Rendering & Graphics Programming with Unreal Engine

by: Julien Popa-Liesz

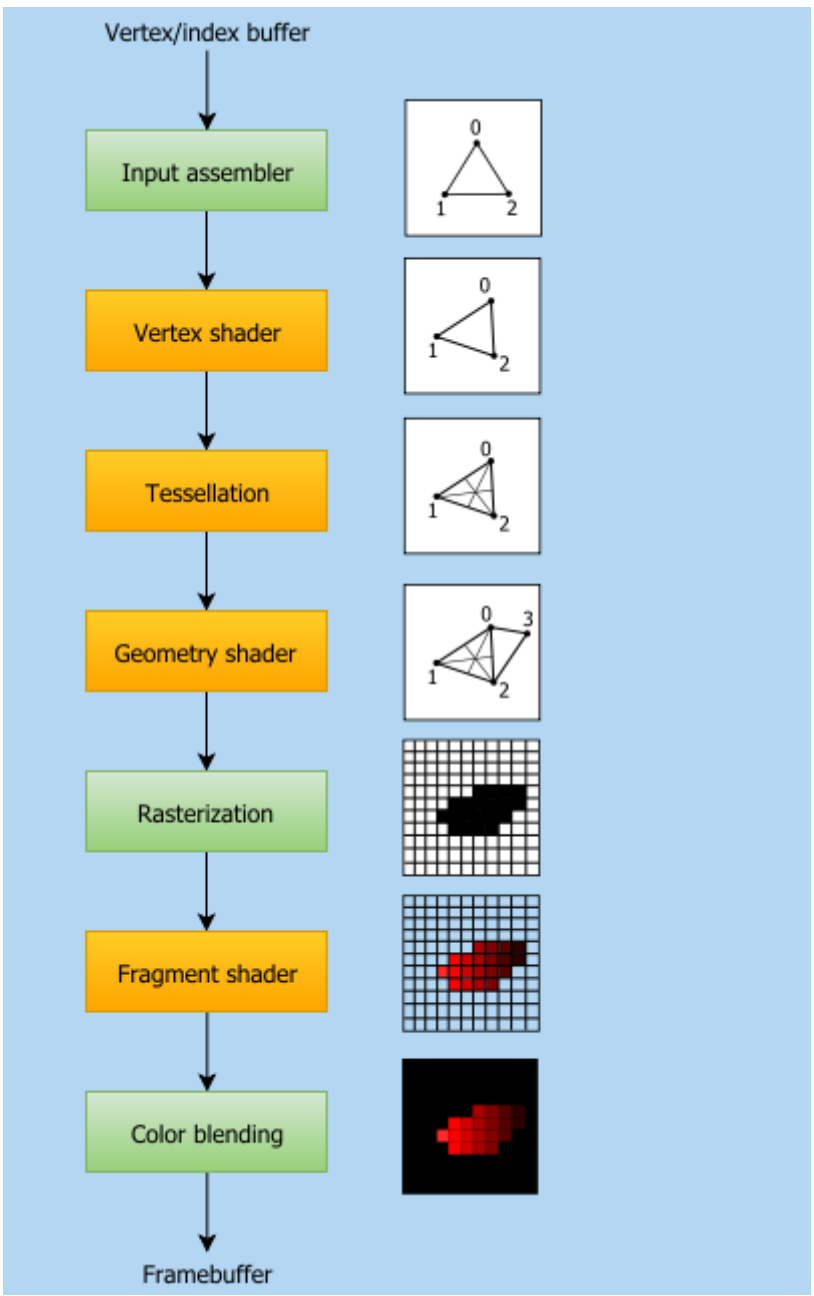
Table of Contents:

Preface: The goal of this document is to highlight the rendering pipeline used inside Unreal Engine 5.1. The paper will outline some basics of the graphics pipeline and how Unreal interfaces with many Graphics APIs to perform rendering.

Fundamentals of Graphics Programming

Graphics Pipeline

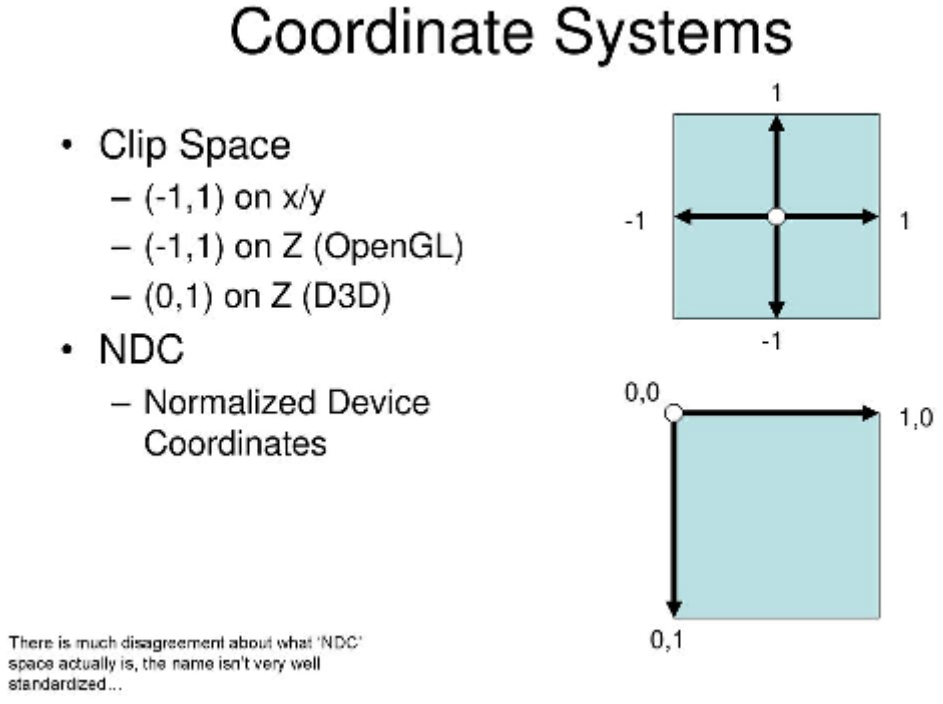
In order to grasp the content of this document a basic understanding of a graphics rendering pipeline is needed. Below is an example of the Vulkan Graphics Pipeline. You can skip this section if you know already about the graphics pipeline.



There are many types of graphics pipelines used across many platforms some examples are Vulkan, DirectX, OpenGL. All pipelines share common steps in the pipeline such as the Input Assembler, Vertex Shader, Rasterization and Fragment Shader (also know and pixel shader). Some API's have different steps before or after the Rasterizer that may optimize the pipeline specific operations to that platform.

Step 1. The input assembler collects the raw vertex data from the buffers you specify and as far as my knowledge in combination with shader code like HLSL will allow you to bind to input Semantics which I will explore later on.

Step 2. The Vertex shader is run for every vertex and generally applies transformations to turn vertex positions from model space to screen space. It also passes per-vertex data down the pipeline. In general before doing any Matrix Transformations from the vertex to world space, local space or screen space. The vertex data will be interpreted in the standard Clip Space.



Step 3. The Geometry shader is run on every primitive (triangle, line, point) and can discard it or output more primitives than came in. This is similar to the tessellation shader, but much more flexible. However, it is not used much in today's applications, reiterating some API's are different.

Step 4. The rasterization stage discretizes the primitives into fragments. These are the pixel elements that they fill on the framebuffer. Any fragments that fall outside the screen are discarded and the attributes outputted by the vertex shader are interpolated across the fragments, as shown in the figure. Usually the fragments that are behind other primitive fragments are also discarded here because of depth testing. Depth testing is used along side a depth buffer which holds info for the culling process. From my understanding the Rasterizer is the only place in the pipeline that's not programable.

Step 5. Fragment Shader or the Pixel shader as Unreal calls it. The pixel shader is invoked for every fragment that survives the rasterization stage and determines which framebuffer(s) the fragments are written to along with which color and depth values. It can do this using the interpolated data from the vertex shader, which can include things like texture coordinates and normals for lighting.

If interpolation is confusing imagine this example scenario between vertex shader and pixel shader.

Suppose you had two vertex values A & B. Both vertices contain a unique 2D screen position and a 3D pixel color. Now if vertex A was a point on the screen with a color value of Red and vertex B was another point with the color value blue and you drew a line connecting A to B you'd have something like this.

Red(VA)——Purple—-Blue(B), The vertex shader would draw the line connecting A to B. Then pass that off to the Pixel shader which would interpolate the color between A to B. Point A is purely red and Point B is purely blue giving us a middle color of purple (which is a mix of Red and blue).

Step 6. API specific operations can be seen before the Pixel Shader hands off the final image to the framebuffer.

GPU Buffers

Key thing to understand is that a buffer is just a resource stored in memory on the GPU. These resources are declared on the CPU and then Mem copied to the GPU to be used on various rendering processes. Alignment is important in most cases when defining data on the cpu before passing it of to the GPU, so be prepared to see things about padding and alignment when interfacing with the Graphics APIs. There are so many kinds of buffers but some of the most common buffers are listed below.

Vertex Buffer - Holds data that defines all vertices the input assembler will then read this and bind it to the vertex shader we specified in the GPU Pipeline.

Index Buffer - The Index buffer is an array of pointers to vertices in the vertex buffer, the index buffer usually reads three vertices at a time. However you can specify offsets, this allows us to create multiple combinations from the vertex buffer to feed to the vertex shader. We can define different draw calls to use some examples are screen space only quads, triangles or other primitives.

Command Buffer - Command buffers are used to record the commands that are required to render a frame. These commands include tasks such as setting up the viewport, binding shaders, textures, and issuing draw calls to render the geometry. Once the command buffer is recorded, it can be submitted to the GPU for execution

Depth Buffer - The depth buffer (also known as the Z-buffer) is a memory buffer that is used in 3D graphics to store the depth information for each pixel on the screen. It is used to determine the relative depth of the objects in a scene, and to ensure that objects that are closer to the viewer are drawn on top of objects that are further away.

GBuffer - The GBuffer, short for Geometry Buffer, is a set of multiple off-screen render targets that are used to store various information about the geometry in a 3D scene. This information can include things like depth, normals, albedo (color), specular, and other data. The GBuffer is typically used in deferred shading, which is a technique that separates the process of capturing the geometry of a scene from the process of applying lighting and shading to that geometry.

Texture Buffer - A texture buffer is a GPU buffer that is used to store texture data. A texture is a 2D or 3D image that is applied to the surfaces of 3D models to add visual detail and realism to a scene. Texture buffers are used to store the pixel data for textures, which can include things like color, alpha, and normal information.

HLSL

In Unreal Engine shader code is written using HLSL with the file extension types .usf and .ush.

I like to imagine the HLSL like assembly code. Where you have Input registers and Output registers. You define your inputs from your vertex shader and it's outputs as inputs to your pixel shader. Regular semantics are used to define the meaning of input and output data for a particular stage of the pipeline. This could include things like position, color, normal, and texture coordinates. You can call these "Regular Semantics" anything like but you will need to bind resources (Buffers) to them using the Input Assembler as illustrated earlier.

"System semantics", on the other hand, are used to define the meaning of input and output data that is specific to a particular platform or API. These semantics are used to define how the data is passed between the GPU and the API, such as the DirectX or OpenGL API.

For example, you would use the "SV_VERTEXID" semantic to indicate that a particular input variable contains the vertex ID, which is a system semantic that is specific to DirectX. This semantic is tied to a DirectX specific draw call where the SV_VERTEXID semantic is incremented based on the current vertex being processed the vertex buffer the vertex shader is using.

Below is the Triangle HLSL code we will be binding too to draw our triangle later on.

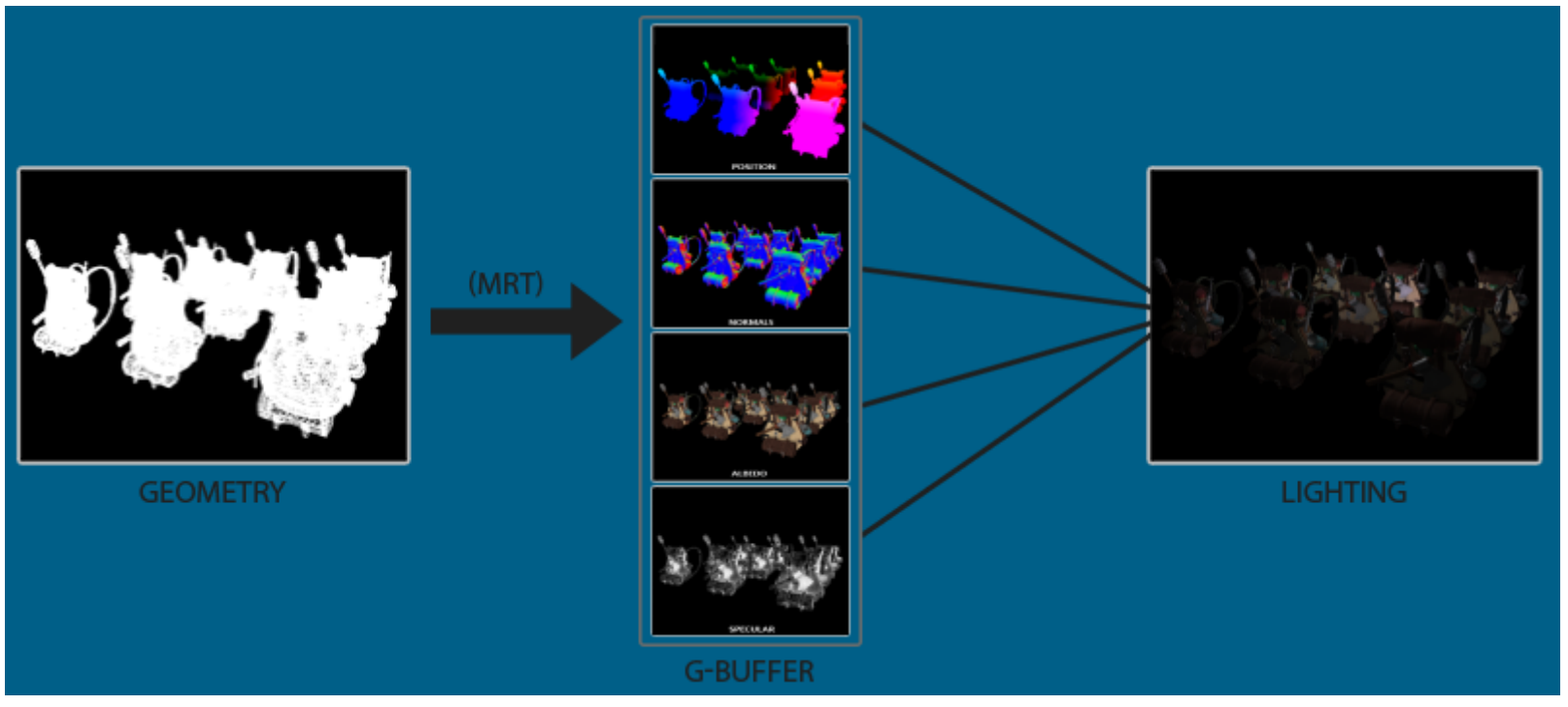
```
// TriangleVS Binable Name for entry point for a custom vertex shader
void TriangleVS(
    in float2 InPosition : ATTRIBUTE0, // First Input Bindable Regular Symantic
    in float4 InColor : ATTRIBUTE1,   // Second Input Bindable Regular Symantic
    out float4 OutPosition : SV_POSITION, // System Symantic Ouput position to Pixel Shader
    out float4 OutColor : COLOR0        // System Symantic Output Color to Pixel Shader
    )
{
    OutPosition = float4(InPosition, 0, 1);
    OutColor = InColor;
}

// TrianglePS Bindable entry point for a custom Pixel Shader
void TrianglePS(
    in float4 InPosition : SV_POSITION, //System Symantic Input position to Pixel Shader
    in float4 InColor : COLOR0, // System Symantic Input Color to Pixel Shader
{
    out float4 OutColor : SV_Target0) // System Symantic Render Target, IE. The 2D texture resource Render to.
    OutColor = InColor;
}
```

Unreal Engine Rendering Pipeline

Before getting into what the rendering pipeline of Unreal, the concept of a rendering pass and deferred rendering needs to addressed.

