Analysis of Unreal Rendering System (04) - Deferred Rendering Pipeline

Table of contents

4.1 Overview

4.1.1 Overview of this article

This article mainly describes the specific implementation process of UE FDeferredShadingSceneRendererand the processes involved. This involves the key rendering technology: Deferred Shading. It is a rendering path with different steps and designs from forward rendering.

Before explainingFDeferredShadingSceneRenderer,we will first explain the technologies of rendering, forward rendering and their variants, so as to better understand the implementation of UE. More specifically, this article mainly includes the following points:

- Render Path
    - Forward rendering and its variants
    - Deferred rendering and its variants
- FDeferredShadingSceneRenderer (deferred shading scene renderer)
    - InitViews
    - PrePass
    - BasePass
    - Lighting Pass
    - TranslucentPass
    - Postprocess
- Optimization and future of deferred rendering

4.1.2 Basic Concepts

Some of the basic concepts and analysis of rendering involved in this article are as follows:

concept	abbreviation	English translation	Analysis
Render Target	RT	Render Target, Render Texture	A texture buffer located in GPU memory that can be used as a target for color writing and blending.
Geometry Buffer	GBuffer	Geometry buffer	A special Render Target, a geometric data storage container for deferred shading technology.
Rendering Path	-	Rendering Path	The steps and techniques used in the process of rendering scene objects into render textures.
Forward Shading	-	Forward Rendering	A commonly available and natively supported rendering path that is simpler and more convenient than the deferred shading path.

4.2 Deferred Rendering Technology

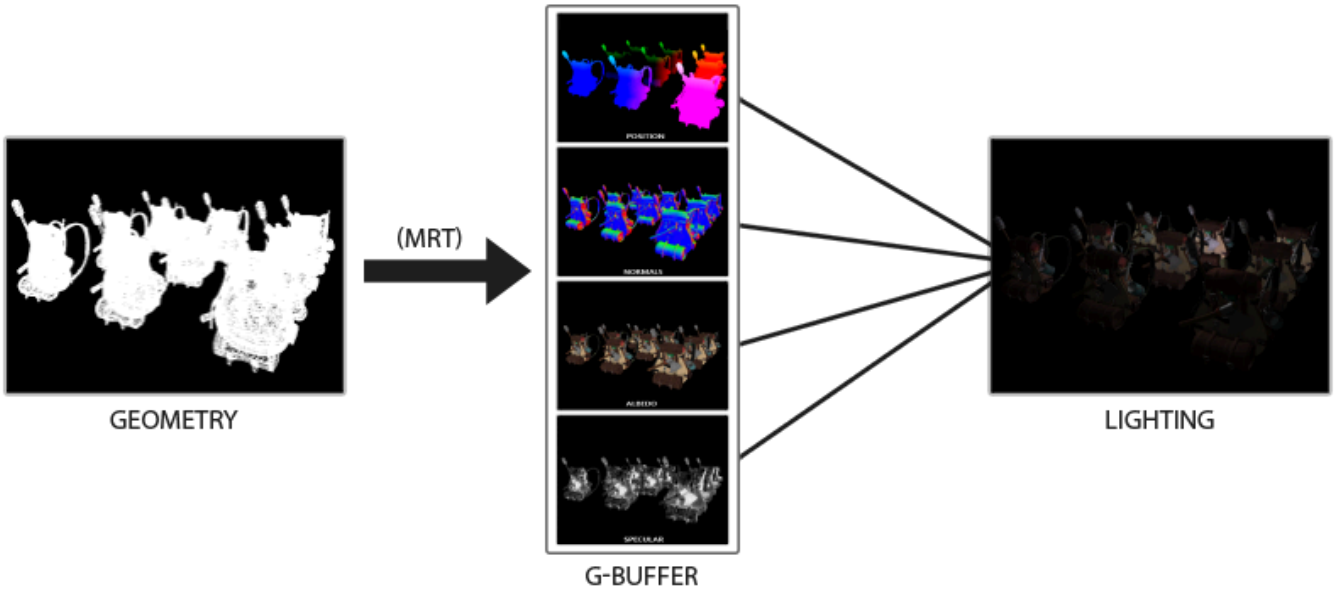
4.2.1 Introduction to Deferred Rendering

