guards against operating on viewport contexts from more than one thread at the time. */
same
	FCriticalSection*		ContextUsageGuard;
};

// Engine\Source\Runtime\OpenGLDrv\Private\Lumin\LuminOpenGL.cpp

// LuminSystematicOpenGLequipment.
structFPlatformOpenGLDevice {

	void	SetCurrentSharedContext();
	void	SetCurrentRenderingContext();
	void	SetCurrentNULLContext();

	FPlatformOpenGLDevice();
	~FPlatformOpenGLDevice();

	void	Init();
	void	LoadEXT();
	void	Terminate();
	void	ReInit();
};

// Engine\Source\Runtime\OpenGLDrv\Private\Linux\OpenGLLinux.cpp

// LinuxSystematicOpenGLequipment.
structFPlatformOpenGLDevice {

	FPlatformOpenGLContext		SharedContext;
	FPlatformOpenGLContext		RenderingContext;
	int32				NumUsedContexts;
```

```
    FCriticalSection*                   ContextUsageGuard;
};

// Engine\Source\Runtime\OpenGLDrv\Private\Lumin\LuminGL4.cpp

// LuminSystematicOpenGLequipment.
structFPlatformOpenGLDevice {

    FPlatformOpenGLContext          SharedContext;
    FPlatformOpenGLContext          RenderingContext;
    TArray<FPlatformOpenGLContext*> bool        ViewportContexts;
                                    TargetDirty;
    FCriticalSection*               ContextUsageGuard;
};
```

The above shows that the definition of OpenGL device objects in different operating systems. In fact, the OpenGL context also varies depending on the operating system. The following takes Windows as an example:

```
// Engine\Source\Runtime\OpenGLDrv\Private\Windows\OpenGLWindows.cpp

struct    FPlatformOpenGLContext
{
    //    Window handle
    HWNDWindowHandle;
    //    The device context.
    HDC DeviceContext;
    //    OpenGLContext.
    HGLRC   OpenGLContext;

    //Other actual.
    bool bReleaseWindowOnDestroy;
    int32    SyncInterval;
    GLuint        ViewportFramebuffer;
    GLuint        VertexArrayObject;          // one has to be generated and set for each context
(OpenGL 3.2 Core requirements)
    GLuint        BackBufferResource;
    GLenum        BackBufferTarget;
};
```

The UML diagram drawn by FOpenGLDynamicRHI is as follows:



10.3.3.3 FD3D12DynamicRHI

The core type definition of FD3D12DynamicRHI is as follows:

```
// Engine\Source\Runtime\D3D12RHI\Private\D3D12RHIPrivate.h

classFD3D12DynamicRHI: public FDynamicRHI {

	(......)

protected:
	//The selected adapter.
	TArray<TSharedPtr<FD3D12Adapter>>		ChosenAdapters;

	// D3D12equipment.
	inlineFD3D12Device*GetRHIDevice(uint32 GPUIndex) {

		returnGetAdapter().GetDevice(GPUIndex);
	}

	(......)
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12Adapter.h

classFD3D12Adapter: public FNoncopyable {

public:
	void	Initialize(FD3D12DynamicRHI* RHI);
	void	InitializeDevices();
	void	InitializeRayTracing();

	//Resource creation.
	HRESULTCreateCommittedResource(...)
	HRESULT CreateBuffer(...);
	template	<typename BufferType>
	BufferType*	CreateRHIBuffer(...);
```

```
        inline      FD3D12CommandContextRedirector&       GetDefaultContextRedirector();
        inline      FD3D12CommandContextRedirector&       GetDefaultAsyncComputeContextRedirector();
        FD3D12FastConstantAllocator&          GetTransientUniformBufferAllocator();

        void BlockUntilIdle();

        (......)

protected:
        virtualvoid CreateRootDevice(boolbWithDebug);

        FD3D12DynamicRHI* OwningRHI;

        // LDASetting has aID3D12Device
        TRefCountPtr<ID3D12Device>          RootDevice;
        TRefCountPtr<ID3D12Device1>           RootDevice1;

        TRefCountPtr<IDXGIAdapter> DxgiAdapter;

        TRefCountPtr<IDXGIFactory>          DxgiFactory;
        TRefCountPtr<IDXGIFactory2>           DxgiFactory2;

        //Each device represents a physicalGPUnode".
        FD3D12Device* Devices[MAX_NUM_GPUS];

        FD3D12CommandContextRedirector       DefaultContextRedirector;
        FD3D12CommandContextRedirector       DefaultAsyncComputeContextRedirector;

        TArray<FD3D12Viewport*> Viewports;
        TRefCountPtr<FD3D12Viewport> DrawingViewport;

        (......)
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12RHICommon.h

classFD3D12AdapterChild {

protected:
        FD3D12Adapter* ParentAdapter;

        (......)
};

classFD3D12DeviceChild {

protected:
        FD3D12Device* Parent;

        (......)
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12Device.h

classFD3D12Device: public FD3D12SingleNodeGPUObject, public FNoncopyable, public FD3D12AdapterChild

{
```

```
public:
    TArray<FD3D12CommandListHandle> PendingCommandLists;

    void Initialize();
    void    CreateCommandContexts();
    void InitPlatformSpecific();
    virtual     void Cleanup();
    bool GetQueryData(FD3D12RenderQuery& Query,boolbWait);

    ID3D12Device*GetDevice();

    void    BlockUntilIdle();
    bool    IsGPUIdle();

    FD3D12SamplerState*CreateSampler(constFSamplerStateInitializerRHI& Initializer);

    (......)

protected:
    //                  CommandListManager
    FD3D12CommandListManager*     CommandListManager;
    FD3D12CommandListManager*     CopyCommandListManager;
    FD3D12CommandListManager*     AsyncCommandListManager;
    FD3D12CommandAllocatorManager       TextureStreamingCommandAllocatorManager;

    //                  Allocator                   FD3D12OfflineDescriptorManager
    FD3D12ROTfVflAinlleoDcaetsocrr;iptorManager
    FD3D12OfflineDescriptorManager
    FD3D12DOSfVflAinlleoDcaetsocr;iptorManager
    FD3D12DefaultBufferAllocator DefaultBuffSeBrVAAllollocactaotor;r;
                                                    UAVAllocator;


    //FD3D12CommandContext
    TArray<FD3D12CommandContext*>       CommandContextArray;
    TArray<FD3D12CommandContext*>       FreeCommandContexts;
    TArray<FD3D12CommandContext*>       AsyncComputeContextArray;

    (......)
};

// Engine\Source\Runtime\D3D12RHI\Public\D3D12Viewport.h

classFD3D12Viewport: public FRHIViewport, public FD3D12AdapterChild {

public:
    void Init();
    void Resize(uint32 InSizeX, uint32 InSizeY,boolbInIsFullscreen, EPixelFormat PreferredPixelFormat);


    void    ConditionalResetSwapChain(boolbIgnoreFocus); Present(bool
    bool    bLockToVsync);

    void    WaitForFrameEventCompletion();
    bool    CurrentOutputSupportsHDR()const;

    (......)

private:
```

```
        HWND WindowHandle;

#if    D3D12_VIEWPORT_EXPOSES_SWAP_CHAIN
        TRefCountPtr<IDXGISwapChain1>          SwapChain1;
        TRefCountPtr<IDXGISwapChain4>          SwapChain4;
#endif

        TArray<TRefCountPtr<FD3D12Texture2D>> BackBuffers;
        TRefCountPtr<FD3D12Texture2D> DummyBackBuffer_RenderThread; uint32
        CurrentBackBufferIndex_RHIThread;
        FD3D12Texture2D* BackBuffer_RHIThread;
        TArray<TRefCountPtr<FD3D12Texture2D>> SDRBackBuffers;
        TRefCountPtr<FD3D12Texture2D> SDRDummyBackBuffer_RenderThread;
        FD3D12Texture2D* SDRBackBuffer_RHIThread;


        bool   CheckHDRSupport();
        void   EnableHDR();
        void   ShutdownHDR();

        (......)
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12CommandContext.h

classFD3D12CommandContextBase: public IRHICommandContext, public FD3D12AdapterChild {

public:
        FD3D12CommandContextBase(class FD3D12Adapter* InParent, FRHIGPUMask InGPUMask,bool
InIsDefaultContext,boolInIsAsyncComputeContext);

        void RHIBeginDrawingViewport(FRHIViewport* Viewport, FRHITexture* RenderTargetRHI) final override;

        void RHIEndDrawingViewport(FRHIViewport* Viewport,boolbPresent,boolbLockToVsync) final override;

        void RHIBeginFrame() final override; void
        RHIEndFrame() final override;

        (......)

protected:
        virtual FD3D12CommandContext*GetContext(uint32 InGPUIndex) =0;

        FRHIGPUMask GPUMask;

        (......)
};

classFD3D12CommandContext: public FD3D12CommandContextBase, public FD3D12DeviceChild {

public:
        FD3D12CommandContext(class FD3D12Device* InParent,boolInIsDefaultContext,bool
        InIsAsyncComputeContext);
        virtual ~FD3D12CommandContext();

        void   EndFrame();
        void   ConditionalObtainCommandAllocator();
        void   ReleaseCommandAllocator();
```

```
        FD3D12CommandListManager&     GetCommandListManager();
        void OpenCommandList();
        void CloseCommandList();

        FD3D12CommandListHandleFlushCommands(boolWaitForCompletion =false,
EFlushCommandsExtraAction ExtraAction = FCEA_None);
        void Finish(TArray<FD3D12CommandListHandle>& CommandLists);

        FD3D12FastConstantAllocator           ConstantsAllocator;
        FD3D12CommandListHandle     CommandListHandle;
        FD3D12CommandAllocator*     CommandAllocator;
        FD3D12CommandAllocatorManager       CommandAllocatorManager;

        FD3D12DynamicRHI& OwningRHI;

        // State Block.
        FD3D12RenderTargetView*       CurrentRenderTargets[D3D12_SIMULTANEOUS_RENDER_TARGET_COUNT];
        FD3D12DepthStencilView* FD3D1C2uTrreexntutDreBpathseS*te ncilTarget;
        CurrentDepthTexture; uint32
        NumSimultaneousRenderTargets;

        //     Uniform     Buffer.
        FD3D12UniformBuffer*      BoundUniformBuffers[SF_NumStandardFrequencies][MAX_CBS];
        FUniformBufferRHIRef      BoundUniformBufferRefs[SF_NumStandardFrequencies][MAX_CBS];
        uint16 DirtyUniformBuffers[SF_NumStandardFrequencies];

        //Constant buffer.
        FD3D12ConstantBuffer      VSConstantBuffer;
        FD3D12ConstantBuffer      HSConstantBuffer;
        FD3D12ConstantBuffer      DSConstantBuffer;
        FD3D12ConstantBuffer      PSConstantBuffer;
        FD3D12ConstantBuffer      GSConstantBuffer;
        FD3D12ConstantBuffer      CSConstantBuffer;

        template <classShaderType>void SetResourcesFromTables(constShaderType* RESTRICT); template <class
        ShaderType> uint32SetUAVPSResourcesFromTables(constShaderType* RESTRICT
                Shader);
        void CommitGraphicsResourceTables();
        void    CommitComputeResourceTables(FD3D12ComputeShader* ComputeShader);
        void ValidateExclusiveDepthStencilAccess(FExclusiveDepthStencil Src)const; void
        CommitRenderTargetsAndUAVs();

        virtualvoid SetDepthBounds(floatMinDepth,floatMaxDepth);
        virtualvoid SetShadingRate(EVRSShadingRate ShadingRate, EVRSRateCombiner Combiner);

        (.....)

protected:
        FD3D12CommandContext*GetContext(uint32 InGPUIndex) final override;
        TArray<FRHIUniformBuffer*> GlobalUniformBuffers;
};

classFD3D12CommandContextRedirectorfinal: public FD3D12CommandContextBase {

public:
        FD3D12CommandContextRedirector(class FD3D12Adapter* InParent,boolInIsDefaultContext, bool
        InIsAsyncComputeContext);
```

```
        virtualvoid RHISetComputeShader(FRHIComputeShader* ComputeShader) final override; virtualvoid
        RHISetComputePipelineState(FRHIComputePipelineState* ComputePipelineState) final override;

        virtualvoid RHIDispatchComputeShader(uint32 ThreadGroupCountX, uint32
ThreadGroupCountY, uint32 ThreadGroupCountZ) final override;

        (......)

private:
        FRHIGPUMask PhysicalGPUMask; FD3D12CommandContext*
        PhysicalContexts[MAX_NUM_GPUS];
};

// Engine\Source\Runtime\D3D12RHI\Private\D3D12CommandContext.cpp

classFD3D12CommandContextContainer: public IRHICommandContextContainer {

        FD3D12Adapter* Adapter;
        FD3D12CommandContext* CmdContext;
        FD3D12CommandContextRedirector*     CmdContextRedirector;
        FRHIGPUMask GPUMask;
        TArray<FD3D12CommandListHandle>     CommandLists;

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

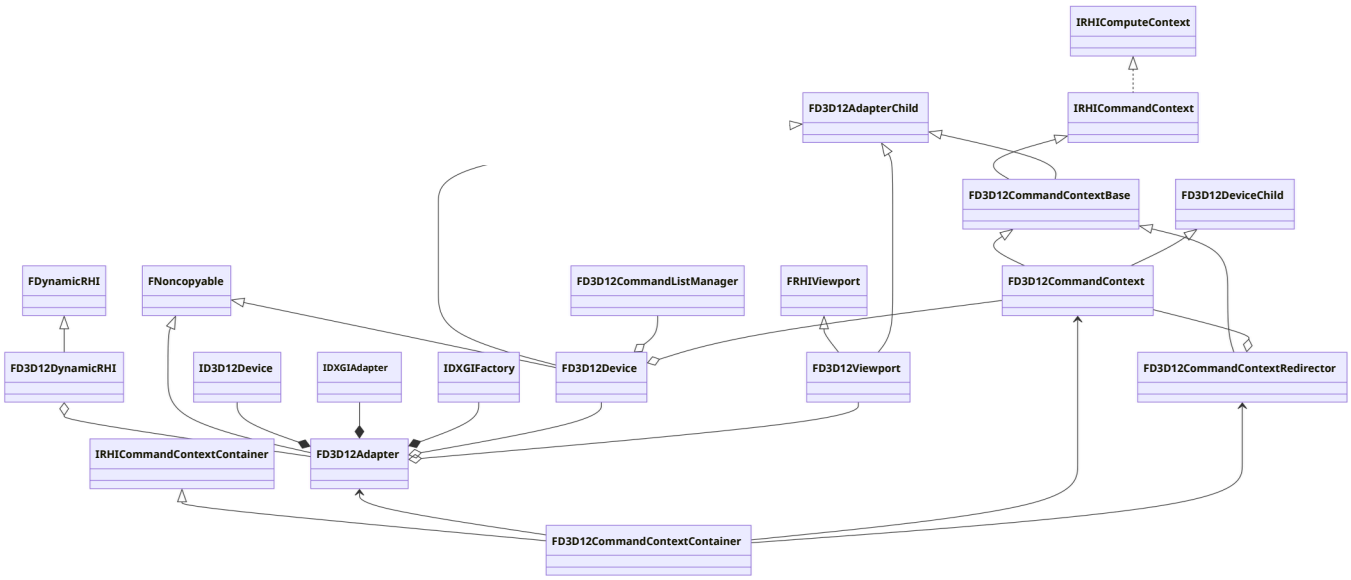
As can be seen above, D3D12 involves many core types, involving multi-level complex data structure chains, and its memory layout is as follows:

```
[Engine]--
            |
            |-[RHI]--
                        |
                        |-[Adapter]-- |      (LDA)
                                      |
                        |             |-   [Device]
                        |             |
                        |             |-   [Device]
                        |
                        |-[Adapter]--
                                    |
                                    |-   [Device]--
                                                  |
                                                  |-[CommandContext]
                                                  |
                                                  |-[CommandContext]---
                                                                       |
                                                                       |-[StateCache]
```

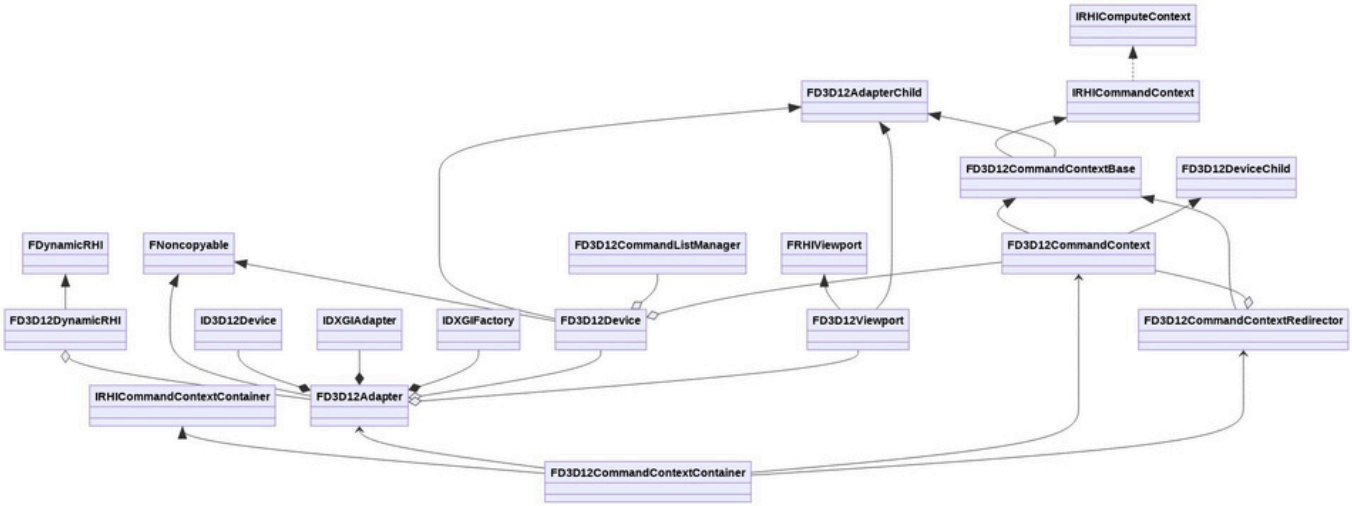
In this scheme, FD3D12Device represents 1 node, belonging to 1 physical adapter. This structure allows one RHI to control several different types of hardware settings, for example:

- Single GPU system (conventional case). Multi-GPU
- systems such as LDA (Crossfire/SLI).
- Asymmetric multi-GPU systems, such as separate and integrated GPU collaboration systems.

After abstracting the core class of D3D12 into a UML diagram, it is as follows:



If you can't see it clearly, you can click on the image below:



10.3.3.4 FVulkanDynamicRHI

The core classes involved in FVulkanDynamicRHI are as follows:

```
// Engine\Source\Runtime\VulkanRHI\Public\VulkanDynamicRHI.h

classFVulkanDynamicRHI: public FDynamicRHI {

public:
    // FDynamicRHI interface. virtualvoid Init() final
    override; virtualvoid PostInit() final override; virtual
    void Shutdown() final override;; void InitInstance();



    (.....)

protected:
    //Examples.
    VkInstance      Instance;
```