A Rendering Pass is set of one to many draw calls executed on the GPU. Usually many draw calls are grouped together to ensure proper order of execution. This is because the output of a previous pass may be used as input for other sequential passes.

Deferred rendering is a default method in Unreal that renders lights and materials in a separate pass. This separate pass waits for the base pass to accumulate the information about key information such as opacity, specular, diffusion, normals etc. An example below shows how the deferred rendering works.



Instead of computing lighting and shading for each pixel as it rasterized, Unreal uses deferred rendering to capture information about the scene's geometry into the "off-screen" Gbuffers. Then it's used on a second pass to apply lighting and shading to the scene. The advantage of this is that it allows for more efficient use of the GPU. Separation of the geometry from lighting and shading can improve performance since it allows the GPU to process large number of lights and effects simultaneously. Additionally this allows for flexibility with dynamic lighting and complex lighting setups.

Pass Order in Unreal Engine

These Passes may change but in general this is the order of things. I recommend downloading RenderDoc and hooking the engine to see for yourself.

Base Pass

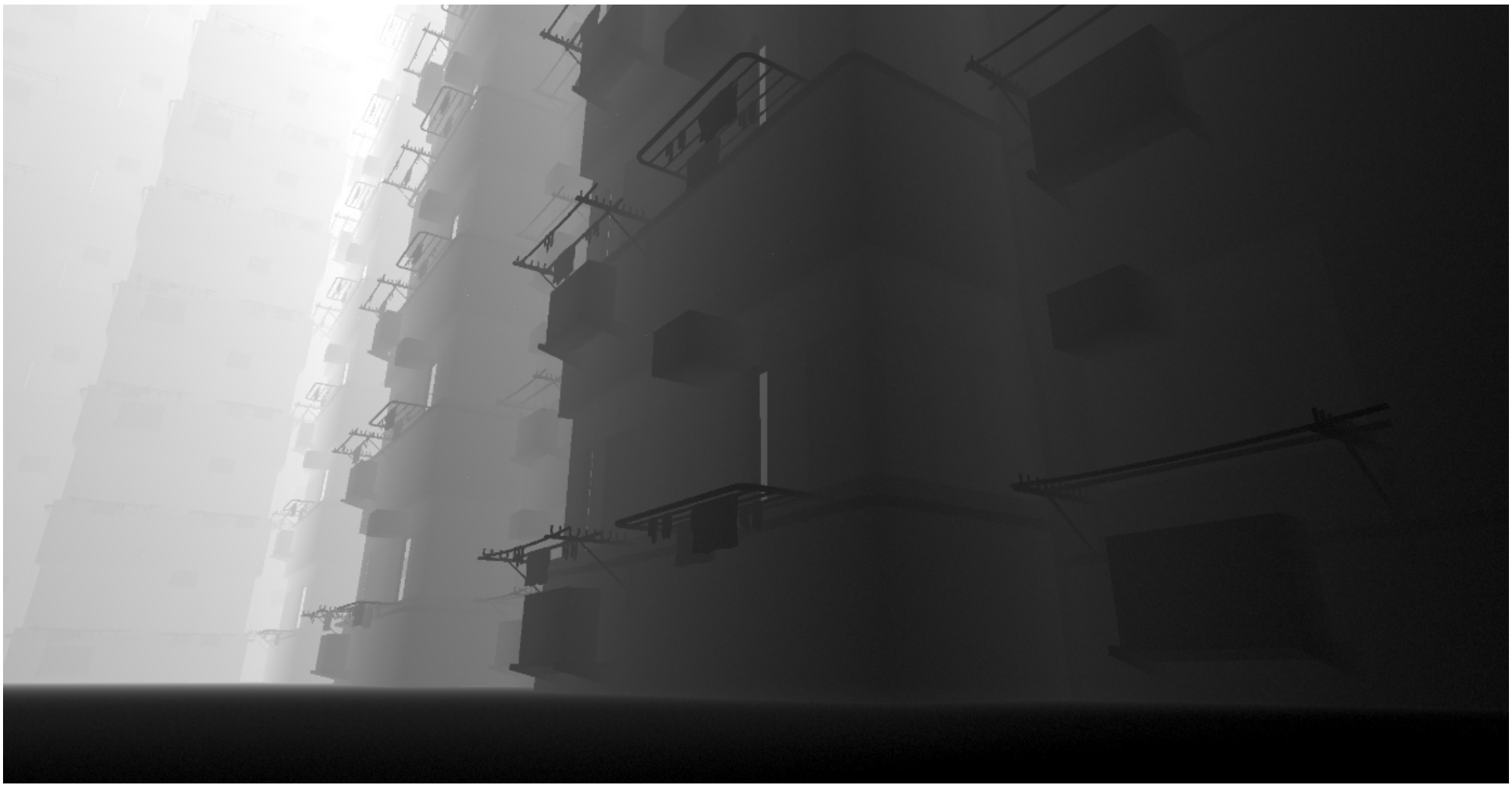
- Rendering final attributes of **Opaque** or **Masked** materials to the G-Buffer
- Reading static lighting and saving it to the G-Buffer
- Applying DBuffer decals
- Applying fog
- Calculating final velocity (from packed 3D velocity)
- In forward renderer: dynamic lighting

In Deferred mode the base pass saves the properties of materials into the GBuffer as highlighted earlier and leaves it for calculation of lighting later on.

Geometry Passes

The Geometry pass is where the meshes get drawn and prioritized before lighting.

PrePass



Early Rendering of Depth Z-Buffer, which is used to optimize out meshes bases on translucency. This also is used to optimize out meshes that are hidden behind other meshes to cull them out (to not render them)

HZB

- Generates a hierarchy Z-Buffer

The HZB is used by an occlusion culling method and by screen space techniques for ambient occlusion and reflection.

Render Velocities

- Saves velocity of each vertex (used later by motion blur and temporal anti-aliasing)

Velocity is a buffer that measures the velocity of every moving vertex and saves it into the motion blur velocity buffer. The Velocity buffer compares the difference between the current frame and one frame behind to create a mask. In Doom 2016 they use this mask to render only meshes that are not static in a scene to optimize rendering of meshes that moved in the next frame.

Lighting Pass

This is the most hardcore part of the frame, especially with a lot of dynamic and shadowed light sources.

Direct Lighting

- Optimized lighting in forward shading

Non-Shadowed Lights

- Lights in deferred rendering that don’t cast shadows

Shadowed Lights

- Lights that obviously cast dynamic shadows

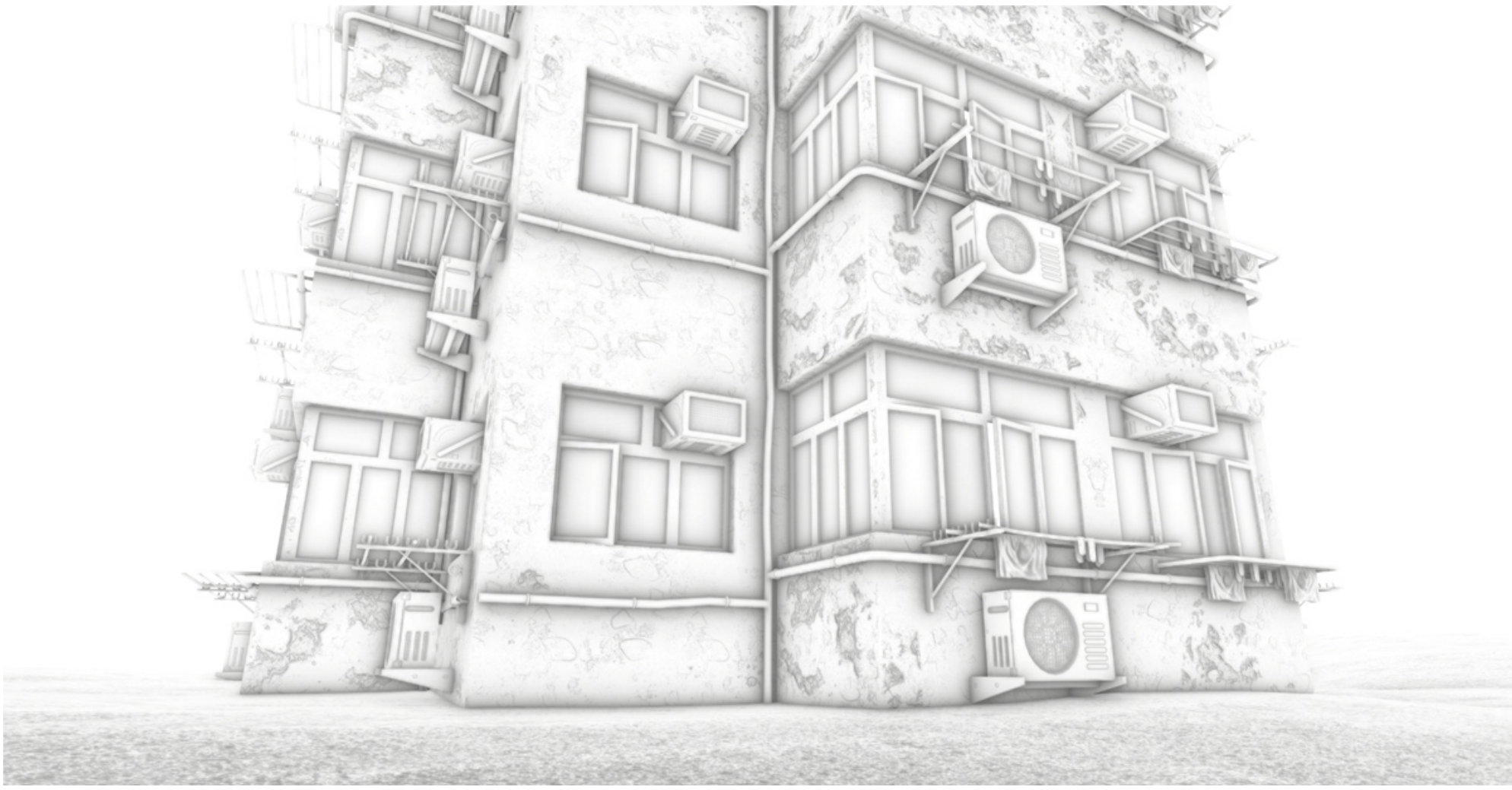
Shadow Depths

- Generates depth maps for shadow-casting lights

Shadow Projection

- Final Rendering of Shadows

Indirect Lighting



- Screen space ambient occlusion

- Decals (non-Buffer type)

Composition After Lighting

- Handles subsurface scattering

Translucency and lighting

- Renders translucent materials
- Lighting of materials that use surface forward shading.

