Analysis of Unreal Rendering System (11) - RDG

Table of contents

11.1 Overview

RDGstands for**Rendering Dependency Graph**,which means**rendering dependency graph**.It is a new rendering subsystem introduced in UE4.22. It is a scheduling system based on directed acyclic graph (DAG) and is used to perform full-frame optimization of the rendering pipeline.

It takes advantage of modern graphics APIs (DirectX 12, Vulkan, and Metal 2) to improve performance through automatic asynchronous compute scheduling and more efficient memory management and barrier management.

Traditional graphics APIs (DirectX 11, OpenGL) require the driver to invoke complex heuristics to determine when and how to perform critical scheduling operations on the GPU. For example, flushing caches, managing and reusing memory, performing layout transformations, and so on. Due to the immediate mode nature of the interface, complex record keeping and state tracking are required to handle various corner cases. These cases ultimately have a negative impact on
performance and hinder parallelism.

Modern graphics APIs (DirectX 12, Vulkan, and Metal 2) differ from traditional graphics APIs by shifting the burden of low-level GPU management to the application. This allows applications to take advantage of high-level context of the rendering pipeline to drive scheduling , thereby improving performance and simplifying the rendering stack.

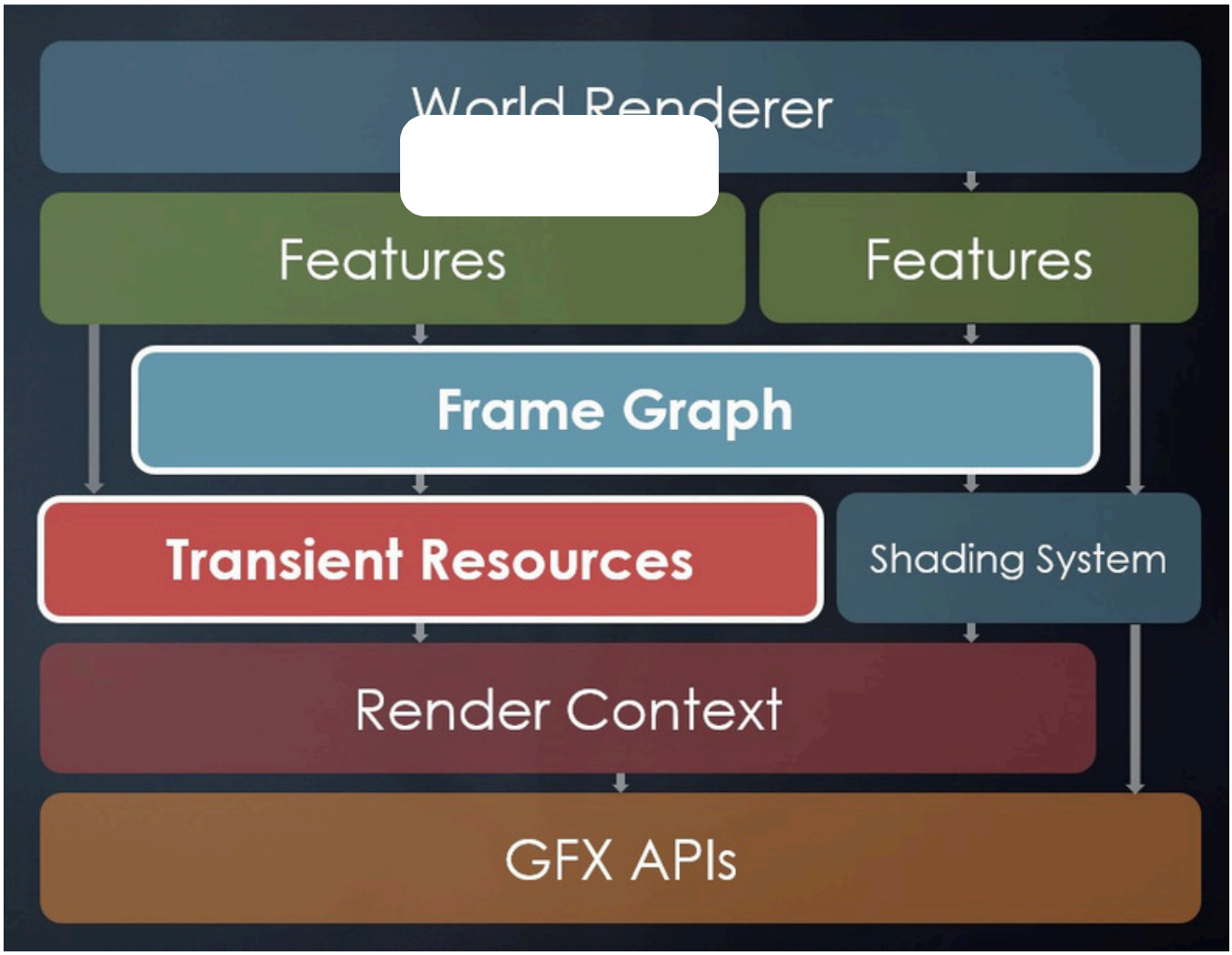
The idea of RDG is not to execute the Pass immediately on the GPU, but to collect all the Passes that need to be rendered first, and then compile and execute the graph in the order of dependencies, during which various types of cropping and optimization will be performed.

The frame-wide awareness of the dependency graph data structure combined with the capabilities of modern graphics APIs enables RDG to perform complex scheduling tasks in the background:

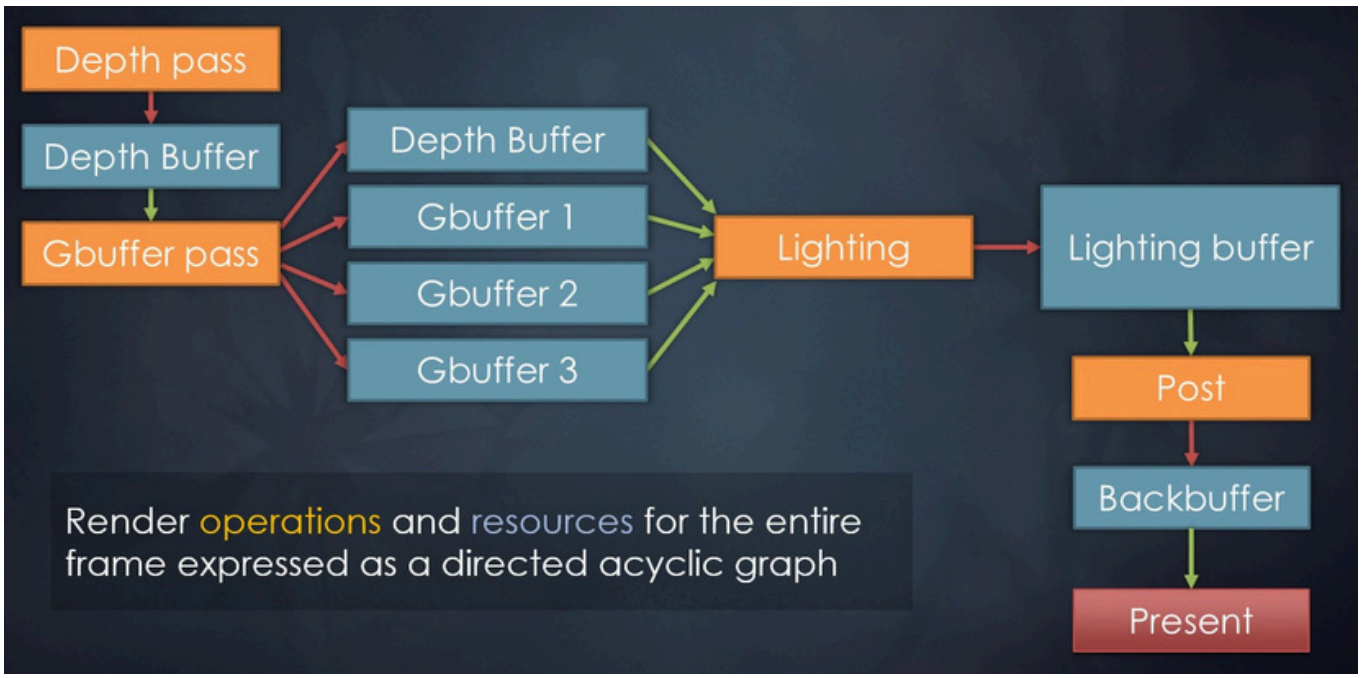
- Performs automatic scheduling and isolation of asynchronous computation channels.
- Keeps aliased memory between resources active during disjoint intervals of frames. Start
- barrier and layout transformations early to avoid pipeline delays.

Additionally, RDG improves the development process by providing rich validation during channel setup using dependency graphs to automatically capture issues that impact functionality and performance.

RDG is not a concept and technology created by UE. As early as GDC 2017, Frostbite had already implemented and applied Frame Graph technology. Frame Graph is designed to isolate the engine's various rendering functions (Features) from the upper rendering logic (Renderer) and lower resources (Shader, RenderContext, graphics API, etc.) for further decoupling and optimization, the most important of which is multi-threading and parallel rendering.



FrameGraph is a high-level representation of Render Passes and resources, containing all the information used in a frame. The order and dependencies between Passes can be specified, and the following figure is an example:



The order and dependency graph of deferred rendering implemented by Frostbite Engine using frame graph method.

It is no exaggeration to say that UE's RDG is customized and implemented based on Frame Graph. By UE4.26, RDG has been widely popularized, including scene rendering, post-processing, ray tracing and other modules all use RDG instead of the original direct call RHI command method.

This article mainly describes the following contents of UE RDG:

- Basic concepts and types of RDG.
- How to use RDG.
- The internal mechanism and principles of RDG.

11.2 RDG Basics

This chapter first explains the main types, concepts, interfaces, etc. involved in RDG.

11.2.1 RDG Basic Types

RDG basic types and interfaces are mainly concentrated in RenderGraphUtils.h and RenderGraphDefinitions.h. Some of the analysis is as follows:

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

// RDG Passtype.
enum classERDGPassFlags: uint8 {

	None =0,                                //    For no parametersAddPassfunction.
	Raster =1<<0, Compute =1<<1             // PassUse rasterization on the graphics pipeline.
	, AsyncCompute =1<<2, Copy              // PassUsed in the graphics pipelinecompute.
	=1<<3,                                    // PassUse on asynchronous computation pipelinescompute //
                                              PassUse the copy command on the graphics pipeline.
	NeverCull =1<<4, SkipRenderPass =       //Not optimized for clipping, used for specialpass.
	1<<5, RasterValid when binding.         //neglectBeginRenderPass/EndRenderPass,Leave it to the user to call.
Will be disabledPassmerge.
	UntrackedAccess =1<<6,                    // PassVisit the originalRHIResources, which may be registered toRDGHowever, all the
Sources remain in their current state. This flag prevents graphics scheduling across a channel's split barrier. Any split is delayed untilpassAfter execution. The resource may not be pass
Change state during execution. Barrier that affects performance. Cannot be used withAsyncComputecombination.
	Readback = Copy | NeverCull,            // PassUse the copy command, but write to the staging resource (staging resource).

	CommandMask = Raster | Compute | AsyncCompute | Copy,//Flag mask indicating submission topassofRHIThe type of command.

	ScopeMask = NeverCull | UntrackedAccess//Flags mask that can be used by the passed flag scope
};

// Buffermark.
enum classERDGBufferFlags: uint8 {

	None =0,//No mark. MultiFrame =1<<0//
	Lasts for multiple frames.
};

//Texture markings.
enum classERDGTextureFlags: uint8
```

```
{
    None =0,
    MultiFrame =1<<0,//Lasts for multiple frames. MaintainCompression =1<<1,//
    Prevents decompression of metadata on this Texture.
};

// UAVmark.
enum classERDGUnorderedAccessViewFlags: uint8 {

    None =0,
    SkipBarrier =1<<0//Ignore barriers.
};

//The parent resource type.
enum classERDGParentResourceType: uint8 {

    Texture,
    Buffer,
    MAX
};

//The view type.
enum classERDGViewType: uint8 {

    TextureUAV,    //  TextureUAV(For  writing  data)
    TextureSRV,    //  TextureSRV(For reading data)
    BufferUAV,     //  bufferUAV(For writing data)
    BufferSRV,     //  bufferSRV(For reading data)
    MAX
};

//Used to specify texture metadata planes when creating a view
enum classERDGTextureMetaDataAccess: uint8 {

    None =0,                    //  The main plane is compressed by default.
    CompressedSurface,          //  The main plane is used without compression.
    Depth,          //The depth plane is compressed by default.
    Stencil,    //  Template planes are compressed by default.
    HTile,          // HTileflat.
    FMask,      // FMaskflat.
    CMask       // CMaskflat.
};

//SimpleC++Object allocator, withMemStackAllocators track and destroy
objects. classFRDGAllocatorfinal {

public:
    FRDGAllocator();
    ~FRDGAllocator();

    //Allocate raw memory.
    FORCEINLINEvoid*Alloc(uint32 SizeInBytes, uint32 AlignInBytes) {

        returnMemStack.Alloc(SizeInBytes, AlignInBytes);
    }
    //distributePODmemory without tracking
    destructors. template <typename PODType>
    FORCEINLINE PODType*AllocPOD()
```

```
    {
        return    reinterpret_cast<PODType*>(Alloc(sizeof(PODType),                alignof(PODType)));
    }
    //    With destruction trackingC++Object allocation.
    template <typename ObjectType, typename...TArgs> FORCEINLINE
    ObjectType*AllocObject(TArgs&&... Args) {

        TTrackedAlloc<ObjectType>* TrackedAlloc = new(MemStack) TTrackedAlloc<ObjectType>
        (Forward<TArgs&&>(Args)...);
        check(TrackedAlloc);
        TrackedAllocs.Add(TrackedAlloc); return
        TrackedAlloc->Get();
    }
    //Without destruction trackingC++Object allocation. (Dangerous, use with
    caution) template <typename ObjectType, typename...TArgs> FORCEINLINE
    ObjectType*AllocNoDestruct(TArgs&&... Args) {

        returnnew (MemStack) ObjectType(Forward<TArgs&&>(Args)...);
    }

    //Releases all allocated memory.
    void ReleaseAll();

private:
    class    FTrackedAlloc
    {
    public:
        virtual ~FTrackedAlloc() =default;
    };

    template <typename ObjectType>
    classTTrackedAlloc: public FTrackedAlloc {

    public:
        template <typename...TArgs>
        FORCEINLINETTrackedAlloc(TArgs&&... Args) :Object(Forward<TArgs&&>(Args)...) {}

        FORCEINLINE ObjectType*Get() {return&Object; }

    private:
        ObjectType    Object;
    };

    //Allocator.
    FMemStackBase MemStack; //
    All allocated objects.
    TArray<FTrackedAlloc*,        SceneRenderingAllocator>        TrackedAllocs;
};

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

//Clean up unused resources.
externRENDERCORE_APIvoid ClearUnusedGraphResourcesImpl(constFShaderParameterBindings& ShaderBindings, ...);

(.....)

//Registers an external texture, optionally with a fallback instance.
FRDGTextureRefRegisterExternalTextureWithFallback(FRDGBuilder& GraphBuilder, ...);
```

```
inlineFRDGTextureRefTryRegisterExternalTexture(FRDGBuilder& GraphBuilder, ...); inlineFRDGBufferRef
TryRegisterExternalBuffer(FRDGBuilder& GraphBuilder, ...);

//Utility classes for compute shaders.
struct      RENDERCORE_API FComputeShaderUtils
{
        //    The ideal group size is8x8,existGCNAt least onewave,existNvidiaOccupies twowarp.
        staticconstexpr int32 kGolden2DGroupSize =8;
        staticFIntVectorGetGroupCount(constint32 ThreadCount,constint32 GroupSize);

        //Dispatching compute shaders toRHIA list of commands, along with
        their arguments. template<typename TShaderClass>
        static void Dispatch(FRHIComputeCommandList& RHICmdList,const TShaderRef<TShaderClass>&
ComputeShader,consttypename TShaderClass::FParameters& Parameters, FIntVector GroupCount);

        //Dispatching indirect compute shaders toRHIA list of commands, along with
        their arguments. template<typename TShaderClass>
        static void DispatchIndirect(FRHIComputeCommandList& RHICmdList,const TShaderRef<TShaderClass>&
ComputeShader,consttypename TShaderClass::FParameters& Parameters, FRHIVertexBuffer*
IndirectArgsBuffer, uint32 IndirectArgOffset);
        //Dispatching compute shaders torender graph builder,Carry its
        parameters. template<typename TShaderClass>
        static void AddPass(FRDGBuilder& GraphBuilder,FRDGEventName&& PassName,ERDGPassFlags PassFlags,const
TShaderRef<TShaderClass>& ComputeShader,typename TShaderClass::FParameters* Parameters,FIntVector
GroupCount);

        (......)