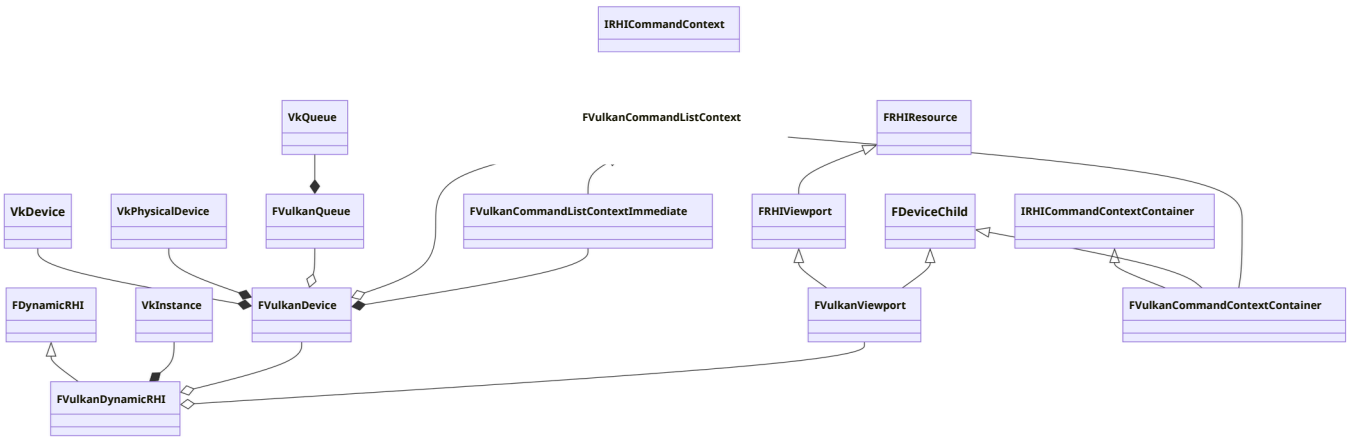
```
    //equipment.
    TArray<FVulkanDevice*>      Devices;
    FVulkanDevice* Device;

    //Viewport.
    TArray<FVulkanViewport*> Viewports;

    (......)
};

// Engine\Source\Runtime\VulkanRHI\Private\VulkanDevice.h

classFVulkanDevice
{
public:
    FVulkanDevice(FVulkanDynamicRHI* InRHI, VkPhysicalDevice Gpu);
    ~FVulkanDevice();

    bool  QueryGPU(int32    DeviceIndex);
    void  InitGPU(int32     DeviceIndex);
    void  CreateDevice();
    void  PrepareForDestroy();
    void  Destroy();

    void  WaitUntilIdle();
    void  PrepareForCPURead();
    void  SubmitCommandsAndFlushGPU();

    (......)

private:
    void  SubmitCommands(FVulkanCommandListContext* Context);

    // vkequipment.
    VkDevice Device;
    // vkPhysical equipment.
    VkPhysicalDevice
                        GPU;

    VkPhysicalDeviceProperties GpuProps;
    VkPhysicalDeviceFeatures PhysicalFeatures;

    //Manager.
    VulkanRHI::FDeviceMemoryManager DeviceMemoryManager;
    VulkanRHI::FMemoryManager MemoryManager;
    VulkanRHI::FDeferredDeletionQueue2 DeferredDeletionQueue;
    VulkanRHI::FStagingManager StagingManager;
    VulkanRHI::FFenceManager FenceManager; FVulkanDescriptorPoolsManager*
    DescriptorPoolsManager = nullptr;

    FVulkanDescriptorSetCache* DescriptorSetCache = nullptr;
    FVulkanShaderFactory ShaderFactory;

    //queue.
    FVulkanQueue*   GfxQueue;
    FVulkanQueue*   ComputeQueue;
    FVulkanQueue*   TransferQueue;
    FVulkanQueue*   PresentQueue;
```

```
        //GPUbrand.
        EGpuVendorId VendorId = EGpuVendorId::NotQueried;

        //The command queue context.
        FVulkanCommandListContextImmediate* ImmediateContext;
        FVulkanCommandListContext* ComputeContext;
        TArray<FVulkanCommandListContext*> CommandContexts;

        FVulkanDynamicRHI* RHI = nullptr;
        classFVulkanPipelineStateCacheManager*PipelineStateCache;

        (......)
};

// Engine\Source\Runtime\VulkanRHI\Private\VulkanQueue.h

classFVulkanQueue
{
public:
        FVulkanQueue(FVulkanDevice* InDevice, uint32 InFamilyIndex);
        ~FVulkanQueue();

        void Submit(FVulkanCmdBuffer* CmdBuffer, uint32 NumSignalSemaphores =0, VkSemaphore* SignalSemaphores
= nullptr);
        void Submit(FVulkanCmdBuffer* CmdBuffer, VkSemaphore SignalSemaphore);

        void GetLastSubmittedInfo(FVulkanCmdBuffer*& OutCmdBuffer, uint64& OutFenceCounter) const;


        (......)

private:
        // vkqueue
        VkQueue Queue;
        //Family Index.
        uint32 FamilyIndex;
        //The queue index.
        uint32 QueueIndex;
        FVulkanDevice* Device;


        // vkCommand buffer.
        FVulkanCmdBuffer*       LastSubmittedCmdBuffer;
        uint64     LastSubmittedCmdBufferFenceCounter;
        uint64 SubmitCounter;
        mutable    FCriticalSection CS;

        void UpdateLastSubmittedCommandBuffer(FVulkanCmdBuffer* CmdBuffer);
};

// Engine\Source\Runtime\VulkanRHI\Public\VulkanMemory.h

//Device child node.
classFDeviceChild
{
public:
        FDeviceChild(FVulkanDevice* InDevice = nullptr);

        (......)
```

```
  protected:
        FVulkanDevice*       Device;
};

// Engine\Source\Runtime\VulkanRHI\Private\VulkanContext.h

classFVulkanCommandListContext: public IRHICommandContext {

public:
        FVulkanCommandListContext(FVulkanDynamicRHI* InRHI, FVulkanDevice* InDevice, FVulkanQueue*
InQueue, FVulkanCommandListContext* InImmediate);
        virtual ~FVulkanCommandListContext();

        staticinlineFVulkanCommandListContext&GetVulkanContext(IRHICommandContext& CmdContext);


        inlinebool IsImmediate()const;

        virtualvoid RHISetStreamSource(uint32 StreamIndex, FRHIVertexBuffer* VertexBuffer, uint32 Offset) final
        override;
        virtualvoid RHISetViewport(floatMinX,floatMinY,floatMinZ,floatMaxX,float MaxY,floatMaxZ) final override;

        virtualvoid RHISetScissorRect(boolbEnable, uint32 MinX, uint32 MinY, uint32 MaxX, uint32 MaxY) final override;


        (.....)

        inlineFVulkanDevice*GetDevice()const;
        void PrepareParallelFromBase(constFVulkanCommandListContext& BaseContext);

protected:
        FVulkanDynamicRHI* RHI;
        FVulkanCommandListContext*        Immediate;
        FVulkanDevice*       Device;
        FVulkanQueue*     Queue;

        FVulkanUniformBufferUploader*          UniformBufferUploader;
        FVulkanCommandBufferManager*      CommandBufferManager;
        staticFVulkanLayoutManager           LayoutManager;

private:
        FVulkanGPUProfiler GpuProfiler; TArray<FRHIUniformBuffer*>
        GlobalUniformBuffers;

        (.....)
};

//An immediate-mode command queue context.
classFVulkanCommandListContextImmediate: public FVulkanCommandListContext {

public:
        FVulkanCommandListContextImmediate(FVulkanDynamicRHI* InRHI, FVulkanDevice* InDevice, FVulkanQueue*
        InQueue);
};

//Command context container.
structFVulkanCommandContextContainer: publicIRHICommandContextContainer, public
```

```
VulkanRHI::FDeviceChild {

    FVulkanCommandListContext*        CmdContext;

    FVulkanCommandContextContainer(FVulkanDevice* InDevice);

    virtual IRHICommandContext*GetContext() override final; virtualvoid
    FinishContext() override final;
    virtualvoid SubmitAndFreeContextContainer(int32 Index, int32 Num) override final;

    void* operatornew(size_tSize); voidoperatordelete(
    void* RawMemory);

    (......)
};

// Engine\Source\Runtime\VulkanRHI\Private\VulkanViewport.h

classFVulkanViewport: public FRHIViewport, public VulkanRHI::FDeviceChild {

public:
    FVulkanViewport(FVulkanDynamicRHI* InRHI, FVulkanDevice* InDevice,void* InWindowHandle, uint32
InSizeX,uint32 InSizeY,boolbInIsFullscreen, EPixelFormat InPreferredPixelFormat);

    ~FVulkanViewport();

    void   AdvanceBackBufferFrame(FRHICommandListImmediate& RHICmdList);
    void   WaitForFrameEventCompletion();

    virtualvoid SetCustomPresent(FRHICustomPresent* InCustomPresent) override final; virtual
    FRHICustomPresent*GetCustomPresent()constoverride final; virtualvoid Tick(floatDeltaTime) override final;

    bool Present(FVulkanCommandListContext* Context, FVulkanCmdBuffer* CmdBuffer, FVulkanQueue*
Queue, FVulkanQueue* PresentQueue,boolbLockToVsync);

    (......)

protected:
    TArray<VkImage, TInlineAllocator<NUM_BUFFERS*2>> BackBufferImages;
    TArray<VulkanRHI::FSemaphore*, TInlineAllocator<NUM_BUFFERS*2>>
    RenderingDoneSemaphores;
    TArray<FVulkanTextureView, TInlineAllocator<NUM_BUFFERS*2>>                TextureViews;
    TRefCountPtr<FVulkanBackBuffer>          RHIBackBuffer;
    TRefCountPtr<FVulkanTexture2D>              RenderingBackBuffer;

    /** narrow-scoped section that locks access to back buffer during its recreation*/ FCriticalSection
    RecreatingSwapchain;

    FVulkanDynamicRHI*      RHI;
    FVulkanSwapChain*     SwapChain;
    void* WindowHandle;
    VulkanRHI::FSemaphore*        AcquiredSemaphore;
    FCustomPresentRHIRef CustomPresent; FVulkanCmdBuffer*
    LastFrameCommandBuffer = nullptr;

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

If the core types of Vulkan RHI are drawn as UML diagrams, they are as follows:



10.3.3.5 FMetalDynamicRHI

The core type definition of FMetalDynamicRHI is as follows:

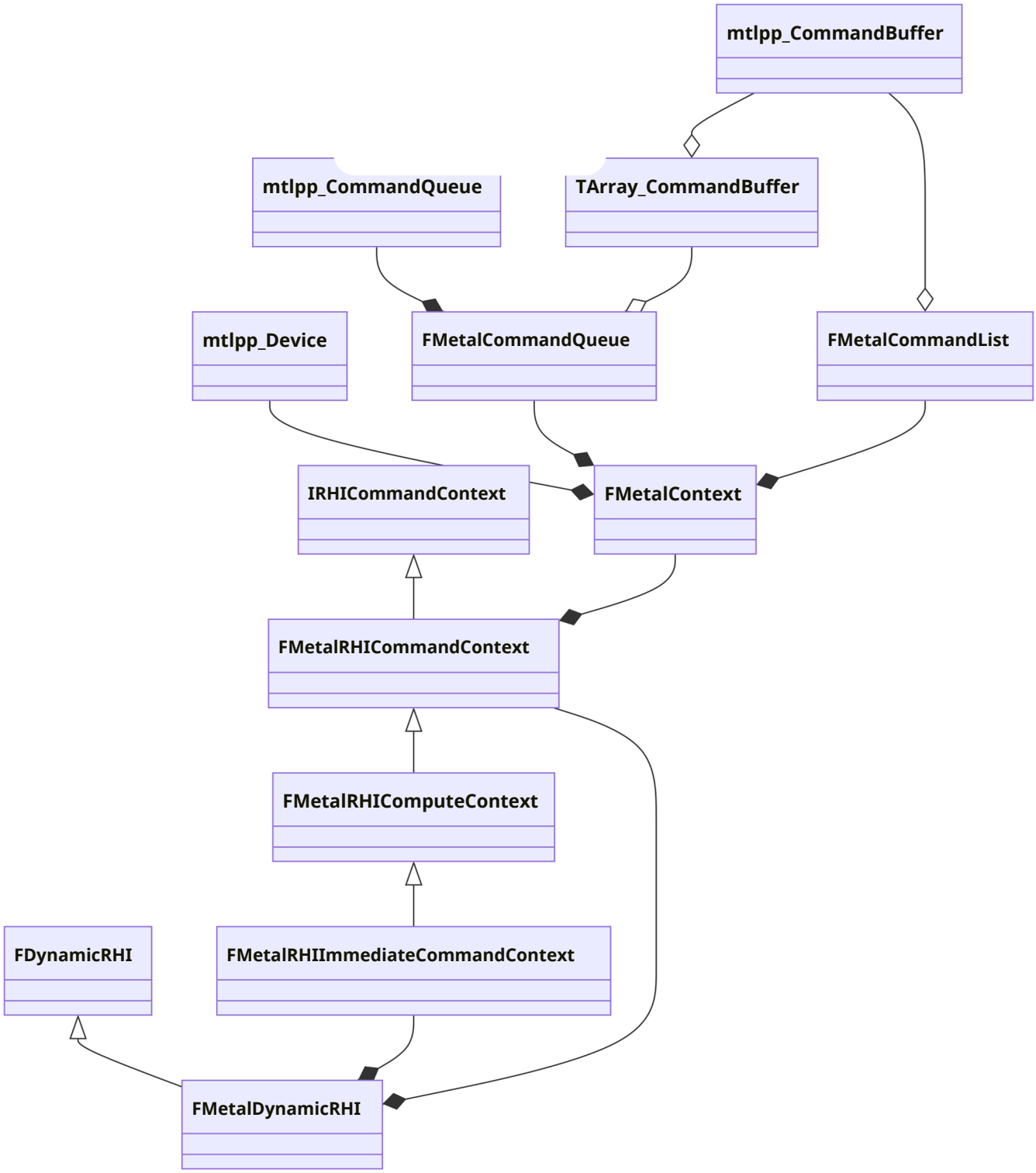
```
// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalDynamicRHI.h

classFMetalDynamicRHI: public FDynamicRHI {

public:
    // FDynamicRHI interface. virtual
    void Init(); virtualvoid Shutdown() {}



    (......)

private:
    //An immediate mode context.
    FMetalRHIImmediateCommandContext //An                ImmediateContext;
    asynchronous computation context.
    FMetalRHICommandContext* AsyncComputeContext;

    (......)
};

// Engine\Source\Runtime\Apple\MetalRHI\Public\MetalRHIContext.h

classFMetalRHICommandContext: public IRHICommandContext {

public:
    FMetalRHICommandContext(class FMetalProfiler* InProfiler, FMetalContext* WrapContext); virtual
    ~FMetalRHICommandContext();

    virtualvoid RHISetComputeShader(FRHIComputeShader* ComputeShader) override; virtualvoid
    RHISetComputePipelineState(FRHIComputePipelineState* ComputePipelineState) override;

    virtualvoid RHIDispatchComputeShader(uint32 ThreadGroupCountX, uint32
ThreadGroupCountY, uint32 ThreadGroupCountZ) final override;

    (......)

protected:
    //MetalContext.
```

```
    FMetalContext* Context;

    TSharedPtr<FMetalCommandBufferFence, ESPMode::ThreadSafe> CommandBufferFence; class
    FMetalProfiler*Profiler; FMetalBuffer PendingVertexBuffer;


    TArray<FRHIUniformBuffer*> GlobalUniformBuffers;

    (.....)
};

classFMetalRHIComputeContext: public FMetalRHICommandContext {

public:
    FMetalRHIComputeContext(class FMetalProfiler* InProfiler, FMetalContext* WrapContext); virtual
    ~FMetalRHIComputeContext();

    virtualvoid RHISetAsyncComputeBudget(EAsyncComputeBudget Budget) final override; virtualvoid
    RHISetComputeShader(FRHIComputeShader* ComputeShader) final override; virtualvoid
    RHISetComputePipelineState(FRHIComputePipelineState* ComputePipelineState) final override;

    virtualvoid RHISubmitCommandsHint() final override;
};

classFMetalRHIImmediateCommandContext: public FMetalRHICommandContext {

public:
    FMetalRHIImmediateCommandContext(class FMetalProfiler* InProfiler, FMetalContext* WrapContext);


    // FRHICommandContext API accessible only on the immediate device context virtualvoid
    RHIBeginDrawingViewport(FRHIViewport* Viewport, FRHITexture* RenderTargetRHI) final
    override;
    virtualvoid RHIEndDrawingViewport(FRHIViewport* Viewport,boolbPresent,bool bLockToVsync) final
    override;

    (.....)
};

// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalContext.h

//Context.
classFMetalContext
{
public:
    FMetalContext(mtlpp::Device InDevice, FMetalCommandQueue& Queue,bool const bIsImmediate);

    virtual ~FMetalContext();

    mtlpp::Device&GetDevice();

    bool PrepareToDraw(uint32 PrimitiveType, EMetalIndexType IndexType =
    EMetalIndexType_None);
    void SetRenderPassInfo(constFRHIRenderPassInfo& RenderTargetsInfo,bool const bRestart =false);


    void SubmitCommandsHint(uint32constbFlags = EMetalSubmitFlagsCreateCommandBuffer); void
    SubmitCommandBufferAndWait();
```

```
    void ResetRenderCommandEncoder();

    void DrawPrimitive(uint32 PrimitiveType, uint32 BaseVertexIndex, uint32 NumPrimitives, uint32 NumInstances);

    void DrawPrimitiveIndirect(uint32 PrimitiveType, FMetalVertexBuffer* VertexBuffer, uint32 ArgumentOffset);

    void DrawIndexedPrimitive(FMetalBufferconst& IndexBuffer, ...); void
    DrawIndexedIndirect(FMetalIndexBuffer* IndexBufferRHI, ...); void
    DrawIndexedPrimitiveIndirect(uint32 PrimitiveType, ...); void DrawPatches(uint32
    PrimitiveType, ...);

    (......)

protected:
    //MetalThe underlying device.
    mtlpp::Device Device;

    FMetalCommandQueue& CommandQueue;
    FMetalCommandList CommandList;

    FMetalStateCache      StateCache;
    FMetalRenderPass      RenderPass;

    dispatch_semaphore_tCommandBufferSemaphore;
    TSharedPtr<FMetalQueryBufferPool,            ESPMode::ThreadSafe>      QueryBuffer;
    TRefCountPtr<FMetalFence>        StartFence;
    TRefCountPtr<FMetalFence>        EndFence;

    int32 NumParallelContextsInPass;

    (......)
};

// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalCommandQueue.h

classFMetalCommandQueue {

public:
    FMetalCommandQueue(mtlpp::Device Device, uint32constMaxNumCommandBuffers =0);
    ~FMetalCommandQueue(void);

    mtlpp::CommandBufferCreateCommandBuffer(void);
    void CommitCommandBuffer(mtlpp::CommandBuffer& CommandBuffer);
    void SubmitCommandBuffers(TArray<mtlpp::CommandBuffer> BufferList, uint32 Index, uint32 Count);

    FMetalFence*CreateFence(ns::Stringconst& Label)const;
    void GetCommittedCommandBufferFences(TArray<mtlpp::CommandBufferFence>& Fences);

    mtlpp::Device&GetDevice(void);

    staticmtlpp::ResourceOptionsGetCompatibleResourceOptions(mtlpp::ResourceOptions Options);

    staticinlinebool SupportsFeature(EMetalFeatures InFeature); staticinlinebool
    SupportsSeparateMSAAAndResolveTarget();

    (......)

private:
```

```
        //equipment.
        mtlpp::Device       Device;
        //Command queue.
        mtlpp::CommandQueue
                                    CommandQueue;
        //Command buffer list. (Note that it is an array of arrays)
        TArray<TArray<mtlpp::CommandBuffer>>            CommandBuffers;

        TLockFreePointerListLIFO<mtlpp::CommandBufferFence> CommandBufferFences; uint64
        ParallelCommandLists;
};

// Engine\Source\Runtime\Apple\MetalRHI\Private\MetalCommandList.h

classFMetalCommandList {

public:
        FMetalCommandList(FMetalCommandQueue& InCommandQueue,bool constbInImmediate);
        ~FMetalCommandList(void);

        void Commit(mtlpp::CommandBuffer& Buffer, TArray<ns::Object<mtlpp::CommandBufferHandler>>
CompletionHandlers,bool constbWait,bool constbIsLastCommandBuffer);

        void Submit(uint32 Index, uint32 Count);

        bool    IsImmediate(void)         const;
        bool IsParallel(void)             const;
        void SetParallelIndex(uint32 Index, uint32 Num); uint32
                    GetParallelIndex(void)const;
        uint32 GetParallelNum(void)const;

        (......)

private:
        //Belong toFMetalCommandQueue.
        FMetalCommandQueue& CommandQueue; //List
        of submitted command buffers.
        TArray<mtlpp::CommandBuffer> SubmittedBuffers;
};
```

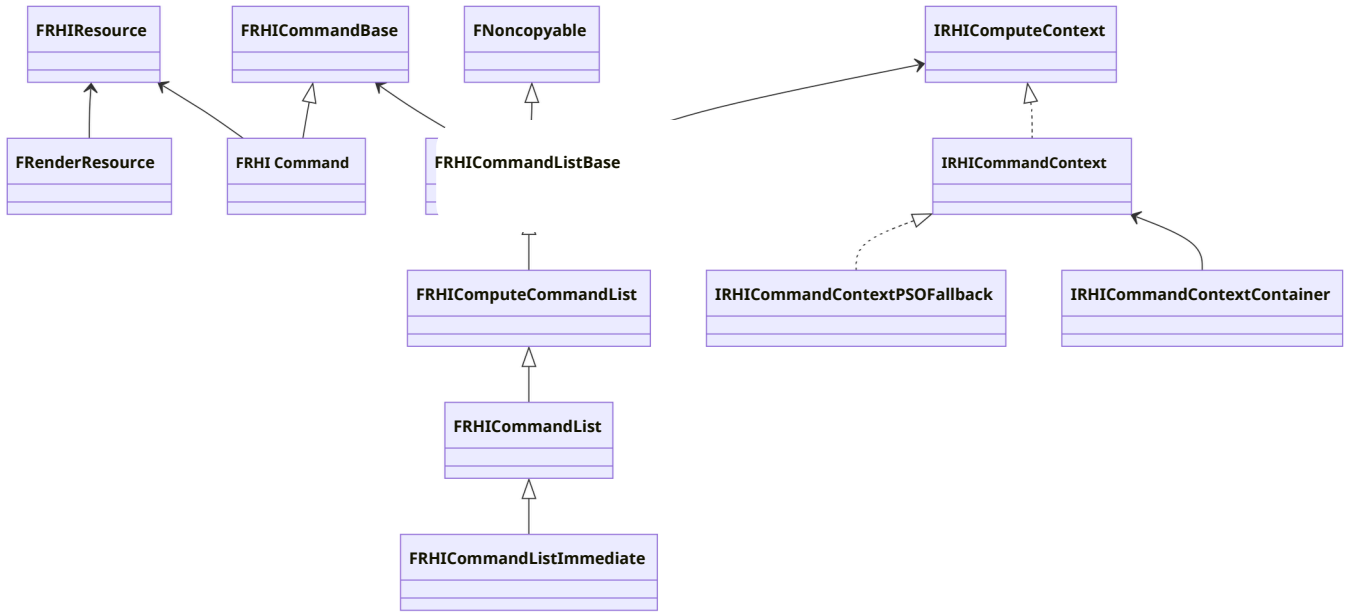
Compared with other modern graphics APIs, the concepts and interfaces of FMetalDynamicRHI are much simpler. Its UML diagram is as follows:



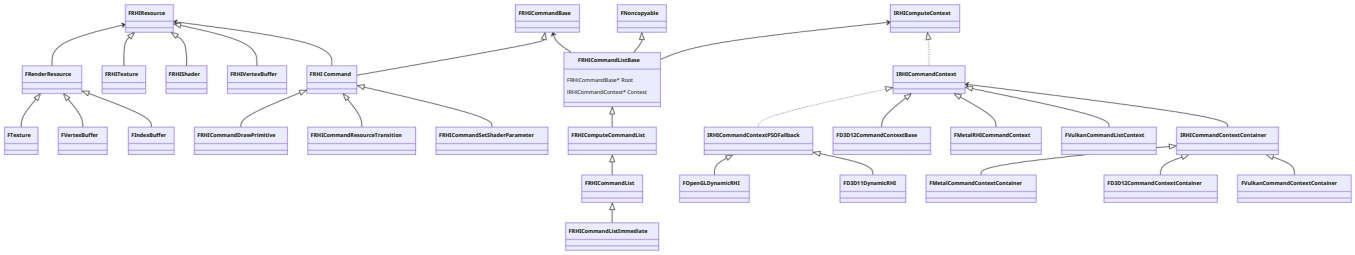
10.3.4 Overview of the RHI System

Chapters 10.2 and 10.3 elaborate on the basic concepts and inheritance system of the RHI system, including rendering layer resources, RHI layer resources, commands, contexts, and dynamic RHI. They also elaborate on the specific implementations of each mainstream graphics API and the relationship between the RHI abstraction layer.

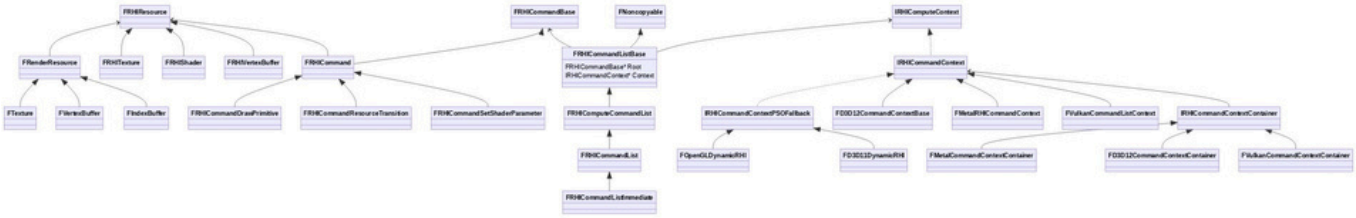
If we ignore the specific implementation details of the graphics API and the numerous RHI specific subclasses, and summarize the top-level concepts of RHI Context/CommandList/Command/Resource into a UML relationship diagram, it looks like this:



The following figure is a UML that refines the subclass based on the above:



If you can't see it clearly, click on the image below to enlarge it:



10.4 RHI Mechanism

This chapter will describe the operating mechanism and principles of RHI system design.

10.4.1 RHI Command Execution

10.4.1.1 FRHICommandListExecutor

FRHICommandListExecutor is responsible for translating (or directly calling) the RHI intermediate commands of the Renderer layer to the graphics API of the target platform. It plays a vital role in the RHI system and is defined as follows:

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

classRHI_API FRHICommandListExecutor
{
public:
	enum
	{
		DefaultBypass = PLATFORM_RHITHREAD_DEFAULT_BYPASS
	};
	FRHICommandListExecutor()
		: bLatchedBypass(!!DefaultBypass) ,
		bLatchedUseParallelAlgorithms(false)
	{
	}

	//Static interface, get the immediate command list.
	staticinlineFRHICommandListImmediate&GetImmediateCommandList(); //Static interface, get the list of
	immediate asynchronous computing commands.
	staticinlineFRHIAsyncComputeCommandListImmediate&
GetImmediateAsyncComputeCommandList();

	//Execute a list of commands.
	void	ExecuteList(FRHICommandListBase& CmdList); ExecuteList
	void	(FRHICommandListImmediate& CmdList); LatchBypass();
	void

	//waitRHIThread fence.
	static void WaitOnRHIThreadFence(FGraphEventRef& Fence);

	//Whether to bypass command generation mode, if so, call the graphics of the target
	platform directlyAPI. FORCEINLINE_DEBUGGABLEbool Bypass() {

#if	CAN_TOGGLE_COMMAND_LIST_BYPASS
		return	bLatchedBypass;
#else
		return	!!DefaultBypass;
#endif
	}
	//Whether to use parallel algorithms.
	FORCEINLINE_DEBUGGABLE	bool UseParallelAlgorithms()
	{
#if	CAN_TOGGLE_COMMAND_LIST_BYPASS
		return bLatchedUseParallelAlgorithms;
#else
		return	FApp::ShouldUseThreadingForPerformance() && !Bypass() &&
(GSupportsParallelRenderingTasksWithSeparateRHIThread		|| !IsRunningRHIInSeparateThread());
#endif
	}
	static	void	CheckNoOutstandingCmdLists();
	static	bool	IsRHIThreadActive();
	static	bool	IsRHIThreadCompletelyFlushed();

private:
	//Internal execution.
	void ExecuteInner(FRHICommandListBase& //Internal execution,	CmdList);
	actually performs the translation.
	static void ExecuteInner_DoExecute(FRHICommandListBase& CmdList);
```

```
    bool   bLatchedBypass;
    bool   bLatchedUseParallelAlgorithms;

    //Synchronize variables.
    FThreadSafeCounter      UIDCounter;
    FThreadSafeCounter      OutstandingCmdListCount;

    //The immediate mode command queue.
    FRHICommandListImmediate CommandListImmediate; //An immediate-mode
    asynchronous computation command queue.
    FRHIAsyncComputeCommandListImmediate AsyncComputeCmdListImmediate;
};
```

The following is the implementation code of some important interfaces of FRHICommandListExecutor:

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

//DetectionRHIWhether the thread is active.
bool FRHICommandListExecutor::IsRHIThreadActive() {