Reflections

- Reading and blending reflection capture actors' results into a full-screen reflection buffer

Screen Space Reflections

- Real-Time dynamic reflections
- Done in Post process using a screen-space ray tracing technique

Post Processing

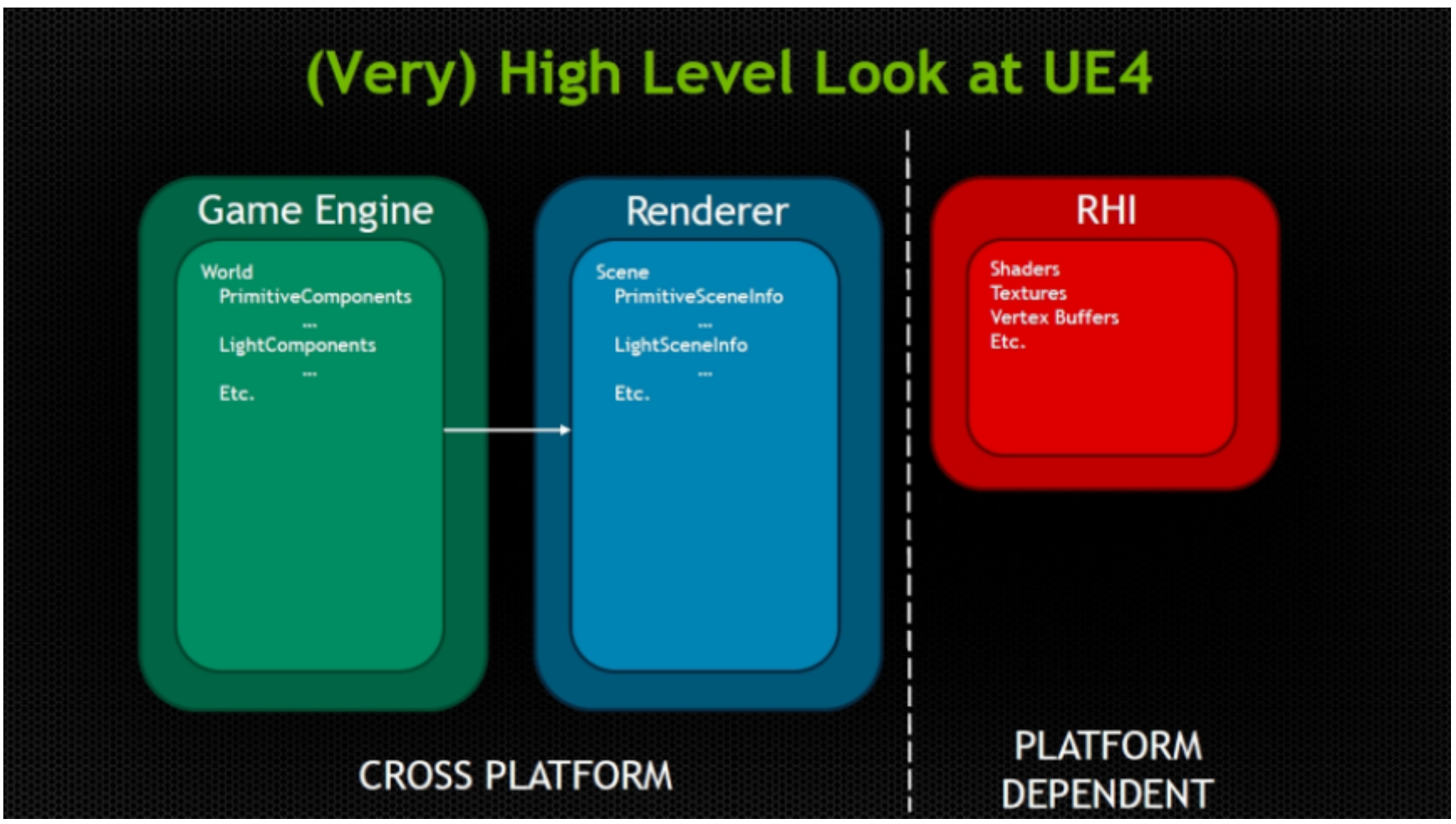
The post processing is the last pass of the render pipeline and is the part of the rendering process we will draw our triangle later on.

- Depth of Field (**BokehDOFRecombine**)
- Temporal anti-aliasing (**TemporalAA**)
- Reading velocity values (**VelocityFlatten**)
- Motion blur (**MotionBlur**)
- Auto exposure (**PostProcessEyeAdaptation**)
- Tone mapping (**Tonemapper**)
- Upscaling from rendering resolution to display's resolution (**PostProcessUpscale**)

Render Hardware Interface (RHI)

The original RHI was designed based on the D3D11 API, including some resource management and command interfaces. Since Unreal Engine is a ubiquitous tool that supports many platforms like mobile, console and PC in which then can use (DirectX, Vulkan, OpenGL, Metal). To address this Unreal abstracted an interface between all these API's so they could keep the Rendering code as comprehensive as possible.

This achieved using different render threads as shown below:



We have a Game thread, Render thread and RHI Thread. The important thing to understand is that anything that's rendered has a twin object between game thread and the render thread aside from some other specific cases.

Game Thread

- Primitive Components
- Light Components

Render Thread

- Primitive Proxy
- Light Proxy

RHI Thread

- Translates RHI “immediate” instructions from the Rendering thread to GPU based on the API specified. Note RHI Immediate really means immediate it’s not the same as a regular RHI command which is usually deferred.
- DX12, Vulkan and Host support parallelism and if a RHI immediate instruction is a generated parallel command then the RHI thread will translate it in parallel.

Basics of RHI

FRenderResource

The FRenderResource is a rendering resource representation on the rendering thread. This resource is managed and passed by the rendering thread as the intermediate data between the game thread and

```
/**
 * A rendering resource which is owned by the rendering thread.
 * NOTE - Adding new virtual methods to this class may require stubs added to FViewport/FDummyViewport, otherwise certain modules may have link errors
 */
class RENDERCORE_API FRenderResource
{
public:
    ////////////////////////////////////////////////////////////////////////////
    // The following methods may not be called while asynchronously initializing / releasing render resources.

    /** Release all render resources that are currently initialized. */
    static void ReleaseRHIForAllResources();

    /** Initialize all resources initialized before the RHI was initialized. */
    static void InitPreRHIResources();

    /**
     * Initializes the dynamic RHI resource and/or RHI render target used by this resource.
     * Called when the resource is initialized, or when reseting all RHI resources.
     * Resources that need to initialize after a D3D device reset must implement this function.
     * This is only called by the rendering thread.
     */
    virtual void InitDynamicRHI() {}

    /**
     * Releases the dynamic RHI resource and/or RHI render target resources used by this resource.
     * Called when the resource is released, or when reseting all RHI resources.
     * Resources that need to release before a D3D device reset must implement this function.
     * This is only called by the rendering thread.
     */
    virtual void ReleaseDynamicRHI() {}

    /**
     * Initializes the RHI resources used by this resource.
     * Called when entering the state where both the resource and the RHI have been initialized.
     * This is only called by the rendering thread.
     */
    virtual void InitRHI() {}

    /**
     * Releases the RHI resources used by this resource.
     * Called when leaving the state where both the resource and the RHI have been initialized.
     * This is only called by the rendering thread.
     */
    virtual void ReleaseRHI() {}
```

```
    /**
     * Initializes the resource.
     * This is only called by the rendering thread.
     */
    virtual void InitResource();

    /**
     * Prepares the resource for deletion.
     * This is only called by the rendering thread.
     */
    virtual void ReleaseResource();

    /**
     * If the resource's RHI resources have been initialized, then release and reinitialize it.  Otherwise, do nothing.
     * This is only called by the rendering thread.
     */
    void UpdateRHI();

 (...)
};
```

There are many subclasses that inherit from this class so that the rendering thread can transfer data and operations of the game thread to the RHI thread at different levels of abstraction.

FRHIResource

FRHIResource is used for reference counting, delayed deletion, tracking, runtime data and marking. FRHIResource can be divided into state blocks, shader bindings, shaders, pipeline states, buffers, textures, views, and other miscellaneous items. It should be noted that we can create platform specific types with this class. Check the source for FRHIUniformBuffer.

```
/** The base type of RHI resources. */
class RHI_API FRHIResource
{
public:
    UE_DEPRECATED(5.0, "FRHIResource(bool) is deprecated, please use FRHIResource(ERHIResourceType)")
    FRHIResource(bool InbDoNotDeferDelete=false)
        : ResourceType(RRT_None)
        , bCommitted(true)
#if RHI_ENABLE_RESOURCE_INFO
        , bBeingTracked(false)
#endif
    {
    }

    FRHIResource(ERHIResourceType InResourceType)
        : ResourceType(InResourceType)
        , bCommitted(true)
#if RHI_ENABLE_RESOURCE_INFO
        , bBeingTracked(false)
#endif
    {
#if RHI_ENABLE_RESOURCE_INFO
        BeginTrackingResource(this);
#endif
    }

    virtual ~FRHIResource()
    {
        check(IsEngineExitRequested() || CurrentlyDeleting == this);
        check(AtomicFlags.GetNumRefs(std::memory_order_relaxed) == 0); // this should not have any outstanding refs
        CurrentlyDeleting = nullptr;

#if RHI_ENABLE_RESOURCE_INFO
        EndTrackingResource(this);
#endif
    }

    FORCEINLINE_DEBUGGABLE uint32 AddRef() const
    {...};

private:
    // Separate function to avoid force inlining this everywhere. Helps both for code size and performance.
```

```
    inline void Destroy() const
    {...};
public:
    FORCEINLINE_DEBUGGABLE uint32 Release() const
    {...};

    FORCEINLINE_DEBUGGABLE uint32 GetRefCount() const
    {...};
    static int32 FlushPendingDeletes(FRHICommandListImmediate& RHICmdList);

    static bool Bypass();

    bool IsValid() const
    {...};
    void Delete()
    {...};
    inline ERHIResourceType GetType() const { return ResourceType; }

#if RHI_ENABLE_RESOURCE_INFO
    // Get resource info if available.
    // Should return true if the ResourceInfo was filled with data.
    virtual bool GetResourceInfo(FRHIResourceInfo& OutResourceInfo) const
    {...};

    static void BeginTrackingResource(FRHIResource* InResource);
    static void EndTrackingResource(FRHIResource* InResource);
    static void StartTrackingAllResources();
    static void StopTrackingAllResources();
#endif

private:
    class FAtomicFlags
    {
        static constexpr uint32 MarkedForDeleteBit    = 1 << 30;
        static constexpr uint32 DeletingBit           = 1 << 31;
        static constexpr uint32 NumRefsMask           = ~(MarkedForDeleteBit | DeletingBit);

        std::atomic_uint Packed = { 0 };

    public:
        int32 AddRef(std::memory_order MemoryOrder)
        {...};
        int32 Release(std::memory_order MemoryOrder)
        {...};
        bool MarkForDelete(std::memory_order MemoryOrder)
        {...};

        bool UnmarkForDelete(std::memory_order MemoryOrder)
        {...};

        bool Deleteing()
        {...};

    mutable FAtomicFlags AtomicFlags;

    const ERHIResourceType ResourceType;
    uint8 bCommitted : 1;
#if RHI_ENABLE_RESOURCE_INFO
    uint8 bBeingTracked : 1;
#endif

    static std::atomic<TClosableMpscQueue<FRHIResource*>*> PendingDeletes;
    static FHazardPointerCollection PendingDeletesHPC;
    static FRHIResource* CurrentlyDeleting;

    // Some APIs don't do internal reference counting, so we have to wait an extra couple of frames before deleting resources
    // to ensure the GPU has completely finished with them. This avoids expensive fences, etc.
    struct ResourcesToDelete
    {...};
};
```

FRHICommandList

The RHI Command list is an instruction queue that is used to manage and execute a group of command objects. The parent class for this is **FRHICommandListBase**. **FRHICommandListBase** defines the basic data (Command list, Device context) and interface (Command refresh, wait, enqueue, memory allocation etc..) which is required by the command queue. **FRHIComputeCommandList** defines the interfaces between compute shaders, state transition of GPU resources and the settings for the shader parameters. **FRHICommandList** defines the interface of common rendering pipelines, these include binding Vertex Shaders, Pixel Shaders, Geometry Shaders, Primitive Drawing, shader parameters, resource management etc.

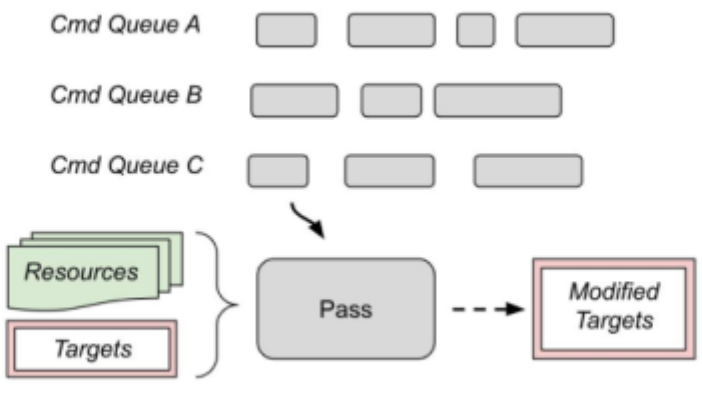
RHIContext & DynamicRHI

Lastly the RHIContext and DynamicRHI is another interface class which defines a set of graphics API related operations. As mentioned earlier some APIs can process commands in parallel and so this separate object is used to define that.

In summary, the RHI classes are the lowest level abstraction used in Unreal to talk to our Graphics APIs. The ones listed are the main ones you should be aware of. I've provided an in depth article that goes through the RHI in more detail under the references section. Also if the command lists doesn't make sense just look up how a basic command list/buffer is processed to the GPU. You can look at one of the Graphics API's to see how it's queued.

Render Dependency Graph (RDG)

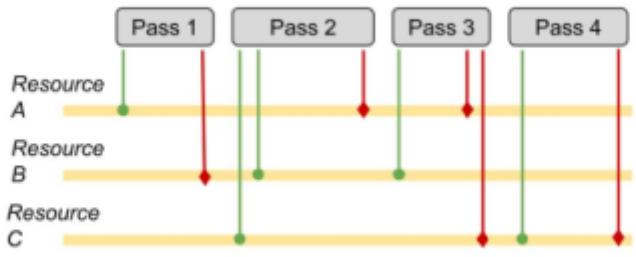
In 2017 Yuriy O'Donnell pioneered a render graph system while working for Frostbite and presented the first Frame Graph at GDC. Consequently the series of advantages this system provided was taken into Unreal Engine and as of 2021 the use of the render graph has become the standard in AAA game engine development.



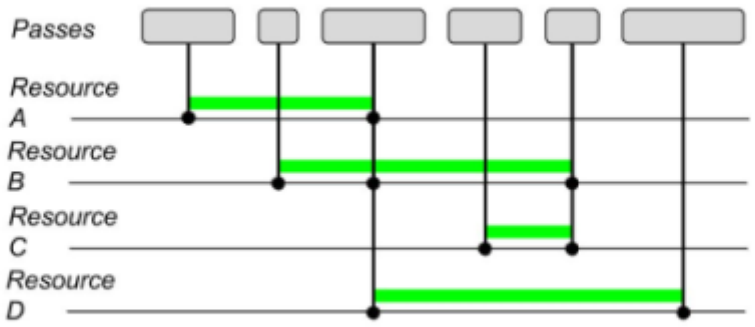
Properties

With the Render Graph we are able to abstract render operations to a single way to produce render code. This allows for clarity of the code and debuggability for tools to interpret resource lifetimes and render pass dependencies to effectively reduce development time.

The next generation of Graphics APIs such as DX12 and Vulkan manage resource states and transitions depending on operations we want to perform.