Deferred Rendering (Deferred Shading)was first proposed by Michael Deering in his 1988 paper The triangle processor and normal vector shader: a VLSI system for high performance graphics . It is a screen-space shading technology and the most common**rendering path**technology. Similar to it is**forward shading**.

Its core idea is to separate the drawing of scene objects into two passes: geometry pass and lighting pass, with the purpose of postponing the lighting pass with large computational workload and decoupling the number of objects and lighting to improve shading efficiency. It has wide and sufficient support in current mainstream renderers and commercial engines.

4.2.2 Deferred Rendering Technology

Deferred rendering is different from forward rendering in that there are two main passes: Geometry Pass (called Base Pass in UE) and Lighting Pass. Its rendering process is shown in the following figure:



There are two rendering passes in deferred rendering: the first is the geometry pass, which rasterizes the object information of the scene into multiple GBuffers; the second is the lighting stage, which uses the geometric information of the GBuffer to calculate the lighting result of each pixel.

The respective processes and functions of the two channels are as follows:

- **Geometry Channel**

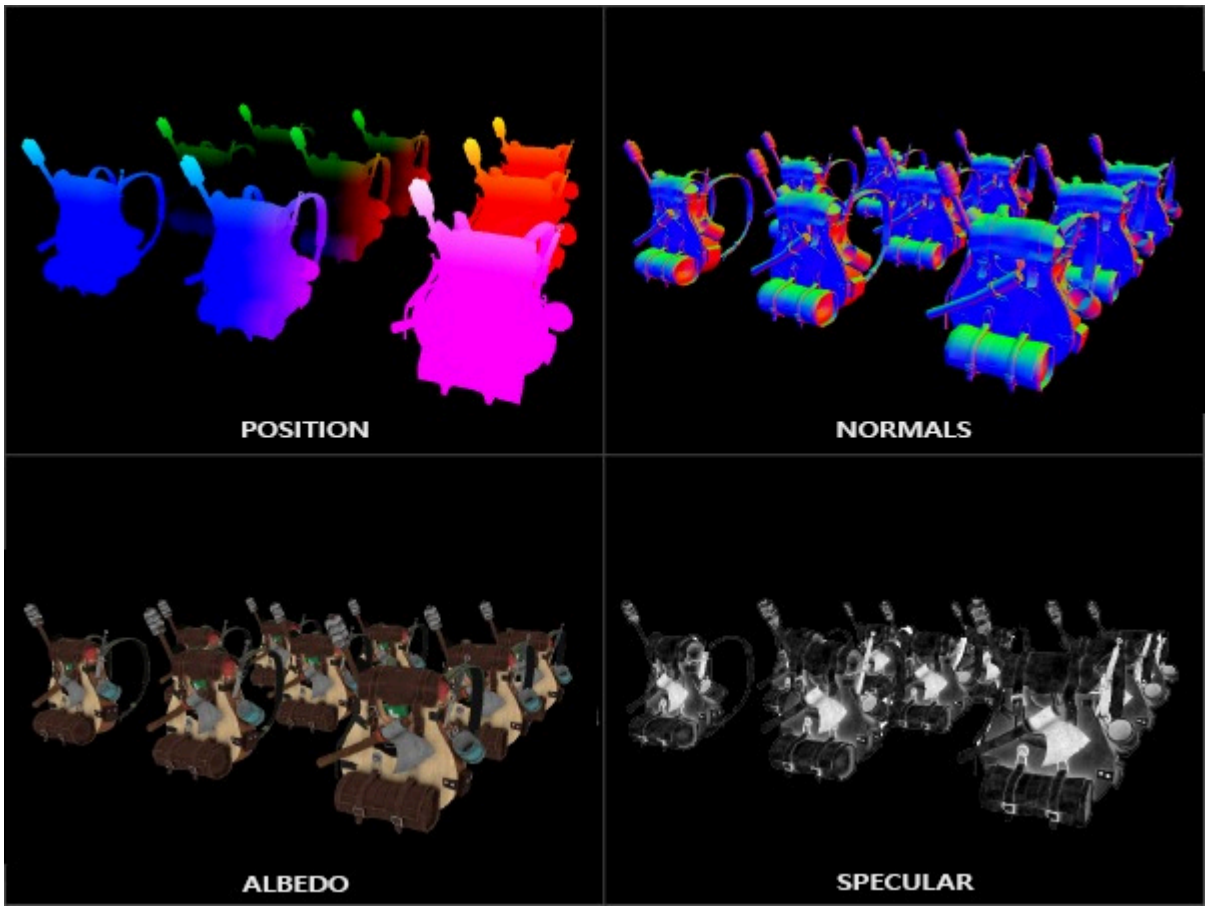
This stage renders all opaque and masked objects in the scene with unlit materials, and then writes the geometric information of the objects into the corresponding render texture.
The geometric information of the object includes:

1. Position (Position, can also be depth, because as long as there is depth and screen space UV, the position can be reconstructed).
2. Normal.

3. Material parameters (Diffuse Color, Emissive Color, Specular Scale, Specular Power, AO, Roughness, Shading Model, ...).
The pseudo code for the geometric channel is as follows:

```
void RenderBasePass()
{
    SetupGBuffer();//Set up the geometry data buffer.

    //Iterate over the non-translucent objects in the scene
    foreach(Object in OpaqueAndMaskedObjectsInScene) {

        SetUnlitMaterial(Object);          //  Setting up an unlit material
        DrawObjectToGBuffer(Object);       //  RenderingObjectThe geometric information ofGBuffer, usually inGPUCompleted in rasterization.
    }
}
```

After the above rendering, the GBuffer information of these non-transparent objects will be obtained, such as color, normal, depth, AO, etc. (see the figure below):



The deferred rendering technology obtains the GBuffer information of the object after the geometry channel, where the upper left is the position, the upper right is the normal, the lower left is the color (base color), and the lower right is the highlight.

It is important to note that the geometry channel stage does not perform lighting calculations, thereby eliminating a lot of redundant lighting calculation logic, that is, avoiding lighting calculations for many objects that are originally occluded, and avoiding overdraw.

- **Lighting Channel**

In the lighting stage, the GBuffer data generated in the geometry channel stage is used to perform lighting calculations. Its core logic is to traverse the number of pixels with the same rendering resolution, sample each pixel from the GBuffer according to its UV coordinates, and then perform lighting calculations. The pseudo code is as follows:

```
void RenderLightingPass() {

    BindGBuffer();//Binds the geometry data buffer.
    SetupRenderTarget();//Sets the render texture.

    //TraversalRTAll pixels
    foreach(pixel in RenderTargetPixels) {

        //GetGBufferdata.
        pixelData = GetPixelDataFromGBuffer(pixel.uv); //Clear
        accumulated color
        color =0;
        //Iterate over all lights and accumulate the lighting calculation results of each light into the color.
```

```
            foreach(light in Lights)

                color += CalculateLightColor(light, pixelData);
            }
            //Write color toRT.
            WriteColorToRenderTarget(color);
        }
}
```

Among them,CalculateLightColorlighting models such as Gouraud, Lambert, Phong, Blinn-Phong, BRDF, BTDF, BSSRDF, etc. can be used.