        //CleaningUAV.
        static void ClearUAV(FRDGBuilder& GraphBuilder, FGlobalShaderMap* ShaderMap, FRDGBufferUAVRef
UAV, uint32 ClearValue);
        static void ClearUAV(FRDGBuilder& GraphBuilder, FGlobalShaderMap* ShaderMap, FRDGBufferUAVRef
UAV, FVector4 ClearValue);
};

//Add copy texturePass.
void AddCopyTexturePass(FRDGBuilder& GraphBuilder, FRDGTextureRef InputTexture, FRDGTextureRef
OutputTexture,constFRHICopyTextureInfo& CopyInfo); (......)


//Add copy to parsing targetPass.
void AddCopyToResolveTargetPass(FRDGBuilder& GraphBuilder, FRDGTextureRef InputTexture, FRDGTextureRef
OutputTexture,constFResolveParams& ResolveParams);

//Clean up various resourcesPass.
void AddClearUAVPass(FRDGBuilder& GraphBuilder, FRDGBufferUAVRef BufferUAV, uint32 Value); void
AddClearUAVFloatPass(FRDGBuilder& GraphBuilder, FRDGBufferUAVRef BufferUAV,float Value);

void AddClearUAVPass(FRDGBuilder& GraphBuilder, FRDGTextureUAVRef TextureUAV,const FUintVector4&
ClearValues);
void AddClearRenderTargetPass(FRDGBuilder& GraphBuilder, FRDGTextureRef Texture); void
AddClearDepthStencilPass(FRDGBuilder& GraphBuilder,FRDGTextureRef Texture,bool bClearDepth,floatDepth,
boolbClearStencil,uint8 Stencil);
void AddClearStencilPass(FRDGBuilder& GraphBuilder, FRDGTextureRef Texture); (......)


//Added readback texturePass.
void AddEnqueueCopyPass(FRDGBuilder& GraphBuilder, FRHIGPUTextureReadback* Readback,
```

```
FRDGTextureRef SourceTexture, FResolveRect Rect = FResolveRect()); //Increase the
readback bufferPass.
void AddEnqueueCopyPass(FRDGBuilder& GraphBuilder, FRHIGPUBufferReadback* Readback, FRDGBufferRef
SourceBuffer, uint32 NumBytes);

//Create a resource.
FRDGBufferRefCreateStructuredBuffer(FRDGBuilder& GraphBuilder, ...); FRDGBufferRef
CreateVertexBuffer(FRDGBuilder& GraphBuilder, ...);

//No parametersPassIncrease.
template <typename ExecuteLambdaType>
void AddPass(FRDGBuilder& GraphBuilder, FRDGEventName&& Name, ExecuteLambdaType&&
ExecuteLambda);
template <typename ExecuteLambdaType>
void AddPass(FRDGBuilder& GraphBuilder, ExecuteLambdaType&& ExecuteLambda);

//Other specialPass
void AddBeginUAVOverlapPass(FRDGBuilder& GraphBuilder);
void AddEndUAVOverlapPass(FRDGBuilder& GraphBuilder);

(.....)
```

11.2.2 RDG Resources

RDG resources do not use RHI resources directly, but wrap RHI resource references, and then encapsulate different types of resources separately and add additional information. Some RDG definitions are as follows:

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

classFRDGResource
{
public:
    //Delete the copy constructor.
    FRDGResource(constFRDGResource&) = delete; virtual
    ~FRDGResource() =default;

    ////////////////////////////////////////////////////// ////////////////////////// //The following interfaces can only
    beRDGofPassCalled during execution.

    //Marks whether this resource is used, if not, it will be cleaned up.
#if RDG_ENABLE_DEBUG
    virtual void MarkResourceAsUsed();
#else
    inline void MarkResourceAsUsed() {}
#endif
    // GetRDGofRHIResource references.
    FRHIResource* GetRHI() const
    {
        ValidateRHIAccess();
        returnResourceRHI;
    }

    ////////////////////////////////////////////////////// //////////////////////////
```

```
protected:
    FRDGResource(constTCHAR* InName);

    //Assign this resource toRHIA simple pass-through container for
    resources. void SetPassthroughRHI(FRHIResource* {                InResourceRHI)

        ResourceRHI = InResourceRHI;
#if     RDG_ENABLE_DEBUG
        DebugData.bAllowRHIAccess       =true;
        DebugData.bPassthrough =       true;
#endif
    }

    bool IsPassthrough()          const
    {
#if  RDG_ENABLE_DEBUG
        return    DebugData.bPassthrough;
#else
        return    false;
#endif
    }

    /** Verify that the RHI resource can be accessed at a pass execution. */ void ValidateRHIAccess()
    const {

#if  RDG_ENABLE_DEBUG
        checkf(DebugData.bAllowRHIAccess,
            TEXT("Accessing the RHI resource of %s at this time is not allowed. If you hit
this check in pass, ")
            TEXT("that is due to this resource not being referenced in the parameters of
your pass."),
            Name);
#endif
    }

    FRHIResource*GetRHIUnchecked()const {

        returnResourceRHI;
    }

    //RHIResource references.
    FRHIResource* ResourceRHI = nullptr;

private:
    //Debug information.
#if  RDG_ENABLE_DEBUG
    classFDebugData
    {
    private:
        //Tracks whether a resource is used at runtimepassoflambdaIn actual use, to detectpassUnnecessary
        resource dependencies. boolbIsActuallyUsedByPass =false; //trackPassDuring execution, the
        underlyingRHIDo you allow access? boolbAllowRHIAccess=false;

        //Iftrue, the resource is not attached to any builder, but exists as a virtual container for temporarily storing
        code inRDG. boolbPassthrough =false; }DebugData;

#endif
};
```

```
classFRDGUniformBuffer: public FRDGResource {

public:
    //GetRHI.
    FRHIUniformBuffer*GetRHI()const {

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

    (......)

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

private:
    //Parameter structure.
    constFRDGParameterStruct //RHIParameterStruct;
    resource.
    TRefCountPtr<FRHIUniformBuffer> //RDGUniformBufferRHI;
    Handle.
    FRDGUniformBufferHandle Handle; //
    Global or local binding.
    uint8 bGlobal :1;
};

//RDGUniformBufferTemplate class.
template <typename ParameterStructType> class
TRDGUniformBuffer: public FRDGUniformBuffer {

public:
    constTRDGParameterStruct<ParameterStructType>&GetParameters()const;
    TUniformBufferRef<ParameterStructType>GetRHIRef()const; constParameterStructType*
    operator->()const;

    (......)
};

//A render graph resource whose lifecycle is allocated by the graph tracker. It may have child resources
(such as views) that reference it. classFRDGParentResource: public FRDGResource {

public:
    //The parent resource type.
    const   ERDGParentResourceType Type;
    bool  IsExternal()const;

protected:
    FRDGParentResource(constTCHAR* InName, ERDGParentResourceType InType);

    //Is it an external resource?
    uint8 bExternal :1; //Whether
    the resource is extracted.
    uint8 bExtracted :1;
    //Is this resource required?acquire / discard.
```

```
        uint8 bTransient :1; //Whether it
        was allocated by the last owner.
        uint8 bLastOwner :1; //will
        be cropped.
        uint8 bCulled :1; //Whether it is
        calculated asynchronouslyPassuse.
        uint8 bUsedByAsyncComputePass:1;

private:
        //Number of citations.
        uint16 ReferenceCount =0; //The initial and final states of
        user-allocated resources (if known) ERHIAccess
        AccessInitial = ERHIAccess::Unknown; ERHIAccess
        AccessFinal = ERHIAccess::Unknown;

        FRDGPassHandle AcquirePass;
        FRDGPassHandle FirstPass;
        FRDGPassHandle LastPass;

        (.....)
};

//Creates a render texture description.
structRENDERCORE_API FRDGTextureDesc
{
        static FRDGTextureDesc Create2D(...);
        static FRDGTextureDesc Create2DArray(...);
        static FRDGTextureDesc Create3D(...);
        static FRDGTextureDesc CreateCube(...);
        static FRDGTextureDesc CreateCubeArray(...);

        bool IsTexture2D() const;
        bool IsTexture3D() const;
        bool IsTextureCube() const;
        bool IsTextureArray() const;
        bool IsMipChain()const;
        bool IsMultisample() const;
        FIntVector GetSize() const;

        //The layout of the subresources.
        FRDGTextureSubresourceLayout bool GetSubresourceLayout() const;
        IsValid()const;

        //Clean up the value.
        FClearValueBinding ClearValue;
        ETextureDimension Dimension = ETextureDimension::Texture2D; //Clean up the
        markup.
        ETextureCreateFlags Flags = TexCreate_None; //Pixel
        format.
        EPixelFormat Format = PF_Unknown; //
        Texture inxandyRange in
        FIntPoint Extent = FIntPoint(1,1); // 3DThe
        depth of the texture.
        uint16 Depth =1; uint16
        ArraySize =1; //Number of
        texture levels.
        uint8 NumMips =1; //Number
        of samples.
```

```
    uint8 NumSamples =1;
};

//The poolRTDescription toRDGTexture description.
inlineFRDGTextureDescTranslate(constFPooledRenderTargetDesc& InDesc,
ERenderTargetTexture InTexture = ERenderTargetTexture::Targetable); //WillRDGTexture
description converted to poolRTdescribe.
inlineFPooledRenderTargetDescTranslate(constFRDGTextureDesc& InDesc);

//The texture of the pool.
classRENDERCORE_API        FRDGPooledTexture
{
public:
    //describe.
    constFRDGTextureDesc        Desc;

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

private:
    FRDGPooledTexture(FRHITexture* InTexture,constFRDGTextureDesc& InDesc,const
FUnorderedAccessViewRHIRef& FirstMipUAV);

    //Initialize the cacheUAV.
    void InitViews(constFUnorderedAccessViewRHIRef& FirstMipUAV); void
        Finalize();
    void Reset();

    //CorrespondingRHITexture.
    FRHITexture* Texture = nullptr; //The
    texture object.
    FRDGTexture* Owner = nullptr; //
    Subresource layout.
    FRDGTextureSubresourceLayout //The status    Layout;
    of the subresource.
    FRDGTextureSubresourceState        State;

    //forRHITexture CacheUAV/SRV. TArray<FUnorderedAccessViewRHIRef,
    TInlineAllocator<1>> TArray<TPair<FRHITextureSRVCreateInfo,                MipUAVs;
                                                    FShaderResourceViewRHIRef>,
TInlineAllocator<1>>SRVs;
    FUnorderedAccessViewRHIRef        HTileUAV;
    FShaderResourceViewRHIRef        HTileSRV;
    FUnorderedAccessViewRHIRef        StencilUAV;
    FShaderResourceViewRHIRef        FMaskSRV;
    FShaderResourceViewRHIRef        CMaskSRV;

    mutable uint32 RefCount =0;
};

//RDGTexture.
classRENDERCORE_API FRDGTexturefinal: public FRDGParentResource {

public:
    //For not yet deliveredRDGofPassCreate a suitableRDGParameter fillingRHIPass-through textures for uniform buffers.
    staticFRDGTextureRefGetPassthrough(constTRefCountPtr<IPooledRenderTarget>&
```

```
PooledRenderTarget);

    //Description and tags.
    const    FRDGTextureDesc     Desc;
    const    ERDGTextureFlags      Flags;

    ////////////////////////////////////////////////// ////////////////////////// //! The following methods may only
    be called during pass execution.

    IPooledRenderTarget*GetPooledRenderTarget()const FRHITexture*
    GetRHI()const

    ////////////////////////////////////////////////// //////////////////////////

    FRDGTextureSubresourceLayout         GetSubresourceLayout()const;
    FRDGTextureSubresourceRange       GetSubresourceRange()const;
    FRDGTextureSubresourceRange       GetSubresourceRangeSRV()const;

private:
    FRDGTextureconstTCHAR* InName,constFRDGTextureDesc& InDesc, ERDGTextureFlags InFlags,
ERenderTargetTexture InRenderTargetTexture);

    void SetRHI(FPooledRenderTarget* PooledRenderTarget, FRDGTextureRef&
    OutPreviousOwner);
    void Finalize();
    FRHITexture*GetRHIUnchecked()const; bool IsLastOwner()const
    ; FRDGTextureSubresourceState&GetState(); const
    ERenderTargetTexture RenderTargetTexture;



    //A layout used to facilitate subresource transitions.
    FRDGTextureSubresourceLayout Layout; //During execution, you
    havePooledTextureThe next texture to allocate. FRDGTextureHandle
    NextOwner; //The handle that has been registered with the builder.

    FRDGTextureHandle Handle;

    //Pool texture.
    IPooledRenderTarget* PooledRenderTarget = nullptr;
    FRDGPooledTexture* PooledTexture = nullptr; //State pointer from
    the pool texture cache
    FRDGTextureSubresourceState* State = nullptr; //Strictly valid while
    holding a strong reference,
    TRefCountPtr<IPooledRenderTarget> Allocation;

    //Track merged subresource status while building the graph
    FRDGTextureTransientSubresourceStateIndirect //During graph  MergeState;
    construction, track the producer of each subresource passed.
    TRDGTextureSubresourceArray<FRDGPassHandle>                  LastProducers;
};

//Pooled buffer.
classRENDERCORE_API       FRDGPooledBuffer
{
public:
    constFRDGBufferDesc         Desc;

    FRHIUnorderedAccessView*GetOrCreateUAV(FRDGBufferUAVDesc UAVDesc);
```

```
        FRHIShaderResourceView*GetOrCreateSRV(FRDGBufferSRVDesc SRVDesc);

        FRHIVertexBuffer*       GetVertexBufferRHI()const;
        FRHIIndexBuffer*        GetIndexBufferRHI()const;
        FRHIStructuredBuffer*           GetStructuredBufferRHI()            const;

        uint32    GetRefCount()      const;
        uint32    AddRef()     const;
        uint32    Release()      const;

        (.....)

private:
        FRDGPooledBufferconstFRDGBufferDesc& InDesc);

        //Vertex/index/struct buffers.
        FVertexBufferRHIRef         VertexBuffer;
        FIndexBufferRHIRef        IndexBuffer;
        FStructuredBufferRHIRef //        StructuredBuffer;
        UAV/SRV.
        TMap<FRDGBufferUAVDesc,       FUnorderedAccessViewRHIRef, FUnordFeDrefdaAucltcSeestsAVllioewcaRtHorl,Ref>>
        TUAVFuncs<FRDGBufferUAVDesc,FShaderResourceViewRHIRef, FDefaultSUeAtAVlslo;cator,
        TMap<FRDGBufferSRVDesc,       FShaderResourceViewRHIRef>> SRVs;
        TSRVFuncs<FRDGBufferSRVDesc,

        void    Reset();
        void    Finalize();

        constTCHAR* Name = nullptr;

        //Owner.
        FRDGBufferRef Owner = nullptr;
        FRDGSubresourceState State;

        mutable uint32 RefCount =0; uint32
        LastUsedFrame =0;
};

//Buffer for rendering graph traces.
classRENDERCORE_API FRDGBufferfinal: public FRDGParentResource {

public:
        const    FRDGBufferDesc      Desc;
        const    ERDGBufferFlags       Flags;

        ////////////////////////////////////////////////// ////////////////////////// //! The following methods may only
        be called during pass execution.

        //GetRHIresource.
        FRHIVertexBuffer*       GetIndirectRHICallBuffer()              const
        FRHIVertexBuffer*       GetRHIVertexBuffer()const
        FRHIStructuredBuffer*           GetRHIStructuredBuffer()            const

        ////////////////////////////////////////////////// //////////////////////////

private:
        FRDGBuffer(constTCHAR* InName,constFRDGBufferDesc& InDesc, ERDGBufferFlags InFlags);
```

```
    //set upRHIresource.
    void SetRHI(FRDGPooledBuffer* InPooledBuffer, FRDGBufferRef& OutPreviousOwner); void Finalize();


    FRDGSubresourceState&GetState()          const
    //RDGHandle.
    FRDGBufferHandle Handle; //The last
    person to process this resource.
    FRDGPassHandle    LastProducer;
    //The next owner.
    FRDGBufferHandle     NextOwner;

    //The assigned pooling buffer.
    FRDGPooledBuffer* PooledBuffer = nullptr; //The status of the
    subresource.
    FRDGSubresourceState*    State    =    nullptr;
    TRefCountPtr<FRDGPooledBuffer>   Allocation;
    FRDGSubresourceState* MergeState = nullptr;
};

(.....)
```

In the RDG system, basically all RHI resources are encapsulated and wrapped to further control and manage RHI resources, accurately control their life cycle, reference relationship and debugging information, etc., and further optimize and trim them to improve rendering performance.