Using render graphs these operations can be handled automatically without manual input. As seen in the diagram above, a graphics programmer can declare what resources are needed for their shader input, render targets and or depth-stencils. Resource transitions are handled by the graph, you can visualize the green lines as "read" and red lines as "write" operations. Each render graph node has knowledge of these interactions and allows it to place "Barriers" for resource transitions. This means that a optimal barrier ensures that there is an optimal command queue setup. So if resource A is used as a shader resource for Pass 1 but as a render target for pass 2 then we will still need a resource transition to render between the two targets. This reduces calls and saves memory allocation.



In the image above, for example, resource A is used only up to the third pass. On the other hand, resource C starts getting used in the fourth pass, and so its lifetime does not overlap the one of resource A, meaning that we can use the same memory for both resources.
The same concept applies for resource A and D, and in general we will have multiple ways to overlap our memory allocations, so we will also need clever ways to detect the best allocation strategy.

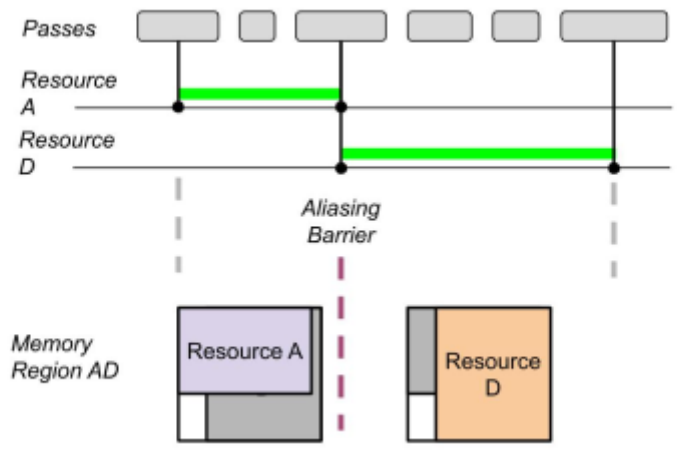
RDG Resources

There are resources that are used on “Per-frame” basis: which are technically called graph or transient resources since their lifetime can be fully handled by the render graph. Some examples of “Per-Frame” resources are Gbuffers and Camera Depth which are deferred in lighting pass.

Transient resources are meant for a render graph to last for a specific time within a single frame so there is a lot of potential for memory re-use. The are other resources that are used outside and are dependent on other resources like a window swapchain back buffer, so the graph in this case will limit itself to just managing their state. This is know as “external resources”.

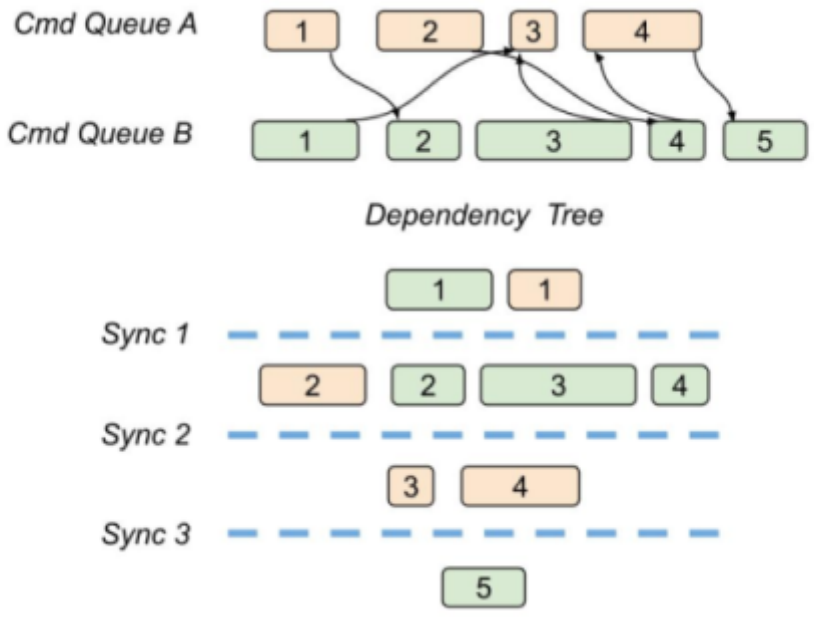
Transient Resource System

The lifetime of transient resources can have what’s called “resource aliasing” for these resources (according to DX12 terminology).



Aliased resources can spare more than 50% of the used resource allocation space, especially when using a render graph. They add an additional managing resource complexity to the scene, but if we want to spare memory, they are almost always worth it.

Build Cross-Queues Synchronization



Lastly the graph allows us to use multiple command queues and run them in parallel, using a dependency tree we can synchronize these mechanism to prevent race conditions on the shared resources.

An acyclic graph of render passes, which we have after we lay down every pass from the dependent queues. Each level of a dependency tree, called **dependency level**, will contain passes independent of each other, in the sense of their resource usage. By doing that we can ensure that every pass in the same dependency level can potentially run asynchronously. We can still have cases of multiple passes belonging to the same queue in the same dependency level, but this does not bother us.
As a consequence, we can put a synchronization point with a GPU fence at the end of every dependency level: this will execute the needed resource transitions, for Every queue on a single graphics command list. This approach of course does not come for free, since using fences and syncing different command queues has a time cost. In addition to that, it will not always be the optimal and smallest amount of synchronizations, but it will produce acceptable performance and it should cover all the possible edge cases.

Thus this highlights the advantages of the Render Graph in a nutshell. Better resource management, easier debugging tools and parallel command list synchronization. Additional information on this is linked in the references.

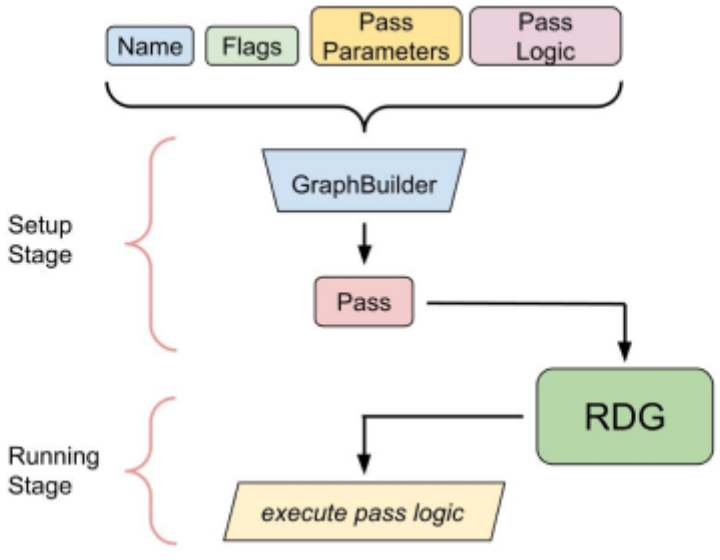
RDG Dynamics

Some new terminology needs to be addressed on top of what we already seen in the context of RDG.

- **View**: a single "viewport" looking at the FScene. When playing in split screen or rendering left and right eye in VR for example, we are going to have two views.
- **Vertex Factory**: class encapsulating vertex data and it is linked to the input of a vertex shader. We have different vertex factory types depending on the kind of mesh we are rendering.
- **Pooled Resource**: graphics resource created and handled by the RDG. Their availability is guaranteed during RDG passes execution only.
- **External Resource**: graphics resource created independently from the RDG.

The workflow of Unreal Engines RDG can be Identified in a 3 step process:

- **Setup phase**: declares which render passes will exist and what resources will be accessed by them.
- **Compile phase**: figure out resources lifetime and make resource allocations accordingly.
- **Running/Execute phase**: all the graph nodes get executed.



Setup Stage

The setup stage begins inside FRenderModule, which is triggered by only the render thread main function. Which builds passes for the visible views and all objects associated with them.

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

The generic syntax for all RDG Passes will be created in this Generic Case

```
// Instantiate the resources we need for our pass
 FShaderParameterStruct* PassParameters = GraphBuilder.AllocParameters&lt;FShaderParameterStruct>();
// Fill in the pass parameters
 PassParameters->MyParameter = GraphBuilder.CreateSomeResource(MyResourceDescription, TEXT("MyResourceName"));
// Define pass and add it to the RDG builder
GraphBuilder.AddPass( RDG_EVENT_NAME("MyRDGPassName"), PassParameters, ERDGPassFlags::
Raster, [PassParameters, OtherDataToCapture](FRHICommandList&RHICmdList) {
```

```
// … pass logic here, render something! ...
 }
```

- **Pass Name**: This will ultimately be represented by an object of type FRDGEventName containing the description of the pass. It is used for debugging and profiling tools.
- **Pass Parameters**: This object is expected to derive from a **Shader Parameter Struct** which has to be created with GraphBuilder.AllocParameters() and needs to be defined with the macro BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters,) The PassParameters will need to distinguish between at least a shader resources and render targets in order to detect proper transitions. (more on this subject in the Shader Parameters) and it can come from either:
    - A shader uniform buffer proper to a shader (e.g. in the case we have only one shader like a compute shader) and so in this case the PassParameters will be of type FMyShader::FParameters.
    - A generically defined shader uniform buffer, which will usually be defined in the source (.cpp) file. The usual name of these buffers will include “PassParameters” to specify that they are used for a whole pass instead of a single shader.
- **Pass Flags**: Set of flags of type ERDGPassFlags, they are mainly used to specify the kind of operations we will be doing inside the pass, e.g. raster, copy and compute.
- **Lambda Function**: This will contain the “body” of our pass, and so the logic to execute at run time. With the lambda we can capture any number of objects we want to later use to set up our rendering operations. Remember, after collection, they are not executed immediately and will be delayed but there are situations where it can be immediate.

Compile Phase

The compile phase is completely autonomous and a “non-programmable” stage, in the sense that the render pass programmer does not have influence on it. In this phase the graph gets inspected to find all the possible flow optimizations, it will:

1. Exclude unreferenced but defined resources and passes: if we want to draw a second debug view of the scene, we might be interested to draw only certain passes for it.
2. Compute and handle used resources lifetime
3. Resources Allocation
4. Build optimized resource transition graph

Running Stage

With the term Running Stage we mean the time when the lambda function of an RDG pass gets executed. This will happen asynchronously and the exact moment is completely up to the RDG.
When the lambda body executes, the available input will be the variables captured by the lambda and a command list (either RHIComputeCommandList& for Compute / AsyncCompute workloads or FRHICommandList& for raster operations).
What essentially happens inside the lambda body is the following

- Set Pipeline state object: e.g. setting rasterizer, blend and depth/stencil states.
- Set Shaders and their Attributes: select what shaders to use, bind them to the current pipeline and set parameters for them, which means binding resources to the shader slots, on the current command list.
- Send copy/draw/dispatch command: send render commands on the command list.

Execute Phase

Execution phase is as simple as navigating through all the passes that survived the compile phase culling and executing the draw and dispatch commands on the list. Up until the execute phase all the resources were handled by opaque and abstract references, while at execute phase we access the real GPU API resources and set them in the pipeline. The preparation of command lists, on the CPU side, can be potentially parallelized quite a lot: in most of the cases, each pass command list setup is independent from each other. Aside from that, command list submissions on a single command queue is not thread safe, and in any case we would first need to determine if adding parallelization would bring significant gains.

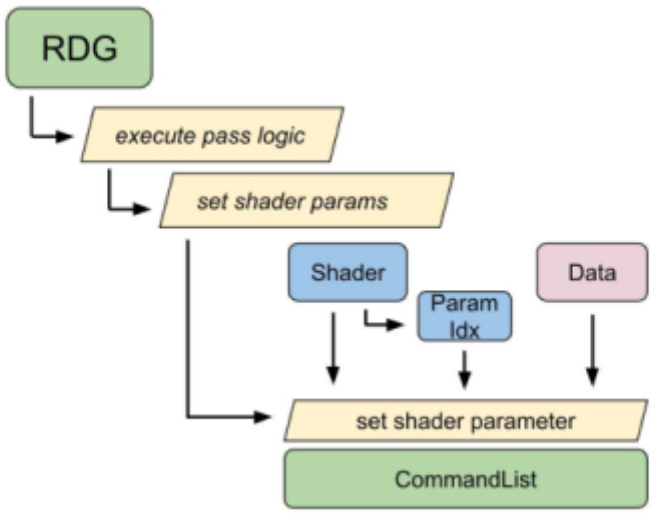
Shader Types

The base class for shaders is FShader, but we find two main types of shaders that we can use:

- **FGlobalShader**: all the shaders deriving from it are part of the global shaders group. A global shader produces a single instance across the engine, and it can only use global parameters.
- **FMaterialShader**: all the derived classes are the ones that use parameters tied to materials. If they also use parameters tied to the vertex factory, then the class to refer to is **FMeshMaterialShader**.

Note* I’ll be using the Global Shader since I rendered inside the post processing pass. The Material shaders are heavily tied to a vertex factory, which I didn’t have time to cover and or experiment with. I will provide some resources for that in the reference section if you’re interested.

Shader Parameters



The shader parameters are the objects that are gonna identify the resource slots used by a shader. These parameters are used when setting resources for a graphics compute or computer operation.

The process of setting a shader parameter will consist in binding a resource to the command list at the index specified by the shader parameter.

Note* if binding and context is confusing, this is because an understanding a lower level GPU and graphics API knowledge is probably missing. To address this I've given a brief explanation in the supplemental explanation section.

We have the following types of Shader Parameters, as seen in ShaderParametersUtils.h and ShaderParameters.h:

- **FShaderParameter**: shader parameter's register binding. e.g. float1/2/3/4, can be an array, UAV.
- **FShaderResourceParameter**: shader resource binding (textures or samplerstates).
- **FRWShaderParameter**: class that binds either a UAV or SRV resource.
- **TShaderUniformBufferParameter**: shader uniform buffer binding with a specific structure (templated). This parameter references a struct which contains all the resources defined for a specific shader. More info later.

As many cases in Unreal Engine, shader parameter classes use macros to define how they are composed. The most important part of the shader parameters is its Layout, an internal variable defined at compile time that specifies its structure, composed of **Layout Fields**.

```
LAYOUT_FIELD(MyDataType, MyFiledName);
LAYOUT_FIELD(FShaderResourceParameter, UAVParameter);
```

The way the layout will be used depends on the type of shader parameters and it can contain any data (e.g. a parameter index). Its purpose is always to hold information about shader parameters (e.g. CBVs, SRVs, UAVs in D3D12) so that we can use them to bind to resources at the moment of executing the shader in the command list.