    //Whether to submit asynchronously.
    boolbAsyncSubmit = CVarRHICmdAsyncRHIThreadDispatch.GetValueOnRenderThread() >0; //
        1.First check whether there are any unfinished subcommand list submission
    if    tasks. (bAsyncSubmit)
    {
        if(RenderThreadSublistDispatchTask.GetReference() &&
RenderThreadSublistDispatchTask->IsComplete())
        {
            RenderThreadSublistDispatchTask = nullptr;
        }
        if(RenderThreadSublistDispatchTask.GetReference()) {

            returntrue;// it might become active at any time
        }
        // otherwise we can safely look at RHIThreadTask
    }

    // 2.Check again if there are any unfinishedRHIThread tasks.
    if(RHIThreadTask.GetReference() && RHIThreadTask->IsComplete()) {

        RHIThreadTask = nullptr;
        PrevRHIThreadTask = nullptr;
    }
    return!!RHIThreadTask.GetReference();
}

//DetectionRHIWhether the thread has completely flushed the data.
bool FRHICommandListExecutor::IsRHIThreadCompletelyFlushed() {

    if(IsRHIThreadActive() || GetImmediateCommandList().HasCommands()) {

        returnfalse;
    }
    if(RenderThreadSublistDispatchTask.GetReference() && RenderThreadSublistDispatchTask-
```

```
> IsComplete())
	{
#if	NEEDS_DEBUG_INFO_ON_PRESENT_HANG
		bRenderThreadSublistDispatchTaskClearedOnRT		=	IsInActualRenderingThread();
		bRenderThreadSublistDispatchTaskClearedOnGT		=	IsInGameThread();
#endif
		RenderThreadSublistDispatchTask = nullptr;
	}
	return!RenderThreadSublistDispatchTask;
}

void FRHICommandListExecutor::ExecuteList(FRHICommandListImmediate& CmdList) {

	{
		SCOPE_CYCLE_COUNTER(STAT_ImmedCmdListExecuteTime);
		ExecuteInner(CmdList);
	}
}

void FRHICommandListExecutor::ExecuteList(FRHICommandListBase& CmdList) {

	//Flush existing commands before performing command queue conversion.
	if(IsInRenderingThread() && !GetImmediateCommandList().IsExecuting()) {


GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
	}

	//Internal execution.
	ExecuteInner(CmdList);
}

void FRHICommandListExecutor::ExecuteInner(FRHICommandListBase& CmdList) {

	//Whether it is in the rendering thread.
	boolbIsInRenderingThread = IsInRenderingThread(); //Whether it is
	in the game thread.
	boolbIsInGameThread = IsInGameThread();

	//	Enabled dedicatedRHIThreads.
	if	(IsRunningRHIInSeparateThread())
	{
		boolbAsyncSubmit =false;
		ENamedThreads::Type RenderThread_Local = ENamedThreads::GetRenderThread_Local(); if
		(bIsInRenderingThread) {

			if(!bIsInGameThread &&
!FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
			{
				//Get rid of everything that needs to be passed on.
				FTaskGraphInterface::Get().ProcessThreadUntilIdle(RenderThread_Local);
			}
			//Check if the subcommand list task is completed.

			bAsyncSubmit = CVarRHICmdAsyncRHIThreadDispatch.GetValueOnRenderThread() >0; if
			(RenderThreadSublistDispatchTask.GetReference() &&
RenderThreadSublistDispatchTask->IsComplete())
			{
				RenderThreadSublistDispatchTask = nullptr;
```

```
                if(bAsyncSubmit && RHIThreadTask.GetReference() && RHIThreadTask-
> IsComplete())
                {
                        RHIThreadTask = nullptr;
                        PrevRHIThreadTask = nullptr;
                }
        }
        //    DetectionRHIWhether the thread task is completed.
        if(!bAsyncSubmit && RHIThreadTask.GetReference() && RHIThreadTask-
> IsComplete())
        {
                RHIThreadTask = nullptr;
                PrevRHIThreadTask = nullptr;
        }
    }

    if(CVarRHICmdUseThread.GetValueOnRenderThread() >0&& bIsInRenderingThread && !bIsInGameThread)

    {
        //Swap the preamble andRTA list of thread
        tasks. FRHICommandList*     SwapCmdList;
        FGraphEventArray            Prereq;
        Exchange(Prereq,            CmdList.RTTasks);
        {
                QUICK_SCOPE_CYCLE_COUNTER(STAT_FRHICommandListExecutor_SwapCmdLists);
                SwapCmdList = new FRHICommandList(CmdList.GetGPUMask());

                static_assert(sizeof(FRHICommandList) ==sizeof(FRHICommandListImmediate),
"We are memswapping FRHICommandList and FRHICommandListImmediate; they need to be swappable.");

                SwapCmdList->ExchangeCmdList(CmdList);
                CmdList.CopyContext(*SwapCmdList); CmdList.GPUMask =
                SwapCmdList->GPUMask; CmdList.InitialGPUMask =
                SwapCmdList->GPUMask; CmdList.PSOContext =
                SwapCmdList->PSOContext; CmdList.Data.bInsideRenderPass
                                                        =SwapCmdList->Data.bInsideRenderPass;
                CmdList.Data.bInsideComputePass          = SwapCmdList->Data.bInsideComputePass;
        }

        //Submit the task.
        QUICK_SCOPE_CYCLE_COUNTER(STAT_FRHICommandListExecutor_SubmitTasks);

        //createFDispatchRHIThreadTask,and willAll Outstanding Tasksand
RenderThreadSublistDispatchTaskas its predecessor task.
        if(AllOutstandingTasks.Num() ||
RenderThreadSublistDispatchTask.GetReference())
        {
                Prereq.Append(AllOutstandingTasks);
                AllOutstandingTasks.Reset();
                if(RenderThreadSublistDispatchTask.GetReference()) {

                        Prereq.Add(RenderThreadSublistDispatchTask);
                }
                RenderThreadSublistDispatchTask =
TGraphTask<FDispatchRHIThreadTask>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(SwapCmdList, bAsyncSubmit);

        }
```

```
                //createFExecuteRHIThreadTask,and willRHIThreadTaskas its predecessor task.
                else
                {
                    if(RHIThreadTask.GetReference()) {

                        Prereq.Add(RHIThreadTask);
                    }
                    PrevRHIThreadTask = RHIThreadTask;
                    RHIThreadTask = TGraphTask<FExecuteRHIThreadTask>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(SwapCmdList);
                }

                if(CVarRHICmdForceRHIFlush.GetValueOnRenderThread() >0) {

                    //Check if the rendering thread is deadlocked.
                    if
(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
                    {
                        // this is a deadlock. RT tasks must be done by now or they won't be
done. We could add a third queue...
                        UE_LOG(LogRHI, Fatal, TEXT("Deadlock in
FRHICommandListExecutor::ExecuteInner 2."));
                    }

                    //  DetectionRenderThreadSublistDispatchTaskIs it completed?
                    if  (RenderThreadSublistDispatchTask.GetReference())
                    {

FTaskGraphInterface::Get().WaitUntilTaskCompletes(RenderThreadSublistDispatchTask, RenderThread_Local);

                        RenderThreadSublistDispatchTask = nullptr;
                    }

                    //waitRHIThreadTaskFinish.
                    while(RHIThreadTask.GetReference()) {

                        FTaskGraphInterface::Get().WaitUntilTaskCompletes(RHIThreadTask,
RenderThread_Local);
                        if(RHIThreadTask.GetReference() && RHIThreadTask->IsComplete()) {

                            RHIThreadTask = nullptr;
                            PrevRHIThreadTask = nullptr;
                        }
                    }
                }

                return;
            }

            //  implementRTTasks/RenderThreadSublistDispatchTask/RHIThreadTaskAnd other
            if  tasks. (bIsInRenderingThread)
            {
                if(CmdList.RTTasks.Num()) {

                    if
(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
                    {
                        UE_LOG(LogRHI, Fatal, TEXT("Deadlock in
```

```
FRHICommandListExecutor::ExecuteInner (RTTasks)."));
                        }
                        FTaskGraphInterface::Get().WaitUntilTasksComplete(CmdList.RTTasks,
RenderThread_Local);
                        CmdList.RTTasks.Reset();

                    }
                    if    (RenderThreadSublistDispatchTask.GetReference())
                    {
                        if
(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
                        {
                            // this is a deadlock. RT tasks must be done by now or they won't be
done. We could add a third queue...
                            UE_LOG(LogRHI, Fatal, TEXT("Deadlock in
FRHICommandListExecutor::ExecuteInner (RenderThreadSublistDispatchTask)."));
                        }

FTaskGraphInterface::Get().WaitUntilTaskCompletes(RenderThreadSublistDispatchTask, RenderThread_Local);

#ifNEEDS_DEBUG_INFO_ON_PRESENT_HANG
                        bRenderThreadSublistDispatchTaskClearedOnRT             = IsInActualRenderingThread();
                        bRenderThreadSublistDispatchTaskClearedOnGT             = bIsInGameThread;
#endif
                        RenderThreadSublistDispatchTask = nullptr;
                    }
                    while(RHIThreadTask.GetReference()) {

                        if
(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
                        {
                            // this is a deadlock. RT tasks must be done by now or they won't be
done. We could add a third queue...
                            UE_LOG(LogRHI, Fatal, TEXT("Deadlock in
FRHICommandListExecutor::ExecuteInner (RHIThreadTask)."));
                        }
                        FTaskGraphInterface::Get().WaitUntilTaskCompletes(RHIThreadTask,
RenderThread_Local);
                        if(RHIThreadTask.GetReference() && RHIThreadTask->IsComplete()) {

                            RHIThreadTask = nullptr;
                            PrevRHIThreadTask = nullptr;
                        }
                    }
                }
            }
            //NoRHIDedicated thread.
            else
            {
                if(bIsInRenderingThread && CmdList.RTTasks.Num()) {

                    ENamedThreads::Type RenderThread_Local =
ENamedThreads::GetRenderThread_Local();
                    if(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local)) {

                        // this is a deadlock. RT tasks must be done by now or they won't be done.
We could add a third queue...
                        UE_LOG(LogRHI, Fatal, TEXT("Deadlock in
```

```
FRHICommandListExecutor::ExecuteInner (RTTasks)."));
                }
                FTaskGraphInterface::Get().WaitUntilTasksComplete(CmdList.RTTasks,
RenderThread_Local);
                CmdList.RTTasks.Reset();
            }
        }

        //Internal execution command.
        ExecuteInner_DoExecute(CmdList);
}

void FRHICommandListExecutor::ExecuteInner_DoExecute(FRHICommandListBase& CmdList) {

        FScopeCycleCounterScopeOuter(CmdList.ExecuteStat);

        CmdList.bExecuting =          true;
        check(CmdList.Context          || CmdList.ComputeContext);

        FMemMarkMark(FMemStack::Get());

        //Set up moreGPUofMask.
#if     WITH_MGPU
        if(CmdList.Context != nullptr) {

            CmdList.Context->RHISetGPUMask(CmdList.InitialGPUMask);
        }
        if(CmdList.ComputeContext != nullptr && CmdList.ComputeContext != CmdList.Context) {

            CmdList.ComputeContext->RHISetGPUMask(CmdList.InitialGPUMask);
        }
#endif

        FRHICommandListDebugContext DebugContext;
        FRHICommandListIteratorIter(CmdList); //Statistics
        execution information.
#if     STATS
        boolbDoStats = CVarRHICmdCollectRHIThreadStatsFromHighLevel.GetValueOnRenderThread()
>  0&& FThreadStats::IsCollectingData() && (IsInRenderingThread() || IsInRHIThread()); if(bDoStats)

        {
            while    (Iter.HasCommandsLeft())
            {
                TStatIdDataconst* Stat = GCurrentExecuteStat.GetRawPointer();
                FScopeCycleCounterScope(GCurrentExecuteStat);
                while(Iter.HasCommandsLeft() && Stat == GCurrentExecuteStat.GetRawPointer()) {

                    FRHICommandBase* Cmd = Iter.NextCommand(); Cmd-
                    >ExecuteAndDestruct(CmdList, DebugContext);
                }
            }
        }
        else
        //      Counts the specified events.
#elif     ENABLE_STATNAMEDEVENTS
        boolbDoStats = CVarRHICmdCollectRHIThreadStatsFromHighLevel.GetValueOnRenderThread()
>  0&& GCycleStatsShouldEmitNamedEvents && (IsInRenderingThread() || IsInRHIThread()); if(bDoStats)
```

```
    {
        while    (Iter.HasCommandsLeft())
        {
            PROFILER_CHARconst* Stat = GCurrentExecuteStat.StatString;
            FScopeCycleCounterScope(GCurrentExecuteStat);
            while(Iter.HasCommandsLeft() && Stat == GCurrentExecuteStat.StatString) {

                FRHICommandBase* Cmd = Iter.NextCommand(); Cmd-
                >ExecuteAndDestruct(CmdList, DebugContext);
            }
        }
    }
    else
#endif
    //    A non-debug or non-statistics version.
    {
        //Loop through all commands, execute and destroy them.
        while(Iter.HasCommandsLeft()) {

            FRHICommandBase* Cmd = Iter.NextCommand();
            GCurrentCommand = Cmd;
            Cmd->ExecuteAndDestruct(CmdList, DebugContext);
        }
    }
    //    Recharge command list.
    CmdList.Reset();
}
```

From this, we can see that FRHICommandListExecutor handles various complex tasks and determines the order, waiting, and dependency of tasks, as well as the dependency and waiting relationships between threads. The above code involves two important task types:

```
//DistributionRHIThread tasks.
classFDispatchRHIThreadTask {

    FRHICommandListBase* RHICmdList;//A list of commands to be dispatched.
    boolbRHIThread;//Is it inRHIDispatched in thread.

public:
    FDispatchRHIThreadTask(FRHICommandListBase* InRHICmdList,boolbInRHIThread)
        : RHICmdList(InRHICmdList) ,
        bRHIThread(bInRHIThread)
    { }
    FORCEINLINE                                TStatIdGetStatId()const;
    staticESubsequentsMode::TypeGetSubsequentsMode()    {return
    ESubsequentsMode::TrackSubsequents; }


    //The expected thread is determined by whetherRHIThread/Is it in independentRHIDetermined by
    variables such as threads. ENamedThreads::TypeGetDesiredThread() {

        returnbRHIThread ? (IsRunningRHIInDedicatedThread() ? ENamedThreads::RHIThread :
CPrio_RHIThreadOnTaskThreads.Get()) : ENamedThreads::GetRenderThread_Local();
    }

    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
```

```
MyCompletionGraphEvent)
	{
		//The previous task is
		RHIThreadTask. FGraphEventArrayPrereq;
		if	(RHIThreadTask.GetReference())
		{
			Prereq.Add(RHIThreadTask);
		}
		//	Put the current task intoPrevRHIThreadTaskmiddle.
		PrevRHIThreadTask	=RHIThreadTask;
		//createFExecuteRHIThreadTaskTasks and assign toRHIThreadTask.
		RHIThreadTask = TGraphTask<FExecuteRHIThreadTask>::CreateTask(&Prereq,
		CurrentThread).ConstructAndDispatchWhenReady(RHICmdList);
	}
};

//implementRHIThread tasks.
classFExecuteRHIThreadTask {

	FRHICommandListBase* RHICmdList;

public:
	FExecuteRHIThreadTask(FRHICommandListBase*		InRHICmdList)
		: RHICmdList(InRHICmdList)
	{
	}

	FORCEINLINE TStatIdGetStatId()const;
	staticESubsequentsMode::TypeGetSubsequentsMode() {return
	ESubsequentsMode::TrackSubsequents; }

	//Depending on whetherRHIThread selectionRHIor the
	rendering thread. ENamedThreads::TypeGetDesiredThread() {

		returnIsRunningRHIInDedicatedThread() ? ENamedThreads::RHIThread :
		CPrio_RHIThreadOnTaskThreads.Get();
	}

	void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
	MyCompletionGraphEvent)
	{
		//	Setting Global VariablesGRHIThreadId
		if	(IsRunningRHIInTaskThread())
		{
			GRHIThreadId = FPlatformTLS::GetCurrentThreadId();
		}

		//implementRHICommand queue.
		{
			//Critical section, ensuring thread access safety.
			FScopeLockLock(&GRHIThreadOnTasksCritical);

			FRHICommandListExecutor::ExecuteInner_DoExecute(*RHICmdList); delete
			RHICmdList;
		}

		//	Clear global variablesGRHIThreadId
		if	(IsRunningRHIInTaskThread())
```

```
        {
            GRHIThreadId    = 0;
        }
    }
};
```

As can be seen above, when dispatching and translating command queues, it may be executed in a dedicated RHI thread, or it may be executed in a rendering thread or a worker thread.

10.4.1.2 GRHICommandList

At first glance, GRHICommandList looks like an instance of FRHICommandListBase, but its actual type is FRHICommandListExecutor. Its declaration and implementation are as follows:

```
// Engine\Source\Runtime\RHI\Public\RHICommandList.h extern RHI_API
FRHICommandListExecutor GRHICommandList;

// Engine\Source\Runtime\RHI\Private\RHICommandList.cpp RHI_API
FRHICommandListExecutor GRHICommandList;
```

The global or static interfaces of GRHICommandList are as follows:

```
FRHICommandListImmediate& FRHICommandListExecutor::GetImmediateCommandList() {

    return GRHICommandList.CommandListImmediate;
}

FRHIAsyncComputeCommandListImmediate&
FRHICommandListExecutor::GetImmediateAsyncComputeCommandList() {

    return GRHICommandList.AsyncComputeCmdListImmediate;
}
```

There are a lot of GRHICommandList use cases in UE's rendering module and RHI module, take one of them:

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

void ServiceLocalQueue() {


FTaskGraphInterface::Get().ProcessThreadUntilIdle(ENamedThreads::GetRenderThread_Local());

    if(IsRunningRHIInSeparateThread()) {


FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::Dis
patchToRHIThread);
    }
}
```

In the RHI command queue module, in addition to GRHICommandList, many global task variables are also involved:

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

static    FGraphEventArray    AllOutstandingTasks;
static    FGraphEventArray    WaitOutstandingTasks;
static    FGraphEventRef    RHIThreadTask;
static    FGraphEventRef    PrevRHIThreadTask;
static    FGraphEventRef    RenderThreadSublistDispatchTask;
```

Their codes for creating or adding tasks are as follows:

```
void FRHICommandListBase::QueueParallelAsyncCommandListSubmit(FGraphEventRef*
AnyThreadCompletionEvents, ...)
{
    (.....)

    if(Num && IsRunningRHIInSeparateThread()) {

        (.....)

        //createFParallelTranslateSetupCommandListTask. FGraphEventRef
        TranslateSetupCompletionEvent =
TGraphTask<FParallelTranslateSetupCommandList>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(CmdList, Num,          &RHICmdLists[0],
bIsPrepass);
        QueueCommandListSubmit(CmdList); //Add toAll Outstanding Tasks.
        AllOutstandingTasks.Add(TranslateSetupCompletionEvent);



        (.....)

        FGraphEventArray Prereq;
        FRHICommandListBase** RHICmdLists =
(FRHICommandListBase**)Alloc(sizeof(FRHICommandListBase*)1+ Last - Start), alignof
(FRHICommandListBase*));
        //All external tasksAnyThreadCompletionEventsAdd to the corresponding
        list. for(int32 Index = Start; Index <= Last; Index++) {

            FGraphEventRef& AnyThreadCompletionEvent = AnyThreadCompletionEvents[Index];
            FRHICommandList* CmdList = CmdLists[Index];
            RHICmdLists[Index - Start] = CmdList; if
            (AnyThreadCompletionEvent.GetReference()) {

                Prereq.Add(AnyThreadCompletionEvent);
                AllOutstandingTasks.Add(AnyThreadCompletionEvent);
                WaitOutstandingTasks.Add(AnyThreadCompletionEvent);
            }
        }

        (.....)

        //Parallel translation tasksFParallelTranslateCommandList. FGraphEventRef TranslateCompletionEvent =
        TGraphTask<FParallelTranslateCommandList>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(&RHICmdLists[0],1+ Last

- Start, ContextContainer, bIsPrepass);
AllOutstandingTasks.Add(TranslateCompletionEvent);
```

```
        (.....)
}

void FRHICommandListBase::QueueAsyncCommandListSubmit(FGraphEventRef&
AnyThreadCompletionEvent, class FRHICommandList* CmdList)
{
    (.....)

    //  Handling external tasksAnyThreadCompletionEvent
    if  (AnyThreadCompletionEvent.GetReference())
    {
        if(IsRunningRHIInSeparateThread()) {

            AllOutstandingTasks.Add(AnyThreadCompletionEvent);
        }
        WaitOutstandingTasks.Add(AnyThreadCompletionEvent);
    }

    (.....)
}

classFDispatchRHIThreadTask {

    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        (.....)

        //createRHIThread TasksFExecuteRHIThreadTask.
        PrevRHIThreadTask = RHIThreadTask;
        RHIThreadTask = TGraphTask<FExecuteRHIThreadTask>::CreateTask(&Prereq,
        CurrentThread).ConstructAndDispatchWhenReady(RHICmdList);
    }
};

classFParallelTranslateSetupCommandList {

    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        (.....)