11.2.3 RDG Pass

The RDG Pass module involves concepts such as barriers, resource conversion, and RDGPass:

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

//Used to indicateRHIAn opaque data structure of pending resource
transitions in . structFRHITransition
{
public:
    template <typename T>
    inlineT*GetPrivateData() {

        uintptr_tAddr = Align(uintptr_t(this +1),
        GRHITransitionPrivateData_AlignInBytes);
        returnreinterpret_cast<T*>(Addr);
    }

    template <typename T>
    inlineconstT*GetPrivateData()const {

        returnconst_cast<FRHITransition*>(this)->GetPrivateData<T>();
    }

private:
    FRHITransition(constFRHITransition&) = delete; FRHITransition(FRHITransition&&) = delete;
    FRHITransition(ERHIPipeline SrcPipelines, ERHIPipeline DstPipelines);
```

```
        ~FRHITransition();

        //Get the total allocation size.
        staticuint64GetTotalAllocationSize() //Get the
        alignment byte count.
        staticuint64GetAlignment();

        //Start tagging.
        inlinevoid MarkBegin(ERHIPipeline Pipeline)const {

                int8 Mask = int8(Pipeline);
                int8 PreviousValue = FPlatformAtomics::InterlockedAnd(&State, ~Mask); if(PreviousValue ==
                Mask) {

                        Cleanup();
                }
        }
        //End tag.
        inlinevoid MarkEnd(ERHIPipeline Pipeline)const {

                int8 Mask = int8(Pipeline) << int32(ERHIPipeline::Num);
                int8 PreviousValue = FPlatformAtomics::InterlockedAnd(&State, ~Mask); if(PreviousValue ==
                Mask) {

                        Cleanup();
                }
        }
        //Clean up conversion resources, includingRHIConversion and
        allocated memory. inlinevoid Cleanup()const;

        mutable int8 State;

#if     DO_CHECK
        mutable    ERHIPipeline       AllowedSrc;
        mutable    ERHIPipeline       AllowedDst;
#endif

#if     ENABLE_RHI_VALIDATION
        //Fence.
        RHIValidation::FFence* Fence = nullptr; //Pending
        start operation.
        RHIValidation::FOperationsList //Pending PendingOperationsBegin;
        end operations.
        RHIValidation::FOperationsList                  PendingOperationsEnd;
#endif
};

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

//RDGBarrier Batch
classRENDERCORE_API        FRDGBarrierBatch
{
public:
        FRDGBarrierBatch(constFRDGBarrierBatch&) = delete; bool
        IsSubmitted()                 const
        FStringGetName()         const;

protected:
```

```
        FRDGBarrierBatch(constFRDGPass* InPass,constTCHAR* InName); void
        SetSubmitted();
        ERHIPipelineGetPipeline()const

private:
        boolbSubmitted =false; //
        GraphicsorAsyncCompute
        ERHIPipeline Pipeline;

#if    RDG_ENABLE_DEBUG
        const    FRDGPass* Pass;
        const    TCHAR* Name;
#endif
};

//Barrier batch starts
classRENDERCORE_API FRDGBarrierBatchBeginfinal: public FRDGBarrierBatch {

public:
        FRDGBarrierBatchBegin(constFRDGPass* InPass,constTCHAR* InName,
TOptional<ERHIPipeline> InOverridePipelineForEnd = {});
        ~FRDGBarrierBatchBegin();

        //Added resource conversion to batches.
        void AddTransition(FRDGParentResourceRef Resource,constFRHITransitionInfo& Info);

        constFRHITransition*GetTransition()const; bool
                IsTransitionValid()const;
        void SetUseCrossPipelineFence();
        //Submit barrier/resource transition.
        void Submit(FRHIComputeCommandList&          RHICmdList);

private:
        TOptional<ERHIPipeline> OverridePipelineToEnd; bool
        bUseCrossPipelineFence =false;

        //The resource transformation stored after submission, which is assigned back when the batch
        is finishednull. constFRHITransition* Transition = nullptr; //An array of asynchronous resource
        transformations to perform.
        TArray<FRHITransitionInfo, TInlineAllocator<1, SceneRenderingAllocator>> Transitions;

#if    RDG_ENABLE_DEBUG
        //andTransitionsArray matchingRDGArray of resources, for debugging
        only. TArray<FRDGParentResource*, SceneRenderingAllocator>                Resources;
#endif
};

//Barrier batch ends
classRENDERCORE_API FRDGBarrierBatchEndfinal: public FRDGBarrierBatch {

public:
        FRDGBarrierBatchEnd(constFRDGPass* InPass,constTCHAR* InName);
        ~FRDGBarrierBatchEnd();

        //Reserve memory.
        void ReserveMemory(uint32 ExpectedDependencyCount); //Insert
        dependencies on start batch, a start batch can insert multiple end batches.
        void AddDependency(FRDGBarrierBatchBegin* BeginBatch);
```

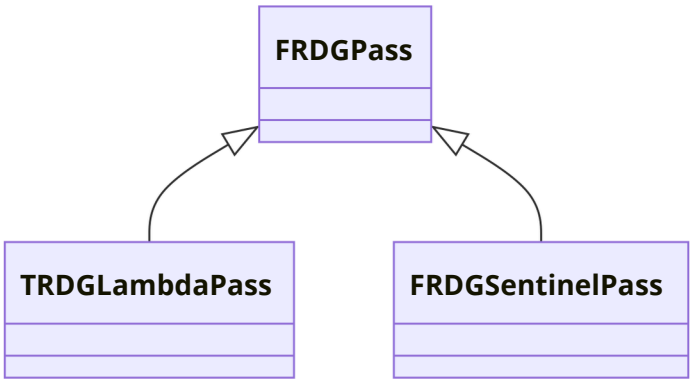
```
    //Submit resource conversion.
    void Submit(FRHIComputeCommandList& RHICmdList);

private:
    //This end batch can be called after the start batch conversion is completed.
    TArray<FRDGBarrierBatchBegin*, TInlineAllocator<1, Dependencies;SceneRenderingAllocator>>

};

//RGDChannel base class.
classRENDERCORE_API        FRDGPass
{
public:
    FRDGPass(FRDGEventName&& InName, FRDGParameterStruct InParameterStruct, ERDGPassFlags InFlags);

    FRDGPass(constFRDGPass&) = delete; virtual
    ~FRDGPass() =default;

    //Channel data interface.
    constTCHAR*GetName()const;
    FORCEINLINEconstFRDGEventName&GetEventName()const;
    FORCEINLINE ERDGPassFlagsGetFlags()const; FORCEINLINE ERHIPipeline
    GetPipeline()const; // RDG Passparameter.

    FORCEINLINE FRDGParameterStructGetParameters()const; FORCEINLINE
    FRDGPassHandleGetHandle()const; bool
        IsMergedRenderPassBegin()const;
    bool  IsMergedRenderPassEnd()const;
    bool  SkipRenderPassBegin()const;
    bool  SkipRenderPassEnd()const;
    bool  IsAsyncCompute()const;
    bool IsAsyncComputeBegin()        const;
    bool IsAsyncComputeEnd()       const;
    bool IsGraphicsFork()         const;
    bool IsGraphicsJoin()         const;
    //The producer handle.
    constFRDGPassHandleArray&GetProducers()const; //Cross-
    pipeline producer.
    FRDGPassHandle   GetCrossPipelineProducer()           const;
    //Cross-pipeline consumers.
    FRDGPassHandle   GetCrossPipelineConsumer()           const;
    //ForkPass.
    FRDGPassHandle   GetGraphicsForkPass()        const;
    //mergePass.
    FRDGPassHandle   GetGraphicsJoinPass()        const;

#if  RDG_CPU_SCOPES
    FRDGCPUScopes GetCPUScopes()       const;
#endif
#if  RDG_GPU_SCOPES
    FRDGGPUScopes GetGPUScopes()       const;
#endif

private:
    //Precursor barrier.
    FRDGBarrierBatchBegin&       GetPrologueBarriersToBegin(FRDGAllocator&                Allocator);
    FRDGBarrierBatchEnd&GetPrologueBarriersToEnd(FRDGAllocator& //Post-order  Allocator);
    barrier.
```

```
    FRDGBarrierBatchBegin&        GetEpilogueBarriersToBeginForGraphics(FRDGAllocator&
Allocator);
    FRDGBarrierBatchBegin&        GetEpilogueBarriersToBeginForAsyncCompute(FRDGAllocator&
Allocator);
    FRDGBarrierBatchBegin&        GetEpilogueBarriersToBeginFor(FRDGAllocator& Allocator,
ERHIPipelinePipelineForEnd);

    ////////////////////////////////////////////////// ////////////////////////// //! User Methods to Override


    //Execute implementation.
    virtualvoid ExecuteImpl(FRHIComputeCommandList& RHICmdList) =0;

    ////////////////////////////////////////////////// ////////////////////////////

    //implement.
    void Execute(FRHIComputeCommandList& RHICmdList);

    // Passdata.
    const    FRDGEventName Name;
    const FRDGParameterStruct          ParameterStruct;
    const ERDGPassFlags         Flags;
    const ERHIPipeline          Pipeline;
    FRDGPassHandle     Handle;

    // Passmark.
    union
    {
        struct
        {
            uint32 bSkipRenderPassBegin :1; uint32
            bSkipRenderPassEnd :1; uint32
            bAsyncComputeBegin :1; uint32
            bAsyncComputeEnd :1; uint32
            bAsyncComputeEndExecute:1; uint32
            bGraphicsFork :1; uint32 bGraphicsJoin :1;
            uint32 bUAVAccess :1;


            IF_RDG_ENABLE_DEBUG(uint32 bFirstTextureAllocated:1);
        };
        uint32 PackedBits =0;
    };

    //The handle of the latest producer across the pipeline.
    FRDGPassHandle CrossPipelineProducer; //The handle of the
    earliest consumer that crossed the pipeline.
    FRDGPassHandle CrossPipelineConsumer;

    // (OnlyAsyncCompute)Graphics pass, the channel is asynchronous calculation intervalfork / join.
    FRDGPassHandle       GraphicsForkPass;
    FRDGPassHandle     GraphicsJoinPass;

    //The channel that handles the predecessor/successor barriers of this channel.
    FRDGPassHandle     PrologueBarrierPass;
    FRDGPassHandle     EpilogueBarrierPass;

    //ProducerPassList.
    FRDGPassHandleArray Producers;
```

```
    //Texture state.
    structFTextureState
    {
        FRDGTextureTransientSubresourceState              State;
        FRDGTextureTransientSubresourceStateIndirect uint16       MergeState;
        ReferenceCount =0;
    };

    //Buffer status.
    structFBufferState
    {
        FRDGSubresourceState       State;
        FRDGSubresourceState* MergeState = nullptr; uint16
        ReferenceCount =0;
    };

    //Map the texture/buffer toPassHow to use information in.
    TSortedMap<FRDGTexture*, FTextureState, SceneRenderingAllocator> TextureStates;
    TSortedMap<FRDGBuffer*, FBufferState, SceneRenderingAllocator> BufferStates; //In executing thisPass
    During the period, the plan startedPassA list of parameters.
    TArray<FRDGPass*, TInlineAllocator<1, SceneRenderingAllocator>> ResourcesToBegin; TArray<FRDGPass*, TInlineAllocator<
    1, SceneRenderingAllocator>> ResourcesToEnd; //existacquireList of textures acquired *after* completion, *before*
    discarding. TArray<FRHITexture*, SceneRenderingAllocator> //existPassAGeft ethre c toexmtuprel eavtainilagb le* ,f*or yaoll ual lowcailtli ognes.t (a
    that, the texture list is discarded. Processing, and the last alias of the teTxetuxrteu r(aelsiaT)UoAsec qautiore-d;iscard behavior (to s
    cleaner handoffs to users or returns to the pool).                    Discard only applies to data that is marked as transient (transient)Pattern

    TArray<FRHITexture*, SceneRenderingAllocator> TexturesToDiscard;

    FRDGBarrierBatchBegin* PrologueBarriersToBegin = nullptr; FRDGBarrierBatchEnd*
    PrologueBarriersToEnd = nullptr; FRDGBarrierBatchBegin* EpilogueBarriersToBeginForGraphics
    = nullptr; FRDGBarrierBatchBegin* EpilogueBarriersToBeginForAsyncCompute = nullptr;


    EAsyncComputeBudget AsyncComputeBudget = EAsyncComputeBudget::EAll_4;
};

// RDG Pass LambdaExecute the function.
template <typename ParameterStructType, typename ExecuteLambdaType> class
TRDGLambdaPass: public FRDGPass {

    (.....)

    TRDGLambdaPass(FRDGEventName&& InName,constParameterStructType* InParameterStruct, ERDGPassFlags
InPassFlags, ExecuteLambdaType&& InExecuteLambda);

private:
    //Execute implementation.
    void ExecuteImpl(FRHIComputeCommandList& RHICmdList) override {

        check(!kSupportsRaster || RHICmdList.IsImmediate()); //CallLambda
        Examples. ExecuteLambda(static_cast<TRHICommandList&>(RHICmdList));

    }

    LambdaExamples.
    ExecuteLambdaType ExecuteLambda;
};
```

```
//With emptyLambdaofPass.
template <typename ExecuteLambdaType>
classTRDGEmptyLambdaPass: public TRDGLambdaPass<FEmptyShaderParameters,
ExecuteLambdaType>
{
public:
    TRDGEmptyLambdaPass(FRDGEventName&& InName, ERDGPassFlags InPassFlags,
ExecuteLambdaType&& InExecuteLambda);

private:
    FEmptyShaderParameters      EmptyShaderParameters;
};

//For pre/postPass.
classFRDGSentinelPassfinal: public FRDGPass {

public:
    FRDGSentinelPass(FRDGEventName&& Name);

private:
    void  ExecuteImpl(FRHIComputeCommandList&) override;
    FEmptyShaderParameters      EmptyShaderParameters;
};
```

The above shows that RDG Pass is relatively complex and is one of the most core types in the RDG system, involving data and processing such as consumers, producers, conversion dependencies, and various resource states. RDG Pass has the following types:



There is no one-to-one correspondence between RDG Pass and rendering Pass. It is possible that multiple passes can be merged into one rendering Pass. See the following chapters for details. The
The most complex aspects of RDG Pass are multi-threaded processing, resource state conversion, and dependency processing, but this section will not cover them first, and they will be discussed in detail in subsequent chapters.

11.2.4 FRDGBuilder

FRDGBuilder is the heart and engine of the RDG system, and is also a housekeeper responsible for collecting rendering passes and parameters, compiling passes and data, handling resource

dependencies, cutting and optimizing various data, and providing execution interfaces. Its declaration is as follows:

```
classRENDERCORE_API FRDGBuilder {

public:
	FRDGBuilder(FRHICommandListImmediate& InRHICmdList, FRDGEventName InName = {},const char*
 UnaccountedCSVStat = kDefaultUnaccountedCSVStat);
	 FRDGBuilder(constFRDGBuilder&) = delete;

	//Look for an external texture, and return if not foundnull.
	FRDGTextureRefFindExternalTexture(FRHITexture* Texture)const; FRDGTextureRefFindExternalTexture
	(IPooledRenderTarget* ExternalPooledTexture, ERenderTargetTexture Texture)const;


	//Register external poolRTarriveRDG,so thatRDGTracking. PondRTIt may include twoRHITexture:MSAAand
	nonMSAA. FRDGTextureRefRegisterExternalTexture(
		constTRefCountPtr<IPooledRenderTarget>& ExternalPooledTexture, ERenderTargetTexture
		Texture = ERenderTargetTexture::ShaderResource, ERDGTextureFlags Flags =
		ERDGTextureFlags::None);
	FRDGTextureRef	RegisterExternalTexture(TRefCountPtr<IPooledRenderTarget>&
		const	ExternalPooledRenderTarget, const TCHAR* Name = nullptr, ERenderTargetTexture	=	ERenderTargetTexture::ShaderResource,
		ERDGTextureFlags Flags = ERDGTextureFlags::None);



	//Register external buffer toRDG,so thatRDGTrack it.
	FRDGBufferRefRegisterExternalBuffer(constTRefCountPtr<FRDGPooledBuffer>&
ExternalPooledBuffer, ERDGBufferFlags Flags = ERDGBufferFlags::None);
	FRDGBufferRefRegisterExternalBuffer(constTRefCountPtr<FRDGPooledBuffer>&
ExternalPooledBuffer, ERDGBufferFlags Flags, ERHIAccess AccessFinal);
	FRDGBufferRef	RegisterExternalBuffer(TRefCountPtr<FRDGPooledBuffer>&
		const	ExternalPooledBuffer, const TCHAR* Name = nullptr, ERDGBufferFlags Flags = ERDGBufferFlags::None);
		ERDGBufferFlags::None);