Shader Uniform Buffer Parameter

The concept of **Uniform Buffer Parameter** in Unreal Engine is very different from what we are used to in standard computer graphics: here it is essentially defined as a struct of shader parameters.
Uniform Buffers, as previously mentioned in the RDG chapter, can be defined using a **Shader Parameter Struct** macro, either inside a shader class declaration or in global scope.

```
BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters, )

    SHADER_PARAMETER_RDG_TEXTURE(Texture2D, InputTexture)

    SHADER_PARAMETER_SAMPLER(SamplerState, InputSampler)

    RENDER_TARGET_BINDING_SLOTS()

END_SHADER_PARAMETER_STRUCT()
```

This family of macros is very flexible and it can contain:

- Shader Parameters: textures, samplers, buffers and descriptors. For a full list of macros reference to ShaderParameterMacros.h.
- Nested Structs: we can encapsulate the definition of a shader parameter struct into another. This is achieved using the macro SHADER_PARAMETER_STRUCT(StructType,MemberName) and SHADER_PARAMETER_STRUCT_ARRAY(..).
- Binding Slots: Available with the macro RENDER_TARGET_BINDING_SLOTS() which adds an array of assignable Render Targets to the parameter struct we use.

  Usage

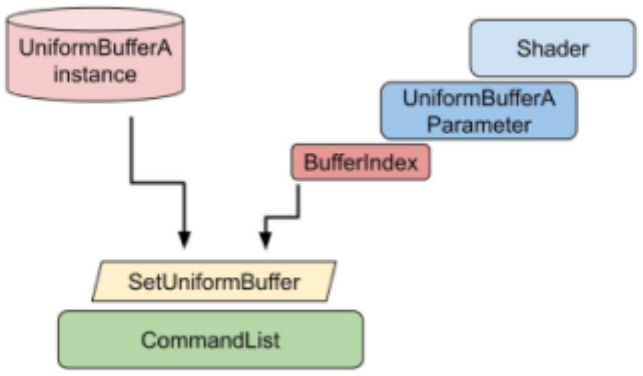
  - If defined outside a shader, the name "FMyShaderParameters" will usually be **F<MyPassName>PassParameters**. These shader parameter structs are used as Pass Parameters for the RDG, as described in the RDG section of this article.

- If defined inside a shader class, the name “FMyShaderParameters” will usually be **FParameters**. We will also need to use this macro at the top of the shader class SHADER_USE_PARAMETER_STRUCT(FMyShaderClass, FBaseClassFromWhatMyShaderDerivesFrom);.

Set Shader Parameters

Most of the times when using a shader inside an RDG pass you can call

```
SetShaderParameters(TRHICmdList& RHICmdList, const TShaderRef&lt;TShaderClass>& Shader, TShaderRHI* ShadeRHI, const typename TShaderClass::FParameters& Parameters)
```



from ShaderParameterStruct.h will be called to bind input resources to a specific shader.
The function will first call ValidateShaderParameters(Shader, Parameters); to check that all the input shader resources cover all the expected shader parameters.
Then it will start to bind all the resources to the relative parameters: every parameter type listed at the beginning of this section (e.g. FShaderParameter, FShaderResourceParameter, etc.) will have their own call for getting bound to the command list. A scheme of when we set a uniform buffer resource is as follows:

BufferIndex is used for all the FParameterStructReference, but also for the basic FParameters elements, since they are stored in buffers as well.
What happens inside CmdList::SetUniformBuffer with such input parameters is completely up to the render platform we are using and it varies a lot from case to case.

Shader Macro Usage & Setup

Local Parameters

Starting with some Parameters, if we want to generate our own Uniform Buffer (Constant Buffer used by many shaders). Let’s show an example between HLSL declarations and our Macro Setup

```
float2 ViewPortSize;
float4 Hello;
float World;
float3 FooBarArray[16];

Texture2D BlueNoiseTexture;
SamplerState BlueNoiseSampler;

// Note Sampler States are objects that we used to "Sample a texture" basically reading
// the sample if we want to do masking, blending or other render wizardry.
Texture2D SceneColorTexture;
SamplerState SceneColorSampler;

RWTexture2D<float4> SceneColorOutput;
```

as explained in the previous section we need to use an internal macro so we can bind these parameters.

```
BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters,)
    SHADER_PARAMETER(FVector2f,ViewPortSize)
    SHADER_PARAMETER(FVector4f,Hello)
    SHADER_PARAMETER(float,World)
    SHADER_PARAMETER(FVector3f,FooBarArray,[16])

    SHADER_PARAMETER_TEXTURE(Texture2D,BlueNoiseTexture)
    SHADER_PARAMETER_TEXTURE(SamplerState,BlueNoiseSampler)

    SHADER_PARAMETER_TEXTURE(Texture2D,SceneColorTexture)
    SHADER_PARAMETER_TEXTURE(SamplerState,SceneColorSampler)
    SHADER_PARAMETER_UAV(Texture2D,SceneColorTexture)
END_SHADER_PARAMETER_STRUCT()
```

The macro **SHADER_PARAMETER_STRUCT** will fill in all the data internally to generate reflective data at compile time.

```
const FShaderParametersMetadata* ParametersMetadata = FShaderParameters::FTypeInfo::GetStructMetadata();
```

Alignment Requirements

You need to conform to alignment. Unreal Adopts the principle of **16-byte** automatic alignment, thus the order of any members matters when declaring that struct.

The main rule is that each member is aligned to the next power of its size, but only if it larger that 4 Bytes. For example:

Pointers are 8-byte aligned

- float,uint32,int32 are 4-byte aligned
- FVector2f, FIntPoint is 8-byte aligned
- FVector and FVector4f are 16-byte aligned

if you don't follow this alignment the engine will throw an assert statement at compile time.

Automatic alignment of each member will be inevitably create padding, as indicated below:

```
BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters,)
    SHADER_PARAMETER(FVector2f,ViewPortSize) // 2 x 4 bytess
    // 2 x 4 byte of padding
    SHADER_PARAMETER(FVector4f,Hello)  // 4 x 4 bytes
    SHADER_PARAMETER(float,World) // 1 x 4 bytes
    // 3 x 4 byte of padding
    SHADER_PARAMETER(FVector3f,FooBarArray,[16]) // 4 x 4 x 16 bytes

    SHADER_PARAMETER_TEXTURE(Texture2D,BlueNoiseTexture) // 8 bytes
    SHADER_PARAMETER_TEXTURE(SamplerState,BlueNoiseSampler) // 8 bytes

    SHADER_PARAMETER_TEXTURE(Texture2D,SceneColorTexture) // 8 bytes
    SHADER_PARAMETER_TEXTURE(SamplerState,SceneColorSampler) // 8 bytes

    SHADER_PARAMETER_UAV(Texture2D,SceneColorTexture) // 8 bytes
END_SHADER_PARAMETER_STRUCT()
```

```
SHADER_PARAMETER(FVector3f,WorldPositionAndRadius) // DONT DO THIS
// ---------------------- //
// Do this
SHADER_PARAMETER(FVector, WorldPosition) // Good
SHADER_PARAMETER(float,WorldRadius) // Good
SHADER_PARAMETER_ARRAY(FVector4f,WorldPositionRadius,[16]) // Good
```

Binding the Shader

After we've setup the shader parameters and alignment is all good declare it with

```
SHADER_USE_PARAMETER_STRUCT(FMyShaderCS,FGlobalShader)
```

```
class FMyShaderCS : public FGlobalShader
{
    DECLARE_GLOBAL_SHADER(FMyShaderCS);
    SHADER_USE_PARAMETER_STRUCT(FMyShaderCS,FGlobalShader)

    static bool ShouldCompilePermutation(const FGlobalShaderPermutationParameters& Parameters) {
        return true;
    }

    using FParameters = FMyShaderParameters;
}

// Additionally we could inline the struct definition like this
class FMyShaderCS : public FGlobalShader
{
    DECLARE_GLOBAL_SHADER(FMyShaderCS);
    SHADER_USE_PARAMETER_STRUCT(FMyShaderCS,FGlobalShader)

    static bool ShouldCompilePermutation(const FGlobalShaderPermutationParameters& Parameters) {
```

```
        return true;
    }

    using FParameters = FMyShaderParameters;

    BEGIN_SHADER_PARAMETER_STRUCT(FMyShaderParameters,)
        SHADER_PARAMETER(FVector2f,ViewPortSize) // 2 x 4 bytess
        // 2 x 4 byte of padding
        SHADER_PARAMETER(FVector4f,Hello)  // 4 x 4 bytes
        SHADER_PARAMETER(float,World) // 1 x 4 bytes
        // 3 x 4 byte of padding
        SHADER_PARAMETER(FVector3f,FooBarArray,[16]) // 4 x 4 x 16 bytes

        SHADER_PARAMETER_TEXTURE(Texture2D,BlueNoiseTexture) // 8 bytes
        SHADER_PARAMETER_TEXTURE(SamplerState,BlueNoiseSampler) // 8 bytes

        SHADER_PARAMETER_TEXTURE(Texture2D,SceneColorTexture) // 8 bytes
        SHADER_PARAMETER_TEXTURE(SamplerState,SceneColorSampler) // 8 bytes

        SHADER_PARAMETER_UAV(Texture2D,SceneColorTexture) // 8 bytes
    END_SHADER_PARAMETER_STRUCT()
}
```

```
// Setting the Parameters in the C++ code before passing it off to the Lambda Function
FMyShaderParameters* PassParameters = GraphBuilder.AllocParameters<FMyShaderParameters>();

PassParameters.ViewPortSize = View.ViewRect.Size();
PassParameters.World = 1.0f;
PassParameters.FooBarArray[4] = FVector(1.0f,0.5f,0.5f);

// Get access to the shader we class we declared Global
TShaderMapRef<FMyShaderCS> ComputeShader(View.Shadermap);
RHICmdList.SetComputeShader(ShaderRHI);

// Setup your Shader Parameters
SetShaderParameters(RHICmdList,*ComputeShader,ComputeShader->GetComputeShader(),Parameters);
RHICmdList.DispatchComputeShader(GroupCount.X,GroupCount.Y,GroupCount.Z);
```

Global Uniform Buffer

Process is the same with some small differences

```
BEGIN_GLOBAL_SHADER_PARAMETER_STRUCT(FSceneTextureUniformParameters,/*Blah_API*/)
    // Scene Color / Depth
    SHADER_PARAMETER_TEXTURE(Texture2D, SceneColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState,SceneColorTextureSampler)
    SHADER_PARAMETER_TEXTURE(Texture2D, SceneColorTexture)
    SHADER_PARAMETER_SAMPLER(SamplerState,SceneDepthTextureSampler)
    SHADER_PARAMETER_TEXTURE(Texture2D<float>, SceneDepthTexturesNonMS)

    //GBuffer
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferATexture)
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferBTexture)
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferCTexture)
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferDTexture)
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferETexture)
    SHADER_PARAMETER_TEXTURE(Texture2D, GBufferVelocityTexture)
    // ...
END_GLOBAL_SHADER_PARAMETER_STRUCT()
```

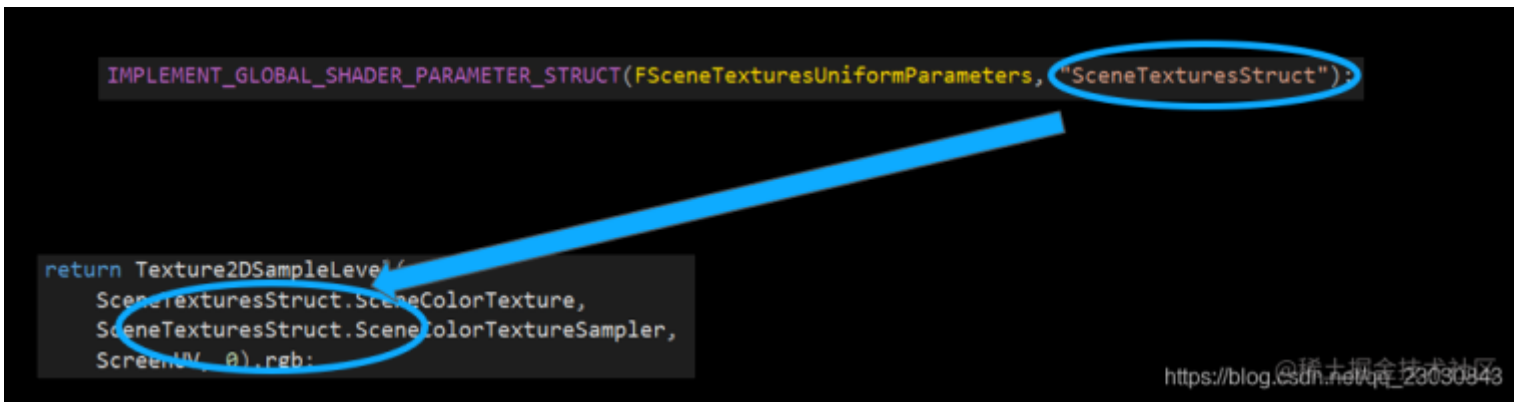
After the struct setup you need to call the implement macro, following the string is the real name defined in the HLSL file.

```
IMPLEMENT_GLOBAL_SHADER_PARAMETER_STRUCT(FSceneTexturesUniformParameters,"SceneTextureStruct");
```

Now inside the Unreal system, Common.ush will refer to the code generated, you will see this Common.ush get included in a lot of HLSL files. There are other includes that you can use that have some useful functions for render code.

Now the uniform buffer we set can be accessed anywhere

```
// Generated file that contains the unifrom buffer declarations that we need to compile the shader want
#include "/Engine/Generated/GeneratedUniformBuffers.ush"
```



Now reference our uniform buffer inside our Parameter struct

```
BEGIN_SHADER_PARAMETER_STRUCT(FParameters,)
    //...
    // Here we ref our unifor buffer
    SHADER_PARAMETER_STRUCT_REF(FViewUniformShaderParameters,ViewUniformBuffer)
END_SHADER_PARAMETER_STRUCT()
```

Again setup the parameter in C++ before passing into the Lambda Function

```
FMyShaderParameters* PassParameters = GraphBuilder.AllocParameters<FDMyShaderParameters>();
PassParameters.ViewPortSize = View.ViewRect.Size();
PassParameters.World = 1.0f;
PassParameters.FooBarArray[4] = FVector(1.0f,0.5f,0.5f);
PassParameters.ViewUniformBuffer = View.ViewUniformBuffer;
```

Drawing a Triangle

With this knowledge the next step is to put it into practice. The “Hello World” of graphics programming is to draw a basic triangle using the Vertex Shader and Pixel Shader. Drawing a triangle in Unreal Engine encapsulates the process for rendering anything in the Engine itself.