//Creating parallel translation tasksFParallelTranslateCommandList. FGraphEventRef
TranslateCompletionEvent = TGraphTask<FParallelTranslateCommandList>::CreateTask(nullptr,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(&RHICmdLists[Start], Last - Start,
ContextContainer, bIsPrepass);                                                                    1  +

        MyCompletionGraphEvent->DontCompleteUntil(TranslateCompletionEvent);
        //useRHICmdListInterfaceFRHICommandWaitForAndSubmitSubListParallelSubmit the task and you will eventually
enter All Outstanding TasksandWaitOutstandingTasks.
        ALLOC_COMMAND_CL(*RHICmdList, FRHICommandWaitForAndSubmitSubListParallel)
(TranslateCompletionEvent, ContextContainer, EffectiveThreads, ThreadIndex++);

};

void FRHICommandListExecutor::ExecuteInner(FRHICommandListBase& CmdList) {
```

```
	(……)

	if(IsRunningRHIInSeparateThread())
	{
		(……)

		if(AllOutstandingTasks.Num() || RenderThreadSublistDispatchTask.GetReference()) {

			(……)
			//Create a rendering thread subcommand to dispatch (submit) tasks
			FDispatchRHIThreadTask. RenderThreadSublistDispatchTask =
TGraphTask<FDispatchRHIThreadTask>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(SwapCmdList, bAsyncSubmit);

		}
		else
		{
			(……)
			PrevRHIThreadTask = RHIThreadTask;
			//Create a rendering thread subcommand translation taskFExecuteRHIThreadTask.
			RHIThreadTask = TGraphTask<FExecuteRHIThreadTask>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(SwapCmdList);
		}

		(……)
}
```

To summarize the functions of these task variables:

Task variables	Thread of execution	describe
AllOutstandingTasks	Rendering, RHI, Work	A list of all tasks being processed or to be processed. The types are FParallelTranslateSetupCommandList and FParallelTranslateCommandList.
WaitOutstandingTasks	Rendering, RHI, Work	The task list to be processed. The type is FParallelTranslateSetupCommandList, FParallelTranslateCommandList.
RHIThreadTask	RHI, work	The RHI thread task being processed. The type is FExecuteRHIThreadTask.
PrevRHIThreadTask	RHI, work	The last processed RHIThreadTask. Type is FExecuteRHIThreadTask.
RenderThreadSublistDispatchTask	Rendering, RHI, Work	The task being dispatched (submitted). The type is FDispatchRHIThreadTask.

10.4.1.3 D3D11 Command Execution

This section will study the general RHI and D3D11 command operation process and mechanism of UE4.26 on the PC platform. Since the default RHI of UE4.26 on the PC platform is D3D11, and the default values of several key console variables are as follows :

```
Cmd: r.RHICmdBypass
r.RHICmdBypass = "1"      LastSetBy: Console
Cmd: r.RHIThread.Enable
LogConsoleResponse: Display: Usage: r.RHIThread.Enable 0=off,  1=dedicated thread,  2=task threads; Currently 0
```

That is to say, the command skip mode is turned on and the RHI thread is disabled. In this case, when an interface of FRHICommandList is called, a separate FRHICommand is not generated, but the Context method is called directly. Take FRHICommandList:: DrawPrimitive as an example:

```
classRHI_API FRHICommandList: public FRHIComputeCommandList {

    void DrawPrimitive(uint32 BaseVertexIndex, uint32 NumPrimitives, uint32 NumInstances) {

        //  By defaultBypassfor1,Enter this
        if  branch. (Bypass())
        {
            //Directly call graphicsAPIThe corresponding method of the context.
            GetContext().RHIDrawPrimitive(BaseVertexIndex, NumPrimitives, NumInstances); return;

        }

        //Assign a separateFRHICommandDrawPrimitiveOrder.
        ALLOC_COMMAND(FRHICommandDrawPrimitive)(BaseVertexIndex,          NumPrimitives,
        NumInstances);
    }
}
```

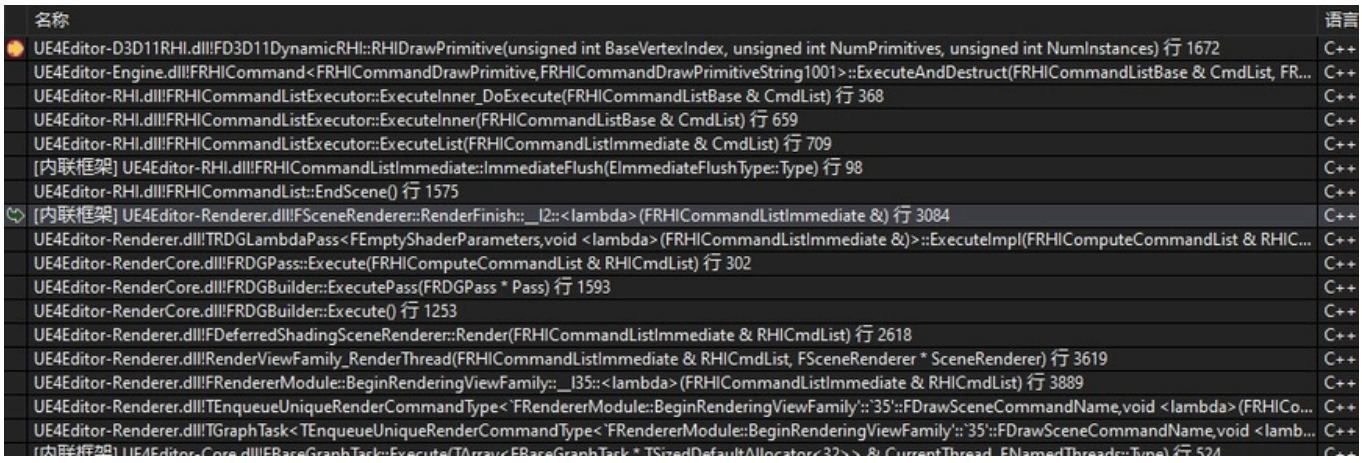
Therefore, under the PC's default graphics API (D3D11), r.RHICmdBypass1 and r.RHIThread.Enable0, FRHICommandList will directly call the interface of the graphics API context, which is equivalent to calling the graphics API synchronously. At this time, the graphics API runs on the rendering thread (if enabled).

Next, set r.RHICmdBypass to 0, but keep r.RHIThread.Enable to 0. At this time, the Context method is no longer called directly, but by generating separate FRHICommands, which are then executed by FRHICommandList-related objects. Taking FRHICommandList ::DrawPrimitive as an example, the call stack is as follows:

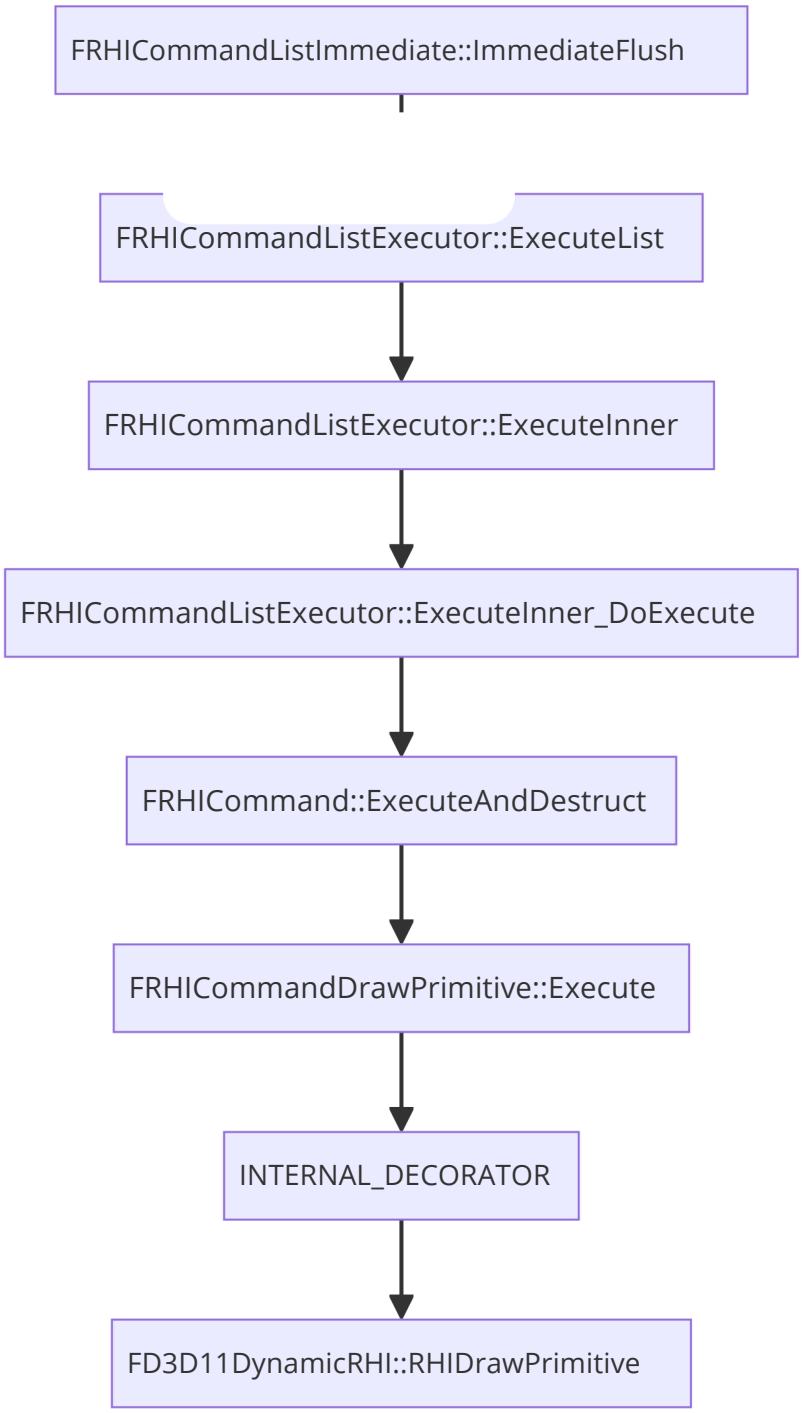
```
classRHI_API FRHICommandList: public FRHIComputeCommandList {

    void FRHICommandList::DrawPrimitive(uint32 BaseVertexIndex, uint32 NumPrimitives, uint32 NumInstances)

    {
        //  By defaultBypassfor1,Enter this
        if  branch. (Bypass())
        {
            //Directly call graphicsAPIThe corresponding method of the context.
            GetContext().RHIDrawPrimitive(BaseVertexIndex, NumPrimitives, NumInstances);
```

```
            return;
        }

        //  Assign a separateFRHICommandDrawPrimitiveOrder.
        //  ALLOC_COMMANDThe macro callsAllocCommandinterface.
        ALLOC_COMMAND(FRHICommandDrawPrimitive)(BaseVertexIndex,  NumPrimitives,
        NumInstances);
    }

    template <typename TCmd>
    void*AllocCommand() {

        returnAllocCommand(sizeof(TCmd),  alignof(TCmd));
    }

    void*AllocCommand(int32 AllocSize, int32 Alignment) {

        FRHICommandBase* Result = (FRHICommandBase*) MemManager.Alloc(AllocSize, Alignment);

        + +NumCommands;
        // CommandLinkPoints to the previous command nodeNext.
        * CommandLink = Result;
        //WillCommandLinkAssign the value to the current
        nodeNext. CommandLink = &Result->Next; return
        Result;
    }
}
```

The command instance allocated using ALLOC_COMMAND will enter the command list of FRHICommandListBase, but it is not executed at this time. Instead, it waits for another suitable time to execute, such as in FRHICommandListImmediate::ImmediateFlush. The following is the call stack for executing FRHICommandList:



From the call stack, we can see that in this case, the command execution process becomes complicated, with many more intermediate execution steps. Taking FRHICommandList::DrawPrimitive as an example, the call flow diagram is as follows:

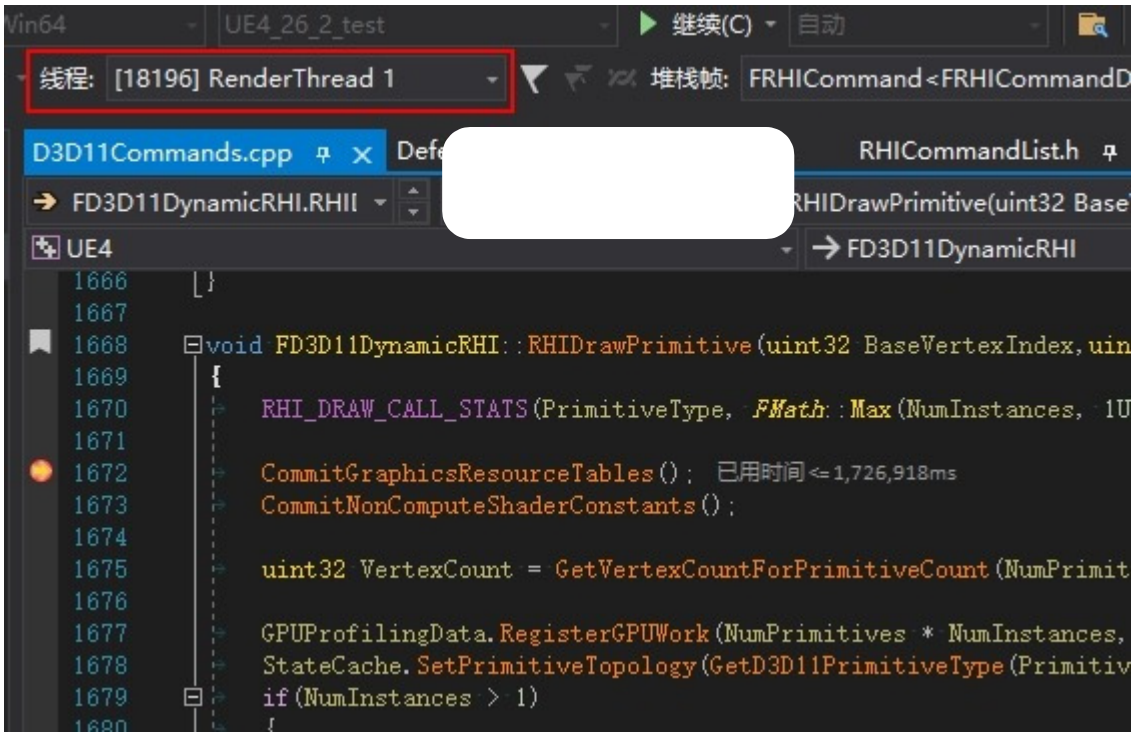


The above figure uses the macro INTERNAL_DECORATOR, and its and related macro definitions are as follows:

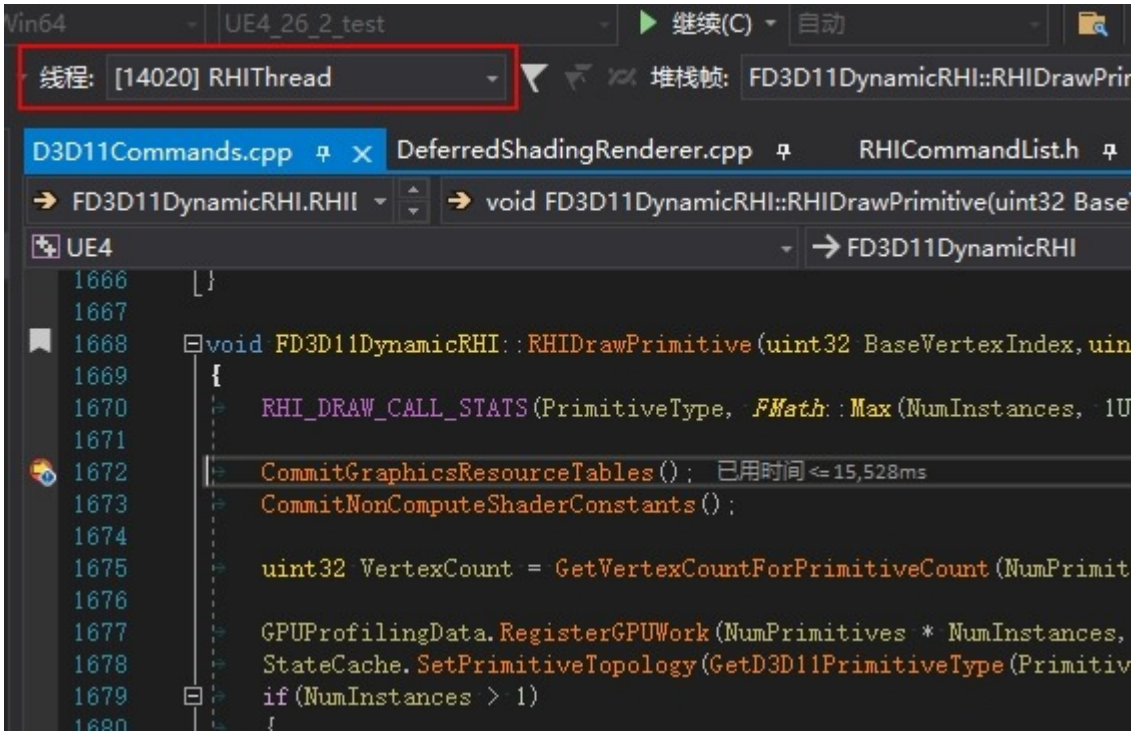
```
// Engine\Source\Runtime\RHI\Public\RHICommandListCommandExecutes.inl

#define    INTERNAL_DECORATOR(Method) CmdList.GetContext().Method
#define    INTERNAL_DECORATOR_COMPUTE(Method) CmdList.GetComputeContext().Method
```

It is equivalent to calling the Context interface of CommandList through a macro.

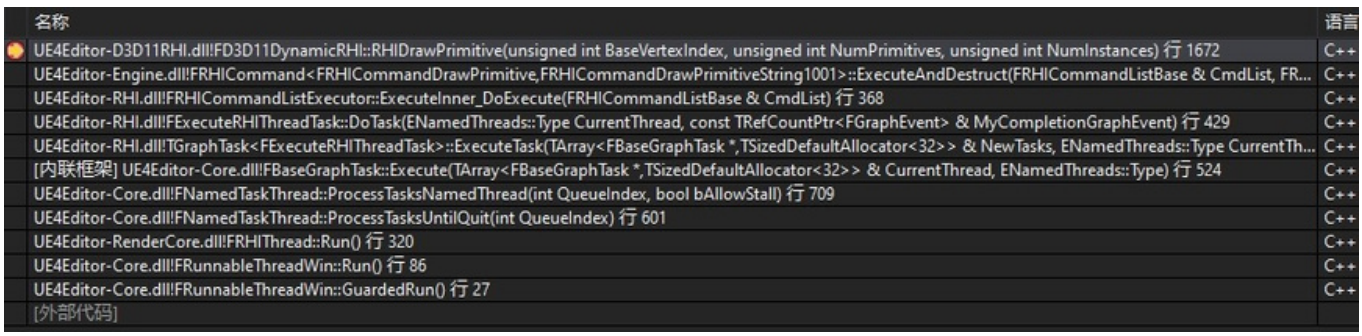
When the RHI is disabled (r.RHIThread.Enable == 0), the above call is executed on the rendering thread:



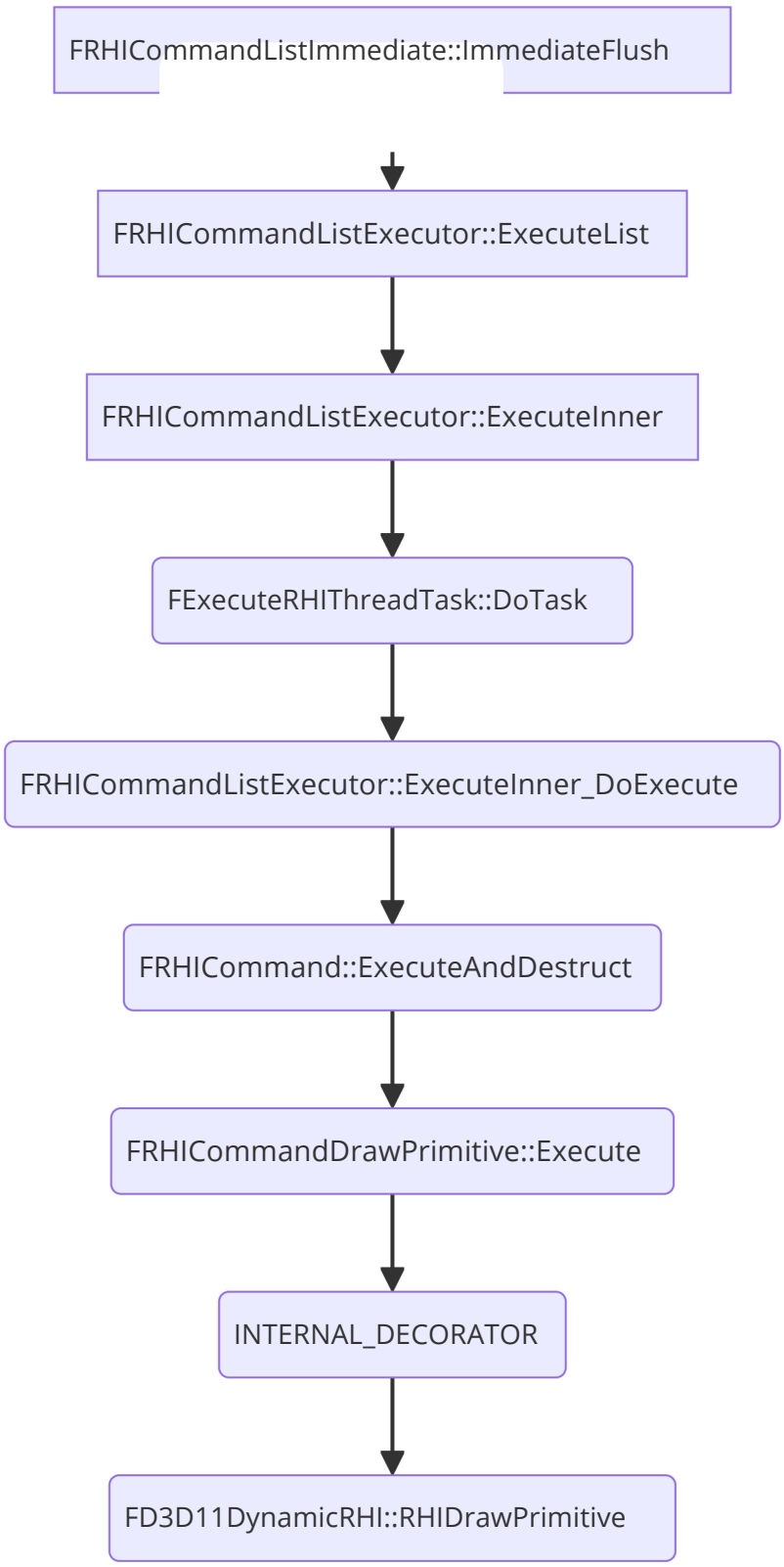
Next, set r.RHIThread.Enable to 1 to enable the RHI thread. The thread that runs the command now becomes RHI:



And the call stack is launching the task from the RHI thread of TaskGraph:



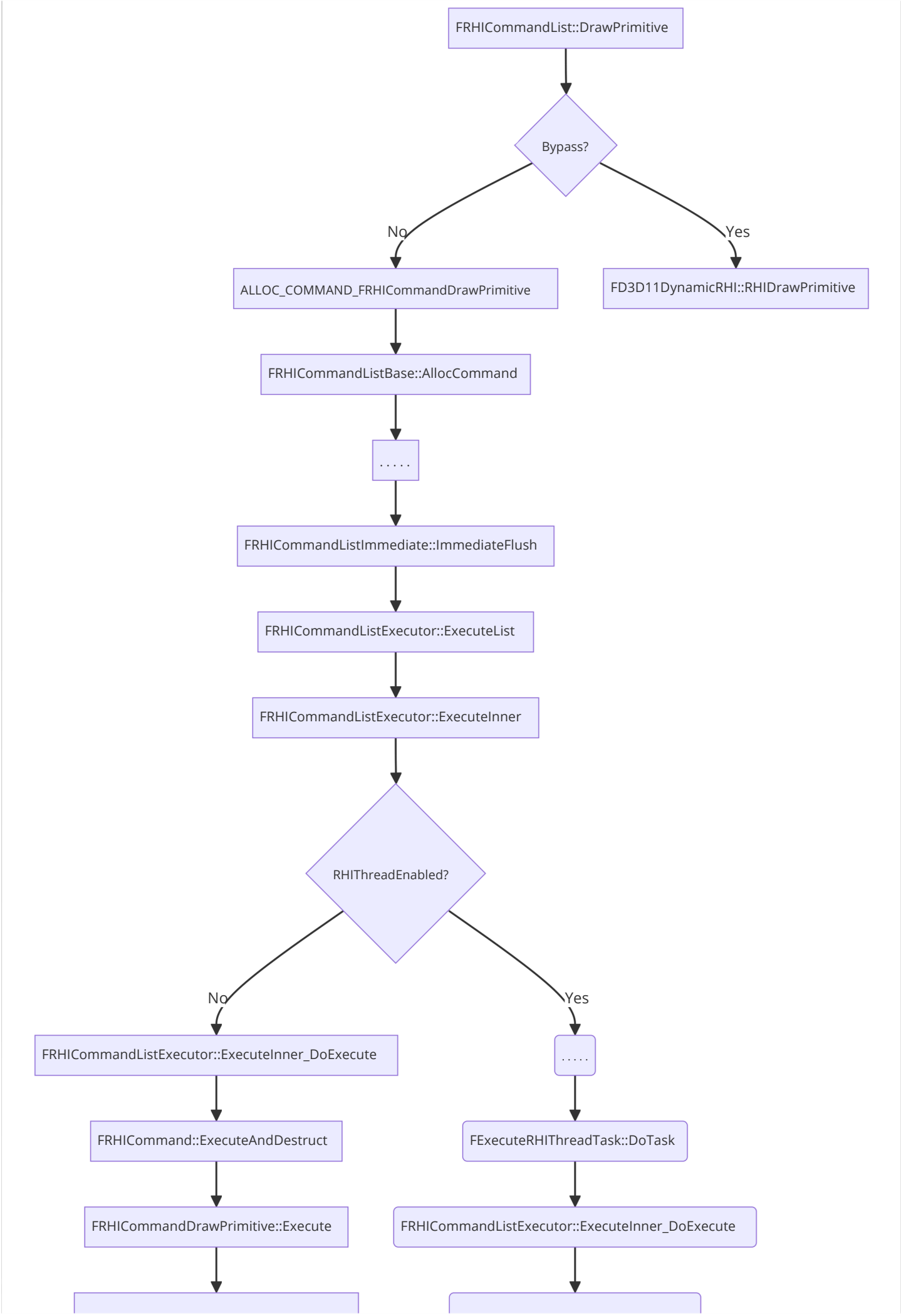
At this point, the flow chart of command execution is as follows:

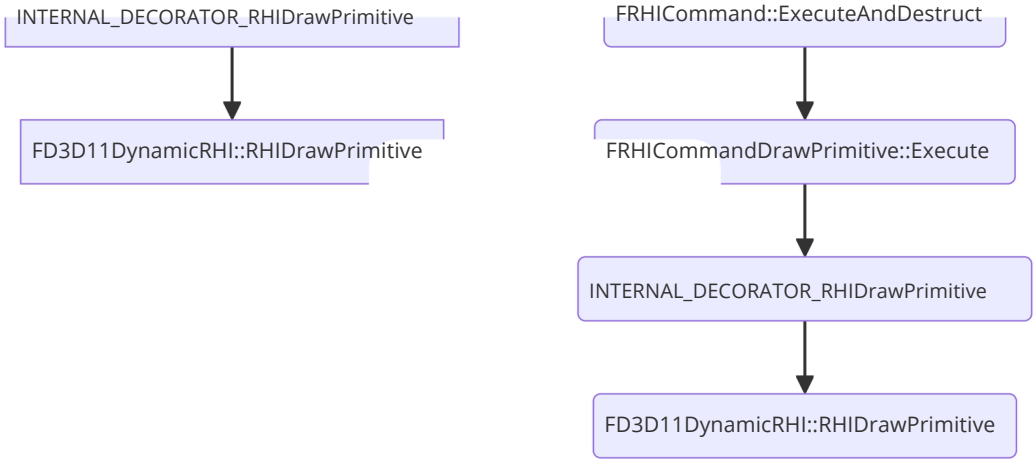


In the above flowchart, the square corners are executed in the rendering thread, while the rounded corners are executed in the RHI thread. After the RHI thread is turned on, its statistics will appear:

Left: Statistics without RHI thread enabled. Right: Statistics after RHI thread enabled.

The following is a flowchart for opening or closing the Bypass and RHI threads (taking the call to DrawPrimitive of D3D11 as an example):


FRHICommandList::DrawPrimitive
Bypass?
No
Yes
ALLOC_COMMAND_FRHICommandDrawPrimitive
FD3D11DynamicRHI::RHIDrawPrimitive
FRHICommandListBase::AllocCommand
.....
FRHICommandListImmediate::ImmediateFlush
FRHICommandListExecutor::ExecuteList
FRHICommandListExecutor::ExecuteInner
RHIThreadEnabled?
No
Yes
FRHICommandListExecutor::ExecuteInner_DoExecute
.....
FRHICommand::ExecuteAndDestruct
FExecuteRHIThreadTask::DoTask
FRHICommandDrawPrimitive::Execute
FRHICommandListExecutor::ExecuteInner_DoExecute



In the above flowchart, square corners indicate execution in the rendering thread, and rounded corners indicate execution in the RHI thread.

10.4.2 ImmediateFlush

In section 10.3.3 FDynamicRHI, the refresh type (FlushType) is mentioned, which refers to the type defined by EImmediateFlushType:

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

namespace  EImmediateFlushType
{
	enum Type
	{
		WaitForOutstandingTasksOnly =0,//Wait for only the currently processing task to
		complete. DispatchToRHIThread,                  //  Distribute toRHIThreads.
		WaitForDispatchToRHIThread,                     //  Waiting for deliveryRHIThreads.
		FlushRHIThread,                                 //  refreshRHIThreads.
		FlushRHIThreadFlushResources,                   //  refreshRHIThreads and Resources
		FlushRHIThreadFlushResourcesFlushDeferredDeletes        //refreshRHIThreads/resources and lazy deletion.
	};
};
```