	//Resource creation interface.
	FRDGTextureRefCreateTexture(constFRDGTextureDesc& Desc,constTCHAR* Name, ERDGTextureFlags
Flags = ERDGTextureFlags::None);
	FRDGBufferRefCreateBuffer(constFRDGBufferDesc& Desc,constTCHAR* Name, ERDGBufferFlags
Flags = ERDGBufferFlags::None);
	FRDGTextureSRVRefCreateSRV(constFRDGTextureSRVDesc& Desc);
	FRDGBufferSRVRefCreateSRV(constFRDGBufferSRVDesc& Desc);
	FORCEINLINE FRDGBufferSRVRefCreateSRV(FRDGBufferRef Buffer, EPixelFormat Format); FRDGTextureUAVRef
	CreateUAV(constFRDGTextureUAVDesc& Desc, ERDGUnorderedAccessViewFlags Flags =
	ERDGUnorderedAccessViewFlags::None);
	FORCEINLINE FRDGTextureUAVRefCreateUAV(FRDGTextureRef Texture,
ERDGUnorderedAccessViewFlags Flags = ERDGUnorderedAccessViewFlags::None);
	FRDGBufferUAVRefCreateUAV(constFRDGBufferUAVDesc& Desc, ERDGUnorderedAccessViewFlags Flags =
	ERDGUnorderedAccessViewFlags::None);
	FORCEINLINE FRDGBufferUAVRefCreateUAV(FRDGBufferRef Buffer, EPixelFormat Format,
ERDGUnorderedAccessViewFlags Flags = ERDGUnorderedAccessViewFlags::None);
	template <typename ParameterStructType> TRDGUniformBufferRef<ParameterStructType>
	CreateUniformBuffer(ParameterStructType* ParameterStruct);



	//Allocate memory, memory is allocated byRDGManage lifecycle.
```

```
        void*Alloc(uint32 SizeInBytes, uint32 AlignInBytes); template
                        <typename PODType>
        PODType* AllocPOD();
        template <typename ObjectType, typename...TArgs> ObjectType*
        AllocObject(TArgs&&... Args); template <typename
        ParameterStructType> ParameterStructType*AllocParameters();


        //Add additional parameters andLambdaofPass.
        template <typename ParameterStructType, typename ExecuteLambdaType>
        FRDGPassRefAddPass(FRDGEventName&& Name,constParameterStructType* ParameterStruct, ERDGPassFlags
Flags, ExecuteLambdaType&& ExecuteLambda);
        //Add no parameters onlyLambdaofPass. template
        <typename ExecuteLambdaType>
        FRDGPassRefAddPass(FRDGEventName&& Name, ERDGPassFlags Flags, ExecuteLambdaType&&
        ExecuteLambda);


        //existBuilderAt the end of execution, extract the texture in the pool to the specified pointer.RDGCreated resources, which will extendGPUThe life of the resource,
until execution, the pointer is filled. If specified, the texture will be converted toAccessFinalstate, otherwise it will be converted tokDefaultAccessFinalstate.
        void QueueTextureExtraction(FRDGTextureRef Texture, TRefCountPtr<IPooledRenderTarget>*
        OutPooledTexturePtr);
        void QueueTextureExtraction(FRDGTextureRef Texture, TRefCountPtr<IPooledRenderTarget>* OutPooledTexturePtr,
ERHIAccess AccessFinal);


        //existBuilderAt the end of execution, extract the buffer to the specified pointer.
        void QueueBufferExtraction(FRDGBufferRef Buffer, TRefCountPtr<FRDGPooledBuffer>* OutPooledBufferPtr);

        void QueueBufferExtraction(FRDGBufferRef Buffer, TRefCountPtr<FRDGPooledBuffer>* OutPooledBufferPtr,
ERHIAccess AccessFinal);


        //Preallocate resources. OnlyRDGThe created resource will force an immediate allocation of the underlying pool resources, effectively promoting it to the external resource. This will increase
memory pressure, but allow the use ofGetPooled{Texture,                Buffer}Query the resources in the pool.     Mainly used to incrementally port code toRDG.
        void    PreallocateTexture(FRDGTextureRef Texture);
        void    PreallocateBuffer(FRDGBufferRef Buffer);


        //Immediately obtain the underlying resource, only allowed for registered or preallocated resources.
        constTRefCountPtr<IPooledRenderTarget>&GetPooledTexture(FRDGTextureRef Texture) const;

        constTRefCountPtr<FRDGPooledBuffer>&GetPooledBuffer(FRDGBufferRef Buffer)const;


        //Set the status after execution.
        void SetTextureAccessFinal(FRDGTextureRef Texture, ERHIAccess Access); void
        SetBufferAccessFinal(FRDGBufferRef Buffer, ERHIAccess Access);

        void    RemoveUnusedTextureWarning(FRDGTextureRef Texture);
        void    RemoveUnusedBufferWarning(FRDGBufferRef Buffer);


        //Execution QueuePass, manage render targets (RHI RenderPassesSetup, resource conversion, and queue texture fetching.
        void Execute();


        //Renders each frame update of the graphics resource pool.
        static void TickPoolElements(); //RDGList
        of commands used.
        FRHICommandListImmediate& RHICmdList;

private:
        static constERHIAccess kDefaultAccessInitial = ERHIAccess::Unknown; static constERHIAccess
        kDefaultAccessFinal = ERHIAccess::SRVMask;
```

```
static const char*constkDefaultUnaccountedCSVStat;

//RDGUsedAsyncComputeList of commands.
FRHIAsyncComputeCommandListImmediate&       RHICmdListAsyncCompute;
FRDGAllocator Allocator;

constFRDGEventName BuilderName;

ERDGPassFlagsOverridePassFlags(constTCHAR* PassName, ERDGPassFlags Flags,bool
bAsyncComputeSupported);
FORCEINLINE FRDGPassHandleGetProloguePassHandle()const; FORCEINLINE
FRDGPassHandleGetEpiloguePassHandle()const;

//RDGObject registry.
FRDGPassRegistry Passes;
FRDGTextureRegistry        Textures;
FRDGBufferRegistry       Buffers;
FRDGViewRegistry Views;
FRDGUniformBufferRegistry          UniformBuffers;

//The croppedPass.
FRDGPassBitArray       PassesToCull;
//Without parametersPass.
FRDGPassBitArray       PassesWithEmptyParameters;

//Track external resources to registered render counterparts for deduplication.
TSortedMap<FRHITexture*, FRDGTexture*, TInlineAllocator<4, SceneRenderingAllocator>> ExternalTextures;

TSortedMap<constFRDGPooledBuffer*, FRDGBuffer*, TInlineAllocator<4,
SceneRenderingAllocator>> ExternalBuffers;

FRDGPass* ProloguePass = nullptr;
FRDGPass* EpiloguePass = nullptr;

//A list of resources to be fetched.
TArray<TPair<FRDGTextureRef, TRefCountPtr<IPooledRenderTarget>*>,                TInlineAllocator<4,
SceneRenderingAllocator>> ExtractedTextures;
TArray<TPair<FRDGBufferRef, TRefCountPtr<FRDGPooledBuffer>*>, TInlineAllocator<4,
SceneRenderingAllocator>> ExtractedBuffers;
//Texture state used for intermediate operations, saved here to avoid reallocation.
FRDGTextureTransientSubresourceStateIndirect ScratchTextureState; EAsyncComputeBudget
AsyncComputeBudgetScope = EAsyncComputeBudget::EAll_4;

//Compile.
void Compile();
//Clean up.
void Clear();

//Start resource conversion.
void BeginResourceRHI(FRDGUniformBuffer* UniformBuffer); void
BeginResourceRHI(FRDGPassHandle, FRDGTexture* Texture); void
BeginResourceRHI(FRDGPassHandle, FRDGTextureSRV* SRV); void
BeginResourceRHI(FRDGPassHandle, FRDGTextureUAV* UAV); void
BeginResourceRHI(FRDGPassHandle, FRDGBuffer* Buffer); void
BeginResourceRHI(FRDGPassHandle, FRDGBufferSRV* SRV); void
BeginResourceRHI(FRDGPassHandle, FRDGBufferUAV* UAV);


//End resource conversion.
```

```
    void EndResourceRHI(FRDGPassHandle, FRDGTexture* Texture, uint32 ReferenceCount); void
    EndResourceRHI(FRDGPassHandle, FRDGBuffer* Buffer, uint32 ReferenceCount);

    // Passinterface.
    void SetupPassInternal(FRDGPass* Pass, FRDGPassHandle PassHandle, ERHIPipeline PassPipeline);

    void   SetupPass(FRDGPass* Pass);
    void   SetupEmptyPass(FRDGPass* Pass);
    void   ExecutePass(FRDGPass* Pass);

    // PassPre-order and post-order.
    void ExecutePassPrologue(FRHIComputeCommandList& RHICmdListPass, FRDGPass* Pass); void
    ExecutePassEpilogue(FRHIComputeCommandList& RHICmdListPass, FRDGPass* Pass);

    //Gather resources and barriers.
    void CollectPassResources(FRDGPassHandle PassHandle);
    void CollectPassBarriers(FRDGPassHandle PassHandle, FRDGPassHandle&
    LastUntrackedPassHandle);

    //IncreasePassrely.
    void AddPassDependency(FRDGPassHandle ProducerHandle, FRDGPassHandle ConsumerHandle);

    //Add post-order conversion.
    void AddEpilogueTransition(FRDGTextureRef Texture, FRDGPassHandle
    LastUntrackedPassHandle);
    void AddEpilogueTransition(FRDGBufferRef Buffer, FRDGPassHandle
    LastUntrackedPassHandle);

    //Added general conversion.
    void AddTransition(FRDGPassHandle PassHandle, FRDGTextureRef Texture,const
FRDGTextureTransientSubresourceStateIndirect& StateAfter, FRDGPassHandle LastUntrackedPassHandle);

    void AddTransition(FRDGPassHandle PassHandle, FRDGBufferRef Buffer,
FRDGSubresourceState StateAfter, FRDGPassHandle LastUntrackedPassHandle);

    void AddTransitionInternal(
        FRDGParentResource*      Resource,
        FRDGSubresourceState     StateBefore,
        FRDGSubresourceState     StateAfter,
        FRDGPassHandle LastUntrackedPassHandle, const
        FRHITransitionInfo& TransitionInfo);

    //Get RenderingPassinformation.
    FRHIRenderPassInfoGetRenderPassInfo(constFRDGPass* Pass)const; //Allocate
    subresources.
    FRDGSubresourceState*AllocSubresource(constFRDGSubresourceState& Other);

#if  RDG_ENABLE_DEBUG
    void VisualizePassOutputs(constFRDGPass* Pass); void
    ClobberPassOutputs(constFRDGPass* Pass);
#endif
};
```

As the driver of the RDG system, FRDGBuilder is responsible for storing data, handling state transitions, automatically managing resource lifecycles and barriers, pruning invalid resources, and

collecting, compiling, and executing Passes, extracting textures or buffers, etc. Its internal execution mechanism is relatively complex, and will be analyzed in detail in subsequent chapters.

11.3 RDG Mechanism

This section will mainly explain the working mechanism, process and principle of RDG, as well as its advantages and characteristics in rendering.

If some students only want to learn how to use RDG, they can skip this chapter and read 11.4 RDG Development directly.

11.3.1 Overview of RDG Mechanism

Rendering Dependency Graph Framework, which sets up Lambda scopes, which are designed as Passes, to issue GPU commands to the RHI using deferred execution. They are created via FRDGBuilder::AddPass(). When a Pass is created, it requires a Shader parameter . It can be any shader parameter, but the framework is most interested in rendering graphics resources.

The structure holding all the Pass parameters should be allocated using FRDGBuider::AllocParameters() to ensure correct lifetime since Lambda execution is deferred.

A render graph resource created with FRDGBuilder::CreateTexture() or FRDGBuilder::CreateBuffer() only records the resource descriptor. When the resource is needed, it will be allocated per graph. The render graph will track the lifecycle of the resource and release and reuse the memory when the remaining passes no longer reference it.

All rendering resources used by the Pass must be in the Pass parameter given by FRDGBuider::AddPass(), because the rendering needs to know which resources each Pass is using.

Resources are only guaranteed to be allocated when a Pass is executed. Therefore, accessing them should only be done within the lambda scope of a Pass created with FRDGBuider::AddPass(). Not listing some resources used by a Pass may cause problems. It is important not to reference more graph resources in the parameters than the pass needs, because this artificially increases the graph information about the life cycle of that resource. This may cause an increase in memory usage or prevent the overlapping execution of passes. An example is ClearUnusedGraphResources(), which can automatically clear resource references that are not used in the shader. If the resource is not used in the pass, a warning will be issued. The lambda scope of a pass execution may occur at any time after FRDGBuilder::AddPass(). For debugging purposes, it may occur directly in AddPass() with Immediate mode. When an error during pass execution, Immediate mode allows you to use the call stack of the pass settings that may contain the cause of the error source. Immediate mode can be enabled via a command line command-rdgimmediateor console variabler.RDG.ImmediateMode=1.

Pooled resource textures generated by legacy code in FPooledRenderTarget can be used in a render graph by using FRDGBuider::RegisterExternalTexture().

With the information of pass dependencies, execution may prioritize different hardware targets, such as prioritizing memory pressure or pass GPU execution concurrency. Therefore, the execution order of passes is not guaranteed. The execution order of passes can only guarantee that work will be executed on intermediate resources, just like immediate mode executes work on the GPU.

Rendermap passes should not modify the state of external data structures, as this may cause edge cases depending on the order in which the passes are executed. Rendermap resources that survive the execution completion (eg viewport back buffer, TAA history...) should be extracted using FRDGBuilder::QueueTextureExtraction(). If a pass is detected to have no use for generating any resource that was scheduled to be extracted or modifying external textures, the pass may not even execute with a warning.

Unless there is a strong technical reason (such as stereo rendering for multiple views at once for VR), do not bundle multiple jobs on different resources in the same Pass. This will end up creating more dependencies on a set of work, and a single job may only need a subset of these dependencies. The scheduler may overlap some of this with other GPU work. This may also keep transient resources allocated for longer, potentially increasing the peak memory pressure of the entire frame.

Although AddPass() only expects lambda scopes to have deferred execution, this does not mean we need to write one. Most cases can be satisfied by using a simpler toolkit (such as FComputeShaderUtils, FPixelShaderUtils).

11.3.2 FRDGBuilder::AddPass

FRDGBuilder::AddPass adds a Pass containing Pass parameters and Lambda to the RDG system. The specific logic is as follows:

```
// Engine\Source\Runtime\RenderCore\Public\RenderGraphBuilder.inl

template <typename ParameterStructType, typename ExecuteLambdaType> FRDGPassRef
FRDGBuilder::AddPass(FRDGEventName&& Name,constParameterStructType* ParameterStruct,
ERDGPassFlags Flags, ExecuteLambdaType&& ExecuteLambda)
{
	using LambdaPassType = TRDGLambdaPass<ParameterStructType, ExecuteLambdaType>;

	(.....)