This section will cover how to create a render pass to draw this triangle in a step by step tutorial using the Render Dependency Graph in Unreal engine.

Plugin/Module & Shader Folder Setup

Using a Plugin or Module depends on your use case but the important thing to know is that both “Startup Module” or “Initialize” functions in the Plugin or Module allow you to run code before Unreal Engine is fully initialized. The reason this matters is because Unreal’s Renderer is a Module. If you don’t understand Modules and the Engine Life Cycle, I recommend doing a little reading on it to give yourself a better understanding of the runtime linking of modules. The Renderer is linked at runtime so shader code is compiled right before the editor starts up.

Plugin Setup

Go ahead and create a basic C++ project in Unreal Engine, First Person shooter will suffice.

Once you’ve create the project, navigate to your folder structure and add a Shader Folder.

Here is a rough example of what it looks like

_____-

├── YourProjectName

..... ├──> YourProjectName.Build.cs

..... ├──> Source Folder

..... └──> ...

├──> Plugins

..... ├──> Your Plugin Folder

..... └──> Shaders (Create the Folder here)

..... └──> .usf Files go here

..... ├──> Source (Create the Folder here)

..... └──>Private

..... └──>Public

..... └──>YourPluginName.Build.cs

..... └──> YourPluginName.uplugin

_____-

Go to the PluginName.Build.cs file and ensure the dependencies are there.

```
using UnrealBuildTool;

public class YourPluginName : ModuleRules
{
    public YourPluginName(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = ModuleRules.PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicIncludePaths.AddRange(
            new string[] {
                // ... add public include paths required here ...
                EngineDirectory + "/Source/Runtime/Renderer/Private"
            }
            );


        PrivateIncludePaths.AddRange(
            new string[] {
                // ... add other private include paths required here ...
            }
            );


        PublicDependencyModuleNames.AddRange(
            new string[]
            {
                "Core",
                "RHI",
                "Renderer",
                "RenderCore",
                "Projects"
                // ... add other public dependencies that you statically link with here ...
            }
            );


        PrivateDependencyModuleNames.AddRange(
            new string[]
            {
                "CoreUObject",
                "Engine",
```

```
                "Slate",
                "SlateCore"
                // ... add private dependencies that you statically link with here ...
            }
            );


        DynamicallyLoadedModuleNames.AddRange(
            new string[]
            {
                // ... add any modules that your module loads dynamically here ...
            }
            );
    }
}
```

Next set the modules loading phase inside YourPluginName.uplugin to “PostConfigInit”

```
{
    "FileVersion": 3,
    "Version": 1,
    "VersionName": "1.0",
    "FriendlyName": "YourPluginName",
    "Description": "",
    "Category": "Other",
    "CreatedBy": "",
    "CreatedByURL": "",
    "DocsURL": "",
    "MarketplaceURL": "",
    "SupportURL": "",
    "CanContainContent": true,
    "IsBetaVersion": false,
    "IsExperimentalVersion": false,
    "Installed": false,
    "Modules": [
        {
            "Name": "YourPluginName",
            "Type": "Runtime",
            "LoadingPhase": "PostConfigInit" // Set it Here
        }
    ]
}
```

The next step is to bind the shader folder with Unreal so it can find and compile our custom shader code.

```
#include "FroyokLensFlarePlugin.h"
#include "Interfaces/IPluginManager.h"

#define LOCTEXT_NAMESPACE "FFroyokLensFlarePluginModule"

void FYourPluginNameModule::StartupModule()
{
    // This code will execute after your module is loaded into memory; the exact timing is specified in the .uplugin file per-module
    FString BaseDir = IPluginManager::Get().FindPlugin(TEXT("YourPluginName"))->GetBaseDir();
    FString PluginShaderDir = FPaths::Combine(BaseDir, TEXT("Shaders"));
    AddShaderSourceDirectoryMapping(TEXT("/CustomShaders"), PluginShaderDir);
}

void FYourPluginNameModule::ShutdownModule()
{
    // This function may be called during shutdown to clean up your module.  For modules that support dynamic reloading,
    // we call this function before unloading the module.
}

#undef LOCTEXT_NAMESPACE

IMPLEMENT_MODULE(FYourPluginNameModule, YourPluginNamePlugin)
```

You need to include "Interfaces/IPluginManager.h" so you can get access to the helper functions that grabs the base directory to your plugin shader folder directory. The “AddShaderSourceDirectoryMapping(TEXT("/CustomShaders"), PluginShaderDir)” line basically binds a virtual folder called “/CustomShaders” I don’t think the name matters but you can name it whatever. I could be wrong.

Next go into YourPlugin Folder to add a MyViewExtensionSubSystem.cpp file and MyViewExtensionSubSystem.h inside your Private and Public folder respectively. The subsystem is ideal to create a custom FSceneViewExtensionBase pointer to this is object during PostConfigInit phase. This allows us to draw our triangle to the editor viewport. You could declare a TSharedPtr<FMyViewExtension, ESPMode::ThreadSafe> object in MyCharacter.h and instantiate it in BeginPlay. So it renders the triangle after pressing play in the editor.

Module Setup

MyViewExtensionSubSystem.cpp

```
// Fill out your copyright notice in the Description page of Project Settings.

#include "UViewExtensionSubSystem.h"
#include "SceneViewExtension.h"
#include "MyViewExtension.h"

void UMyViewExtensionSubSystem::Initialize(FSubsystemCollectionBase& Collection)
{
    Super::Initialize(Collection);
    // Create Shared Pointer Call
    UE_LOG(LogTemp, Warning, TEXT("View Extension SubSystem Init"));
    // This is the Pointer to the FSceneViewExenstion you will see later on
    // You need this line to run your shader.
    this->ShaderTest = FSceneViewExtensions::NewExtension<MyFViewExtension>();
}
```

MyViewExtensionSubSystem.h

```
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"
#include "Subsystems/EngineSubsystem.h"
#include "MyViewExtensionSystem.generated.h"
class FViewExtension;

/**
 *
 */
UCLASS()
class MyViewExtensionSubSystem: public UEngineSubsystem
{
    GENERATED_BODY()
protected:
    // Declaration of the Pointer delegate Unreal's FViewExenstion object gives us
    TSharedPtr<FViewExtension,ESPMode::ThreadSafe> ShaderTest;
public:
    virtual void Initialize(FSubsystemCollectionBase& Collection) override;
};
```

FSceneViewExtensionBase

This base class is important because it allows you to hook into the render pipeline and use its delegates to insert your own custom render pass. Before 5.1, in order to create your own custom rendering pass a plugin or module was still needed to initialize your shader folder. However without this class you will need to do additional C++ work to add your own delegates inside the render pipeline source code for a custom rendering pass. In this tutorial we will be overriding a delegate function in the FSceneViewExtensionBase class to insert a pass in Post Processing phase of the render pipeline.

In Engine Source at line 411 inside PostProcessing.cpp the delegates of FSceneViewExtensionBase are added

```
// ../Engine../PostProcessing.cpp"

const auto AddAfterPass = [&](EPass InPass, FScreenPassTexture InSceneColor) -> FScreenPassTexture
{
    // In some cases (e.g. OCIO color conversion) we want View Extensions to be able to add extra custom post processing after the pass.
```

```
    FAfterPassCallbackDelegateArray& PassCallbacks = PassSequence.GetAfterPassCallbacks(InPass);

    if (PassCallbacks.Num())
    {
        FPostProcessMaterialInputs InOutPostProcessAfterPassInputs = GetPostProcessMaterialInputs(InSceneColor);

        for (int32 AfterPassCallbackIndex = 0; AfterPassCallbackIndex < PassCallbacks.Num(); AfterPassCallbackIndex++)
        {
            InOutPostProcessAfterPassInputs.SetInput(EPostProcessMaterialInput::SceneColor, InSceneColor);

            FAfterPassCallbackDelegate& AfterPassCallback = PassCallbacks[AfterPassCallbackIndex];
            PassSequence.AcceptOverrideIfLastPass(InPass, InOutPostProcessAfterPassInputs.OverrideOutput,
AfterPassCallbackIndex);
            InSceneColor = AfterPassCallback.Execute(GraphBuilder, View, InOutPostProcessAfterPassInputs);
        }
    }
```

You see at AddAfterPass, FScreenPassTexture InSceneColor is passing a reference of “SceneColor” to the delegates overridden. SceneColor is a FScreenPassTexture that describes a texture paired with a viewport rect. The Scene Color texture is written too many times throughout PostProcessing pipeline and will be the texture we will draw our triangle onto. This will be our “Render Target”.

Class Setup

Let’s create another class header and CPP file inside our plugin public/private source folder. Call it MyViewExtension (or whatever you like).

```
#pragma once
#include "TriangleShader.h"
#include "SceneViewExtension.h"
#include "RenderResource.h"

class YOURPLUGINNAME_API FMyViewExtension : public FSceneViewExtensionBase {

public:
     FMyViewExtension(const FAutoRegister& AutoRegister);

    //~ Begin FSceneViewExtensionBase Interface
    virtual void SetupViewFamily(FSceneViewFamily& InViewFamily) override {}
    virtual void SetupView(FSceneViewFamily& InViewFamily, FSceneView& InView) override {};
    virtual void BeginRenderViewFamily(FSceneViewFamily& InViewFamily) override;
    virtual void PreRenderViewFamily_RenderThread(FRDGBuilder& GraphBuilder, FSceneViewFamily& InViewFamily) override{};
    virtual void PreRenderView_RenderThread(FRDGBuilder& GraphBuilder, FSceneView& InView) override;
    virtual void PostRenderBasePass_RenderThread(FRHICommandListImmediate& RHICmdList, FSceneView& InView) override {};
    virtual void PrePostProcessPass_RenderThread(FRDGBuilder& GraphBuilder, const FSceneView& View, const
FPostProcessingInputs& Inputs) override;
    virtual void SubscribeToPostProcessingPass(EPostProcessingPass Pass, FAfterPassCallbackDelegateArray& InOutPassCallbacks,
bool bIsPassEnabled)override;
    //~ End FSceneViewExtensionBase Interface
};
```

Here we have a couple delegates denoted with _RenderThread and other setup functions. We will be overriding “SubscribeToPostProcessingPass” only.

Let’s setup some functions in the MyViewExtension.cpp file.

```
#include "ViewExtension.h"
#include "TriangleShader.h"
#include "PixelShaderUtils.h"
#include "PostProcess/PostProcessing.h"
#include "PostProcess/PostProcessMaterial.h"
#include "SceneTextureParameters.h"
#include "ShaderParameterStruct.h"


// This Line Declares the Name of your render Pass so you can see it in the render debugger
DECLARE_GPU_DRAWCALL_STAT(TrianglePass);

FMyViewExtension::FMyViewExtension(const FAutoRegister& AutoRegister) : FSceneViewExtensionBase(AutoRegister) {

}

// Begin FLensFlareScene View
```

```
void FMyViewExtension::SubscribeToPostProcessingPass(EPostProcessingPass Pass, FAfterPassCallbackDelegateArray&
InOutPassCallbacks, bool bIsPassEnabled)
{
    if (Pass == EPostProcessingPass::Tonemap)
    {
        // Create Raw Delegate Here, see later on
    }
}
```

Inside SubscribeToPostProcessingPass function I want to bring attention to the if statement. The if statement uses EPostProcessing Enum to define where in the Post Processing phase you want to insert a render pass. I've opted to do it after Tonemap, however you can choose other points in the pipeline defined by that enum. For now the goal is to draw the triangle to the Scene Color since it's available during the entire Post Process Pass.

Setting up Global Shaders

The next step is to create two Global shaders. One being our custom Vertex shader that handles the triangle vertex data stored in the vertex buffer. The second shader being a pixel shader which will color our triangle after the rasterizer processed our vertices.

Again, inside the Plugin source folder we create a separate cpp/header file.

TriangleShader.cpp and TriangleShader.h

Vertex Shader Class

```
// TrangleShader.h
// Defined here so we can access it in View Extension.
BEGIN_SHADER_PARAMETER_STRUCT(FTriangleVSParams,)
    //RENDER_TARGET_BINDING_SLOTS()
END_SHADER_PARAMETER_STRUCT()

class FTriangleVS : public FGlobalShader
{
public:
    DECLARE_GLOBAL_SHADER(FTriangleVS);
    SHADER_USE_PARAMETER_STRUCT(FTriangleVS, FGlobalShader)
    using FParameters = FTriangleVSParams;


    static bool ShouldCompilePermutation(const FGlobalShaderPermutationParameters& Parameters) {
        return true;
    }
};
```

Keeping it simple we aren't creating any new crazy macro specific buffers or resources for the vertex shader. The RDG at minimum requires the macro declaration.

Pixel Shader Class

```
// TrangleShader.h
BEGIN_SHADER_PARAMETER_STRUCT(FTrianglePSParams,)
    RENDER_TARGET_BINDING_SLOTS()
END_SHADER_PARAMETER_STRUCT()

class FTrianglePS: public FGlobalShader
{
    DECLARE_GLOBAL_SHADER(FTrianglePS);
    using FParameters = FTrianglePSParams;
    SHADER_USE_PARAMETER_STRUCT(FTrianglePS, FGlobalShader)
};
```

RENDER_TARGET_BINDING_SLOTS() is the only resource we are passing, we are binding the viewport information or Render Target to our custom shader HLSL code.

In both classes we need to declare them as global shaders and define the local parameter struct the shader needs. In C++ "using" gives access to the type FParamters as "FTrianglePSParams" when declaring a pointer of that type for use elsewhere.

Let's add the Triangle HLSL code in our Shader Folder.