The differences between the values in EImmediateFlushType are reflected in the implementation code of FRHICommandListImmediate::ImmediateFlush:

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

void FRHICommandListImmediate::ImmediateFlush(EImmediateFlushType::Type FlushType) {

	switch(FlushType)
	{
	//Wait for the task to complete.
	caseEImmediateFlushType::WaitForOutstandingTasksOnly:

		{
			WaitForTasks();
		}
		break;
	//DistributionRHIThread (execution command queue)
	caseEImmediateFlushType::DispatchToRHIThread:
```

```
        {
            if (HasCommands())
            {
                GRHICommandList.ExecuteList(*this);
            }
        }
        break;
    // waitRHIThread dispatch.
    case EImmediateFlushType::WaitForDispatchToRHIThread:
        {
            if(HasCommands())
            {
                GRHICommandList.ExecuteList(*this);
            }
            WaitForDispatch();
        }
        break;
    //refreshRHIThreads.
    caseEImmediateFlushType::FlushRHIThread:
        {
            // Dispatch and waitRHIThreads.
            if (HasCommands())
            {
                GRHICommandList.ExecuteList(*this);
            }
            WaitForDispatch();

            // waitRHIThread tasks.
            if (IsRunningRHIInSeparateThread())
            {
                WaitForRHIThreadTasks();
            }

            //Resets the list of tasks being processed.
            WaitForTasks(true);
        }
        break;
    case EImmediateFlushType::FlushRHIThreadFlushResources:
    case EImmediateFlushType::FlushRHIThreadFlushResourcesFlushDeferredDeletes:
        {
            if(HasCommands())
            {
                GRHICommandList.ExecuteList(*this);
            }
            WaitForDispatch();
            WaitForRHIThreadTasks();
            WaitForTasks(true);

            //Refresh the pipeline state cache resources.
            PipelineStateCache::FlushResources(); //Flushes resources
            that are about to be deleted.
            FRHIResource::FlushPendingDeletes(FlushType == 
EImmediateFlushType::FlushRHIThreadFlushResourcesFlushDeferredDeletes);
        }
        break;
    }
}
```

The above code involves several interfaces for processing and waiting tasks, which are implemented as follows:

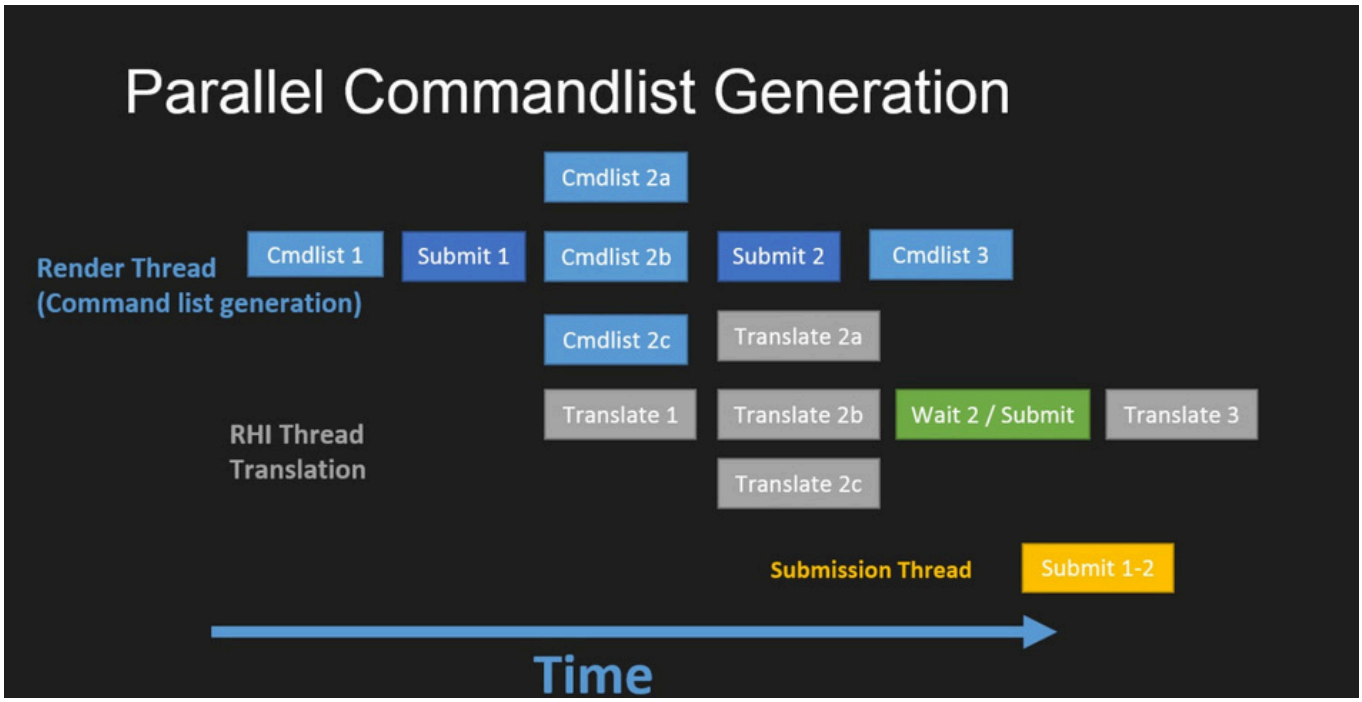
```
//Wait for the task to complete.
void FRHICommandListBase::WaitForTasks(bool bKnownToBeComplete)
{
	if(WaitOutstandingTasks.Num()) {

		//Check if there are any unfinished waiting tasks.
		boolbAny =false;
		for(int32 Index =0; Index < WaitOutstandingTasks.Num(); Index++) {

			if(!WaitOutstandingTasks[Index]->IsComplete()) {

				bAny =true;
				break;
			}
		}
		// Use it if it existsTaskGraphThe interface starts the thread
		if waiting. (bAny)
		{
			ENamedThreads::Type RenderThread_Local =
ENamedThreads::GetRenderThread_Local();
			FTaskGraphInterface::Get().WaitUntilTasksComplete(WaitOutstandingTasks,
RenderThread_Local);
		}
		//Reset the list of pending tasks.
		WaitOutstandingTasks.Reset();
	}
}

//Waiting for the rendering thread to dispatch.
void FRHICommandListBase::WaitForDispatch() {

	//ifRenderThreadSublistDispatchTaskCompleted, Leave it blank.
	if(RenderThreadSublistDispatchTask.GetReference() && RenderThreadSublistDispatchTask-
> IsComplete())
	{
		RenderThreadSublistDispatchTask = nullptr;
	}

	// RenderThreadSublistDispatchTaskThere are unfinished tasks. while
	(RenderThreadSublistDispatchTask.GetReference()) {

		ENamedThreads::Type RenderThread_Local = ENamedThreads::GetRenderThread_Local();
		FTaskGraphInterface::Get().WaitUntilTaskCompletes(RenderThreadSublistDispatchTask, RenderThread_Local);

		if(RenderThreadSublistDispatchTask.GetReference() &&
RenderThreadSublistDispatchTask->IsComplete())
		{
			RenderThreadSublistDispatchTask = nullptr;
		}
	}
}

//waitRHIThe thread task is completed.
void FRHICommandListBase::WaitForRHIThreadTasks()
```

```
{
	boolbAsyncSubmit = CVarRHICmdAsyncRHIThreadDispatch.GetValueOnRenderThread() >0;
	ENamedThreads::Type RenderThread_Local = ENamedThreads::GetRenderThread_Local();

	//	Equivalent to executionFRHICommandListBase::WaitForDispatch()
	if	(bAsyncSubmit)
	{
		if(RenderThreadSublistDispatchTask.GetReference()		&&
RenderThreadSublistDispatchTask->IsComplete())
		{
			RenderThreadSublistDispatchTask		= nullptr;
		}
		while	(RenderThreadSublistDispatchTask.GetReference())
		{
			if	(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
			{
				while(!RenderThreadSublistDispatchTask->IsComplete()) {

					FPlatformProcess::SleepNoStats(0);
				}
			}
			else
			{

FTaskGraphInterface::Get().WaitUntilTaskCompletes(RenderThreadSublistDispatchTask, RenderThread_Local);

			}

			if(RenderThreadSublistDispatchTask.GetReference() &&
RenderThreadSublistDispatchTask->IsComplete())
			{
				RenderThreadSublistDispatchTask = nullptr;
			}
		}
		// now we can safely look at RHIThreadTask
	}

	//ifRHIIf the thread task is completed, clear the task.
	if(RHIThreadTask.GetReference() && RHIThreadTask->IsComplete()) {

		RHIThreadTask = nullptr;
		PrevRHIThreadTask = nullptr;
	}

	//ifRHIThe thread has unfinished tasks.  Then execute and wait.
	while	(RHIThreadTask.GetReference())
	{
		//	If it is already being processed, usesleep(0)Skip this time slice.
		if	(FTaskGraphInterface::Get().IsThreadProcessingTasks(RenderThread_Local))
		{
			while(!RHIThreadTask->IsComplete()) {

				FPlatformProcess::SleepNoStats(0);
			}
		}
		//	The task has not been processed yet, start and wait for it.
		else
		{
```

```
                FTaskGraphInterface::Get().WaitUntilTaskCompletes(RHIThreadTask,
RenderThread_Local);
            }

            //ifRHIIf the thread task is completed, clear the task.
            if(RHIThreadTask.GetReference() && RHIThreadTask->IsComplete()) {

                RHIThreadTask = nullptr;
                PrevRHIThreadTask = nullptr;
            }
        }
}
```

10.4.3 Parallel Rendering

At the beginning of this article, it was mentioned that when the RHI thread is turned on, the RHI thread is responsible for translating the RHI intermediate instructions pushed by the rendering thread into the GPU instructions of the corresponding graphics platform. If the rendering thread generates RHI intermediate instructions in parallel, the RHI thread will also translate them in parallel.



Before formally explaining parallel rendering and translation, you need to understand some basic concepts and types.

10.4.3.1 FParallelCommandListSet

The definition of FParallelCommandListSet is as follows:

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

classFParallelCommandListSet {

public:
    //The view to which it belongs.
    constFViewInfo& View;
```

```
    //The parent command queue.
    FRHICommandListImmediate& //        ParentCmdList;
    ScenarioRTSnapshot.
    FSceneRenderTargets* Snapshot;

    TStatId            ExecuteStat;
    int32 Width;
    int32 NumAlloc;
    int32 MinDrawsPerCommandList;
    //Whether to balance the command queue,Seer.RHICmdBalanceParallelLists
    boolbBalanceCommands;
    // see r.RHICmdSpewParallelListBalance bool
    bSpewBalance;

    //Command queue list.
    TArray<FRHICommandList*,SceneRenderingAllocator> //              CommandLists;
    Synchronous events.
    TArray<FGraphEventRef,SceneRenderingAllocator> Events; //The number of times
    the command queue is drawn, if -1Unknown. Overestimation is better than
    nothing. TArray<int32,SceneRenderingAllocator> NumDrawsIfKnown;

    FParallelCommandListSet(TStatId InExecuteStat,constFViewInfo& InView,
FRHICommandListImmediate& InParentCmdList,boolbInCreateSceneContext);
    virtual ~FParallelCommandListSet();

    //Get the quantity.
    int32NumParallelCommandLists() //Create a const;
    new parallel command queue.
    FRHICommandList*NewParallelCommandList(); //Get the
    predecessor task.
    FORCEINLINE FGraphEventArray*GetPrereqs(); //Added
    parallel command queue.
    void AddParallelCommandList(FRHICommandList* CmdList, FGraphEventRef& CompletionEvent, int32
InNumDrawsIfKnown =-1);
    virtualvoid SetStateOnCommandList(FRHICommandList& CmdList) {} //Wait for the
    task to complete.
    static void WaitForTasks();

protected:
    //Dispatched, must be called by subclasses.
    void Dispatch(boolbHighPriority =false); //Allocate a new
    command queue.
    FRHICommandList*AllocCommandList(); //Whether
    to create a scene context.
    boolbCreateSceneContext;

private:
    void    WaitForTasksInternal();
};
```

The following is the implementation code of the important interface of FParallelCommandListSet:

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

FRHICommandList*FParallelCommandListSet::AllocCommandList() {

    NumAlloc++;
```

```
    returnnew FRHICommandList(ParentCmdList.GetGPUMask());
}

void FParallelCommandListSet::Dispatch(boolbHighPriority) {

    ENamedThreads::Type RenderThread_Local = ENamedThreads::GetRenderThread_Local(); if(bSpewBalance)

    {
        //Waiting for the previous task to complete.
        for(auto& Event : Events) {

            FTaskGraphInterface::Get().WaitUntilTaskCompletes(Event, RenderThread_Local);
        }
    }

    //Whether to translate in parallel.
    boolbActuallyDoParallelTranslate = GRHISupportsParallelRHIExecute && CommandLists.Num() >=
    CVarRHICmdMinCmdlistForParallelSubmit.GetValueOnRenderThread();
    if(bActuallyDoParallelTranslate) {

        int32 Total =0;
        boolbIndeterminate =false; for(int32 Count :
        NumDrawsIfKnown) {

            //Not sure how much is in there.    Assume that parallel translation should be done.
            if(Count <0)
            {
                bIndeterminate =        true;
                break;
            }
            Total += Count;
        }

        //The number of command queues is too small, and no parallel translation is performed.
        if(!bIndeterminate && Total < MinDrawsPerCommandList) {

            bActuallyDoParallelTranslate=false;
        }
    }

    if(bActuallyDoParallelTranslate) {

        //Ensure support for parallelRHIimplement.
        check(GRHISupportsParallelRHIExecute); NumAlloc
        -= CommandLists.Num();

        //Enqueue parallel asynchronous command queue submission using parent command queue.
        ParentCmdList.QueueParallelAsyncCommandListSubmit(&Events[0], bHighPriority, &CommandLists[0],
&NumDrawsIfKnown[0], CommandLists.Num(), (MinDrawsPerCommandList *4) / 3, bSpewBalance);

        SetStateOnCommandList(ParentCmdList); //Finish
        PassRendering.
        ParentCmdList.EndRenderPass();
    }
    else//Non-parallel mode.
    {
        for(int32 Index =0; Index < CommandLists.Num(); Index++) {
```

```
                ParentCmdList.QueueAsyncCommandListSubmit(Events[Index], CommandLists[Index]); NumAlloc--;

            }
        }

        //Reset data.
        CommandLists.Reset();
        Snapshot = nullptr;
        Events.Reset();


        //Waiting for the rendering thread to complete.
        FTaskGraphInterface::Get().ProcessThreadUntilIdle(RenderThread_Local);
}

FParallelCommandListSet::~FParallelCommandListSet() {

        GOutstandingParallelCommandListSet = nullptr;
}

FRHICommandList*FParallelCommandListSet::NewParallelCommandList() {

        //Create a new command queue.
        FRHICommandList* Result = AllocCommandList(); Result-
        >ExecuteStat = ExecuteStat;
        SetStateOnCommandList(*Result); if
        (bCreateSceneContext) {

            FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(ParentCmdList); //
                Create a sceneRTSnapshot.
            if  (!Snapshot)
            {
                Snapshot = SceneContext.CreateSnapshot(View);
            }
            //  WillRTThe snapshot is set on the command queue.
            Snapshot->SetSnapshotOnCmdList(*Result);
        }
        returnResult;
}

//Added parallel command queue.
void FParallelCommandListSet::AddParallelCommandList(FRHICommandList*                    CmdList,
FGraphEventRef& CompletionEvent, int32 InNumDrawsIfKnown)
{
        //Add command queue.
        CommandLists.Add(CmdList); //Add
        wait event.
        Events.Add(CompletionEvent); //
        Increase quantity.
        NumDrawsIfKnown.Add(InNumDrawsIfKnown);
}

void FParallelCommandListSet::WaitForTasks() {

        if(GOutstandingParallelCommandListSet) {

            GOutstandingParallelCommandListSet->WaitForTasksInternal();
        }
}
```

```
void FParallelCommandListSet::WaitForTasksInternal() {

	//Collects events waiting to be processed.
	FGraphEventArray WaitOutstandingTasks;
	for(int32 Index =0; Index < Events.Num(); Index++) {

		if(!Events[Index]->IsComplete()) {

			WaitOutstandingTasks.Add(Events[Index]);
		}
	}

	// If there are tasks being processed, wait for them to complete.
	if (WaitOutstandingTasks.Num())
	{
		ENamedThreads::Type RenderThread_Local = ENamedThreads::GetRenderThread_Local();
		FTaskGraphInterface::Get().WaitUntilTasksComplete(WaitOutstandingTasks, RenderThread_Local);

	}
}
```

FParallelCommandListSet has the following subclasses to meet the parallel rendering logic of different Passes or occasions:

- FAnisotropyPassParallelCommandListSet: Parallel rendering command queue set of various anisotropy passes.
- FPrePassParallelCommandListSet: Parallel rendering command queue set for advance depth Pass.

- FShadowParallelCommandListSet: Parallel rendering command queue set for shadow rendering.
- FRDGParallelCommandListSet: Parallel rendering command queue set of the RDG system.

The following takes FPrePassParallelCommandListSet and FShadowParallelCommandListSet as the analysis objects:

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

classFPrePassParallelCommandListSet: public FParallelCommandListSet {

public:
	FPrePassParallelCommandListSet(FRHICommandListImmediate& InParentCmdList, FSceneRenderer&co nst
InSceneRenderer,constFViewInfo& InView,boolbInCreateSceneContext)
		: FParallelCommandListSet(GET_STATID(STAT_CLP_Prepass), InView, InParentCmdList,
		bInCreateSceneContext)
		, SceneRenderer(InSceneRenderer)
	{
	}

	virtual ~FPrePassParallelCommandListSet() {

		//Dispatching the command list in the destructor.
		Dispatch(true);
```

```
    }

    //Sets the status on a command list.
    virtualvoid SetStateOnCommandList(FRHICommandList& CmdList) override {

        FParallelCommandListSet::SetStateOnCommandList(CmdList);
        FSceneRenderTargets::Get(CmdList).BeginRenderingPrePass(CmdList,false);
        SetupPrePassView(CmdList, View, &SceneRenderer);
    }

private:
    const    FSceneRenderer&     SceneRenderer;
};

classFShadowParallelCommandListSet: public FParallelCommandListSet {

public:
    FShadowParallelCommandListSet(
        FRHICommandListImmediate&       InParentCmdList,
        const    FViewInfo& InView,
        bool bInCreateSceneContext,
        FProjectedShadowInfo&        InProjectedShadowInfo,
        FBeginShadowRenderPassFunction          InBeginShadowRenderPass)
        : FParallelCommandListSet(GET_STATID(STAT_CLP_Shadow), InView, InParentCmdList,
        bInCreateSceneContext)
        , ProjectedShadowInfo(InProjectedShadowInfo)
        , BeginShadowRenderPass(InBeginShadowRenderPass)
    {
        bBalanceCommands =false;
    }

    virtual ~FShadowParallelCommandListSet() {

        //Dispatching the command list in the destructor.
        Dispatch();
    }

    virtualvoid SetStateOnCommandList(FRHICommandList& CmdList) override {

        FParallelCommandListSet::SetStateOnCommandList(CmdList);
        BeginShadowRenderPass(CmdList,false);
        ProjectedShadowInfo.SetStateForView(CmdList);
    }

private:
    //Cast shadow information.
    FProjectedShadowInfo&        ProjectedShadowInfo;
    //Start shadow renderingpassfunction.
    FBeginShadowRenderPassFunction //Shadow       BeginShadowRenderPass;
    depth rendering mode.
    EShadowDepthRenderMode RenderMode;
};
```

Using the above logic is relatively simple, taking PrePass as an example:

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

```
bool FDeferredShadingSceneRenderer::RenderPrePassViewParallel(constFViewInfo& View,
FRHICommandListImmediate& ParentCmdList, TFunctionRef<void()> AfterTasksAreStarted,bool bDoPrePre)

{
	boolbDepthWasCleared =false;

	{
		//structureFPrePassParallelCommandListSetExamples.
		FPrePassParallelCommandListSetParallelCommandListSet(ParentCmdList, *this, View,
			CVarRHICmdFlushRenderThreadTasksPrePass.GetValueOnRenderThread() ==0&&
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() ==0);

		//CallFParallelMeshDrawCommandPass::DispatchDraw.

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

		if(bDoPrePre)
		{
			bDepthWasCleared = PreRenderPrePass(ParentCmdList);
		}
	}

	if(bDoPrePre)
	{
		AfterTasksAreStarted();
	}

	returnbDepthWasCleared;
}

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

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

	if(MaxNumDraws <=0) {

		return;
	}

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

	//	Parallel mode.
	if	(ParallelCommandListSet)
	{
		if(TaskContext.bUseGPUScene) {

			//In completionFMeshDrawCommandPassSetupTaskback,RHIThread will uploadPrimitiveIdVertexBufferlife
make.
			FRHICommandListImmediate &RHICommandList =
GetImmediateCommandList_ForRenderCommand();

			if(TaskEventRef.IsValid()) {

				RHICommandList.AddDispatchPrerequisite(TaskEventRef);
			}
```

```
                RHICommandList.EnqueueLambda([
                    VertexBuffer = PrimitiveIdsBuffer, VertexBufferData = TaskContext.PrimitiveIdBufferData,
                    VertexBufferDataSize = TaskContext.PrimitiveIdBufferDataSize,
                    PrimitiveIdVertexBufferPoolEntry = PrimitiveIdVertexBufferPoolEntry]

(FRHICommandListImmediate& CmdList)
                {
                    // Upload vertex buffer data. void*
                    RESTRICT Data = (void*
RESTRICT)CmdList.LockVertexBuffer(VertexBuffer,0, VertexBufferDataSize, RLM_WriteOnly);
                    FMemory::Memcpy(Data, VertexBufferData, VertexBufferDataSize);
                    CmdList.UnlockVertexBuffer(VertexBuffer);

                    FMemory::Free(VertexBufferData);
                });

                RHICommandList.RHIThreadFence(true);

                bPrimitiveIdBufferDataOwnedByRHIThread =true;
            }

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

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

            //    based onNumEstimatedDrawsDistribute the work evenly among availabletask graphWorker threads. Each task will be
            //    based onFVisibleMeshDrawCommandProcessTaskAs a result, its working range is adjusted.
            constint32 NumThreads = FMath::Min<int32>
            (FTaskGraphInterface::Get().GetNumWorkerThreads(),       ParallelCommandListSet->Width);
            constint32 NumTasks = FMath::Min<int32>(NumThreads, FMath::DivideAndRoundUp(MaxNumDraws,
ParallelCommandListSet->MinDrawsPerCommandList));
            constint32 NumDrawsPerTask = FMath::DivideAndRoundUp(MaxNumDraws, NumTasks);

            //EstablishNumTasksindivualFRHICommandList,Add toParallelCommandListSet. for
            (int32 TaskIndex =0; TaskIndex < NumTasks; TaskIndex++) {

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

                //Create a new command queue.
                FRHICommandList* CmdList = ParallelCommandListSet->NewParallelCommandList();

                //Create a taskFDrawVisibleMeshCommandsAnyThreadTask,Get the event object.
                FGraphEventRef AnyThreadCompletionEvent =
TGraphTask<FDrawVisibleMeshCommandsAnyThreadTask>::CreateTask(&Prereqs, RenderThread)
                    . ConstructAndDispatchWhenReady(*CmdList, TaskContext.MeshDrawCommands,
TaskContext.MinimalPipelineStatePassSet, PrimitiveIdsBuffer, BasePrimitiveIdsOffset,
```

```
TaskContext.bDynamicInstancing, TaskContext.InstanceFactor, TaskIndex, NumTasks);
                    //Add command/event and other data toParallelCommandListSet.
                    ParallelCommandListSet->AddParallelCommandList(CmdList,
AnyThreadCompletionEvent, NumDraws);
            }
        }
        else//Non-parallel mode.
        {
            (……)
        }
}
```

From the above, we can know that after FParallelMeshDrawCommandPass::DispatchDraw is called, several FRHICommandList, FDrawVisibleMeshCommandsAnyThreadTask tasks and task synchronization events are created, and then all of them are added to the list of ParallelCommandListSet. In this way, when ParallelCommandListSet is destroyed, the command queue can be truly dispatched.