	//distributeRDG PassExamples.
	FRDGPass* Pass = Allocator.AllocObject<LambdaPassType>(
		MoveTemp(Name),
		ParameterStruct,
		OverridePassFlags(Name.GetTCHAR(), Flags, LambdaPassType::kSupportsAsyncCompute),
		MoveTemp(ExecuteLambda));

	//join inPassList.
	Passes.Insert(Pass);
```

```
    //set upPass.
    SetupPass(Pass);

    returnPass;
}
```

The logic of AddPass is relatively simple. It constructs an FRDGPass instance with the passed data, then adds it to the list and sets the Pass data. The following is the specific logic of SetupPass:

```
void FRDGBuilder::SetupPass(FRDGPass* Pass) {

    //GetPassdata.
    constFRDGParameterStruct PassParameters = Pass->GetParameters(); const
    FRDGPassHandle PassHandle = Pass->GetHandle(); constERDGPassFlags PassFlags =
    Pass->GetFlags(); constERHIPipeline PassPipeline = Pass->GetPipeline();



    boolbPassUAVAccess =false;

    //----Handling Texture State----

    Pass->TextureStates.Reserve(PassParameters.GetTextureParameterCount() +
(PassParameters.HasRenderTargets() ? (MaxSimultaneousRenderTargets +1) :0));
    //Traverse all textures and perform status/data/reference processing on each texture.
    EnumerateTextureAccess(PassParameters, PassFlags, [&](FRDGViewRef TextureView, FRDGTextureRef
Texture, ERHIAccess Access, FRDGTextureSubresourceRange Range)
    {
        constFRDGViewHandle NoUAVBarrierHandle = GetHandleIfNoUAVBarrier(TextureView); const
        EResourceTransitionFlags TransitionFlags = GetTextureViewTransitionFlags(TextureView, Texture);



        auto& PassState = Pass->TextureStates.FindOrAdd(Texture);
        PassState.ReferenceCount++;

        const boolbWholeTextureRange = Range.IsWholeResource(Texture-
> GetSubresourceLayout());
        boolbWholePassState = IsWholeResource(PassState.State);

        // Convert the pass state to subresource dimensionality if we've found a subresource range.

        if(!bWholeTextureRange && bWholePassState) {

            InitAsSubresources(PassState.State, Texture->Layout); bWholePassState
            =false;
        }

        constautoAddSubresourceAccess = [&](FRDGSubresourceState& State) {

            State.Access = MakeValidAccess(State.Access | Access); State.Flags |=
            TransitionFlags;
            State.NoUAVBarrierFilter.AddHandle(NoUAVBarrierHandle); State.Pipeline
            = PassPipeline;
        };

        if(bWholePassState)
        {
```

```
                AddSubresourceAccess(GetWholeResource(PassState.State));
            }
            else
            {
                EnumerateSubresourceRange(PassState.State, Texture->Layout,                Range,
AddSubresourceAccess);
            }

            bPassUAVAccess |= EnumHasAnyFlags(Access, ERHIAccess::UAVMask);
        });

        //----Processing buffer status----

        Pass->BufferStates.Reserve(PassParameters.GetBufferParameterCount()); //Traverse all buffers and
        perform status/data/reference processing on each buffer. EnumerateBufferAccess(PassParameters,
        PassFlags, [&](FRDGViewRef BufferView, FRDGBufferRef Buffer, ERHIAccess Access)

        {
            constFRDGViewHandle NoUAVBarrierHandle = GetHandleIfNoUAVBarrier(BufferView);

            auto& PassState = Pass->BufferStates.FindOrAdd(Buffer);
            PassState.ReferenceCount++;
            PassState.State.Access = MakeValidAccess(PassState.State.Access | Access);
            PassState.State.NoUAVBarrierFilter.AddHandle(NoUAVBarrierHandle); PassState.State.Pipeline =
            PassPipeline;

            bPassUAVAccess |= EnumHasAnyFlags(Access, ERHIAccess::UAVMask);
        });

        Pass->bUAVAccess = bPassUAVAccess;

        const boolbEmptyParameters = !Pass->TextureStates.Num() && !Pass->BufferStates.Num();
        PassesWithEmptyParameters.Add(bEmptyParameters);

        //existGraphicsPipeline,PassCan start/endPassOwn resources. Asynchronous computations are
        orchestrated during compilation. if(PassPipeline == ERHIPipeline::Graphics && !bEmptyParameters) {

            Pass->ResourcesToBegin.Add(Pass); Pass-
            >ResourcesToEnd.Add(Pass);
        }

        //Internal settingsPass.
        SetupPassInternal(Pass, PassHandle, PassPipeline);
}
```

Let's continue parsing SetupPassInternal:

```
void FRDGBuilder::SetupPassInternal(FRDGPass* Pass, FRDGPassHandle PassHandle, ERHIPipeline
PassPipeline)
{
        //Set variousPassFor its own
        handle. Pass->GraphicsJoinPass  =  PassHandle;
        Pass->GraphicsForkPass          = PassHandle;
        Pass->PrologueBarrierPass Pass-   =PassHandle;
        >EpilogueBarrierPass                = PassHandle;

        (.....)
```

```
    //If it is immediate mode and not post-orderPass,
    if(GRDGimmediateMode && Pass != EpiloguePass)
    {
        //SimplymergeStatus redirectionpassstatus, because the graph will not be compiled.

        //TexturedMergestate.
        for(auto& TexturePair : Pass->TextureStates) {

            auto& PassState = TexturePair.Value;
            constuint32 SubresourceCount = PassState.State.Num();
            PassState.MergeState.SetNum(SubresourceCount);
            for(uint32 Index =0; Index < SubresourceCount; ++Index) {

                if(PassState.State[Index].Access != ERHIAccess::Unknown) {

                    PassState.MergeState[Index] = &PassState.State[Index];
                    PassState.MergeState[Index]->SetPass(PassHandle);
                }
            }
        }
        //BufferMergestate.
        for(auto& BufferPair : Pass->BufferStates) {

            auto& PassState = BufferPair.Value;
            PassState.MergeState = &PassState.State;
            PassState.MergeState->SetPass(PassHandle);
        }

        FRDGPassHandle LastUntrackedPassHandle = GetProloguePassHandle(); //collectPass
        resource.
        CollectPassResources(PassHandle); //collect
        Passbarrier.
        CollectPassBarriers(PassHandle, LastUntrackedPassHandle); //Direct
        ExecutionPass.
        ExecutePass(Pass);

    }
}
```

In summary, AddPass will build an instance of RDG Pass based on the passed parameters, then set the texture and buffer data of the Pass, and then use the internal settings of the Pass's dependent Pass handles. If it is in immediate mode, it will redirect the Merge state of the texture and buffer to the Pass state and execute directly.

11.3.3 FRDGBuilder::Compile

The compilation logic of FRDGBuilder is very complex and performs a lot of processing and optimization, as follows:

```
void FRDGBuilder::Compile() {

    uint32 RasterPassCount =0; uint32
    AsyncComputePassCount =0;
```

```
	// PassMark position.
	FRDGPassBitArray	PassesOnAsyncCompute(false, Passes.Num()); PassesOnRaster(
	FRDGPassBitArray	false, Passes.Num()); PassesWithUntrackedOutputs(false,
	FRDGPassBitArray	Passes.Num()); PassesToNeverCull(false, Passes.Num());
	FRDGPassBitArray

	constFRDGPassHandle ProloguePassHandle = GetProloguePassHandle(); const
	FRDGPassHandle EpiloguePassHandle = GetEpiloguePassHandle();

	constautoIsCrossPipeline = [&](FRDGPassHandle A, FRDGPassHandle B) {

		returnPassesOnAsyncCompute[A] != PassesOnAsyncCompute[B];
	};

	constautoIsSortedBefore = [&](FRDGPassHandle A, FRDGPassHandle B) {

		returnA < B;
	};

	constautoIsSortedAfter = [&](FRDGPassHandle A, FRDGPassHandle B) {

		returnA > B;
	};

	//Build producer/consumer dependencies in the graph and build packed metadata bit arrays to facilitate searching for nodes that match specific criteria.PassGet better cache coherence when
	//Search roots are also used for filtering.RHIOutput(eg SHADER_PARAMETER_{BUFFER, TEXTURE}_UAV) ofPassCannot be cropped, nor can any external resources be writtenPass.
	//Resource extraction extends the life cycle to the end (Epilogue) Pass,endPassis always the root of the graph. The preface and epilogue are auxiliaryPass, and therefore will never be eliminated.
	{
		SCOPED_NAMED_EVENT(FRDGBuilder_Compile_Culling_Dependencies, FColor::Emerald);

			//Added pruning dependency.
		constautoAddCullingDependency = [&](FRDGPassHandle& ProducerHandle, FRDGPassHandle
PassHandle, ERHIAccess Access)
		{
			if(Access != ERHIAccess::Unknown) {

				if(ProducerHandle.IsValid()) {

					//IncreasePassrely.
					AddPassDependency(ProducerHandle, PassHandle);
				}

				//	If writable, stores the new producer.
				if	(IsWritableAccess(Access))
				{
					ProducerHandle = PassHandle;
				}
			}
		};

		//Traverse allPass,Process eachPassTexture and buffer states, etc.
		for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
		{
			FRDGPass* Pass = Passes[PassHandle];
```

```
            boolbUntrackedOutputs = Pass->GetParameters().HasExternalOutputs();

            //deal withPassAll texture states of .
            for(auto& TexturePair : Pass->TextureStates) {

                FRDGTextureRef Texture = TexturePair.Key; auto&
                LastProducers = Texture->LastProducers; auto& PassState =
                TexturePair.Value.State;

                const boolbWholePassState = IsWholeResource(PassState); const bool
                bWholeProducers = IsWholeResource(LastProducers);

                //The producer array needs to be at leastpassThe state array is
                the same size. if(bWholeProducers && !bWholePassState) {

                    InitAsSubresources(LastProducers, Texture->Layout);
                }

                //Added pruning dependency.
                for(uint32 Index =0, Count = LastProducers.Num(); Index < Count;
+ + Index)
                {
                    AddCullingDependency(LastProducers[Index], PassHandle,
PassState[bWholePassState ?0: Index].Access);
                }

                bUntrackedOutputs |= Texture->bExternal;
            }

            //deal withPassThe status of all buffers.
            for(auto& BufferPair : Pass->BufferStates) {

                FRDGBufferRef Buffer = BufferPair.Key;
                AddCullingDependency(Buffer->LastProducer,          PassHandle,
BufferPair.Value.State.Access);
                bUntrackedOutputs |= Buffer->bExternal;
            }

            //deal withPassOther tags and data.
            constERDGPassFlags PassFlags = Pass->GetFlags(); const bool
            bAsyncCompute = EnumHasAnyFlags(PassFlags,
ERDGPassFlags::AsyncCompute);
            const boolbRaster = EnumHasAnyFlags(PassFlags, ERDGPassFlags::Raster); const boolbNeverCull =
            EnumHasAnyFlags(PassFlags, ERDGPassFlags::NeverCull);

            PassesOnRaster[PassHandle] = bRaster;
            PassesOnAsyncCompute[PassHandle]      = bAsyncCompute;
            PassesToNeverCull[PassHandle] =       bNeverCull;
            PassesWithUntrackedOutputs[PassHandle]          = bUntrackedOutputs;
            AsyncComputePassCount += bAsyncCompute?1:0;
            RasterPassCount += bRaster ?1:0;
        }

        // prologue/epilogueSet to not track, they are responsible for importing/exporting external
        resources respectively. PassesWithUntrackedOutputs[ProloguePassHand=le]t rue;
        PassesWithUntrackedOutputs[EpiloguePassHandle]                  = true;
```

```
            //Handle clipping dependencies for fetching textures.
            for(constauto& Query : ExtractedTextures) {

                FRDGTextureRef Texture = Query.Key;
                for(FRDGPassHandle& ProducerHandle : Texture->LastProducers) {

                    AddCullingDependency(ProducerHandle, EpiloguePassHandle, Texture-
> AccessFinal);
                }
                Texture->ReferenceCount++;
            }

            //Handle clipping dependencies of fetch buffers.
            for(constauto& Query : ExtractedBuffers) {

                FRDGBufferRef Buffer = Query.Key;
                AddCullingDependency(Buffer->LastProducer,          EpiloguePassHandle,      Buffer-
> AccessFinal);
                Buffer->ReferenceCount++;
            }
        }

        //--------deal withPassCut--------

        if(GRDGCullPasses)
        {
            TArray<FRDGPassHandle, TInlineAllocator<32, SceneRenderingAllocator>> PassStack; //allPassInitialized to
            culling. PassesToCull.Init(true, Passes.Num());


            //collectPassA root list of outputs that are not tracked or marked as never to be removed.Pass.
            for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
            {
                if(PassesWithUntrackedOutputs[PassHandle] || PassesToNeverCull[PassHandle]) {

                    PassStack.Add(PassHandle);
                }
            }

            //Non-recursive stack traversal, using depth-first search, marking each root reachablePassThe node is not
            pruned. while(PassStack.Num()) {

                constFRDGPassHandle PassHandle = PassStack.Pop();

                if(PassesToCull[PassHandle]) {

                    PassesToCull[PassHandle] =false;
                    PassStack.Append(Passes[PassHandle]->Producers);

                #if    STATS
                    - - GRDGStatPassCullCount;
                #endif
                }
            }
        }
        else//Not EnabledPassCut, AllPassInitialized to no clipping. {
```

```
        PassesToCull.Init(false, Passes.Num());
    }

    //--------deal withPassbarrier--------

    //Traverse the filtered graph and compile barriers for each subresource. Some transitions are redundant, e.g.read-to-read. //
        RDGA conservative heuristic is used; choosing not to merge does not necessarily mean that a transformation will be performed.
    //  They are two different steps. The merge state keeps track of the first and lastPassinterval.PassReferences to are also accumulated on each resource. This must
    //  happen after culling, because after cullingPassNo citations can be provided.

    {
        SCOPED_NAMED_EVENT(FRDGBuilder_Compile_Barriers, FColor::Emerald);

        for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
        {
            //Skip the truncated or no parametersPass.
            if(PassesToCull[PassHandle] || PassesWithEmptyParameters[PassHandle]) {

                continue;
            }

            //Merge subresource status.
            constautoMergeSubresourceStates = [&](ERDGParentResourceType ResourceType,
FRDGSubresourceState*& PassMergeState, FRDGSubresourceState*& ResourceMergeState,const
FRDGSubresourceState& PassState)
            {
                //Skip merging of resources with unknown status.
                if(PassState.Access == ERHIAccess::Unknown) {

                    return;
                }

                if(!ResourceMergeState ||
!FRDGSubresourceState::IsMergeAllowed(ResourceType, *ResourceMergeState,                PassState))
                {
                    //Cross-pipeline, non-mergeable state changes require a newpassDependencies for
                    protection. if(ResourceMergeState && ResourceMergeState->Pipeline !=
PassState.Pipeline)
                    {
                        AddPassDependency(ResourceMergeState->LastPass, PassHandle);
                    }

                    //Allocate a new pending merge state and assign it topassstate.
                    ResourceMergeState = AllocSubresource(PassState);
                    ResourceMergeState->SetPass(PassHandle);
                }
                else
                {
                    //mergePassThe state enters the merged
                    state. ResourceMergeState->Access |=      PassState.Access;
                    ResourceMergeState->LastPass              = PassHandle;
                }

                PassMergeState = ResourceMergeState;
            };

            const boolbAsyncComputePass = PassesOnAsyncCompute[PassHandle];
```

```
                //Get the currently processedPassExamples.
                FRDGPass* Pass = Passes[PassHandle];

                //Processing currentPassThe texture state.
                for(auto& TexturePair : Pass->TextureStates) {

                    FRDGTextureRef Texture = TexturePair.Key; auto&
                    PassState = TexturePair.Value;

                    //Increase the number of citations.
                    Texture->ReferenceCount += PassState.ReferenceCount; Texture-
                    >bUsedByAsyncComputePass |= bAsyncComputePass;

                    const boolbWholePassState = IsWholeResource(PassState.State); const bool
                    bWholeMergeState = IsWholeResource(Texture->MergeState);

                    //For simplicity, merge /PassThe state dimensions should
                    match. if(bWholeMergeState && !bWholePassState) {

                        InitAsSubresources(Texture->MergeState,          Texture->Layout);
                    }
                    else if(!bWholeMergeState && bWholePassState) {

                        InitAsWholeResource(Texture->MergeState);
                    }

                    constuint32 SubresourceCount = PassState.State.Num();
                    PassState.MergeState.SetNum(SubresourceCount);

                    //Merge subresource status.
                    for(uint32 Index =0; Index < SubresourceCount; ++Index) {

                        MergeSubresourceStates(ERDGParentResourceType::Texture,
PassState.MergeState[Index], Texture->MergeState[Index], PassState.State[Index]);
                    }
                }

                //Processing currentPassThe buffer status.
                for(auto& BufferPair : Pass->BufferStates) {

                    FRDGBufferRef Buffer = BufferPair.Key; auto&
                    PassState = BufferPair.Value;

                    Buffer->ReferenceCount += PassState.ReferenceCount; Buffer-
                    >bUsedByAsyncComputePass |= bAsyncComputePass;

                    MergeSubresourceStates(ERDGParentResourceType::Buffer,
PassState.MergeState, Buffer->MergeState, PassState.State);
                }
            }
        }

        //Handling asynchronous computationsPass.
        if(AsyncComputePassCount >0) {

            SCOPED_NAMED_EVENT(FRDGBuilder_Compile_AsyncCompute,          FColor::Emerald);
```

```
FRDGPassBitArray PassesWithCrossPipelineProducer(false, Passes.Num());
FRDGPassBitArray PassesWithCrossPipelineConsumer(false, Passes.Num());

//Iterate through the executing activitiesPass, so that for eachPassFind the latest cross-pipeline producer and the earliest cross-pipeline consumer to narrow the search space when constructing the asynchronous computing overlap area later.
for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
{
    if(PassesToCull[PassHandle] || PassesWithEmptyParameters[PassHandle]) {

        continue;
    }

    FRDGPass* Pass = Passes[PassHandle];

    //Traverse the producers and process the reference relationship between producers and consumers.
    for(FRDGPassHandle ProducerHandle : Pass->GetProducers()) {

        constFRDGPassHandle ConsumerHandle = PassHandle;

        if(!IsCrossPipeline(ProducerHandle, ConsumerHandle)) {

            continue;
        }

        FRDGPass* Consumer = Pass;
        FRDGPass* Producer = Passes[ProducerHandle];

        // Find the earliest consumer on another pipe for a producer.
        if (Producer->CrossPipelineConsumer.IsNull() ||
IsSortedBefore(ConsumerHandle, Producer->CrossPipelineConsumer))
        {
            Producer->CrossPipelineConsumer = PassHandle;
            PassesWithCrossPipelineConsumer[ProducerHandle] = true;
        }