- **Pros and cons of deferred rendering**

Since the most time-consuming lighting calculation is delayed to the post-processing stage, it is decoupled from the number of objects in the scene and is only related to the size of the Render Target. The complexity is O( ). Therefore, deferred rendering is more handy in dealing with complex scenes and scenes with a large number of light sources, and can often achieve very good performance improvements.
$N_{light} \times WRT \times HRT$

However, there are also some disadvantages, such as the need for an extra channel to draw geometric information, the need for support for multiple render textures (MRT), more CPU and GPU memory usage, higher bandwidth requirements, limited material rendering types, difficulty in using hardware anti-aliasing such as MSAA, blurry images, etc. In addition, when dealing with simple scenes, you may not get any improvement in rendering performance.

4.2.3 Deferred Rendering Variants

Deferred rendering can use different optimization and improvement techniques for different platforms and APIs, resulting in many variants. Here are some of them:

4.2.3.1 Deferred Lighting(Light Pre-Pass)

Deferred Lightingis also called Light Pre-Pass. It differs from Deferred Shading in that it requires three Passes:

1. The first Pass is called Geometry Pass: it only outputs the geometric attributes (normals, depth) required for each pixel lighting calculation to the GBuffer.
2. The second Pass is called Lighting Pass: stores light source properties (such as Normal*LightDir, LightColor, Specular) in LBuffer (Light Buffer).

- Light Properties that are stored in light buffer

$$I = A + \sum_i Att_i (N.L_i * LightColor_i * D_{MaterialColor} * D_{Intensity} + (N.H_i)^n * S_{MaterialColor} * S_{Intensity})$$

- Light buffer layout

Channel 1: $\sum_i N.L_i * D_{Red} * Att_i$

Channel 2: $\sum_i N.L_i * D_{Green} * Att_i$

Channel 3: $\sum_i N.L_i * D_{Blue} * Att_i$

Channel 4: $\sum_i lum(N.L_i * (N.H_i)^n * Att_i)$

- $D_{red/green/blue}$ is the light color

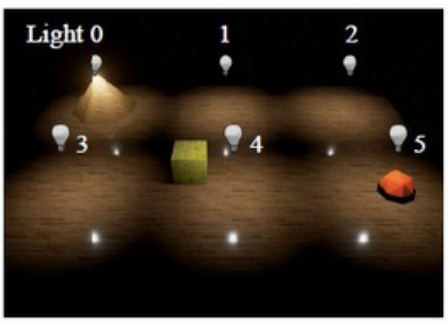
The light buffer stores light data and layout.

3. The third Pass is called Secondary Geometry Pass: Get the data of GBuffer and LBuffer, reconstruct the data required for each pixel to calculate the lighting, and perform lighting calculations.

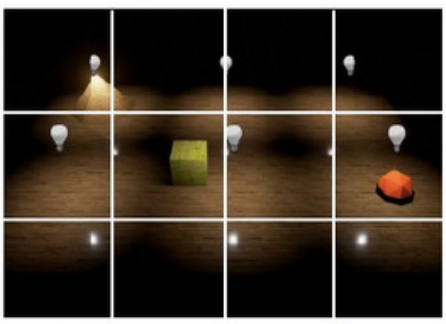
Compared with Deferred Shading, the advantage of Deferred Lighting is that the size of the G-Buffer is drastically reduced, allowing more material types to be presented, better support for MSAA, etc. The disadvantage is that the scene needs to be drawn twice, which increases the Draw Call. In addition, there is an optimized version of Deferred Lighting, which is slightly different from the above method. For details, see the documentLight Pre-Pass .