10.4.3.2 QueueParallelAsyncCommandListSubmit

After calling FParallelCommandListSet::Dispatch in the previous section, you will enter the interface of FRHICommandListBase::QueueParallelAsyncCommandListSubmit:

```
void FRHICommandListBase::QueueParallelAsyncCommandListSubmit(FGraphEventRef*
AnyThreadCompletionEvents,boolbIsPrepass, FRHICommandList** CmdLists, int32* NumDrawsIfKnown,
int32 Num, int32 MinDrawsPerTranslate,boolbSpewMerge) {

    if(IsRunningRHIInSeparateThread()) {

        //Execute all commands queued on the immediate command list before submitting
        the sublist for parallel build. FRHICommandListImmediate&
ImmediateCommandList = FRHICommandListExecutor::GetImmediateCommandList();
        ImmediateCommandList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

        //Clear the fence.
        if(RHIThreadBufferLockFence.GetReference() && RHIThreadBufferLockFence-
> IsComplete())
        {
            RHIThreadBufferLockFence = nullptr;
        }
    }

#if  !UE_BUILD_SHIPPING
    //  Refresh the command before processing, so that you can know what this parallel set broke, or what was
    if  there before. (CVarRHICmdFlushOnQueueParallelSubmit.GetValueOnRenderThread())
    {
        CSV_SCOPED_TIMING_STAT(RHITFlushes, QueueParallelAsyncCommandListSubmit);

FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::Flu shRHIThread);

    }
#endif

    //Make sure it is turned onRHIThreads.
    if(Num && IsRunningRHIInSeparateThread())
```

```
    {
        static constautoICVarRHICmdBalanceParallelLists =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.RHICmdBalanceParallelLists"));

        // r.RHICmdBalanceParallelLists==0andGRHISupportsParallelRHIExecute==trueAnd use the extension
Late context.
        //Unbalanced command queue submission mode.
        if(ICVarRHICmdBalanceParallelLists->GetValueOnRenderThread() ==0&&
CVarRHICmdBalanceTranslatesAfterTasks.GetValueOnRenderThread() >0&& GRHISupportsParallelRHIExecute &&
CVarRHICmdUseDeferredContexts.GetValueOnAnyThread() >0)
        {
            //Process the preceding tasks.
            FGraphEventArray Prereq;
            FRHICommandListBase** RHICmdLists =
(FRHICommandListBase**)Alloc(sizeof(FRHICommandListBase*) * Num, alignof
(FRHICommandListBase*));
            for(int32 Index =0; Index < Num; Index++) {

                FGraphEventRef& AnyThreadCompletionEvent =
AnyThreadCompletionEvents[Index];
                FRHICommandList* CmdList = CmdLists[Index];
                RHICmdLists[Index] = CmdList;
                if(AnyThreadCompletionEvent.GetReference()) {

                    Prereq.Add(AnyThreadCompletionEvent);
                    WaitOutstandingTasks.Add(AnyThreadCompletionEvent);
                }
            }

            //  Ensure that all old buffer locks are completed before starting any parallel
            if  translations. (RHIThreadBufferLockFence.GetReference())
            {
                Prereq.Add(RHIThreadBufferLockFence);
            }

            //NewFRHICommandList.
            FRHICommandList* CmdList = new FRHICommandList(GetGPUMask()); //Copy the
            render thread context.
            CmdList->CopyRenderThreadContexts(*this);
            //Create a translation task (FParallelTranslateSetupCommandList).
            FGraphEventRef TranslateSetupCompletionEvent =
TGraphTask<FParallelTranslateSetupCommandList>::CreateTask(&Prereq,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(CmdList, Num,          &RHICmdLists[0],
bIsPrepass);
            //Enqueue command queue submission.
            QueueCommandListSubmit(CmdList); //Add
            set translation event to the list.
            AllOutstandingTasks.Add(TranslateSetupCompletionEvent); //
                Avoid things after the async command list from being bound to it.
            if  (IsRunningRHIInSeparateThread())
            {

FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::Dis
patchToRHIThread);
            }
            //Refresh command toRHIThreads.
#if!UE_BUILD_SHIPPING
            if(CVarRHICmdFlushOnQueueParallelSubmit.GetValueOnRenderThread())
```

```
                {

FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::Flu shRHIThread);

                }
#endif
                return;
        }

        //Balanced command queue submission mode.
        IRHICommandContextContainer* ContextContainer = nullptr;
        boolbMerge = !!CVarRHICmdMergeSmallDeferredContexts.GetValueOnRenderThread(); int32
        EffectiveThreads =0; int32 Start =0; int32 ThreadIndex =0;


        if(GRHISupportsParallelRHIExecute &&
CVarRHICmdUseDeferredContexts.GetValueOnAnyThread() >0)
        {
                //Since the number of jobs needs to be known in advance, the merge logic is run twice. (Can
                be improved) while(Start < Num) {

                        int32 Last = Start;
                        int32 DrawCnt = NumDrawsIfKnown[Start];

                        if(bMerge && DrawCnt >=0) {

                                while(Last < Num -1&& NumDrawsIfKnown[Last +1] >=0&& DrawCnt +
NumDrawsIfKnown[Last +1] <= MinDrawsPerTranslate)
                                {
                                        Last++;
                                        DrawCnt += NumDrawsIfKnown[Last];
                                }
                        }
                        check(Last >= Start);
                        Start = Last +1;
                        EffectiveThreads++;
                }

                Start =0;
                ContextContainer = RHIGetCommandContextContainer(ThreadIndex,
EffectiveThreads, GetGPUMask());
        }

        if(ContextContainer)
        {
                //Another merge operation.
                while(Start < Num) {

                        int32 Last = Start;
                        int32 DrawCnt = NumDrawsIfKnown[Start];
                        int32 TotalMem = bSpewMerge ? CmdLists[Start]->GetUsedMemory():0;

                        if(bMerge && DrawCnt >=0) {

                                while(Last < Num -1&& NumDrawsIfKnown[Last +1] >=0&& DrawCnt +
NumDrawsIfKnown[Last +1] <= MinDrawsPerTranslate)
                                {
                                        Last++;
```

```
                                DrawCnt += NumDrawsIfKnown[Last];
                                TotalMem += bSpewMerge ? CmdLists[Start]->GetUsedMemory() :0;
                            }
                        }

                    //The following logic is similar to the unbalanced mode and is omitted.

                    (......)

                    return;
                }
            }

            //Non-parallel mode.
            (......)
}
```

From the above, we can see that the following conditions must be met to enable parallel command queue submission:

- The RHI thread is turned on, that is, IsRunningRHIInSeparateThread() is true.
- The graphics API currently used supports parallel execution, that is, GRHISupportsParallelRHIExecute must be true.
- The deferred context is enabled, that is, CVarRHICmdUseDeferredContexts is not 0.

Regardless of which graphics API you use, you need to specify a main CommandList (ParentCommandList) so that you can call its QueueParallelAsyncCommandListSubmit to submit the task of setting up the command queue. The task object submitted to the RHI thread above is FParallelTranslateSetupCommandList, which will be explained in the next section.

10.4.3.3 FParallelTranslateSetupCommandList

FParallelTranslateSetupCommandList is used to establish the task of submitting sub-command queues in parallel (or serially), and is defined as follows:

```
class    FParallelTranslateSetupCommandList
{
        //    A parent command list used to submit a subcommand list.
        FRHICommandList*     RHICmdList;
        //A queue of subcommands to be submitted.
        FRHICommandListBase** RHICmdLists;

        int32 NumCommandLists;
        bool bIsPrepass;
        int32 MinSize;
        int32 MinCount;

public:
        FParallelTranslateSetupCommandList(FRHICommandList* InRHICmdList, FRHICommandListBase**
InRHICmdLists, int32 InNumCommandLists,boolbInIsPrepass)
                : RHICmdList(InRHICmdList) ,
                RHICmdLists(InRHICmdLists)
                , NumCommandLists(InNumCommandLists)
                , bIsPrepass(bInIsPrepass)
```

```
	{
		//The minimum size of a single subcommand queue.
		MinSize = CVarRHICmdMinCmdlistSizeForParallelTranslate.GetValueOnRenderThread() *
1024;
		MinCount = CVarRHICmdMinCmdlistForParallelTranslate.GetValueOnRenderThread();
	}

	staticFORCEINLINE TStatIdGetStatId(); //Expected
	thread.
	staticFORCEINLINE ENamedThreads::TypeGetDesiredThread() {

		returnCPrio_FParallelTranslateSetupCommandList.Get();
	}
	staticFORCEINLINE ESubsequentsMode::TypeGetSubsequentsMode() {return
	ESubsequentsMode::TrackSubsequents; }

	//Perform setup tasks.
	void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
	MyCompletionGraphEvent)
	{
		TArray<int32, TInlineAllocator<64> > Sizes;
		Sizes.Reserve(NumCommandLists);
		for(int32 Index =0; Index < NumCommandLists; Index++) {

			Sizes.Add(RHICmdLists[Index]->GetUsedMemory());
		}

		int32 EffectiveThreads =0; int32 Start
		=0;
		//Merge drawing instructions, calculating the number of threads required.
		while(Start < NumCommandLists) {

			int32 Last = Start; int32 DrawCnt =
			Sizes[Start];

			while(Last < NumCommandLists -1&& DrawCnt + Sizes[Last +1] <= MinSize) {

				Last++;
				DrawCnt  + = Sizes[Last];
			}
			check(Last  > = Start);
			Start = Last +1;
			EffectiveThreads++;
		}

		//If the number of threads required is too small, the subcommand queues are
		submitted serially. if(EffectiveThreads < MinCount) {

			FGraphEventRef Nothing;
			for(int32 Index =0; Index < NumCommandLists; Index++) {

				FRHICommandListBase* CmdList = RHICmdLists[Index]; //Used
				ALLOC_COMMAND_CLAllocate a subcommand queue submission interface.
				ALLOC_COMMAND_CL(*RHICmdList,  FRHICommandWaitForAndSubmitSubList)(Nothing,
CmdList);
#ifWITH_MGPU
				ALLOC_COMMAND_CL(*RHICmdList,  FRHICommandSetGPUMask)(RHICmdList-
> GetGPUMask());
```

```
#endif
                }
            }
            //  Parallel submission.
            else
            {
                Start =  0;
                int32 ThreadIndex =0;

                //Merge too few command queues.
                while(Start < NumCommandLists) {

                    int32 Last = Start; int32 DrawCnt =
                    Sizes[Start];

                    while(Last < NumCommandLists -1&& DrawCnt + Sizes[Last +1] <= MinSize) {

                        Last++;
                        DrawCnt += Sizes[Last];
                    }

                    //GetContextContainer
                    IRHICommandContextContainer* ContextContainer =
RHIGetCommandContextContainer(ThreadIndex, EffectiveThreads, RHICmdList->GetGPUMask());

                    //Creating parallel translation tasksFParallelTranslateCommandList.
                    FGraphEventRef TranslateCompletionEvent =
TGraphTask<FParallelTranslateCommandList>::CreateTask(nullptr,
ENamedThreads::GetRenderThread()).ConstructAndDispatchWhenReady(&RHICmdLists[Start], Last - Start, 1 +
ContextContainer, bIsPrepass);

                    //Before this task ends, ensure that the translation task is completed.
                    MyCompletionGraphEvent->DontCompleteUntil(TranslateCompletionEvent);  //CallRHICmdListof
                    FRHICommandWaitForAndSubmitSubListParallelinterface.  ALLOC_COMMAND_CL(*RHICmdList,
                    FRHICommandWaitForAndSubmitSubListParallel)
(TranslateCompletionEvent, ContextContainer, EffectiveThreads, ThreadIndex++);
                    Start = Last +1;
                }
                check(EffectiveThreads == ThreadIndex);
            }
        }
};
```

In the above code, we can add a few points:

- If the number of commands is too small and the number of threads required is too small, directly use the serial translation interface FRHICommandWaitForAndSubmitSubList.
- In the parallel logic branch, RHIGetCommandContextContainer obtains the context container from the specific RHI subclass, which is only implemented in modern graphics platforms such as D3D12, Vulkan, and Metal. Other graphics platforms all return nullptr. Each thread will submit 1 to N sub-command queues to ensure that the total number of their
- drawing commands is not less than MinSize, thereby improving the submission efficiency of each thread.

- Each thread will create a translation task FParallelTranslateCommandList, and then use FRHICommandWaitForAndSubmitSubListParallel of RHICmdList to wait for the parallel submission of the sub-command list.
- Note that the intended thread of FParallelTranslateSetupCommandList is determined by CPrio_FParallelTranslateSetupCommandList:

```
FAutoConsoleTaskPriority CPrio_FParallelTranslateSetupCommandList
    //The name of the console.
    TEXT("TaskGraph.TaskPriorities.ParallelTranslateSetupCommandList"), //describe.

    TEXT("Task and thread priority for FParallelTranslateSetupCommandList."), //If there is a higher priority thread, use it.
    ENamedThreads::HighThreadPriority, //Use high task priority.
    ENamedThreads::HighTaskPriority,
    //If there is no high-priority thread, a normal-priority thread is used but with a high task priority instead. ENamedThreads::HighTaskPriority );
```

Therefore, it can be seen that the task of setting translation will be executed first by the TaskGraph system, but the thread that initiates the task of setting translation is still the rendering thread rather than the RHI thread.

10.4.3.4 FParallelTranslateCommandList

FParallelTranslateCommandList is the real translation command queue, and its definition is as follows:

```
class FParallelTranslateCommandList
{
    // List of commands to be translated.
    FRHICommandListBase** RHICmdLists;
    //The number of command lists to translate.
    int32 NumCommandLists;
    //The context container.
    IRHICommandContextContainer* ContextContainer;
    //Whether to advance depthpass.
    boolbIsPrepass;

public:
    FParallelTranslateCommandList(FRHICommandListBase** InRHICmdLists, int32 InNumCommandLists, IRHICommandContextContainer* InContextContainer,boolbInIsPrepass)
        : RHICmdLists(InRHICmdLists)
        , NumCommandLists(InNumCommandLists)
        , ContextContainer(InContextContainer)
        , bIsPrepass(bInIsPrepass)
    {
        check(RHICmdLists && ContextContainer && NumCommandLists);
    }

    staticFORCEINLINE TStatIdGetStatId();

    //The expected thread, depending on whetherPrepassIt depends.
```

```
    ENamedThreads::TypeGetDesiredThread() {

        returnbIsPrepass ? CPrio_FParallelTranslateCommandListPrepass.Get() :
        CPrio_FParallelTranslateCommandList.Get();
    }

    staticESubsequentsMode::TypeGetSubsequentsMode() {return
    ESubsequentsMode::TrackSubsequents; }

    //Execute the task.
    void DoTask(ENamedThreads::Type CurrentThread,constFGraphEventRef&
    MyCompletionGraphEvent)
    {
        IRHICommandContext* Context = ContextContainer->GetContext(); for(int32 Index
        =0; Index < NumCommandLists; Index++) {

            //Set the context for the subcommand queue.
            RHICmdLists[Index]->SetContext(Context); //Delete a
            subcommand queue.
            delete RHICmdLists[Index];
        }
        //Clean up the context.
        ContextContainer->FinishContext();
    }
};
```

The above code needs a few additional comments:

- GetDesiredThread is determined by two console traversals based on whether to prepass:

```
FAutoConsoleTaskPriorityCPrio_FParallelTranslateCommandListPrepass(
    TEXT("TaskGraph.TaskPriorities.ParallelTranslateCommandListPrepass"), TEXT("Task and thread
    priority for FParallelTranslateCommandList for the prepass, which we would like to get to the GPU
asap."),
    ENamedThreads::NormalThreadPriority,
    ENamedThreads::HighTaskPriority );


FAutoConsoleTaskPriorityCPrio_FParallelTranslateCommandList(
    TEXT("TaskGraph.TaskPriorities.ParallelTranslateCommandList"), TEXT("Task and thread
    priority for FParallelTranslateCommandList."), ENamedThreads::NormalThreadPriority,

    ENamedThreads::NormalTaskPriority );
```

  From this we can see that if it is a prepass, a normal priority thread but a high task priority is used, and other passes use normal priority threads and normal task priorities.

- The logic of DoTask is very simple. It sets the context for the command queue, deletes the command queue, and cleans up the context. But there is a question here: where is the translation task executed? After several inquiries, it is found in the destructor of FRHICommandListBase. The call stack is as follows:

```
FRHICommandListBase::~FRHICommandListBase() {

	//Refresh the command list.
	Flush();
	GRHICommandList.OutstandingCmdListCount.Decrement();
}

void FRHICommandListBase::Flush() {

	//  If the command exists.
	if  (HasCommands())
	{
		check(!IsImmediate());
		//Execute it using the global command list.GRHICommandListThe type is
		FRHICommandListExecutor. GRHICommandList.ExecuteList(*this);
	}
}

void FRHICommandListExecutor::ExecuteList(FRHICommandListBase& CmdList) {

	if(IsInRenderingThread() && !GetImmediateCommandList().IsExecuting()) {


GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
	}

	ExecuteInner(CmdList);
}

void FRHICommandListExecutor::ExecuteInner(FRHICommandListBase& CmdList) {

	(.....)
}
```

AtFRHICommandListExecutor::ExecuteInnerthis point, it is handed over to FRHICommandListExecutor for processing. For the specific process and analysis, see**10.4.1 RHI command execution**.

However, it is emphasized again that the graphics API needs to support parallel submission and translation to enable true parallel rendering. Otherwise, it can only be executed on the rendering thread as an ordinary task.

10.4.4 Pass Rendering

10.4.4.1 Normal Pass Rendering

The rendering of ordinary Pass involves the following interfaces and types:

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

//Render pass information.
structFRHIRenderPassInfo {
```

```
//Render texture information.
structFColorEntry
{
        FRHITexture*      RenderTarget;
        FRHITexture*      ResolveTarget;
        int32     ArraySlice;
        uint8 MipIndex;
        ERenderTargetActions          Action;
};
FColorEntry ColorRenderTargets[MaxSimultaneousRenderTargets];

//Depth stencil information.
structFDepthStencilEntry {

        FRHITexture*      DepthStencilTarget;
        FRHITexture*      ResolveTarget;
        EDepthStencilTargetActions FExclusivAecDtieopnt;hStencil
        ExclusiveDepthStencil;
};
FDepthStencilEntry DepthStencilRenderTarget;

//Parse the parameters.
FResolveParams ResolveParameters;

//partRHITextures can be used to control sampling and/or shadow resolution in
different areas FTextureRHIRef FoveationTexture = nullptr;

//partRHIA hint is needed that occlusion queries will be used in
this rendering pass uint32 NumOcclusionQueries =0; bool
bOcclusionQueries =false;

//partRHIYou need to know that when converting and generating for some resourcesmipIn case of mapping, whether this render pass will read and write to
the same texture. boolbGeneratingMips =false;

//If thisrenderpassIf it should be multi-view, how many views are
needed. uint8 MultiViewCount =0;

//partRHIA hint that a render pass will have specific subpasses.
ESubpassHint SubpassHint = ESubpassHint::None;

//Is it too much?UAV.
boolbTooManyUAVs=false; bool
bIsMSAA =false;

//Different constructors.