        // Find the latest producer on another pipe for a consumer.
        if (Consumer->CrossPipelineProducer.IsNull() ||
IsSortedAfter(ProducerHandle, Consumer->CrossPipelineProducer))
        {
            Consumer->CrossPipelineProducer = ProducerHandle;
            PassesWithCrossPipelineProducer[ConsumerHandle] = true;
        }
    }
}

//Built for asynchronous computationfork / joinOverlapping regions, used for fencing and resource allocation/reclaiming.fork/joinBefore completion, asynchronous calculation PassTheir resource references cannot be allocated/released because the two pipelines are running in parallel. Therefore, all resource lifecycles of the asynchronous computation are extended to the entire asynchronous region.

constautoIsCrossPipelineProducer = [&](FRDGPassHandle A) {

    returnPassesWithCrossPipelineConsumer[A];
};

constautoIsCrossPipelineConsumer = [&](FRDGPassHandle A) {

    returnPassesWithCrossPipelineProducer[A];
```

```
};

//Find producers across pipelines.
constautoFindCrossPipelineProducer = [&](FRDGPassHandle PassHandle) {

    FRDGPassHandle LatestProducerHandle = ProloguePassHandle;
    FRDGPassHandle ConsumerHandle = PassHandle;

    //It is expected to find the latest producer on other pipelines in order to establish a fork point.NProducer channels consumeNPersonal Information
source, so only the last one matters.
    while(ConsumerHandle != Passes.Begin()) {

        if(!PassesToCull[ConsumerHandle] && !IsCrossPipeline(ConsumerHandle, &&
PassHandle      IsCrossPipelineConsumer(ConsumerHandle))
        {
            constFRDGPass* Consumer = Passes[ConsumerHandle];

            if    (IsSortedAfter(Consumer->CrossPipelineProducer,
LatestProducerHandle))
            {
                LatestProducerHandle = Consumer->CrossPipelineProducer;
            }
        }
        - - ConsumerHandle;
    }

    returnLatestProducerHandle;
};

//Find consumers across pipelines.
constautoFindCrossPipelineConsumer = [&](FRDGPassHandle PassHandle) {

    check(PassHandle != EpiloguePassHandle);

    FRDGPassHandle EarliestConsumerHandle = EpiloguePassHandle;
    FRDGPassHandle ProducerHandle = PassHandle;

    //It is desirable to find the earliest consumer on the other pipe, since this establishes a connection point between the pipes.N
Consumers produce, so only the first consumer that executes is concerned.
    while(ProducerHandle != Passes.End()) {

        if(!PassesToCull[ProducerHandle] && !IsCrossPipeline(ProducerHandle, &&
PassHandle      IsCrossPipelineProducer(ProducerHandle))
        {
            constFRDGPass* Producer = Passes[ProducerHandle];

            if(IsSortedBefore(Producer->CrossPipelineConsumer,
EarliestConsumerHandle))
            {
                EarliestConsumerHandle = Producer->CrossPipelineConsumer;
            }
        }
        + +ProducerHandle;
    }

    returnEarliestConsumerHandle;
};
```

```
//The graphicsPassInserting into asynchronous computationPassIn the fork of .
constautoInsertGraphicsToAsyncComputeFork = [&](FRDGPass* GraphicsPass, FRDGPass*
    AsyncComputePass)
{
    FRDGBarrierBatchBegin& EpilogueBarriersToBeginForAsyncCompute = GraphicsPass-
> GetEpilogueBarriersToBeginForAsyncCompute(Allocator);

    GraphicsPass->bGraphicsFork =1;
    EpilogueBarriersToBeginForAsyncCompute.SetUseCrossPipelineFence();

    AsyncComputePass->bAsyncComputeBegin =1;
    AsyncComputePass-
> GetPrologueBarriersToEnd(Allocator).AddDependency(&EpilogueBarriersToBeginForAsyncCompute
) ;
};

//Asynchronous calculationPassInsert into drawingPassin the process of merging.
constautoInsertAsyncToGraphicsComputeJoin = [&](FRDGPass* AsyncComputePass, FRDGPass*
    GraphicsPass
{
    FRDGBarrierBatchBegin& EpilogueBarriersToBeginForGraphics = AsyncComputePass-
> GetEpilogueBarriersToBeginForGraphics(Allocator);

    AsyncComputePass->bAsyncComputeEnd =1;
    EpilogueBarriersToBeginForGraphics.SetUseCrossPipelineFence();

    GraphicsPass->bGraphicsJoin =1;
    GraphicsPass-
> GetPrologueBarriersToEnd(Allocator).AddDependency(&EpilogueBarriersToBeginForGraphics); };


FRDGPass* PrevGraphicsForkPass = nullptr; FRDGPass*
PrevGraphicsJoinPass = nullptr; FRDGPass*
PrevAsyncComputePass = nullptr;

//Traverse allPass,Extending the life cycle of resources and processing graphicsPassand asynchronous computationPassThe
intersection and merge nodes. for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
{
    if(!PassesOnAsyncCompute[PassHandle] || PassesToCull[PassHandle]) {

        continue;
    }

    FRDGPass* AsyncComputePass = Passes[PassHandle];

    //Find the forkPassand mergePass.
    constFRDGPassHandle GraphicsForkPassHandle =
FindCrossPipelineProducer(PassHandle);
    constFRDGPassHandle GraphicsJoinPassHandle =
FindCrossPipelineConsumer(PassHandle);

    AsyncComputePass->GraphicsForkPass      =  GraphicsForkPassHandle;
    AsyncComputePass->GraphicsJoinPass      =  GraphicsJoinPassHandle;

    FRDGPass* GraphicsForkPass = Passes[GraphicsForkPassHandle]; FRDGPass*
    GraphicsJoinPass = Passes[GraphicsJoinPassHandle];
```

```
                //Extend the lifetime of resources used in asynchronous computations tofork/join
                GraphicsPass. GraphicsForkPass->ResourcesToBegin.Add(AsyncComputePass);
                GraphicsJoinPass->ResourcesToEnd.Add(AsyncComputePass);

                //Forking the graphPassInsert into asynchronous computation forkPass.
                if(PrevGraphicsForkPass != GraphicsForkPass) {

                    InsertGraphicsToAsyncComputeFork(GraphicsForkPass,          AsyncComputePass);
                }

                //Combining asynchronous computationsPassInsert into drawing mergePass.
                if(PrevGraphicsJoinPass != GraphicsJoinPass && PrevAsyncComputePass) {

                    InsertAsyncToGraphicsComputeJoin(PrevAsyncComputePass,
PrevGraphicsJoinPass);
                }

                PrevAsyncComputePass    =  AsyncComputePass;
                PrevGraphicsForkPass    =  GraphicsForkPass;
                PrevGraphicsJoinPass    =  GraphicsJoinPass;
            }

            //  The last asynchronous computation in the diagramPassNeed to connect manually
            if  epilogue pass. (PrevAsyncComputePass)
            {
                InsertAsyncToGraphicsComputeJoin(PrevAsyncComputePass,          EpiloguePass);
                PrevAsyncComputePass->bAsyncComputeEndExecute =1;
            }
        }

        //Traverse all graphics pipelinesPass,And merge all the sameRTRasterizationPassto the sameRHIRendering
        Passmiddle. if(GRDGMergeRenderPasses && RasterPassCount >0) {

            SCOPED_NAMED_EVENT(FRDGBuilder_Compile_RenderPassMerge, FColor::Emerald);

            TArray<FRDGPassHandle, SceneRenderingAllocator> PassesToMerge; FRDGPass*
            PrevPass = nullptr;
            constFRenderTargetBindingSlots* PrevRenderTargets = nullptr;

            constautoCommitMerge = [&] {

                if(PassesToMerge.Num()) {

                    constFRDGPassHandle FirstPassHandle = PassesToMerge[0]; const
                    FRDGPassHandle LastPassHandle = PassesToMerge.Last();

                    //Given aPassThe intervals are merged into a single renderPass: [B, X, X, X, X, E],startPass(B)and
FinishPass(E)Will call separatelyBeginRenderPass/EndRenderPass.
                    //in addition,beginwill process all prologue barriers for the entire merge interval,endAll tail barriers will be processed, which can avoid rendering
Resource conversion within channels, and batch resource conversion more efficiently.
                    //Assume that filtering from the merged set has been done during traversalPassThe dependencies between them.

                    // (B)is the first in the merge sequencePass. {

                        FRDGPass* Pass = Passes[FirstPassHandle]; Pass-
                        >bSkipRenderPassEnd             = 1;
                        Pass->EpilogueBarrierPass        = LastPassHandle;
                    }
```

```
                    // (X)is in the middlePass.
                    for(int32 PassIndex =1, PassCount = PassesToMerge.Num() -1; PassIndex <
PassCount;      + + PassIndex)
                    {
                        constFRDGPassHandle PassHandle = PassesToMerge[PassIndex]; FRDGPass*
                        Pass = Passes[PassHandle];
                        Pass->bSkipRenderPassBegin Pass-=1;
                        >bSkipRenderPassEnd Pass-       =  1;
                        >PrologueBarrierPass Pass-        =FirstPassHandle;
                        >EpilogueBarrierPass                = LastPassHandle;
                    }

                    // (E)It is the last one in the merge sequence.Pass.
                    {
                        FRDGPass* Pass = Passes[LastPassHandle]; Pass-
                        >bSkipRenderPassBegin                = 1;
                        Pass->PrologueBarrierPass        = FirstPassHandle;
                    }

                #if   STATS
                    GRDGStatRenderPassMergeCount      + =  PassesToMerge.Num();
                #endif
                }
                PassesToMerge.Reset();
                PrevPass = nullptr;
                PrevRenderTargets =       nullptr;
            };

            //Iterate over all rastersPass,Merge all identicalRTofPassTo the same renderingPassmiddle.
            for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
            {
                //   Skip the clippedPass.
                if   (PassesToCull[PassHandle])
                {
                    continue;
                }

                //   It is a gratingPassJust processed.
                if   (PassesOnRaster[PassHandle])
                {
                    FRDGPass* NextPass = Passes[PassHandle];

                    //   User controlled renderingPassofPassCannot be used with otherPassMerge, RasterUAVofPassDue to potential interdependence,
and.
                    if   (EnumHasAnyFlags(NextPass->GetFlags(), ERDGPassFlags::SkipRenderPass)
||  NextPass->bUAVAccess)
                    {
                        CommitMerge();
                        continue;
                    }

                    //   Graphics forkPassCannot be used with previous grating
                    if   Passmerge. (NextPass->bGraphicsFork)
                    {
                        CommitMerge();
                    }
```

```
                    constFRenderTargetBindingSlots& RenderTargets = NextPass-
> GetParameters().GetRenderTargets();

                    if(PrevPass)
                    {
                        //  contrastRT,To determine whether they can be merged.
                        if  (PrevRenderTargets->CanMergeBefore(RenderTargets)
                        #if  WITH_MGPU
                            && PrevPass->GPUMask == NextPass->GPUMask
                        #endif
                            )
                        {
                            //  If possible, addPassarrivePassesToMergeList. (!
                            if  PassesToMerge.Num())
                            {
                                PassesToMerge.Add(PrevPass->GetHandle());
                            }
                            PassesToMerge.Add(PassHandle);
                        }
                        else
                        {
                            CommitMerge();
                        }
                    }

                    PrevPass = NextPass;
                    PrevRenderTargets = &RenderTargets;
                }
                else  if(!PassesOnAsyncCompute[PassHandle])
                {
                    //  Non-raster graphics pipelinePasswill makeRTThe merge is invalid.
                    CommitMerge();
                }
            }

            CommitMerge();
        }
}
```

The above code shows that the logic during RDG compilation is very complex and involves many steps, including building the dependency relationship between producers and consumers, determining various tags such as Pass clipping, adjusting the resource life cycle, clipping Pass, processing Pass resource conversion and barriers, processing the dependency and reference relationship of asynchronous computing Pass, finding and establishing forked and merged Pass nodes, and merging all rasterized Passes with the same specific rendering target.

The above code also involves some important interfaces, which are analyzed one by one below:

```
//IncreasePassDependency, the producer (ProducerHandle)Add to Consumer(ConsumerHandle)List of producers (
Producersmiddle. void FRDGBuilder::AddPassDependency(FRDGPassHandle ProducerHandle, FRDGPassHandle
ConsumerHandle)
{
    FRDGPass* Consumer = Passes[ConsumerHandle];
```

```
        auto& Producers = Consumer->Producers;
        if(Producers.Find(ProducerHandle) == INDEX_NONE) {

                Producers.Add(ProducerHandle);
        }
};

//Initialized as a subresource.
template <typename ElementType, typename AllocatorType>
inlinevoid InitAsSubresources(TRDGTextureSubresourceArray<ElementType, AllocatorType>& SubresourceArray,const
FRDGTextureSubresourceLayout& Layout,constElementType& Element = {})

{
        constuint32 SubresourceCount = Layout.GetSubresourceCount();
        SubresourceArray.SetNum(SubresourceCount,false);
        for(uint32 SubresourceIndex =0; SubresourceIndex < SubresourceCount;
+ + SubresourceIndex)
        {
                SubresourceArray[SubresourceIndex] = Element;
        }
}

//Initialize to the whole resource.
template <typename ElementType, typename AllocatorType>
FORCEINLINEvoid InitAsWholeResource(TRDGTextureSubresourceArray<ElementType, AllocatorType>&
SubresourceArray,constElementType& Element = {}) {

        SubresourceArray.SetNum(1,false
        SubresourceArray[0] = Element;
}

//Allocate subresources.
FRDGSubresourceState*FRDGBuilder::AllocSubresource(constFRDGSubresourceState& Other) {

        FRDGSubresourceState* State = Allocator.AllocPOD<FRDGSubresourceState>();
        * State    = Other;
        return     State;
}
```

11.3.4 FRDGBuilder::Execute

After collecting Pass (AddPass) and compiling the rendering as mentioned above, the rendering can be executed, which is undertaken by FRDGBuider::Execute:

```
void FRDGBuilder::Execute() {

        SCOPED_NAMED_EVENT(FRDGBuilder_Execute, FColor::Emerald);

        //Before compiling, at the end of the graph createepilogue pass.
        EpiloguePass = Passes.Allocate<FRDGSentinelPass>(Allocator, RDG_EVENT_NAME("Graph Epilogue"));

        SetupEmptyPass(EpiloguePass);

        constFRDGPassHandle ProloguePassHandle = GetProloguePassHandle(); const
        FRDGPassHandle EpiloguePassHandle = GetEpiloguePassHandle();
```

```
    FRDGPassHandle LastUntrackedPassHandle = ProloguePassHandle;

    //  Not immediate mode.
    if  (!GRDGImmediateMode)
    {
        //Compile before execution, see11.3.3chapter.
        Compile();

        {
            SCOPE_CYCLE_COUNTER(STAT_RDG_CollectResourcesTime);

            //collectPassresource.
            for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
            {
                if(!PassesToCull[PassHandle]) {

                    CollectPassResources(PassHandle);
                }
            }

            //End texture extraction.
            for(constauto& Query : ExtractedTextures) {

                EndResourceRHI(EpiloguePassHandle, Query.Key,               1);
            }

            //End buffer extraction.
            for(constauto& Query : ExtractedBuffers) {

                EndResourceRHI(EpiloguePassHandle, Query.Key,               1);
            }
        }

        //collectPassbarrier.
        {
            SCOPE_CYCLE_COUNTER(STAT_RDG_CollectBarriersTime);

            for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
            {
                if(!PassesToCull[PassHandle]) {

                    CollectPassBarriers(PassHandle, LastUntrackedPassHandle);
                }
            }
        }
    }

    //Iterate over all textures, adding a tail transition to each texture.
    for(FRDGTextureHandle TextureHandle = Textures.Begin(); TextureHandle != Textures.End(); +
    +TextureHandle)
    {
        FRDGTextureRef Texture = Textures[TextureHandle];

        if(Texture->GetRHIUnchecked()) {

            AddEpilogueTransition(Texture, LastUntrackedPassHandle);
```

```
                Texture->Finalize();
            }
        }

        //Iterate over all buffers, adding a tail transition to each buffer.
        for(FRDGBufferHandle BufferHandle = Buffers.Begin(); BufferHandle != Buffers.End();
+ + BufferHandle)
        {
            FRDGBufferRef Buffer = Buffers[BufferHandle];

            if(Buffer->GetRHIUnchecked()) {

                AddEpilogueTransition(Buffer, Buffer-    LastUntrackedPassHandle);
                >Finalize();
            }
        }