4.2.3.2 Tiled-Based Deferred Rendering~~(TBDR)~~

Tiled-Based Deferred Rendering is translated as tile-based rendering, abbreviated as**TBDR**.Its core idea is to divide the rendering texture into regular quadrilaterals (called Tiles), and then use the bounding box of the quadrilateral to eliminate useless light sources in the Tile, and only retain the list of effective light sources, thereby reducing the amount of calculation of invalid light sources in the actual lighting calculation. Its implementation diagram and specific steps are as follows:



(a) example scene



(b) screen-space tiles

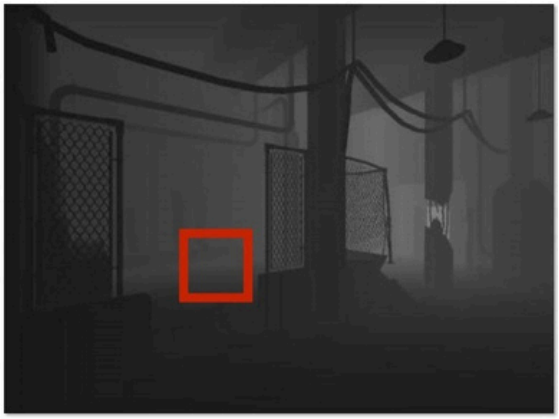
0	0, 1	1, 2	2
0, 3	0, 1, 3, 4	1, 2, 4, 5	2, 5
3	3, 4	4, 5	5

(c) tiled light list

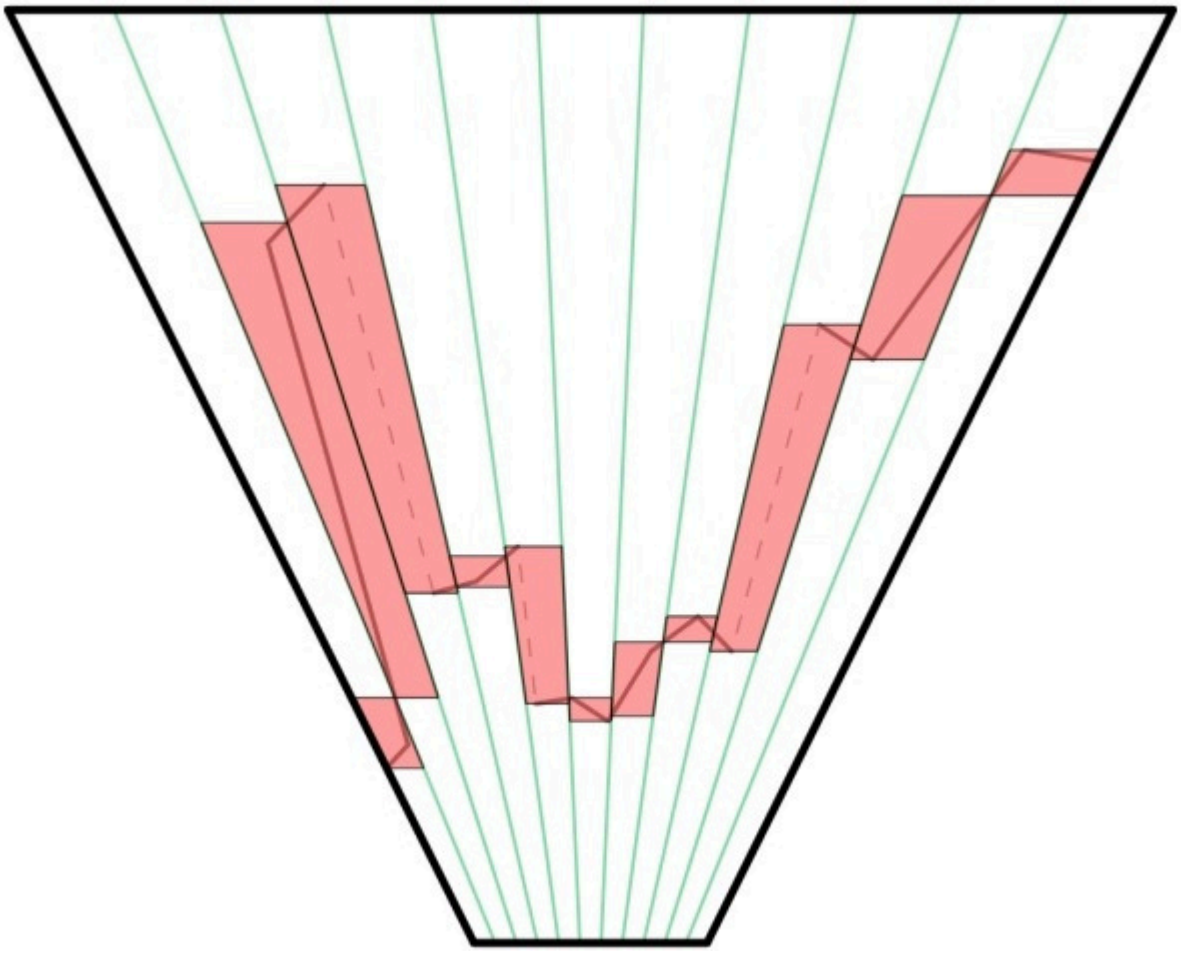
1. Divide the rendered texture into small tiles of equal size. See Figure (b) above.