// Color, no depth, optional resolve, optional mip, optional array slice explicitFRHIRenderPassInfo(FRHITexture*
ColorRT, ERenderTargetActions ColorAction, FRHITexture* ResolveRT = nullptr, uint32 InMipIndex =0, int32
InArraySlice =-1);
// Color MRTs, no depth
explicitFRHIRenderPassInfo(int32 NumColorRTs, FRHITexture* ColorRTs[],
ERenderTargetActions ColorAction);
// Color MRTs, no depth
explicitFRHIRenderPassInfo(int32 NumColorRTs, FRHITexture* ColorRTs[],
ERenderTargetActions ColorAction, FRHITexture* ResolveTargets[]);
// Color MRTs and depth
explicitFRHIRenderPassInfo(int32 NumColorRTs, FRHITexture* ColorRTs[], ERenderTargetActions
ColorAction, FRHITexture* DepthRT, EDepthStencilTargetActions
```

```
DepthActions, FExclusiveDepthStencil InEDS =
FExclusiveDepthStencil::DepthWrite_StencilWrite);
        // Color MRTs and depth
        explicitFRHIRenderPassInfo(int32 NumColorRTs, FRHITexture* ColorRTs[], ERenderTargetActions
ColorAction, FRHITexture* ResolveRTs[], FRHITexture* DepthRT, EDepthStencilTargetActions DepthActions,
FRHITexture* ResolveDepthRT, FExclusiveDepthStencil InEDS = FExclusiveDepthStencil::DepthWrite
_StencilWrite);
        // Depth, no color
        explicitFRHIRenderPassInfo(FRHITexture* DepthRT, EDepthStencilTargetActions DepthActions,
FRHITexture* ResolveDepthRT = nullptr, FExclusiveDepthStencil InEDS =
FExclusiveDepthStencil::DepthWrite_StencilWrite);
        // Depth, no color, occlusion queries
        explicitFRHIRenderPassInfo(FRHITexture* DepthRT, uint32 InNumOcclusionQueries,
EDepthStencilTargetActions DepthActions, FRHITexture* ResolveDepthRT = nullptr, FExclusiveDepthStencil
InEDS = FExclusiveDepthStencil::DepthWrite_StencilWrite);
        // Color and depth
        explicitFRHIRenderPassInfo(FRHITexture* ColorRT, ERenderTargetActions ColorAction, FRHITexture* DepthRT,
EDepthStencilTargetActions DepthActions, FExclusiveDepthStencil InEDS =
FExclusiveDepthStencil::DepthWrite_StencilWrite);
        // Color and depth with resolve
        explicitFRHIRenderPassInfo(FRHITexture* ColorRT, ERenderTargetActions ColorAction, FRHITexture*
        ResolveColorRT,
                FRHITexture* DepthRT, EDepthStencilTargetActions DepthActions, FRHITexture* ResolveDepthRT,
FExclusiveDepthStencil InEDS =
FExclusiveDepthStencil::DepthWrite_StencilWrite);
        // Color and depth with resolve and optional sample density
        explicitFRHIRenderPassInfo(FRHITexture* ColorRT, ERenderTargetActions ColorAction, FRHITexture*
        ResolveColorRT,
                FRHITexture* DepthRT, EDepthStencilTargetActions DepthActions, FRHITexture* ResolveDepthRT,
FRHITexture* InFoveationTexture, FExclusiveDepthStencil InEDS =
FExclusiveDepthStencil::DepthWrite_StencilWrite);

        enumENoRenderTargets
        {
                NoRenderTargets,
        };
        explicit        FRHIRenderPassInfo(ENoRenderTargets             Dummy);
        explicit        FRHIRenderPassInfo();

        inlineint32GetNumColorRenderTargets()const; RHI_APIvoid
        Validate()const;
        RHI_APIvoid ConvertToRenderTargetsInfo(FRHISetRenderTargetsInfo& OutRTInfo)const;

        (......)
};

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

classRHI_API FRHICommandList: public FRHIComputeCommandList {

public:
        void BeginRenderPass(constFRHIRenderPassInfo& InInfo,constTCHAR* Name) {

                if(InInfo.bTooManyUAVs) {

                        UE_LOG(LogRHI, Warning, TEXT("RenderPass %s has too many UAVs"));
                }
                InInfo.Validate();
```

```
        //  Direct callRHIThe interface.
        if  (Bypass())
        {
            GetContext().RHIBeginRenderPass(InInfo,                Name);
        }
        //  distributeRHIOrder.
        else
        {
            TCHAR* NameCopy = AllocString(Name);
            ALLOC_COMMAND(FRHICommandBeginRenderPass)(InInfo,          NameCopy);
        }
        //Set inRenderPassInternal
        mark. Data.bInsideRenderPass =true;

        //Cache activeRT.
        CacheActiveRenderTargets(InInfo); //Reset
        subPass.
        ResetSubpass(InInfo.SubpassHint);
        Data.bInsideRenderPass =true;
    }

    void EndRenderPass()
    {
        //  Call or assignRHIinterface.
        if  (Bypass())
        {
            GetContext().RHIEndRenderPass();
        }
        else
        {
            ALLOC_COMMAND(FRHICommandEndRenderPass)();
        }
        //Reset atRenderPassInternal
        mark. Data.bInsideRenderPass=false;
        //Reset subPassMark asNone.
        ResetSubpass(ESubpassHint::None);
    }
};
```

Their use cases are as follows:

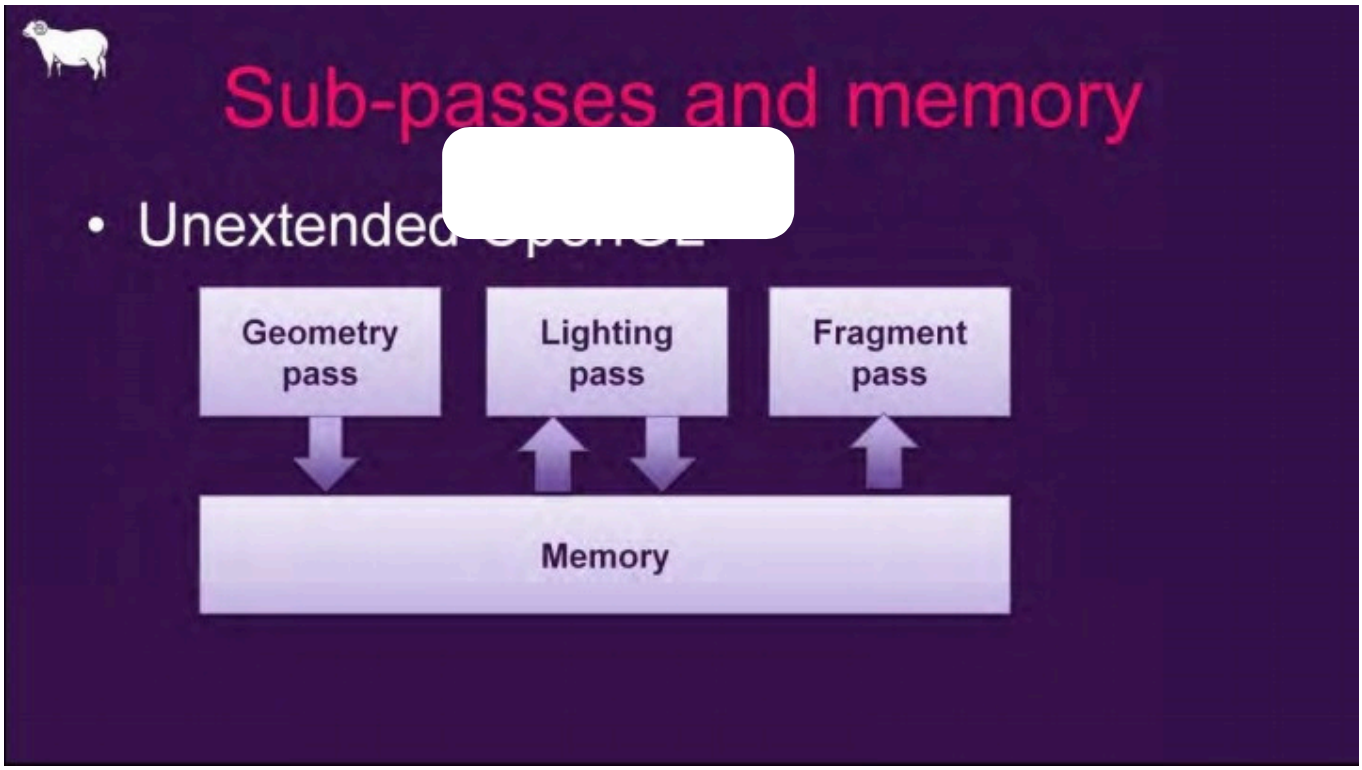
```
void FSceneRenderer::RenderShadowDepthMaps(FRHICommandListImmediate& RHICmdList) {

    (.....)

    for(int32 AtlasIndex =0; AtlasIndex <
    SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases.Num();          AtlasIndex++)
    {
        constFSortedShadowMapAtlas& ShadowMapAtlas =
        SortedShadowsForShadowDepthPass.TranslucencyShadowMapAtlases[AtlasIndex];
        FIntPoint TargetSize = ShadowMapAtlas.RenderTargets.ColorTargets[0]-
> GetDesc().Extent;

        FSceneRenderTargetItem ColorTarget0 =
ShadowMapAtlas.RenderTargets.ColorTargets[0]->GetRenderTargetItem();
        FSceneRenderTargetItem ColorTarget1 =
```

```
ShadowMapAtlas.RenderTargets.ColorTargets[1]->GetRenderTargetItem();

        FRHITexture* RenderTargetArray[2] =
        {
            ColorTarget0.TargetableTexture,
            ColorTarget1.TargetableTexture
        };

        //createFRHIRenderPassInfoExamples.
        FRHIRenderPassInfoRPInfo(UE_ARRAY_COUNT(RenderTargetArray),                RenderTargetArray,
        ERenderTargetActions::Load_Store);
        TransitionRenderPassTargets(RHICmdList, RPInfo); //Start
        RenderingPass.
        RHICmdList.BeginRenderPass(RPInfo, TEXT("RenderTranslucencyDepths")); {

            //Render shadows.
            for(int32 ShadowIndex =0; ShadowIndex < ShadowMapAtlas.Shadows.Num();
ShadowIndex++)
            {
                FProjectedShadowInfo* ProjectedShadowInfo =
ShadowMapAtlas.Shadows[ShadowIndex];
                ProjectedShadowInfo->SetupShadowUniformBuffers(RHICmdList,          Scene);
                ProjectedShadowInfo->RenderTranslucencyDepths(RHICmdList,          this);
            }
        }
        //End RenderingPass.
        RHICmdList.EndRenderPass();

        RHICmdList.Transition(FRHITransitionInfo(ColorTarget0.TargetableTexture,
ERHIAccess::Unknown, ERHIAccess::SRVMask));
        RHICmdList.Transition(FRHITransitionInfo(ColorTarget1.TargetableTexture,
ERHIAccess::Unknown, ERHIAccess::SRVMask));
    }

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

10.4.4.2 Subpass Rendering

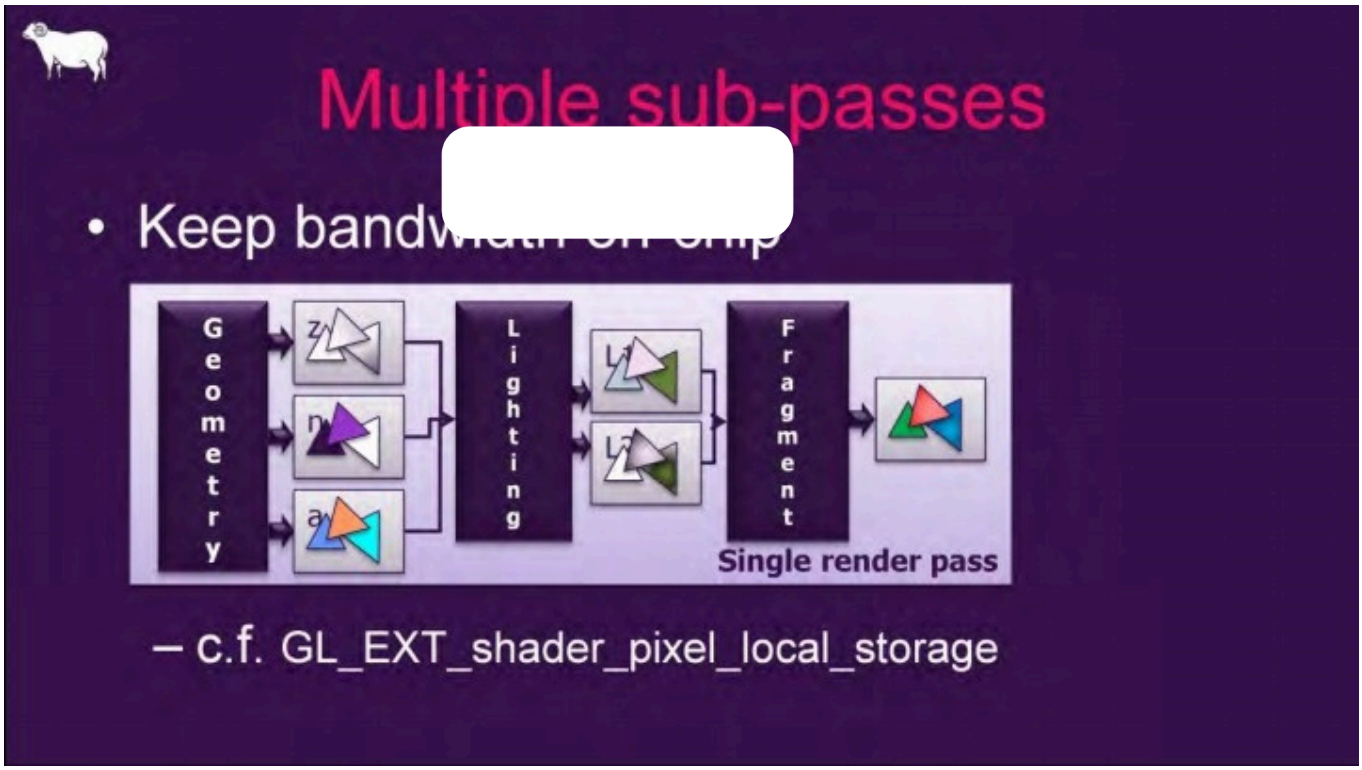
Let's first talk about the origin, function and characteristics of Subpass.

In traditional multi-pass rendering, a set of rendering textures is usually rendered at the end of each Pass, and some of them become shader parameters for the next Pass to sample and read. This texture sampling method is not subject to any restrictions, and can read any field pixels and use any texture filtering method. Although this method is flexible to use, it will have a large consumption in devices with TBR (Tile-Based Renderer) hardware architecture: the Pass that renders the texture usually stores the rendering results in the On-chip Tile Memory, and writes them back to the GPU video memory (VRAM) after the Pass ends. Writing back to the GPU video memory is a time-consuming and power-consuming operation.



The traditional multi-pass memory access model occurs multiple times between On-Chip and global memory.

If a special texture usage occurs: the texture rendered by the previous Pass is immediately used by the next Pass, and the next Pass only samples the data of the pixel position itself, without sampling the position of the neighboring pixels. This situation meets the use scenario of Subpass. The texture results rendered by Subpass will only be stored in Tile Memory, and will not be written back to VRAM after the Subpass ends, but the data of Tile Memory will be directly provided to the next Subpass for sampling and reading. This avoids the time-consuming and power-consuming operation of writing back to GPU memory at the end of the traditional Pass and reading data from GPU memory in the next Pass, thereby improving performance.



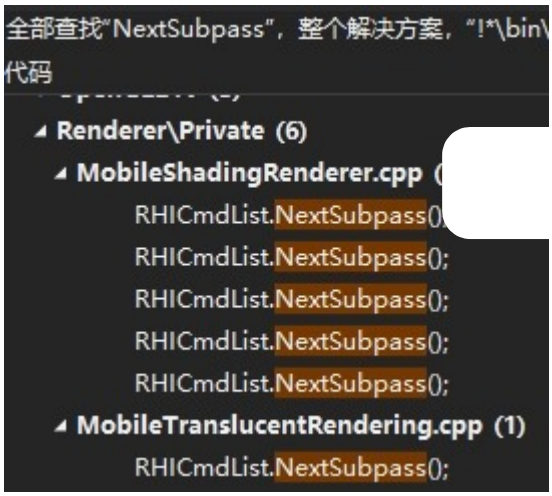
The memory access model between subpasses all occurs on-chip.

The interfaces and types of UE involving Subpass are as follows:

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

//Provide toRHIofSubpassmark. enum
classESubpassHint: uint8 {

	None,                       //  Traditional rendering (nonSubpass)
	DepthReadSubpass,           //  Depth readingSubpass.
	DeferredShadingSubpass,     //Deferred Shading on MobileSubpass.
};

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

classRHI_API FRHICommandListBase: public FNoncopyable {

	(.....)

protected:
	//PSOContext.
	structFPSOContext
	{
		uint32 CachedNumSimultanousRenderTargets =0;
		TStaticArray<FRHIRenderTargetView, MaxSimultaneousRenderTargets>
		CachedRenderTargets;
		FRHIDepthRenderTargetView CachedDepthStencilTarget;

		// SubpassPrompt mark.
		ESubpassHint SubpassHint = ESubpassHint::None; uint8
		SubpassIndex =0; uint8 MultiViewCount =0;

		boolHasFragmentDensityAttachment =false; PSOContext;
	}
```

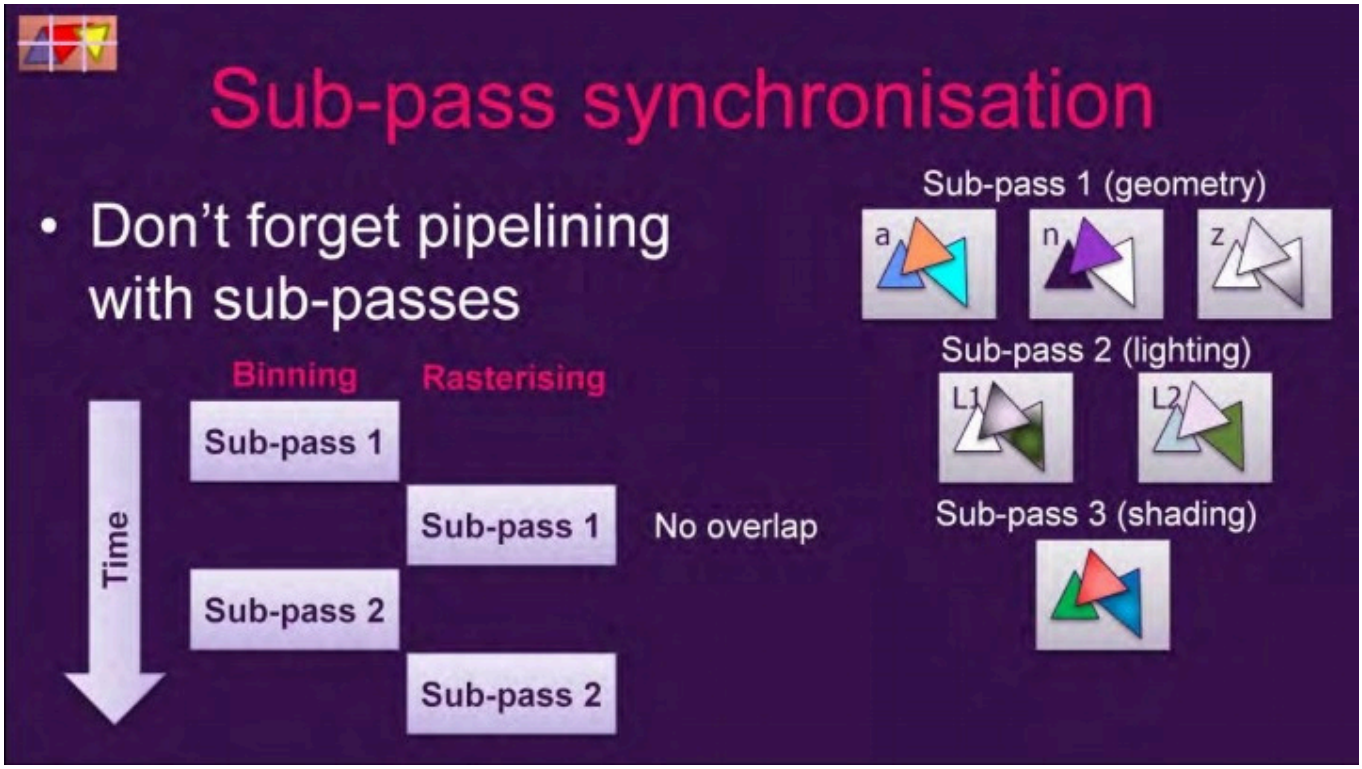
```
};

classRHI_API FRHICommandList: public FRHIComputeCommandList {

public:
	void BeginRenderPass(constFRHIRenderPassInfo& InInfo,constTCHAR* Name) {

		(.....)

		CacheActiveRenderTargets(InInfo); //set up
		Subpassdata.
		ResetSubpass(InInfo.SubpassHint);
		Data.bInsideRenderPass =true;
	}

	void EndRenderPass()
	{
		(.....)

		//ResetSubpassMark asNone.
		ResetSubpass(ESubpassHint::None);
	}

	//NextSubpass.
	void NextSubpass()
	{
		//   Assign or callRHIinterface.
		if   (Bypass())
		{
			GetContext().RHINextSubpass();
		}
		else
		{
			ALLOC_COMMAND(FRHICommandNextSubpass)();
		}

		//IncreaseSubpasscount.
		IncrementSubpass();
	}

	//Increasesubpasscount. void
	IncrementSubpass() {

		PSOContext.SubpassIndex++;
	}

	//ResetSubpassdata.
	void ResetSubpass(ESubpassHint {          SubpassHint)

		PSOContext.SubpassHint        =  SubpassHint;
		PSOContext.SubpassIndex       =  0;
	}
};
```

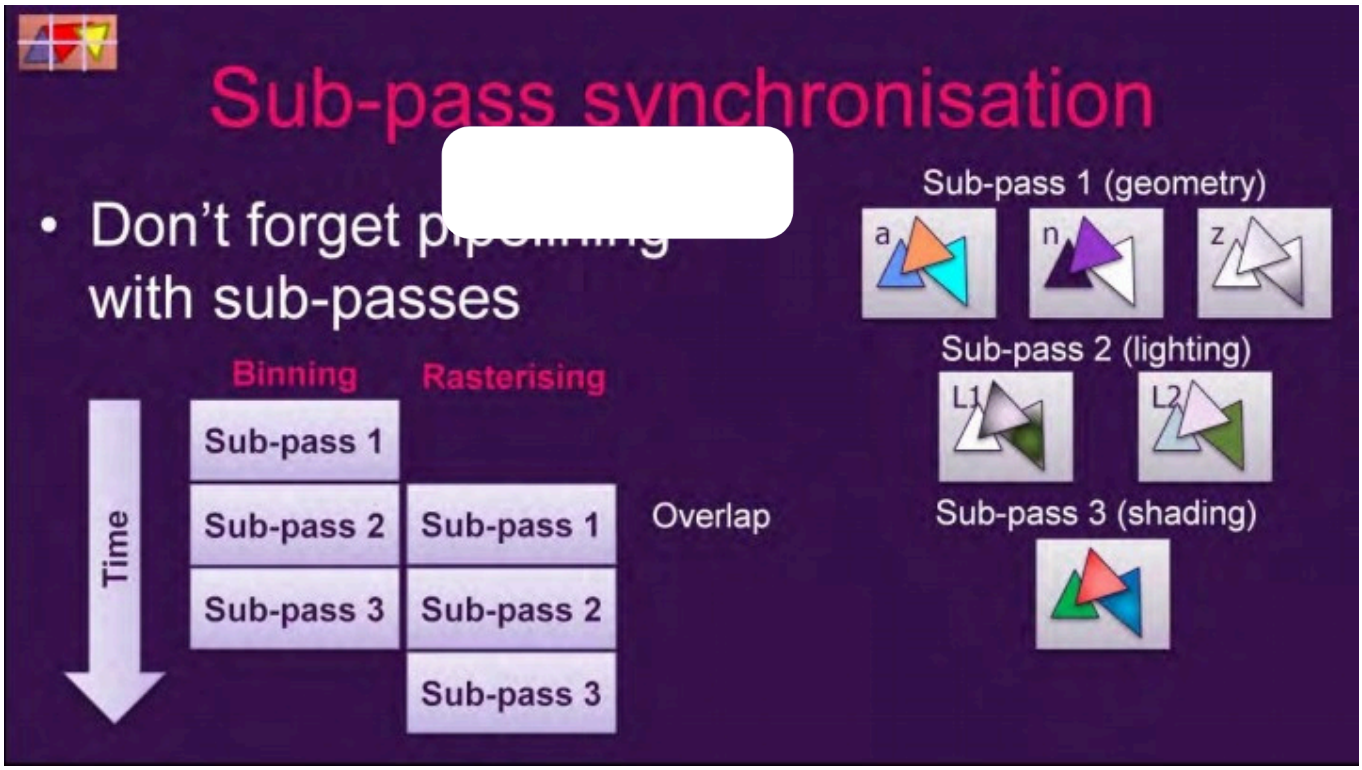
UE's Subpass is mainly focused on mobile renderers:



The reason is that there are more and more hardware devices with TBR architecture on mobile terminals, and their share is getting larger and larger. It is inevitable and reasonable for Subpass to become the first choice for the main renderer on mobile terminals.

In Subpass rendering, the Pass Overlap problem is still involved. Using Overlap can increase the utilization rate of the GPU and improve rendering performance (see the figure below).





Top: Subpass pipeline without Overlap technology; Bottom: Subpass pipeline with Overlap technology.

RHI's instructions on Overlap are mainly for UAVs:

```
classRHI_API FRHIComputeCommandList: public FRHICommandListBase {

	(.....)

	void BeginUAVOverlap() {

		if(Bypass())
		{
			GetContext().RHIBeginUAVOverlap(); return;

		}
		ALLOC_COMMAND(FRHICommandBeginUAVOverlap)();
	}

	void EndUAVOverlap()
	{
		if(Bypass())
		{
			GetContext().RHIEndUAVOverlap(); return;

		}
		ALLOC_COMMAND(FRHICommandEndUAVOverlap)();
	}

	void BeginUAVOverlap(FRHIUnorderedAccessView* UAV) {

		FRHIUnorderedAccessView* UAVs[1] = { UAV };
		BeginUAVOverlap(MakeArrayView(UAVs,1));
	}
```

```
    void EndUAVOverlap(FRHIUnorderedAccessView* UAV) {

        FRHIUnorderedAccessView* UAVs[1] = { UAV };
        EndUAVOverlap(MakeArrayView(UAVs,1));
    }

    void BeginUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs)

        if(Bypass())
        {
            GetContext().RHIBeginUAVOverlap(UAVs); return;