        //    implementPass.
        if    (!GRDGImmediateMode)
        {
            QUICK_SCOPE_CYCLE_COUNTER(STAT_FRDGBuilder_Execute_Passes);

            for(FRDGPassHandle PassHandle = Passes.Begin(); PassHandle != Passes.End();
+ + PassHandle)
            {
                //    Perform non-croppedPass.
                if    (!PassesToCull[PassHandle])
                {
                    ExecutePass(Passes[PassHandle]);
                }
            }
        }
        else
        {
            ExecutePass(EpiloguePass);
        }

        RHICmdList.SetGlobalUniformBuffers({});

#if    WITH_MGPU
        (......)
#endif

        //Perform a texture fetch.
        for(constauto& Query : ExtractedTextures) {

            * Query.Value = Query.Key->PooledRenderTarget;
        }

        //Perform a buffer fetch.
        for(constauto& Query : ExtractedBuffers) {

            * Query.Value = Query.Key->PooledBuffer;
        }

        //Clean up.
```

```
    Clear();
}
```

The execution process involves the ExecutePass interface for executing Pass, and its logic is as follows:

```
void FRDGBuilder::ExecutePass(FRDGPass* Pass) {

    QUICK_SCOPE_CYCLE_COUNTER(STAT_FRDGBuilder_ExecutePass);
    SCOPED_GPU_MASK(RHICmdList, Pass->GPUMask);
    IF_RDG_CPU_SCOPES(CPUScopeStacks.BeginExecutePass(Pass));

    //useGPUscope.
#if  RDG_GPU_SCOPES
    const boolbUsePassEventScope = Pass != EpiloguePass && Pass != ProloguePass; if(bUsePassEventScope)
    {

        GPUScopeStacks.BeginExecutePass(Pass);
    }
#endif

#if  WITH_MGPU
    if(!bWaitedForTemporalEffect && NameForTemporalEffect != NAME_None) {

        RHICmdList.WaitForTemporalEffect(NameForTemporalEffect);
        bWaitedForTemporalEffect =true;
    }
#endif

    //implementpassThe order:1. prologue -> 2. passMain body ->3.epilogue. //The entire process is
    executed using a list of commands on the specified pipe.
    FRHIComputeCommandList& RHICmdListPass = (Pass->GetPipeline() ==
    ERHIPipeline::AsyncCompute)
        ?static_cast<FRHIComputeCommandList&>(RHICmdListAsyncCompute)
        : RHICmdList;

    // 1.implementprologue
    ExecutePassPrologue(RHICmdListPass, Pass);

    // 2.implementpassmain body Pass-
    >Execute(RHICmdListPass);

    // 3.implementepilogue
    ExecutePassEpilogue(RHICmdListPass, Pass);

#if  RDG_GPU_SCOPES
    if(bUsePassEventScope) {

        GPUScopeStacks.EndExecutePass(Pass);
    }
#endif

    //  Once the asynchronous calculation is completed, it is dispatched immediately.
    if  (Pass->bAsyncComputeEnd)
    {
        FRHIAsyncComputeCommandListImmediate::ImmediateDispatch(RHICmdListAsyncCompute);
```

```
        }

        //If it is in debug mode and non-asynchronous computing, submit the command and refresh toGPU,Then waitGPU
        Processing completed. if(GRDGDebugFlushGPU && !GRDGAsyncCompute) {

            RHICmdList.SubmitCommandsAndFlushGPU();
            RHICmdList.BlockUntilGPUIdle();
        }
}
```

There are three main steps to execute a Pass: 1. prologue, 2. pass body, 3. epilogue. Their execution logic is as follows:

```
// 1. prologue
void FRDGBuilder::ExecutePassPrologue(FRHIComputeCommandList& RHICmdListPass, FRDGPass* Pass)

{
        //    Commit the preorder start barrier.
        if    (Pass->PrologueBarriersToBegin)
        {
            Pass->PrologueBarriersToBegin->Submit(RHICmdListPass);
        }

        //    Submit the preorder end barrier.
        if    (Pass->PrologueBarriersToEnd)
        {
            Pass->PrologueBarriersToEnd->Submit(RHICmdListPass);
        }

        //Since the access check will allowRDGCalling on resourcesGetRHI, so the uniform buffer will be initialized
        when first used. Pass->GetParameters().EnumerateUniformBuffers([&](FRDGUniformBufferRef {                    UniformBuffer)

            BeginResourceRHI(UniformBuffer);
        });

        //Set asynchronous computation budget (Budget).
        if(Pass->GetPipeline() == ERHIPipeline::AsyncCompute) {

            RHICmdListPass.SetAsyncComputeBudget(Pass->AsyncComputeBudget);
        }

        constERDGPassFlags PassFlags = Pass->GetFlags();

        if(EnumHasAnyFlags(PassFlags, ERDGPassFlags::Raster)) {

            if(!EnumHasAnyFlags(PassFlags, ERDGPassFlags::SkipRenderPass) && !Pass-
> SkipRenderPassBegin()) {

                //Calling the command queueBeginRenderPassinterface.
                static_cast<FRHICommandList&>(RHICmdListPass).BeginRenderPass(Pass-
> GetParameters().GetRenderPassInfo(), Pass->GetName()); }

        }
}

// 2. passmain body
```

```
void FRDGPass::Execute(FRHIComputeCommandList& RHICmdList) {

	QUICK_SCOPE_CYCLE_COUNTER(STAT_FRDGPass_Execute); //Set up the
	uniform buffer.
	RHICmdList.SetGlobalUniformBuffers(ParameterStruct.GetGlobalUniformBuffers()); //implementPass
	Implementation of.
	ExecuteImpl(RHICmdList);
}

void TRDGLambdaPass::ExecuteImpl(FRHIComputeCommandList& RHICmdList) override {

	//implementLambda.
	ExecuteLambda(static_cast<TRHICommandList&>(RHICmdList));
}

// 3. epilogue
void FRDGBuilder::ExecutePassEpilogue(FRHIComputeCommandList& RHICmdListPass, FRDGPass* Pass)

{
	QUICK_SCOPE_CYCLE_COUNTER(STAT_FRDGBuilder_ExecutePassEpilogue);

	constERDGPassFlags PassFlags = Pass->GetFlags();

	//Calling the command queueEndRenderPass.
	if(EnumHasAnyFlags(PassFlags, ERDGPassFlags::Raster) && !EnumHasAnyFlags(PassFlags,
ERDGPassFlags::SkipRenderPass) && !Pass->SkipRenderPassEnd())
	{
		static_cast<FRHICommandList&>(RHICmdListPass).EndRenderPass();
	}

	//Abandon resource conversion.
	for(FRHITexture* Texture : Pass->TexturesToDiscard) {

		RHIDiscardTransientResource(Texture);
	}

	//Get(AcquireConvert resources.
	for(FRHITexture* Texture : Pass->TexturesToAcquire) {

		RHIAcquireTransientResource(Texture);
	}

	constFRDGParameterStruct PassParameters = Pass->GetParameters();

	//  Submit an epilogue barrier for use in the graphics pipeline.
	if  (Pass->EpilogueBarriersToBeginForGraphics)
	{
		Pass->EpilogueBarriersToBeginForGraphics->Submit(RHICmdListPass);
	}

	//  Submit a tail barrier for asynchronous computation.
	if  (Pass->EpilogueBarriersToBeginForAsyncCompute)
	{
		Pass->EpilogueBarriersToBeginForAsyncCompute->Submit(RHICmdListPass);
	}
}
```

As can be seen above, during execution, all Passes are compiled first, and then the prequel, main body, and postqueue of Pass are executed in sequence, which is equivalent to scattering BeginRenderPass, execution rendering code, and EndRenderPass of the command queue between them. The execution body of Pass is actually very simple, which is to call the Lambda instance of the Pass and pass in the command queue instance used.

The final stage of execution is cleanup, see the analysis below:

```
void FRDGBuilder::Clear() {

    //Clean up external resources.
    ExternalTextures.Empty();
    ExternalBuffers.Empty(); //Clean
    up extracted resources.
    ExtractedTextures.Empty();
    ExtractedBuffers.Empty(); //Clean
    up the main body data.
    Passes.Clear();
    Views.Clear();
    Textures.Clear();
    Buffers.Clear();
    //Clean up uniform buffers and allocators.

    UniformBuffers.Clear();
    Allocator.ReleaseAll();
}
```

11.3.5 Summary of RDG Mechanism

UE's RDG system is executed on the rendering thread by default. Although RDG Passes with the same RT will be merged, it does not mean that they will be executed in parallel, but in series. Under normal circumstances, the end of each Pass execution will not be submitted immediately and wait for the GPU to complete, but if it is debug mode and non-asynchronous calculation, it will be.

FRDGBuilder does not have a globally unique instance. It is usually declared as a local variable to complete the entire process of Pass collection, compilation and execution within a certain life cycle. The modules that declare FRDGBuilder instances are: distance field, render texture, scene renderer, scene capturer, ray tracing, post-processing, hair, virtual texture, etc .

The execution cycle of FRDGBuider can be divided into four stages: collecting passes, compiling passes, executing passes, and cleaning up.

The Pass collection phase mainly collects all Passes (Lambda) of the rendering module that can generate RHI rendering instructions. After collection, they are not executed immediately but will be delayed. The AddPass step is to first create an instance of FRDGPass and add it to the Pass list, and then execute SetupPass. The SetupPass process mainly handles the status, references, dependencies, and tags of textures and buffers.

The Pass compilation phase is more complicated and involves many steps, including building the dependency relationship between producers and consumers, determining various tags such as Pass

clipping, adjusting the resource lifecycle, clipping Pass, processing Pass resource conversion and barriers, processing the dependency and reference relationship of asynchronous computing Pass, finding and establishing forked and merged Pass nodes, and merging all rasterized Passes with the same specific rendering target.

During the Pass execution phase, compilation is performed first, and then all eligible Passes are executed according to the compilation results. When executing a single Pass, the prologue, main body, and follow-up are executed in sequence, which is equivalent to executing BeginRenderPass of the command queue, executing the Pass main body (Lambda) rendering code, and EndRenderPass.
When executing the Pass main body, the process is simple, which is to call the Lambda instance of the Pass.
The final stage is the cleanup phase, which cleans or resets all data and memory within the FRDGBuider instance.
During the entire process of FRDGBuilder execution, compared with directly using RHICommandList, the features and optimization measures of FRDGBuilder are as follows:

- All resources referenced by RDG Pass should be allocated or managed by RDG, even if they are externally registered resources, their lifecycle should be guaranteed during RDG. RDG will automatically manage the lifecycle of resources, delay their lifecycle during crossover and merge passes, and release and reuse them when there are no references.
- Resources are not allocated in an instant response, but are allocated or created when they are first used.
- It has the concept of subresources, and integrates them into large resource blocks through reasonable layout. It can dispatch one subresource to another, and can also automatically create views and aliases of subresources, and create resource aliases created by future rendering passes. It can effectively manage resource allocation, release, and reuse, reduce overall memory usage and memory fragmentation, reduce CPU and GPU IO, and improve memory usage efficiency.
- Manage RDG Pass with FRDGBuilder instance as the unit, automatically sort, reference, fork and merge Pass, handle resource references and dependencies of Pass, and cut useless Pass and resources. RDG can also correctly handle the dependencies and references between Graphics Pass and Async Compute Pass, connect them in order according to the DAG graph, and correctly handle their cross-use of resources and state transitions.
- RDG can merge the rendering of RDG Pass, provided that these RDG Pass use the same rendering texture. This can reduce the calls of Begin/EndRenderPass of the RHI layer, and reduce the conversion of RHI rendering commands and resource states.
- RDG can automatically handle resource dependencies, barriers, and state transitions between passes, discard invalid state transitions (such as read-to-read, write-to-write), and merge and batch convert resource states to further reduce the number of rendering instructions.

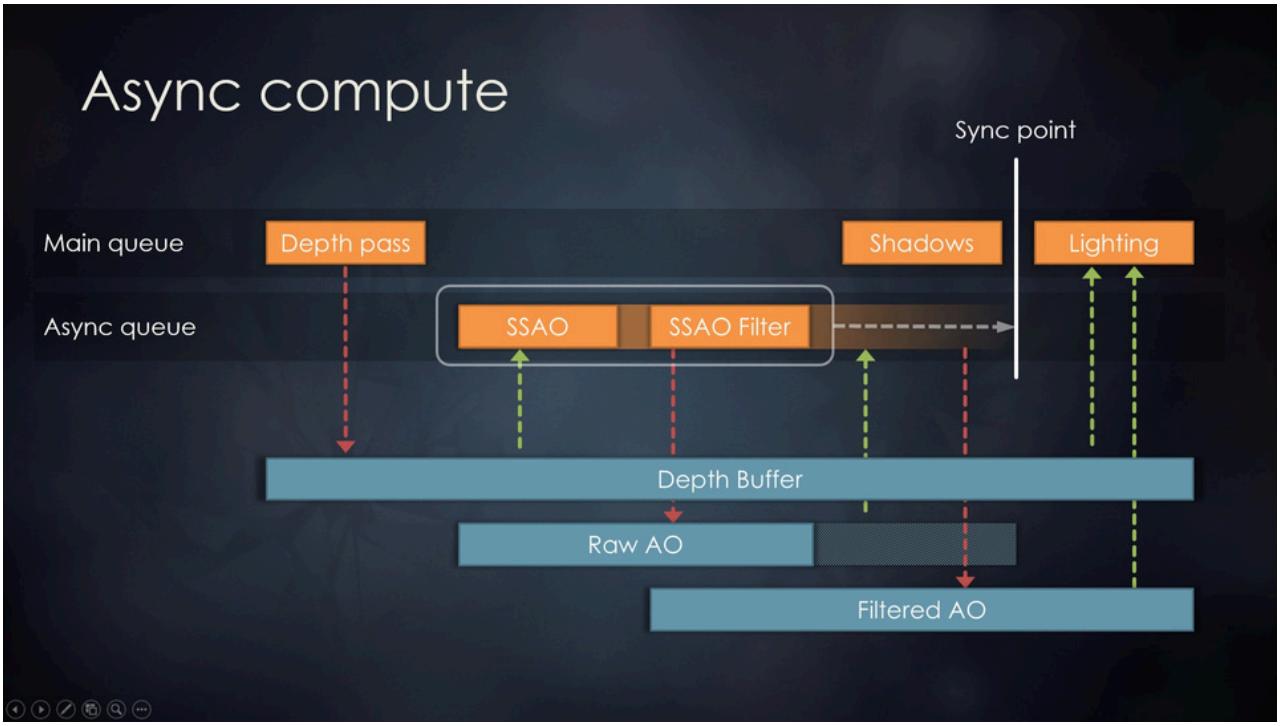
- The execution of RDG Pass occurs in the rendering thread and is serial, rather than being executed in parallel using TaskGraph.

  > Theoretically, it can be executed in parallel, but this is just a guess. Whether it is feasible still needs to be verified in practice.

- RDG has rich debugging modes and information, supports immediate execution mode, and helps developers quickly locate problems, reducing the time and difficulty of bug checking.

Of course, FRDGBuider also has some side effects:

- A layer of concepts and encapsulation is added to the rendering system, which increases the complexity of the rendering layer and the learning cost.
- Increases development complexity. Due to delayed execution, some bugs cannot get immediate feedback.
- The lifecycle of some passes or resources may be extended additionally.



From the Frostbite asynchronous calculation diagram. The Pass of SSAO and SSAO Filter are placed in the asynchronous queue, which will write and read the texture of Raw AO. Even if it ends before the synchronization point, the life cycle of Raw AO will still be extended to the synchronization point.

11.4 RDG Development

This chapter mainly explains how to use the UE's RDG system.

11.4.1 Creating RDG Resources

The sample code for creating RDG resources (textures, buffers, UAVs, SRVs, etc.) is as follows:

```
//----createRDGTexture Demonstration---- //create
RDGTexture Description
FRDGTextureDesc TextureDesc = Input.Texture->Desc;
TextureDesc.Reset();
TextureDesc.Format =          PF_FloatRGBA;
TextureDesc.ClearValue          = FClearValueBinding::None;
TextureDesc.Flags          &=~TexCreate_DepthStencilTargetable;
TextureDesc.Flags          |= TexCreate_RenderTargetable;
TextureDesc.Extent          = OutputViewport.Extent;
//createRDGTexture.
FRDGTextureRef MyRDGTexture = GraphBuilder.CreateTexture(TextureDesc, TEXT(
"MyRDGTexture"));


//----createRDGTextureUAVdemonstration----
FRDGTextureUAVRef MyRDGTextureUAV = GraphBuilder.CreateUAV(MyRDGTexture);