Go into the Shader folder and create a file called "Triangle.usf" then paste the code below and save it.

```
#include "/Engine/Public/Platform.ush"
#include "/Engine/Private/Common.ush"
#include "/Engine/Private/ScreenPass.ush"
```

```
#include "/Engine/Private/PostProcessCommon.ush"

void TriangleVS(
    in float2 InPosition : ATTRIBUTE0,
    in float4 InColor : ATTRIBUTE1,
    out float4 OutPosition : SV_POSITION,
    out float4 OutColor : COLOR0
    )
{
    OutPosition = float4(InPosition, 0, 1);
    OutColor = InColor;
}

void TrianglePS(
    in float4 InPosition : SV_POSITION,
    in float4 InColor : COLOR0,
    out float4 OutColor : SV_Target0)
{
    OutColor = InColor;
}
```

Creating Vertex and Index Buffers

At this point we've introduced mostly all the classes involved in the Rendering Pass. We still need to define the resources classes for our Vertex Buffer and Index Buffer.

Inside TrangleShader.h

Add a struct the defines our Colored Vertex

```
/** The vertex data used to filter a texture. */
// TrangleShader.h
struct FColorVertex
{
public:
    FVector2f Position;
    FVector4f Color;
};
```

Add the FVertexBuffer class

```
// TrangleShader.h
/**
 * Static vertex and index buffer used for 2D screen rectangles.
 */
class FTriangleVertexBuffer : public FVertexBuffer
{
public:
    /** Initialize the RHI for this rendering resource */
    void InitRHI() override {
        TResourceArray<FColorVertex, VERTEXBUFFER_ALIGNMENT> Vertices;
        Vertices.SetNumUninitialized(3);

        Vertices[0].Position = FVector2f(0.0f,0.75f);
        Vertices[0].Color = FVector4f(1, 0, 0, 1);

        Vertices[1].Position = FVector2f(0.75,-0.75);
        Vertices[1].Color = FVector4f(0, 1, 0, 1);

        Vertices[2].Position = FVector2f(-0.75,-0.75);
        Vertices[2].Color = FVector4f(0, 0, 1, 1);

        FRHIResourceCreateInfo CreateInfo(TEXT("FScreenRectangleVertexBuffer"), &Vertices);
        VertexBufferRHI = RHICreateVertexBuffer(Vertices.GetResourceDataSize(), BUF_Static, CreateInfo);
    }
};
```

Inside FTriangleVertexBuffer override InitRHI to initialize the vertex data. In GPU programming you need to create a context so you can bind resources from the CPU before doing a memory copy to the GPU. In order to achieve this a context must be specified holding the name of the resource, size and other properties. Set VertexBufferRHI object with RHICreateVertexBuffer and pass the required information.

Adding the Index Buffer

```
// TrangleShader.h
class FTriangleIndexBuffer : public FIndexBuffer
{
public:
   /** Initialize the RHI for this rendering resource */
   void InitRHI() override
   {
      const uint16 Indices[] = { 0, 1, 2 };

      TResourceArray<uint16, INDEXBUFFER_ALIGNMENT> IndexBuffer;
      uint32 NumIndices = UE_ARRAY_COUNT(Indices);
      IndexBuffer.AddUninitialized(NumIndices);
      FMemory::Memcpy(IndexBuffer.GetData(), Indices, NumIndices * sizeof(uint16));

      FRHIResourceCreateInfo CreateInfo(TEXT("FTriangleIndexBuffer"), &IndexBuffer);
      IndexBufferRHI = RHICreateIndexBuffer(sizeof(uint16), IndexBuffer.GetResourceDataSize(), BUF_Static, CreateInfo);
   }
};
```

Same process here, you don't need to use a index buffer for something a simple as a triangle. Index buffers hold pointers to our vertex data in the vertex buffer. Usually the index buffer will read a combination of three vertices, but you can make any combination you want. This is optimal because we can reuse vertex data to draw a triangle, quad or other primitives depending on your use case.

Lastly we add a global declaration resource for the Vertex Buffer, this is used to define the input layout for the Input assembler so we can bind the correct attributes in the HLSL code.

```
// TrangleShader.h
class FTriangleVertexDeclaration : public FRenderResource
{
public:
    FVertexDeclarationRHIRef VertexDeclarationRHI;

    /** Destructor. */
    virtual ~FTriangleVertexDeclaration() {}

    virtual void InitRHI()
    {
        FVertexDeclarationElementList Elements;
        uint16 Stride = sizeof(FColorVertex);
        Elements.Add(FVertexElement(0, STRUCT_OFFSET(FColorVertex, Position), VET_Float2, 0, Stride));
        Elements.Add(FVertexElement(0, STRUCT_OFFSET(FColorVertex, Color), VET_Float4, 1, Stride));
        VertexDeclarationRHI = PipelineStateCache::GetOrCreateVertexDeclaration(Elements);
    }

    virtual void ReleaseRHI()
    {
        VertexDeclarationRHI.SafeRelease();
    }
};
```

Again we override InitRHI to set our declaration information. For the input layout we define an elements object and the stride. In computer graphics the stride refers to the number of bytes between the start of one element and the start of the next element in an array of elements stored in memory. For the input assembler to read the vertex buffer properly it needs to know the number of bytes between the start of one vertex and the start of the next vertex. We can use sizeof function to compute the space of one vertex as an offset for next vertices. You also notice STRUCT_OFFSET is a macro helper to determine the offset between the values defined in our struct.

Next declare the resource global and extern them so the Renderer API can see it.

```
// TriangleShader.h
extern YOURPLUGIN_API TGlobalResource<FTriangleVertexBuffer> GTriangleVertexBuffer;
extern YOURPLUGIN_API TGlobalResource<FTriangleIndexBuffer> GTriangleIndexBuffer;
extern YOURPLUGIN_API TGlobalResource<FTriangleVertexDeclaration> GTriangleVertexDeclaration;
```

back in Triangle.cpp declare the starting points for our shaders to match the function names in HLSL.

```
#include "TriangleShader.h"
#include "Shader.h"
#include "VertexFactory.h"

// Define our Vertex Shader and Pixel Shader Starting point
// This is needed for all shaders
```

```
IMPLEMENT_SHADER_TYPE(,FTriangleVS, TEXT("/CustomShaders/Triangle.usf"),TEXT("TriangleVS"),SF_Vertex);
IMPLEMENT_SHADER_TYPE(,FTrianglePS,TEXT("/CustomShaders/Triangle.usf"),TEXT("TrianglePS"),SF_Pixel);

TGlobalResource<FTriangleVertexBuffer> GTriangleVertexBuffer;
TGlobalResource<FTriangleIndexBuffer> GTriangleIndexBuffer;
TGlobalResource<FTriangleVertexDeclaration> GTriangleVertexDeclaration;
```

Lastly define our TGlobalResources objects.

Adding an RDG Pass

Returning to update your MyViewExtension cpp/header files.

```
// MyViewExtension.h
class YOURPLUGIN_API FMyViewExtension: public FViewExtension
{
public:

    FLensFlareSceneView(const FAutoRegister& AutoRegister);
    virtual void SubscribeToPostProcessingPass(EPostProcessingPass Pass, FAfterPassCallbackDelegateArray& InOutPassCallbacks,
bool bIsPassEnabled)override;

protected:

    // Copied from PixelShaderUtils
    template <typename TShaderClass>
    static void AddFullscreenPass(
        FRDGBuilder& GraphBuilder,
        const FGlobalShaderMap* GlobalShaderMap,
        FRDGEventName&& PassName,
        const TShaderRef<TShaderClass>& PixelShader,
        typename TShaderClass::FParameters* Parameters,
        const FIntRect& Viewport,
        FRHIBlendState* BlendState = nullptr,
        FRHIRasterizerState* RasterizerState = nullptr,
        FRHIDepthStencilState* DepthStencilState = nullptr,
        uint32 StencilRef = 0);

    template <typename TShaderClass>
    static void DrawFullscreenPixelShader(
        FRHICommandList& RHICmdList,
        const FGlobalShaderMap* GlobalShaderMap,
        const TShaderRef<TShaderClass>& PixelShader,
        const typename TShaderClass::FParameters& Parameters,
        const FIntRect& Viewport,
        FRHIBlendState* BlendState = nullptr,
        FRHIRasterizerState* RasterizerState = nullptr,
        FRHIDepthStencilState* DepthStencilState = nullptr,
        uint32 StencilRef = 0);

    static inline void DrawFullScreenTriangle(FRHICommandList& RHICmdList, uint32 InstanceCount){
        RHICmdList.SetStreamSource(0, GTriangleVertexBuffer.VertexBufferRHI, 0);
        RHICmdList.DrawIndexedPrimitive(
            GTriangleIndexBuffer.IndexBufferRHI,
            /*BaseVertexIndex=*/ 0,
            /*MinIndex=*/ 0,
            /*NumVertices=*/ 3,
            /*StartIndex=*/ 0,
            /*NumPrimitives=*/ 1,
            /*NumInstances=*/ InstanceCount);
    }

    // A delegate that is called when the Tone mapper pass finishes
    FScreenPassTexture TrianglePass_RenderThread(FRDGBuilder& GraphBuilder, const FSceneView& View, const
FPostProcessMaterialInputs& Inputs);

    // For now we try to build a vertex buffer and see if it puts some shit in it

public:
    static void RenderTriangle
```

```
(
FRDGBuilder& GraphBuilder,
const FGlobalShaderMap* ViewShaderMap,
const FIntRect& View,
const FScreenPassTexture& SceneColor);
};
```

Remember the order starts with adding a pass by giving the RDG lambda function the resources and parameters it needs. Next is defining the draw call(s) where you setup the GPU Pipeline. You MUST setup the GPU pipeline because it defines all the parameters for the draw call (Blend States, Rasterizer State, Primitive type, Shaders, Viewport, Render Targets, Commands). The GPU Pipeline stores the state to be interpreted for the low lever graphics API calls being used.

First function is the templated AddFullscreenPass with the required parameters needed for the lambda function, nulling parameters we won't use for the draw call.

```
// MyViewExtension.cpp
template <typename TShaderClass>
void FMyViewExtension::AddFullscreenPass(
        FRDGBuilder& GraphBuilder,
        const FGlobalShaderMap* GlobalShaderMap,
        FRDGEventName&& PassName,
        const TShaderRef<TShaderClass>& PixelShader,
        typename TShaderClass::FParameters* Parameters,
        const FIntRect& Viewport,
        FRHIBlendState* BlendState,
        FRHIRasterizerState* RasterizerState,
        FRHIDepthStencilState* DepthStencilState,
        uint32 StencilRef)
{

    check(PixelShader.IsValid());
    ClearUnusedGraphResources(PixelShader, Parameters);

    GraphBuilder.AddPass(
        Forward<FRDGEventName>(PassName),
        Parameters,
        ERDGPassFlags::Raster,
        [Parameters, GlobalShaderMap, PixelShader, Viewport, BlendState, RasterizerState, DepthStencilState, StencilRef]
(FRHICommandList& RHICmdList)
    {
            FLensFlareSceneView::DrawFullscreenPixelShader<TShaderClass>(RHICmdList, GlobalShaderMap, PixelShader,
*Parameters, Viewport,
            BlendState, RasterizerState, DepthStencilState, StencilRef);
    });
}
```

Setting up the Draw Call

```
template <typename TShaderClass>
    void FMyViewExtension::DrawFullscreenPixelShader(
        FRHICommandList& RHICmdList,
        const FGlobalShaderMap* GlobalShaderMap,
        const TShaderRef<TShaderClass>& PixelShader,
        const typename TShaderClass::FParameters& Parameters,
        const FIntRect& Viewport,
        FRHIBlendState* BlendState,
        FRHIRasterizerState* RasterizerState,
        FRHIDepthStencilState* DepthStencilState,
        uint32 StencilRef)
{
    check(PixelShader.IsValid());
    RHICmdList.SetViewport((float)Viewport.Min.X, (float)Viewport.Min.Y, 0.0f, (float)Viewport.Max.X, (float)Viewport.Max.Y,
1.0f);

    // Begin Setup Gpu Pipeline for this Pass
    FGraphicsPipelineStateInitializer GraphicsPSOInit;
    TShaderMapRef<FTriangleVS> VertexShader(GlobalShaderMap);

    RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);
    GraphicsPSOInit.BlendState = TStaticBlendState<>::GetRHI();
    GraphicsPSOInit.RasterizerState = TStaticRasterizerState<>::GetRHI();
```

```
    GraphicsPSOInit.DepthStencilState = TStaticDepthStencilState<false, CF_Always>::GetRHI();

    GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI = GTriangleVertexDeclaration.VertexDeclarationRHI;
    GraphicsPSOInit.BoundShaderState.VertexShaderRHI = VertexShader.GetVertexShader();
    GraphicsPSOInit.BoundShaderState.PixelShaderRHI = PixelShader.GetPixelShader();
    GraphicsPSOInit.PrimitiveType = PT_TriangleList;

    GraphicsPSOInit.BlendState = BlendState ? BlendState : GraphicsPSOInit.BlendState;
    GraphicsPSOInit.RasterizerState = RasterizerState ? RasterizerState : GraphicsPSOInit.RasterizerState;
    GraphicsPSOInit.DepthStencilState = DepthStencilState ? DepthStencilState : GraphicsPSOInit.DepthStencilState;
    // End Gpu Pipeline setup
    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit, StencilRef);
    SetShaderParameters(RHICmdList, PixelShader, PixelShader.GetPixelShader(), Parameters);
    DrawFullScreenTriangle(RHICmdList, 1);
}
```

I created a RenderTriangle function to wrap my Add Pass and Draw Call Functions

```
// FMyViewExtension.cpp
void FMyViewExtension::RenderTriangle
(
FRDGBuilder& GraphBuilder,
const FGlobalShaderMap* ViewShaderMap,
const FIntRect& ViewInfo,
const FScreenPassTexture& SceneColor)
{
    // Begin Setup
    // Shader Parameter Setup
    FTrianglePSParams* PassParams = GraphBuilder.AllocParameters<FTrianglePSParams>();
    // Set the Render Target In this case is the Scene Color
    PassParams->RenderTargets[0] = FRenderTargetBinding(SceneColor.Texture,ERenderTargetLoadAction::ENoAction);

    // Create FTrianglePS Pixel Shader
    TShaderMapRef<FTrianglePS> PixelShader(ViewShaderMap);

    // Add Pass
    AddFullscreenPass<FTrianglePS>(GraphBuilder,
        ViewShaderMap,
        RDG_EVENT_NAME("TranglePass"),
        PixelShader,
        PassParams,
        ViewInfo);

}
```