        }

        constuint32 AllocSize = UAVs.Num() *sizeof(FRHIUnorderedAccessView*); FRHIUnorderedAccessView**
        InlineUAVs = (FRHIUnorderedAccessView**)Alloc(AllocSize, alignof(FRHIUnorderedAccessView*));

        FMemory::Memcpy(InlineUAVs, ALLOCU_CAOVsM.GMeAtDNaDta(F(R), HAllCloocmSimzea);ndBeginSpecificUAVOverlap)
        (MakeArrayView(InlineUAVs, UAVs.Num()));

    }

    void EndUAVOverlap(TArrayView<FRHIUnorderedAccessView*const> UAVs)

        if(Bypass())
        {
            GetContext().RHIEndUAVOverlap(UAVs); return;

        }

        constuint32 AllocSize = UAVs.Num() *sizeof(FRHIUnorderedAccessView*); FRHIUnorderedAccessView**
        InlineUAVs = (FRHIUnorderedAccessView**)Alloc(AllocSize, alignof(FRHIUnorderedAccessView*));

        FMemory::Memcpy(InlineUAVs, ALLOCU_CAOVsM.GMeAtDNaDta(F(R), HAllCloocmSimzea);ndEndSpecificUAVOverlap)
        (MakeArrayView(InlineUAVs, UAVs.Num()));

    }
}
```

10.4.5 RHI Resource Management

10.2.2 FRHIResourcesection has described the basic interface of RHI resources. FRHIResource itself has reference counting and reference count increase and decrease interfaces:

```
classRHI_API FRHIResource {

public:
    //Increment the reference count.
    uint32AddRef()          const;
    //Decrement the reference count.
    uint32Release()
    //Get the reference count.      const;
    uint32GetRefCount()

                                    const;
};
```

Of course, we do not need to directly reference and manage the instance and count of FRHIResource, but combine the template class of TRefCountPtr to realize automatic management of RHI resources:

```
//VariousRHIResource reference type definition. typedef TRefCountPtr<FRHISamplerState> FSamplerStateRHIRef;
typedef TRefCountPtr<FRHIRasterizerState> FRasterizerStateRHIRef;
typedef TRefCountPtr<FRHIDepthStencilState> FDepthStencilStateRHIRef;
typedef
typedef TRefCountPtr<FRHIVertexDeclaration> FVertexDeclarationRHIRef;
typedef TRefCountPtr<FRHIVertexShader> FVertexShaderRHIRef;
typedef TRefCountPtr<FRHIHullShader> FHullShaderRHIRef;
typedef TRefCountPtr<FRHIDomainShader> TRefCoFDunotmPtari<nFSRhHadIGeerRoHmIeRterfy;Shader>
typedef TRefCountPtr<FRHIComputeShader> TReFfPCioxuenlSthPatrd<eFrRHIIRaeyf;TracingShader>
typedef TRefCountPtr<FRHIComputeFence> TRefCounFGtPetor<mFeRtHryIBShoaudnedrSRhHaIdReerfS;tate>
typedef TRefCountPtr<FRHIUniformBuffer> TRefCouFnCtoPmtrp<FuRteHSIhInadexrRBHuflfReerf>; TR
typedef efCountPtr<FRHIVertexBuffer> TRefCountPtr<FRFHRaISytTrruacctiunrgeSdhBaudfeferRr>H IRef;
typedef TRefCountPtr<FRHITexture> FTextureRHIRef; TRFeCfoCmoupnuttPetFre<nFcReHRIHTeIRxteuf;re2D>
typedef FTexture2DRHIRef; TRefCountPtr<FRHITexture2FDBAorurnayd>S haderStateRHIRef;
typedef FTexture2DArrayRHIRef; TRefCountPtr<FRHFITUenxiftourme3BDu>f fFeTreRxHtIuRref3;DRHIRef;
typedef TRefCountPtr<FRHITextureCube> FTextuFreInCduebxeBRuHffIeRreRf;H IRef;
typedef TRefCountPtr<FRHITextureReference> FTeFxVtuerteeRxeBfuerffeenrcReHRIHRIeRf;ef;
typedef TRefCountPtr<FRHIRenderQuery> FRenderQuerFySRtHruIcRteufr; edBufferRHIRef;
typedef TRefCountPtr<FRHIRenderQueryPool> FRenderQueryPoolRHIRef;
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
typedef
                                                    FTimestampCalibrationQueryRHIRef;
TRefCountPtr<FRHIGPUFence>          FGPUFenceRHIRef;
TRefCountPtr<FRHIViewport>        FViewportRHIRef;
TRefCountPtr<FRHIUnorderedAccessView>          FUnorderedAccessViewRHIRef;
TRefCountPtr<FRHIShaderResourceView>         FShaderResourceViewRHIRef;
TRefCountPtr<FRHIGraphicsPipelineState>            FGraphicsPipelineStateRHIRef;
TRefCountPtr<FRHIRayTracingPipelineState>              FRayTracingPipelineStateRHIRef;
```

After using the above types, the RHI resource is automatically managed by TRefCountPtr with reference counts, and the resource is released in FRHIResource::Release:

```
classRHI_API        FRHIResource
{
    uint32    Release()const
    {
        // count-1.
        int32 NewValue = NumRefs.Decrement(); //If the
        count is0,Handles resource deletion. if(NewValue
        ==0) {

            //   Non-delayed deletion, directdelete.
            if   (!DeferDelete())
            {
                delete this;
```

```
                }
                //Delayed deletion mode.
                else
                {

                        //Using platform-dependent atomic comparisons,      for0Then add it to the list to be deleted.
                        if(FPlatformAtomics::InterlockedCompareExchange(&MarkedForDelete,1,0)
==  0)
                        {
                                PendingDeletes.Push(const_cast<FRHIResource*>(this));
                        }
                }
        }

        //Returns the new value.
        returnuint32(NewValue);
    }

    bool DeferDelete()const {

        //Multithreaded rendering is enabled andGRHINeedsExtraDeletionLatencyfortrue,And the resource is not marked as
        not to delay deletion. return!bDoNotDeferDelete && (GRHINeedsExtraDeletionLatency || !Bypass());
    }
};
```

PendingDeletesIt is a static variable of FRHIResource. Its related data and interfaces are:

```
classRHI_API FRHIResource {

public:
    FRHIResource(boolInbDoNotDeferDelete =false)
        :MarkedForDelete(0)
        , bDoNotDeferDelete(InbDoNotDeferDelete)
        , bCommitted(true)
    { }
    virtual ~FRHIResource() {


        check(PlatformNeedsExtraDeletionLatency() || (NumRefs.GetValue() ==0&& (CurrentlyDeleting == this ||
bDoNotDeferDelete || Bypass())));// this should not have any
    outstanding refs
    }

    //The list of resources to be deleted. Note that it is a lock-free and unordered pointer list.
    staticTLockFreePointerListUnordered<FRHIResource,                PLATFORM_CACHE_LINE_SIZE>
    PendingDeletes;
    //The resource currently being deleted.
    staticFRHIResource* CurrentlyDeleting;

    //The platform requires an additional deletion delay.
    static bool PlatformNeedsExtraDeletionLatency() {

        returnGRHINeedsExtraDeletionLatency && GIsRHIInitialized;
    }

    //List of resources to be deleted.
    structResourcesToDelete {
```

```
        TArray<FRHIResource*>        Resources;
        uint32                       FrameDeleted;
    };

    //Delayed deletion of a queue.
    static    TArray<ResourcesToDelete>    DeferredDeletionQueue;
    static    uint32 CurrentFrame;
};

void FRHIResource::FlushPendingDeletes(boolbFlushDeferredDeletes) {

    FRHICommandListImmediate& RHICmdList =
    FRHICommandListExecutor::GetImmediateCommandList();

    //On DeletionRHIBefore you start, make sure the command list has been refreshed toGPU.
    RHICmdList.ImmediateFlush(EImmediateFlushType::FlushRHIThread); //Make sure there are no waiting
    tasks.
    FRHICommandListExecutor::CheckNoOutstandingCmdLists(); //
        notifyRHIRefresh completed.
    if  (GDynamicRHI)
    {
        GDynamicRHI->RHIPerFrameRHIFlushComplete();
    }

    //Remove anonymous functions.
    autoDelete = [](TArray<FRHIResource*>& ToDelete) {

        for(int32 Index =0; Index < ToDelete.Num(); Index++) {

            FRHIResource* Ref = ToDelete[Index]; check(Ref-
            >MarkedForDelete ==1);
            if(Ref->GetRefCount() ==0)// caches can bring dead objects back to life {

                CurrentlyDeleting       = Ref;
                delete Ref;
                CurrentlyDeleting       =  nullptr;
            }
            else
            {
                Ref->MarkedForDelete        =0;
                FPlatformMisc::MemoryBarrier();
            }
        }
    };

    while   (1)
    {
        if  (PendingDeletes.IsEmpty())
        {
            break;
        }

        //  The platform requires an additional deletion delay.
        if  (PlatformNeedsExtraDeletionLatency())
        {
            constint32 Index = DeferredDeletionQueue.AddDefaulted(); //Join the delayed
            deletion queueDeferredDeletionQueue.
            ResourcesToDelete& ResourceBatch = DeferredDeletionQueue[Index];
```

```
                ResourceBatch.FrameDeleted = CurrentFrame;
                PendingDeletes.PopAll(ResourceBatch.Resources);
            }
            //  Without additional delay, delete the entire list.
            else
            {
                TArray<FRHIResource*> ToDelete;
                PendingDeletes.PopAll(ToDelete);
                Delete(ToDelete);
            }
        }

        constuint32 NumFramesToExpire = RHIRESOURCE_NUM_FRAMES_TO_EXPIRE;

        //  deleteDeferredDeletionQueue.
        if  (DeferredDeletionQueue.Num())
        {
            //  Clear the entireDeferredDeletionQueue
            if  queue. (bFlushDeferredDeletes)
            {
                FRHICommandListExecutor::GetImmediateCommandList().BlockUntilGPUIdle();

                for(int32 Idx =0; Idx < DeferredDeletionQueue.Num(); ++Idx) {

                    ResourcesToDelete& ResourceBatch = DeferredDeletionQueue[Idx];
                    Delete(ResourceBatch.Resources);
                }

                DeferredDeletionQueue.Empty();
            }
            //Delete the list of expired resources.
            else
            {
                int32 DeletedBatchCount =0;
                while(DeletedBatchCount < DeferredDeletionQueue.Num()) {

                    ResourcesToDelete& ResourceBatch =
DeferredDeletionQueue[DeletedBatchCount];
                    if(((ResourceBatch.FrameDeleted + NumFramesToExpire) < CurrentFrame) ||
!GIsRHIInitialized)
                    {
                        Delete(ResourceBatch.Resources);
                        + + DeletedBatchCount;
                    }
                    else
                    {
                        break;
                    }
                }

                if(DeletedBatchCount)
                {
                    DeferredDeletionQueue.RemoveAt(0, DeletedBatchCount);
                }
            }

            + +CurrentFrame;
```

```
    }
}
```

However, it should be pointed out that the destructor of FRHIResource does not release any RHI resources, which usually needs to be executed in the graphics platform-related subclass destructor of FRHIResource, such as FD3D11UniformBuffer:

```
// Engine\Source\Runtime\Windows\D3D11RHI\Public\D3D11Resources.h

classFD3D11UniformBuffer: public FRHIUniformBuffer {

public:
    // D3D11Pinned buffer resources.
    TRefCountPtr<ID3D11Buffer> Resource; //IncludedRHIA table of
    referenced resources. TArray<TRefCountPtr<FRHIResource> >
    ResourceTable;

    FD3D11UniformBuffer(class FD3D11DynamicRHI* InD3D11RHI,constFRHIUniformBufferLayout& InLayout,
ID3D11Buffer* InResource,constFRingAllocation& InRingAllocation);
    virtual ~FD3D11UniformBuffer();

    (......)
};

// Engine\Source\Runtime\Windows\D3D11RHI\Private\D3D11UniformBuffer.cpp

FD3D11UniformBuffer::~FD3D11UniformBuffer() {

    if(!RingAllocation.IsValid() && Resource != nullptr) {

        D3D11_BUFFER_DESC Desc;
        Resource->GetDesc(&Desc);

        //Return this uniform buffer to the free pool.
        if(Desc.CPUAccessFlags == D3D11_CPU_ACCESS_WRITE && Desc.Usage ==
        D3D11_USAGE_DYNAMIC)
        {
            FPooledUniformBuffer NewEntry;
            NewEntry.Buffer = Resource;
            NewEntry.FrameFreed      =GFrameNumberRenderThread;
            NewEntry.CreatedSize      =Desc.ByteWidth;

            // Add to this frame's array of free uniform buffers
            constint32 SafeFrameIndex = (GFrameNumberRenderThread -1)% NumSafeFrames; constuint32
            BucketIndex = GetPoolBucketIndex(Desc.ByteWidth);
            int32 LastNum = SafeUniformBufferPools[SafeFrameIndex][BucketIndex].Num();
            SafeUniformBufferPools[SafeFrameIndex][BucketIndex].Add(NewEntry);

            FPlatformMisc::MemoryBarrier();// check for unwanted concurrency
        }
    }
}
```

The above analysis shows that the release of RHI resources is mainly in the FlushPendingDeletes interface, and the calls involved are:

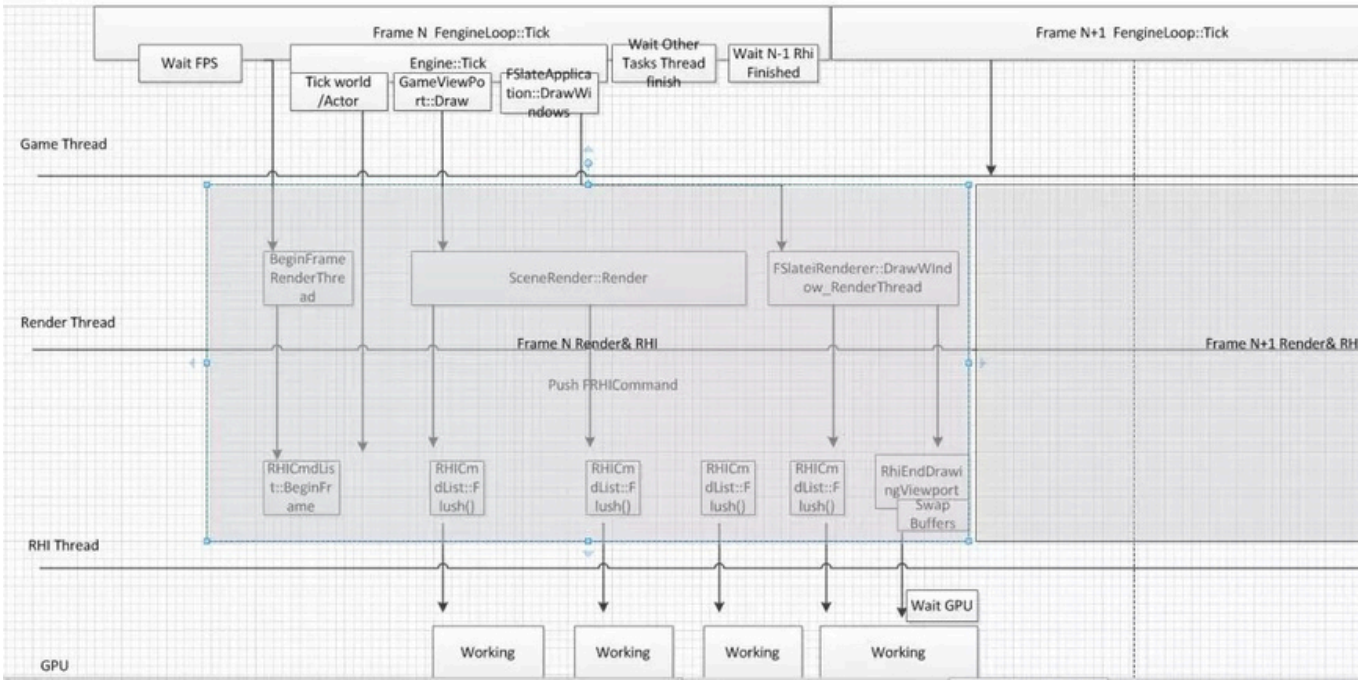
```
// Engine\Source\Runtime\RenderCore\Private\RenderingThread.cpp

void FlushPendingDeleteRHIResources_RenderThread() {

	if(!IsRunningRHIInSeparateThread()) {

		FRHIResource::FlushPendingDeletes();
	}
}

// Engine\Source\Runtime\RHI\Private\RHICommandList.cpp

void  FRHICommandListExecutor::LatchBypass()
{
#if  CAN_TOGGLE_COMMAND_LIST_BYPASS if
	(IsRunningRHIInSeparateThread()) {

		(......)
	}
	else
	{
		(......)

		if(NewBypass && !bLatchedBypass) {

			FRHIResource::FlushPendingDeletes();
		}
	}
#endif

	(......)
}

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

void FRHICommandListImmediate::ImmediateFlush(EImmediateFlushType::Type FlushType) {

	switch(FlushType)
	{
	(......)

	case  EImmediateFlushType::FlushRHIThreadFlushResources:
	case  EImmediateFlushType::FlushRHIThreadFlushResourcesFlushDeferredDeletes:
		{
			(......)

			PipelineStateCache::FlushResources();
			FRHIResource::FlushPendingDeletes(FlushType ==
EImmediateFlushType::FlushRHIThreadFlushResourcesFlushDeferredDeletes);
		}
		break;
	(......)
	}
}
```

The RHI abstraction layer mainly calls FlushPendingDeletes in the above places, but the following graphics platform-related interfaces are also called:

- FD3D12Adapter::Cleanup()
- FD3D12Device::Cleanup()
- FVulkanDevice::Destroy()
- FVulkanDynamicRHI::Shutdown()
- FD3D11DynamicRHI::CleanupD3DDevice()

10.4.6 Multithreaded Rendering

Analysis of Unreal Rendering System (02) - The chapter on multi-threaded rendering has already elaborated on the UE multi-threaded system and rendering mechanism. This section will provide some additional explanations based on the following figure.



In the rendering process of UE, there are up to 4 working threads: game thread, rendering thread, RHI thread and GPU (including driver).

The game thread is the driver of the entire engine, providing all source data and events to drive the rendering thread and RHI thread. The game thread is no more than 1 frame ahead of the rendering thread. More specifically, if the rendering thread of the Nth frame has not completed at the end of the Tick of the game thread of the N+1th frame, the game thread will be stuck by the rendering thread. Conversely, if the game thread is overloaded and fails to send events and data to the rendering thread in time, it will also cause the rendering thread to be stuck.

The rendering thread is responsible for generating intermediate commands of the RHI, dispatching and refreshing commands to the RHI thread at the appropriate time. Therefore, the lag of the rendering thread may also cause the lag of the RHI.

The RHI thread is responsible for dispatching (optional), translating, and submitting instructions, and the last step of rendering requires SwapBuffer, which requires waiting for the GPU to complete the rendering work. Therefore, the busyness of the rendering GPU will also cause the RHI thread to freeze.

In addition to the game thread, there are gaps in the work of the rendering thread, RHI thread and GPU. That is, the timing provided by the game thread to the rendering task will affect the density of the rendering work and the rendering time. Small amounts and multiple times will waste rendering efficiency.

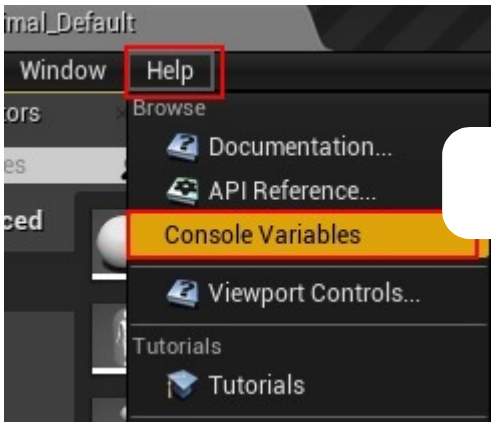
10.4.7 RHI Console Variables

The code in the previous section also shows that the RHI system involves a lot of console variables. Some console variables are listed below to facilitate debugging and optimizing RHI rendering effects or efficiency:

name	describe
r.RHI.Name	Displays the name of the current RHI, such as D3D11.
r.RHICmdAsyncRHIThreadDispatch	Experimental option, whether to perform RHI scheduling asynchronously. This allows data to be refreshed to the RHI thread faster, avoiding lag at the end of the frame.
r.RHICmdBalanceParallelLists	Allows to enable preprocessing of DrawList to try to balance the load between command lists. 0: off, 1: on, 2: experimental option, use the result of the previous frame (does not do anything in splitscreen etc.).
r.RHICmdBalanceTranslatesAfterTasks	Experimental option to balance rendering tasks after translation is completed in parallel. Minimizes the number of deferred contexts, but increases latency in starting translation.
r.RHICmdBufferWriteLocks	Relevant only for RHI threads. Debug option for diagnosing buffer lock problems.
r.RHICmdBypass	Whether to bypass the RHI command list and send RHI commands immediately. 0: Disable (multi-threaded rendering needs to be enabled), 1: Enable.
r.RHICmdCollectRHIThreadStatsFromHighLevel	This will push statistics on the RHI thread of execution so it can be determined which high-level pass they came from. Has a detrimental effect on frame rate. Enabled by default.

name	describe
r.RHICmdFlushOnQueueParallelSubmit	Wait for completion of parallel command list immediately after submission. Problem diagnosis. Applies only to some RHIs.
r.RHICmdFlushRenderThreadTasks	If true, flush the render thread tasks every time it is called. Problem diagnosis. This is a master switch for more fine-grained cvars.
r.RHICmdForceRHIFlush	Force a flush to be sent to the RHI thread for each task. Problem diagnosis.
r.RHICmdMergeSmallDeferredContexts	Merge small parallel translation tasks, based on r.RHICmdMinDrawsPerParallelCmdList.
r.RHICmdUseDeferredContexts	Execute a command list in parallel using deferred contexts. Only works on some RHIs.
r.RHICmdUseParallelAlgorithms	True to use parallel algorithm. Ignored if r.RHICmdBypass is 1.
r.RHICmdUseThread	Using RHI threads. Problem diagnosis.
r.RHICmdWidth	Controls the granularity of tasks for large numbers of things in the parallel renderer.
r.RHIThread.Enable	Enables/disables the RHI thread, and determines whether RHI work runs on a dedicated thread.
RHI.GPUHitchThreshold	The threshold (in milliseconds) for detecting jank on the GPU.
RHI.MaximumFrameLatency	The number of frames that can be queued for rendering.
RHI.SyncThreshold	The number of consecutive "fast" frames before vsync is enabled.
RHI.TargetRefreshRate	If non-zero, the display will never be updated more often than the target refresh rate (in Hz).

It should be noted that the above only lists some RHI-related variables, and there are many more that are not listed. You can view comprehensive commands in the following menus:



10.5 Summary

This article mainly explains the basic concepts, types, and mechanisms of UE's RHI system. I hope that after studying this article, you will no longer be unfamiliar with UE's RHI and will be able to easily master, apply, and expand it.

10.5.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the grasp and understanding of the UE RHI system:

- What types of RHI resources are there? What is the relationship and difference between them and the resources of the rendering layer? How does the rendering system delete RHI resources?

- What are the main types of RHI commands? What is the execution mechanism and process of the command list?
- Briefly describe the relationship between the RHI context and DynamicRHI. Briefly describe the implementation architecture of D3D11.
  How are the multiple threads of UE related? What factors can cause them to freeze?
-

-

-

-

-
-

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming
- Graphics Programming Overview
- Analysis of UE4 rendering module
- UE4 Render System Sheet
- 【UE4 Renderer】<03> PipelineBase Scalability
- for All: Unreal Engine 4 with Intel
- Reflection on Shader's Execution Mechanism from the Bottom to the Top (and Vulkan Learning Summary)
- Learning DirectX 12 – Lesson 1 – Initialize DirectX 12
- Learning DirectX 12 – Lesson 2 – Rendering Learning
- DirectX 12 – Lesson 3 – Framework
- Learning DirectX 12 – Lesson 4 – Textures
- UE Advanced Performance Analysis Technology RHI Attack of Vulkan Mobile
- Development Command Buffer BRINGING UNREAL ENGINE 4 TO OPENGL
- Nick Penwarden Epic Games Subpass Introduction
-
- Best Practice for Mobile