Tile does not have a fixed size. It varies in different platform architectures and renderers, but is generally a power of 2. The length and width are not necessarily equal. It can be 16x16, 32x32, 64x64, etc. It should not be too small or too large, otherwise the optimization effect will not be obvious. PowerVR GPUs usually use 32x32, while ARM Mali GPUs use 16x16.

- **Small enough for a tile of the color buffer and depth buffer (potentially supersampled) to fit in a shader processor core's on-chip storage (i.e., cache)**
- **Tile sizes in range 16x16 to 64x64 pixels are common**
- **ARM Mali GPU: commonly uses 16x16 pixel tiles**



2. Calculate the Bounding Box based on the Depth range within the Tile.



The depth range of each Tile in TBDR may be different, thus obtaining Bounding Boxes of different sizes.

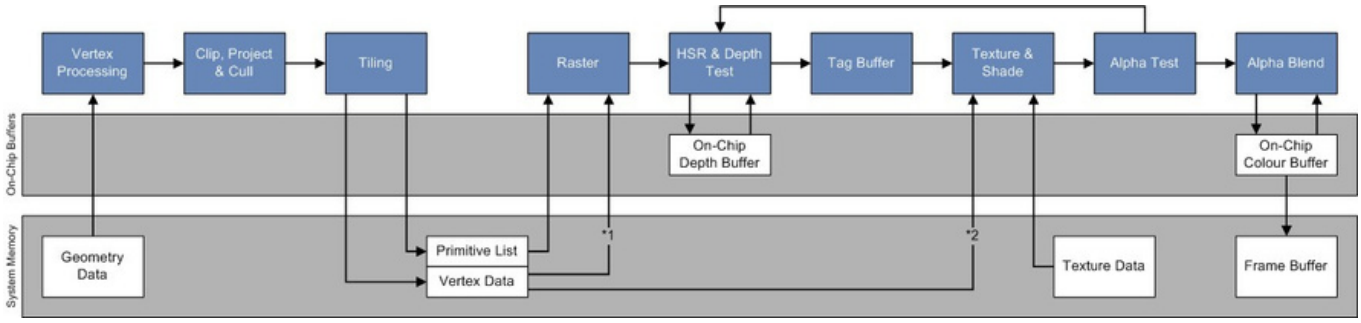
3. According to the Bounding Box of Tile and the Bounding Box of Light, perform intersection.

> Except for the directional light without attenuation, other types of light sources can obtain their Bounding Box based on the position and attenuation calculation.