Binding The Delegate

I created the TrianglePass_RenderThread function.

```
FScreenPassTexture FMyViewExtension::TrianglePass_RenderThread(FRDGBuilder& GraphBuilder, const FSceneView& View, const
FPostProcessMaterialInputs& InOutInputs)
{

    const FScreenPassTexture SceneColor = InOutInputs.GetInput(EPostProcessMaterialInput::SceneColor);

    RDG_GPU_STAT_SCOPE(GraphBuilder, TrianglePass)
    RDG_EVENT_SCOPE(GraphBuilder, "TrianglePass");

    // Casting the FSceneView to FViewInfo
    const FIntRect ViewInfo = static_cast<const FViewInfo&>(View).ViewRect;
    const FGlobalShaderMap* ViewShaderMap = static_cast<const FViewInfo&>(View).ShaderMap;

    RenderTriangle(GraphBuilder,ViewShaderMap,ViewInfo,SceneColor);

    return SceneColor;
}
```

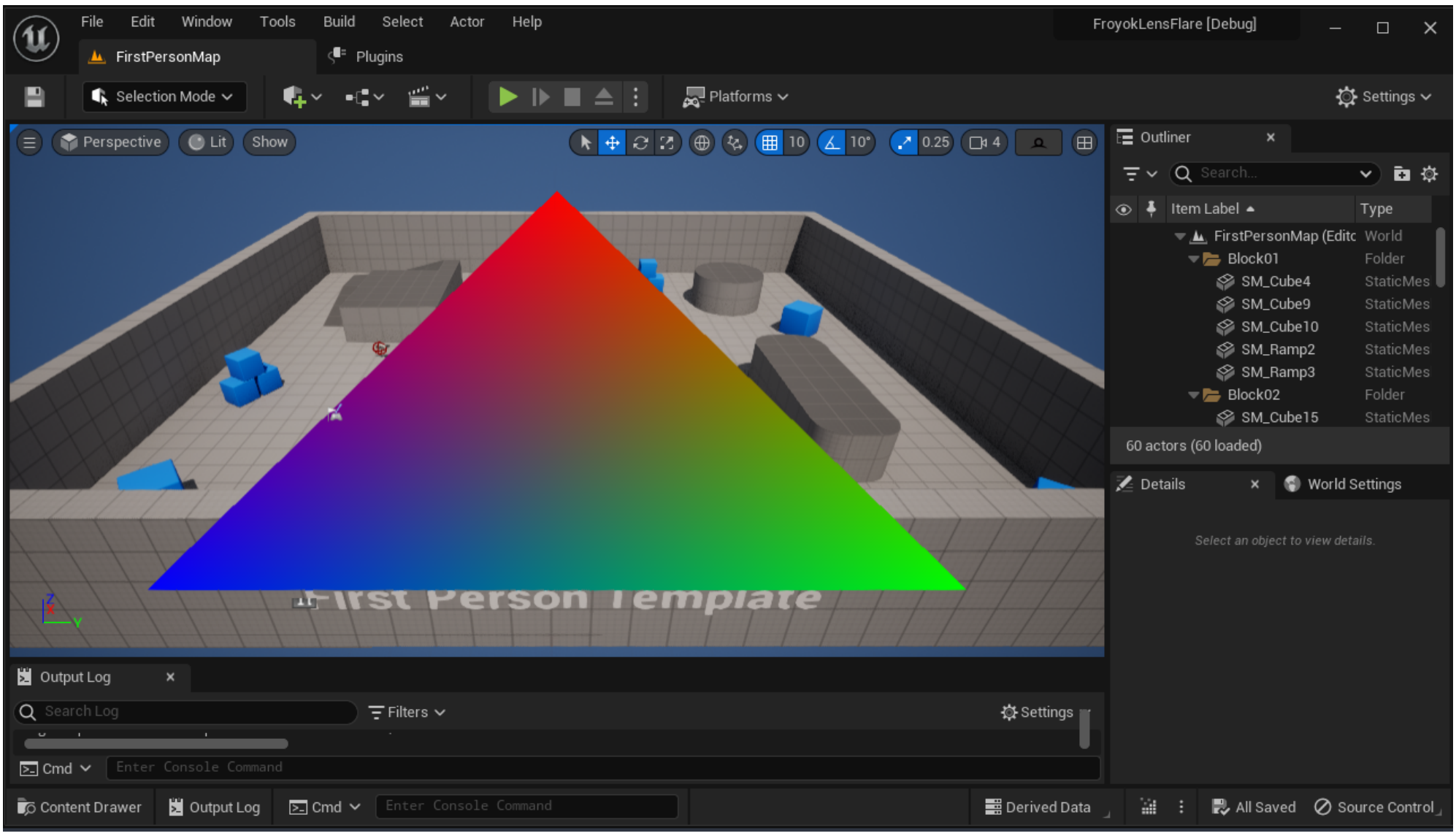
Before calling RenderTriangle, you need to get the SceneColor texture and perform a couple static_casts; one for the viewport dimensions and the other for a pointer to the global shader map. The global shader map points to our custom shader objects we declared using the global macros.

Lastly we add the delegate function TrianglePass_RenderThread inside the if statement I talked about before.

```
void FLensFlareSceneView::SubscribeToPostProcessingPass(EPostProcessingPass Pass, FAfterPassCallbackDelegateArray&
InOutPassCallbacks, bool bIsPassEnabled)
{
    if (Pass == EPostProcessingPass::Tonemap)
    {
        InOutPassCallbacks.Add(FAfterPassCallbackDelegate::CreateRaw(this, &FLensFlareSceneView::TrianglePass_RenderThread));
    }
}
```

Now Compile!

You should see the triangle drawn to the viewport of your editor.



Congratulations you've just drew your first Triangle using Unreal Engines RDG!

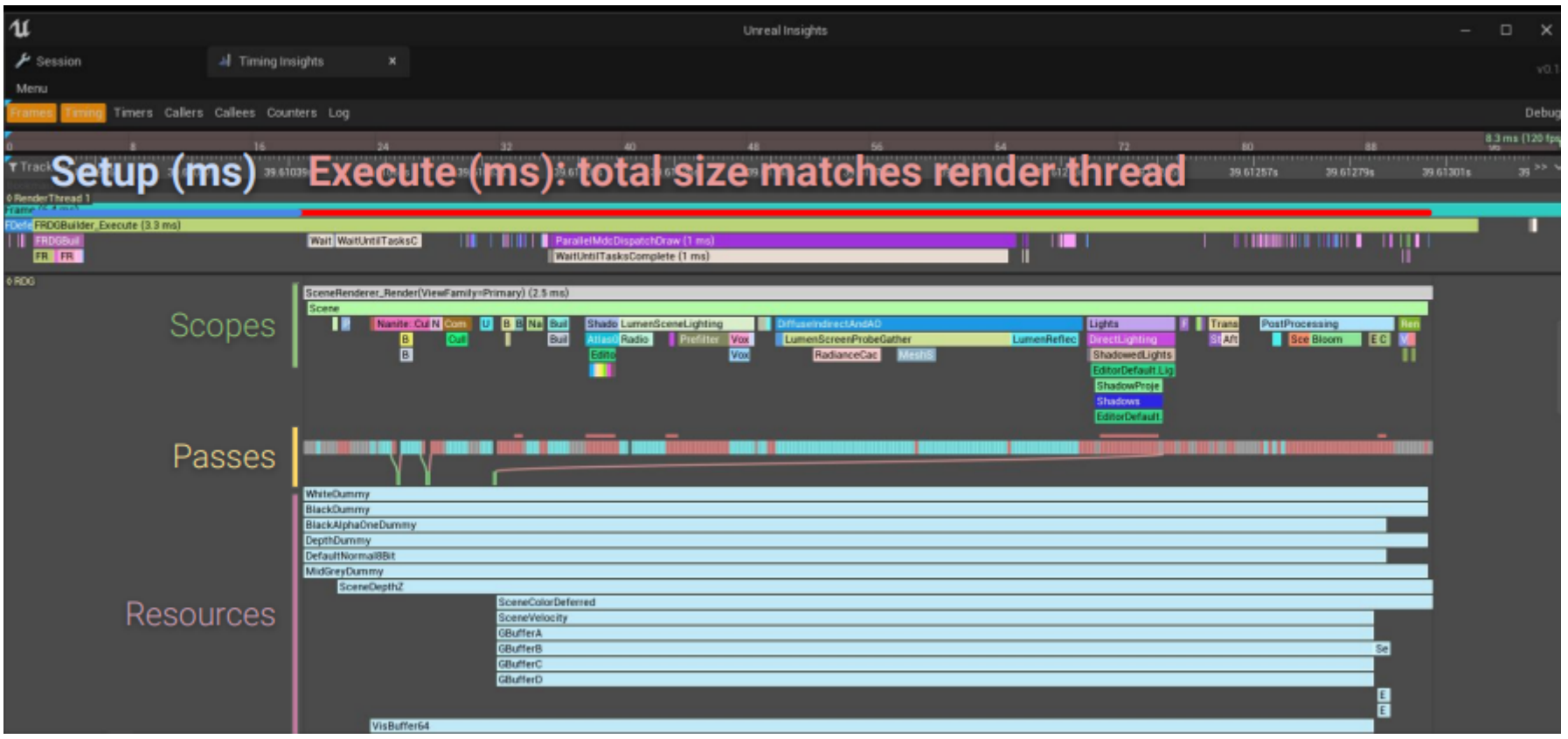
RDG Insights

Insights is a tool I highly recommend you use to profile, track and visualize the Render Dependency graph. Inside Unreal Engine there is a way to enable RDG Insights plugin so when you run Insights with your project you will be able diagnose the Render Dependency graph. This is really useful because It allows us to see how passes are associated, the resource lifetimes tied to those passes and pooled resource overlaps.

The reason this is important is because if you ever use the View Extension class to use a delegate to bind your own pass at a certain point in the render pipeline. There is a chance a resource you need from a previous pass won't be available to you as input resource to your pass. Additionally since all resources can be tracked so you can ensure several things like.

1. Why Doesn't my async compute pass overlap?
2. How is my resource being used across a frame?
3. Does my resource allocation overlap with other resources?
4. Which Resources are used by post processing?
5. is my pass culled.

Here is a visual example of what it looks like when you have it running.



To enable this startup your project and Enable the **RDG Insights** plugin by going to the main menu and selecting **Edit > Plugins > Insights**.

To run insights you need to make sure it's built in the same configuration your project is. If you built from source or not the executable should be inside the "Binaries" folder where Unreal Engine is.

"..UnrealEngine5.1EngineBinariesWin64UnrealInsights-Win64-XX.exe where XX can be Debug, Developer depending on how it was compiled.

Start your project first then insights. It will hook into the live running process. You should see the RDG track at the bottom. Scroll and zoom into the frame to see it.

WARNING: When you run a live capture of the RDG it captures a lot data. I recommend not running to long just capture what you need and close it.

References

In order for me piece this all together it took a bit of time digging through Engine Source code and googling articles. In this section I will link the references I bookmarked and put them in the order of learning. This way you can review this document in order of the references which build upon the knowledge you need to understand rendering in Unreal and basics for Graphic Programming.

Basic Graphics Programming References:

DirectX Pipeline:

https://www.youtube.com/watch?v=pfbWt1BnPIo&list=PLqCJpWy5Fohd3S7ICFXwUomYW0Wv67pDD&index=21&ab_channel=ChiliTomatoNoodle

DirectX Architecture/Swap Chain:

https://www.youtube.com/watch?v=bMxNN9dO4cI&list=PLqCJpWy5Fohd3S7ICFXwUomYW0Wv67pDD&index=19&ab_channel=ChiliTomatoNoodle

Vulkan Uniform Buffers:

https://www.youtube.com/watch?v=may_GMkfs5k&t=607s&ab_channel=BrendanGalea

Vulkan Index and Staging Buffers:

https://www.youtube.com/watch?v=qxuvQVtehII&ab_channel=BrendanGalea

Deferred Rendering:

https://www.youtube.com/watch?v=n5OiqJP2f7w&ab_channel=BenAndrew

Shader Basics:

https://www.youtube.com/watch?v=kfM-yu0iQBk&ab_channel=FreyaHolm%C3%A9r

HLSL Semantics:

https://learn.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-semantics

Unreal Engine Passes & Rendering Basics:

Unreal Engines Render Passes:

https://unrealartoptimization.github.io/book/profiling/passes/

Unreal Engine Rendering: (Drawing Policies are Deprecated)

https://medium.com/@lordned/unreal-engine-4-rendering-overview-part-1-c47f2da65346

Unreal Engine Vertex Factories:

(Basically how you're suppose to create vertex buffers, there are macros for this which you pass to RDG)

https://medium.com/realities-io/creating-a-custom-mesh-component-in-ue4-part-1-an-in-depth-explanation-of-vertex-factories-4a6fd9fd58f2

Unreal Engine RHI

Analysis of Unreal Engine Rendering System: (Chinese)

https://www.cnblogs.com/timlly/p/15156626.html#101-%E6%9C%AC%E7%AF%87%E6%A6%82%E8%BF%B0

Unreal Engine Render Dependency Graph

RDG Architecture:

https://logins.github.io/graphics/2021/05/31/RenderGraphs.html

RDG Analysis: (Chinese)

https://blog.csdn.net/qjh5606/article/details/118246059

https://juejin.cn/post/7085216072202190856

Shaders with RDG:

https://logins.github.io/graphics/2021/03/31/UE4ShadersIntroduction.html

Render Architecture:

https://ikrima.dev/ue4guide/graphics-development/render-architecture/base-usf-shaders-code-flow/

Epic Games RDG Crash Course:

https://epicgames.ent.box.com/s/ul1h44ozs0t2850ug0hrohlzm53kxwrz

Epic Games RDG Documentation:

https://docs.unrealengine.com/5.1/en-US/render-dependency-graph-in-unreal-engine/

Scene View Extension Class

Forum Post:

https://forums.unrealengine.com/t/using-sceneviewextension-to-extend-the-rendering-system/600098

Caius' Blog Global Shaders using Scene View Extension Class

https://itscai.us/blog/post/ue-view-extensions/