//----createRDGTextureSRVdemonstration---- FRDGTextureSRVRef MyRDGTextureSRV =
GraphBuilder.CreateSRV(FRDGTextureSRVDesc::CreateWithPixelFormat(MyRDGTexture, PF_FloatRGBA));
```

Before creating a resource such as a texture, you need to create a resource descriptor. When creating a resource UAV and SRV, you can use the previously created resource as an instance to achieve the purpose of reuse. To create an SRV, you need to use the resource instance as a parameter of the descriptor. After creating the descriptor, create the SRV.

The above code takes the creation of texture related resources as an example. The creation of buffer is similar, so no more examples are given.

11.4.2 Registering External Resources

The resources in the previous section are created and managed by RDG, and the life cycle of the resources is also the responsibility of RDG. If we already have resources that are not created by RDG, can we use them in RDG? The answer is yes , through the FRDGBuilder::RegisterExternalXXX interface, we can register external resources into the RDG system. Let's take registering a texture as an example:

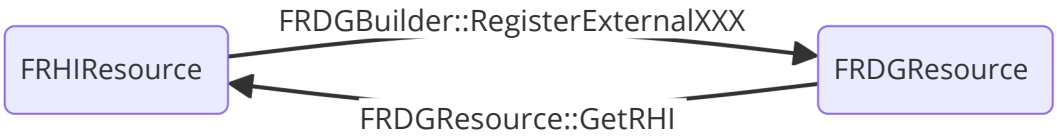
```
//existRDGExternal creationRHIresource.
FRHIResourceCreateInfo CreateInfo;
FTexture2DRHIRef MyRHITexture = RHICreateTexture2D(1024,768, PF_B8G8R8A8,1,1,
TexCreate_CPUReadback, CreateInfo);


//Create externalRHIResource registrationRDGresource.
FRDGTextureRef MyExternalRDGTexture = GraphBuilder.RegisterExternalTexture(MyRHITexture);
```

It should be noted that RDG cannot control and manage the life cycle of externally registered resources. It is necessary to ensure that the life cycle of external resources is in a normal state during the use of RDG, otherwise it will cause exceptions or even program crashes .

If you want to get an instance of an RHI resource from an RDG resource, the following code can achieve this:

```
FRHITexture* MyRHITexture = MyRDGTexture.GetRHI();
```

Use a diagram to show the conversion relationship between RHI resources and RDG resources:



The above code takes the registration of texture related resources as an example, and the registration of buffer is similar.

11.4.3 Extracting Resources

As mentioned in the RDG mechanism in the previous chapter, RDG does not execute immediately after collecting Passes, but delays execution (including delayed creation or allocation of resources). This leads to a problem: if you want to assign the rendered resources to a variable , you cannot use the immediate mode and need to adapt to the delayed execution mode. This adaptation of delayed execution resource extraction is achieved through the following interface:

- FRDGBuilder::QueueTextureExtraction
- FRDGBuilder::QueueBufferExtraction

The following are some usage examples:

```
//createRDGTexture.
FRDGTextureRef MyRDGTexture;

FRDGTextureDesc MyTextureDesc = FRDGTextureDesc::Create2D(OutputExtent, HistoryPixelFormat,
FClearValueBinding::Black, TexCreate_ShaderResource | TexCreate_UAV);

MyRDGTexture = GraphBuilder.CreateTexture(MyTextureDesc,"MyRDGTexture",
ERDGTextureFlags::MultiFrame);

//createUAVand asPassofshader
parameter. (.....)
PassParameters->MyRDGTextureUAV        =  GraphBuilder.CreateUAV(MyRDGTexture);
(.....)

//IncreasePass,To render the image toMyRDGTextureUAV.
FComputeShaderUtils::AddPass(GraphBuilder, RDG_EVENT_NAME("MyCustomPass",                    . . . ) ,
ComputeShader, PassParameters, FComputeShaderUtils::GetGroupCount(8,8));

//Enqueue to extract resources.
TRefCountPtr<IPooledRenderTarget>* OutputRT;
GraphBuilder.QueueTextureExtraction(MyRDGTexture,                &OutputRT);
```

```
//For the extractedOutputRTPerform subsequent
operations. (.....)
```

However, it should be noted that since Pass, resource creation and extraction are all delayed, it means that the extracted resources can only be returned and provided for use in the next frame.

> Thinking: If you want to use the extracted resources in this frame, is it feasible to add a special parameter-free Pass to operate the extracted resources? Why?

11.4.4 Add Pass

The execution unit of the entire RDG system is RDG Pass. Their dependencies, references, inputs, and outputs are all completed through FRDGBuider::AddPass. The following is an example:

```
//createPassofshaderparameter.
FMyPS::FParameters* PassParameters = GraphBuilder.AllocParameters<FMyPS::FParameters>(); PassParameters-
>InputTexture                    = InputTexture;
PassParameters->RenderTargets      =FRenderTargetBinding(InputTexture,
InputTextureLoadAction);
PassParameters->InputSampler       = BilinearSampler;

//Processing shaders.
TShaderMapRef<FScreenPassVS>VertexShader(View.ShaderMap);
TShaderMapRef<FMyPS>PixelShader(View.ShaderMap);

constFScreenPassPipelineStatePipelineState(VertexShader, PixelShader, AdditiveBlendState);


//IncreaseRDG Pass.
GraphBuilder.AddPass(
    RDG_EVENT_NAME("MyRDGPass"),
    PassParameters,
    ERDGPassFlags::Raster,
    // PassofLambda
    [PixelShader, PassParameters, PipelineState] (FRHICommandListImmediate& RHICmdList) {

        //Set the viewport.
        RHICmdList.SetViewport(0,0,0.0f,1024,768,1.0f);

        //set upPSO.
        SetScreenPassPipelineState(RHICmdList, PipelineState);

        //Set shader parameters.
        SetShaderParameters(RHICmdList,        PixelShader,      PixelShader.GetPixelShader(),
* PassParameters);

        //Draw a rectangular area.
        DrawRectangle(RHICmdList,0,0,1024,768,0,0,1.0f,1.0f, FIntPoint(1024,768), FIntPoint(1024,768),
PipelineState.VertexShader, EDRF_Default);
    });
```

The Pass added to the RDG system can be a traditional Graphics Pass, a Compute Shader, or a Pass without parameters. There is no one-to-one correspondence between RDG Pass and RHI Pass.

Several RDG Passes may be combined into one RHI Pass for execution. For details, see the previous section**11.3.4 FRDGBuilder::Execute**.

11.4.5 Create FRDGBuilder

The code to create and use FRDGBuilder is very simple, as shown below:

```
void RenderMyStuff(FRHICommandListImmediate& RHICmdList) {

    //----createFRDGBuilderLocal objects of
    FRDGBuilderGraphBuilder(RHICmdList, RDG_EVENT_NAME("GraphBuilder_RenderMyStuff"));

    (.....)

    //----IncreasePass----

    GraphBuilder.AddPass(...);

    (.....)

    GraphBuilder.AddPass(...);

    (.....)

    //----Increase resource extraction----

    GraphBuilder.QueueTextureExtraction(...);

    (.....)

    //----implementFRDGBuilder ----

    GraphBuilder.Execute();
}
```

It should be pointed out that FRDGBuilder instances are usually local. There are several FRDGBuilder instances in the UE system, which are mainly used for relatively independent modules, such as scene renderers, post-processing, ray tracing and other modules.

FRDGBuilder actually executes three steps: collecting Pass, compiling Pass, and executing Pass. However, FRDGBuilder::Execute already includes compiling and executing Pass, so we no longer need to explicitly call the FRDGBuilder::Compile interface.

11.4.6 RDG Debugging

There are some console commands for the RDG system, their names and descriptions are as follows:

Console variables	describe
r.RDG.AsyncCompute	Controls the asynchronous compute strategy: 0-disabled; 1-enables the flag for asynchronous compute passes (default); 2-enables all compute passes that use the compute command list.
r.RDG.Breakpoint	When certain conditions are met, breakpoints are set to debugger breakpoint locations. 0-disabled, 1~4-different special debugging modes.
r.RDG.ClobberResources	Clears all render targets and texture/buffer UAVs with the specified clear color at allocation time. Useful for debugging.
r.RDG.CullPasses	Whether RDG enables pruning useless passes. 0-disabled, 1-enabled (default).
r.RDG.Debug	Enables output of warnings about inefficiencies found during linking and execution.
r.RDG.Debug.FlushGPU	Enables flushing of instructions to the GPU after each Pass execution. Asynchronous computing is disabled when (r.RDG.AsyncCompute=0) is set.
r.RDG.Debug.GraphFilter	Filter certain debug events into specific graphs.
r.RDG.Debug.PassFilter	Filter certain debug events to specific Passes.
r.RDG.Debug.ResourceFilter	Filter certain debug events to specific resources.
r.RDG.DumpGraph	Dump multiple visualization logs to disk. 0-disable, 1-show producer and consumer pass dependencies, 2-show resource states and transitions, 3-show graphs and overlaps of asynchronous computations.
r.RDG.ExtendResourceLifetimes	RDG will extend the resource lifecycle to the full length of the graph. This will increase memory usage.
r.RDG.ImmediateMode	The pass is executed when it is created. It is very useful to connect the call stack of the code when crashing in the lambda of the pass.

Console variables	describe
r.RDG.MergeRender Passes	The graph will merge identical, consecutive render passes into a single render pass. 0 - disabled, 1 - enabled (default).
r.RDG.Overlap UAVs	RDG will overlap UAV work when needed. If disabled, UAV barriers are always inserted.
r.RDG.TransitionLog	Output resource switches to the console.
r.RDG.Verbose CSVStats	Controls the level of detail of RDG's CSV profiling statistics. 0 - generates a single CSV profile for the graph execution, 1 - generates a single CSV file for each stage of the graph execution.

In addition to the RDG console commands listed above, there are some commands that can display useful information about the RDG system during operation.

visList all valid textures. After input, the following information may be displayed:

```
VisualizeTexture/Vis <CheckpointName> [<Mode>] [PIP/UV0/UV1/UV2] [BMP] [FRAC/SAT] [FULL]: Mode (examples):

   RGB         =RGBinrange0..1(default) = RGB *8
   *8
   A           = alpha channelinrange0..1 = red
   R           channelinrange0..1 = green channelin
   G           range0..1 = blue channelinrange0..1 =
   B           Alpha *16
   A*16
   RGB/2       = RGB /2
SubResource:
   MIP5        = Mip level5(0is default) = Array Element5(0
   INDEX5      is default)
InputMapping:
   PIP         =likeUV1 butaspictureinpicturewithnormal rendering = UVinleft top = full          (default)
   UV0         texture
   UV1
   UV2         = pixel perfect centered
Flags:
   BMP         = save out bitmaptothe screenshots folder (not onconsole, normalized) = Stencil normally displayedin
STENCIL        alpha channelofdepth. Thisoption isused
forBMPto geta stencil only BMP.
   FRAC        = use frac()inshader (default) = use
   SAT         saturate()inshader
   FULLLIST = show full list, otherwise we hide some texturesinthe printout BYNAME
               = sort listbyname = show
   BYSIZE      listbysize
TextureId:
   0            = <off>
 LogConsoleResponse:          13= (2D1x1 PF_DepthStencil)                    DepthDummy1KB
```

LogConsoleResponse: 18= (2D 976x492 PF_FloatRGBA RT) 19= (2D SceneColor 3752KB
LogConsoleResponse: 128x32 PF_G16R16) twenty three= (2D 64x64 PreintegratedGF 16KB
LogConsoleResponse: PF_FloatRGBA VRam) LTCMat 32KB
VRamInKB(Start/Size):<NONE>
LogConsoleResponse: twenty four= (2D 64x64 PF_G16R16F VRam) LTCAmp 16KB
VRamInKB(Start/Size):<NONE>
LogConsoleResponse: 26= (2D 976x492 PF_FloatRGBA UAV) 27= (2D 976x492 SSRTemporalAA 3752KB
LogConsoleResponse: PF_FloatR11G11B10 RT UAV)
SSGITemporalAccumulation0 1876KB
LogConsoleResponse: 29= (2D 976x492 PF_R32_UINT RT UAV) DenoiserMetadata0 1876
KB
LogConsoleResponse: 30= (2D 976x492 PF_FloatRGBA RT UAV VRam) SceneColorDeferred 3752
KB VRamInKB(Start/Size):<NONE>
LogConsoleResponse: 31= (2D 976x492 PF_DepthStencil VRam) SceneDepthZ 2345KB
VRamInKB(Start/Size):<NONE>
LogConsoleResponse: 37= (3D 64x64x16 PF_FloatRGBA UAV) 38= (3D 64 HairLUT 512KB
LogConsoleResponse: x64x16 PF_FloatRGBA UAV) 39= (2D 64x64 HairLUT 512KB
LogConsoleResponse: PF_R32_FLOAT UAV) 47= (2D 98x64 HairCoverageLUT 16KB
LogConsoleResponse: PF_A16B16G16R16) 48= (2D 256x64 SSProfiles 49KB
LogConsoleResponse: 128KB PF_FloatRGBA RT) AtmosphereTransmittance
LogConsoleResponse:
KB 49= (2D 64x16 PF_FloatRGBA RT) AtmosphereIrradiance 8
LogConsoleResponse: 50= (2D 64x16 PF_FloatRGBA RT) 51= (3D 256 AtmosphereDeltaE 8KB
LogConsoleResponse: x128x2 PF_FloatRGBA RT) AtmosphereInscatter 512
KB
LogConsoleResponse: 52= (3D 256x128x2 PF_FloatRGBA RT) 53= (3D 256 AtmosphereDeltaSR 512KB
LogConsoleResponse: x128x2 PF_FloatRGBA RT) 54= (3D 256x128x2 AtmosphereDeltaSM 512KB
LogConsoleResponse: PF_FloatRGBA RT) 55= (2D 1x1 PF_A32B32G32R32F AtmosphereDeltaJ 512KB
LogConsoleResponse: RT UAV) EyeAdaptation 1KB
LogConsoleResponse: 56= (3D 32x32x32 PF_A2B10G10R10 RT VRam) CombineLUTs 128KB
VRamInKB(Start/Size):<NONE>
LogConsoleResponse: 68= (2D 976x492 PF_R8G8 RT UAV)
SSGITemporalAccumulation1 938KB
LogConsoleResponse: 89= (2D 976x246 PF_R32_UINT RT UAV) QuadOverdrawBuffer 938
KB
LogConsoleResponse: 91= (2D 976x492 PF_FloatRGBA RT UAV) LightAccumulation 3752
KB
LogConsoleResponse: 92= (Cube[2] 128PF_FloatRGBA) 93= (3D 64x64x64 ReflectionEnvs 2048KB
LogConsoleResponse: 2048KB PF_FloatRGBA RT UAV) TranslucentVolumeDir2
LogConsoleResponse: 95= (2D 1x1 PF_A32B32G32R32F RT UAV) 96= (3D 64 EyeAdaptation 1KB
LogConsoleResponse: x64x64 PF_FloatRGBA RT UAV) TranslucentVolume2 2048
KB
LogConsoleResponse: 2048KB 97= (3D 64x64x64 PF_FloatRGBA RT UAV) TranslucentVolumeDir1
LogConsoleResponse:
KB
LogConsoleResponse: 2048KB 98= (3D 64x64x64 PF_FloatRGBA RT UAV) TranslucentVolume1 2048
LogConsoleResponse:
KB 99= (3D 64x64x64 PF_FloatRGBA RT UAV) TranslucentVolumeDir0
LogConsoleResponse:
LogConsoleResponse: 101= (3D 64x64x64 PF_FloatRGBA RT UAV) TranslucentVolume0 2048
LogConsoleResponse:
102= (2D 976x492 PF_G8 RT UAV) 106= (2D 488x246 ScreenSpaceAO 469KB
PF_DepthStencil) 107= (2D 1x1 PF_A32B32G32R32F SmallDepthZ 1173KB
RT UAV) EyeAdaptation 1KB
LogConsoleResponse: CheckpointName (what was rendered this frame, use <Name>@<Number> to

```
getintermediate versions):
LogConsoleResponse:Pool:43/112MB (referenced/allocated)
```

11.5 Summary

This article mainly explains the basic concepts, usage methods, rendering process and main mechanisms of UE's RDG, so that readers can have a general understanding of RDG. As for more technical details and principles, readers need to study the UE source code to explore. There are many details of RDG usage that are not covered in this article. You can read the officialRDG 101: A Crash Course to make up for it.

11.5.1 Thoughts on this article

As usual, this article also arranges some small thoughts to help understand and deepen the grasp and understanding of RDG:

- What are the steps of RDG? What is the role of each step? What are the characteristics of each step?
- What are the differences and connections between RDG resources and RHI resources? How to convert between them?
- Use RDG to implement custom CS and PS drawing codes.
-
-
-
-
-

References

- Unreal Engine Source
- Rendering and Graphics

- Materials
- Graphics Programming
- Render Dependency Graph
- FrameGraph: Extensible Rendering Architecture in Frostbite
- RDG 101: A Crash Course
- Frostbite Rendering Architecture and Real-time Procedural Shading & Texturing Techniques

https://github.com/pe7yu