4. Discard the disjoint lights and get the list of lights that have an effect on the tile. See Figure (c) above.

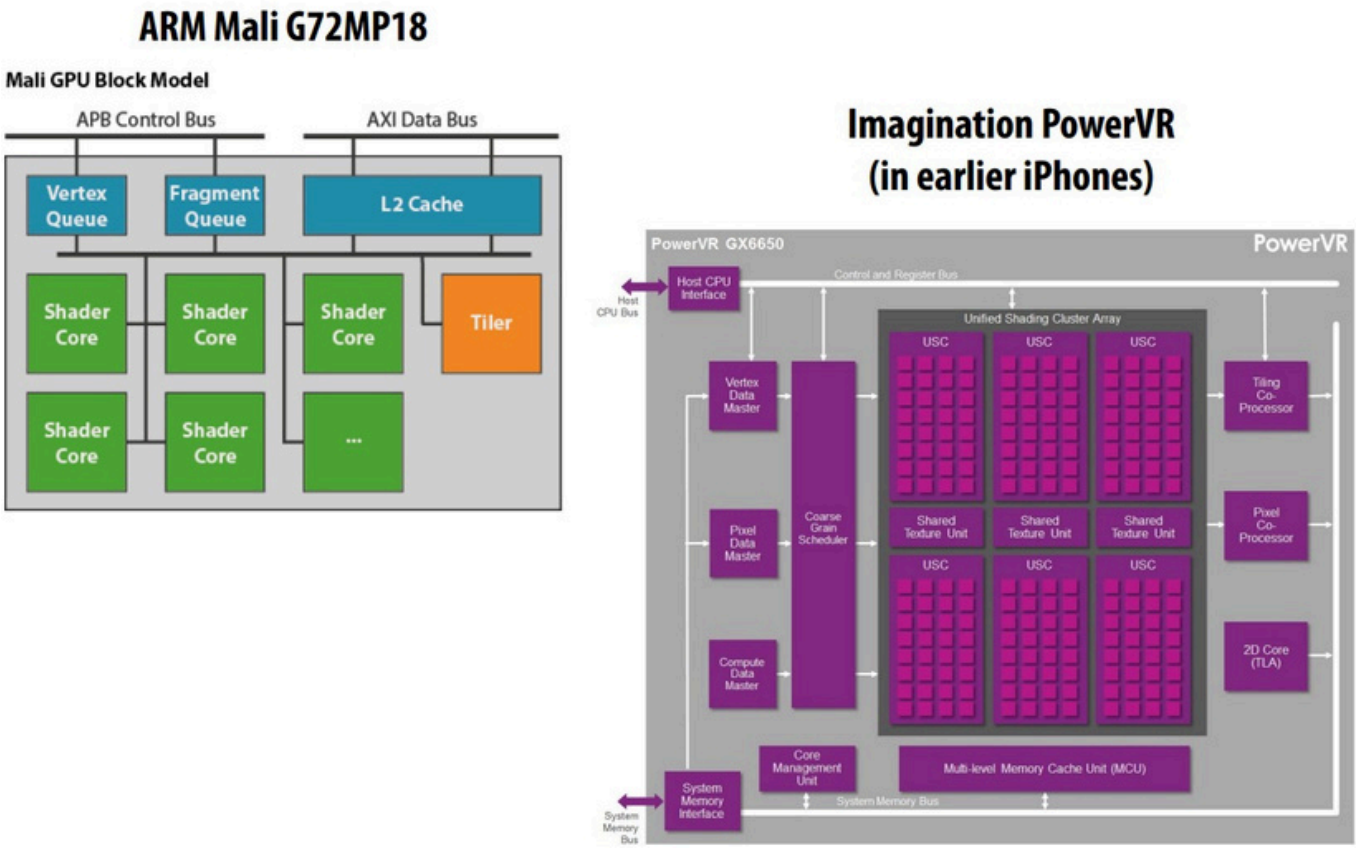
5. Traverse all tiles, obtain the index list of light sources with operations for each tile, and calculate the lighting results of all pixels in the tile.

Since TBDR can discard many useless light sources and avoid many invalid lighting calculations, it has been widely adopted in mobile GPU architectures, forming a TBDR based on hardware acceleration:



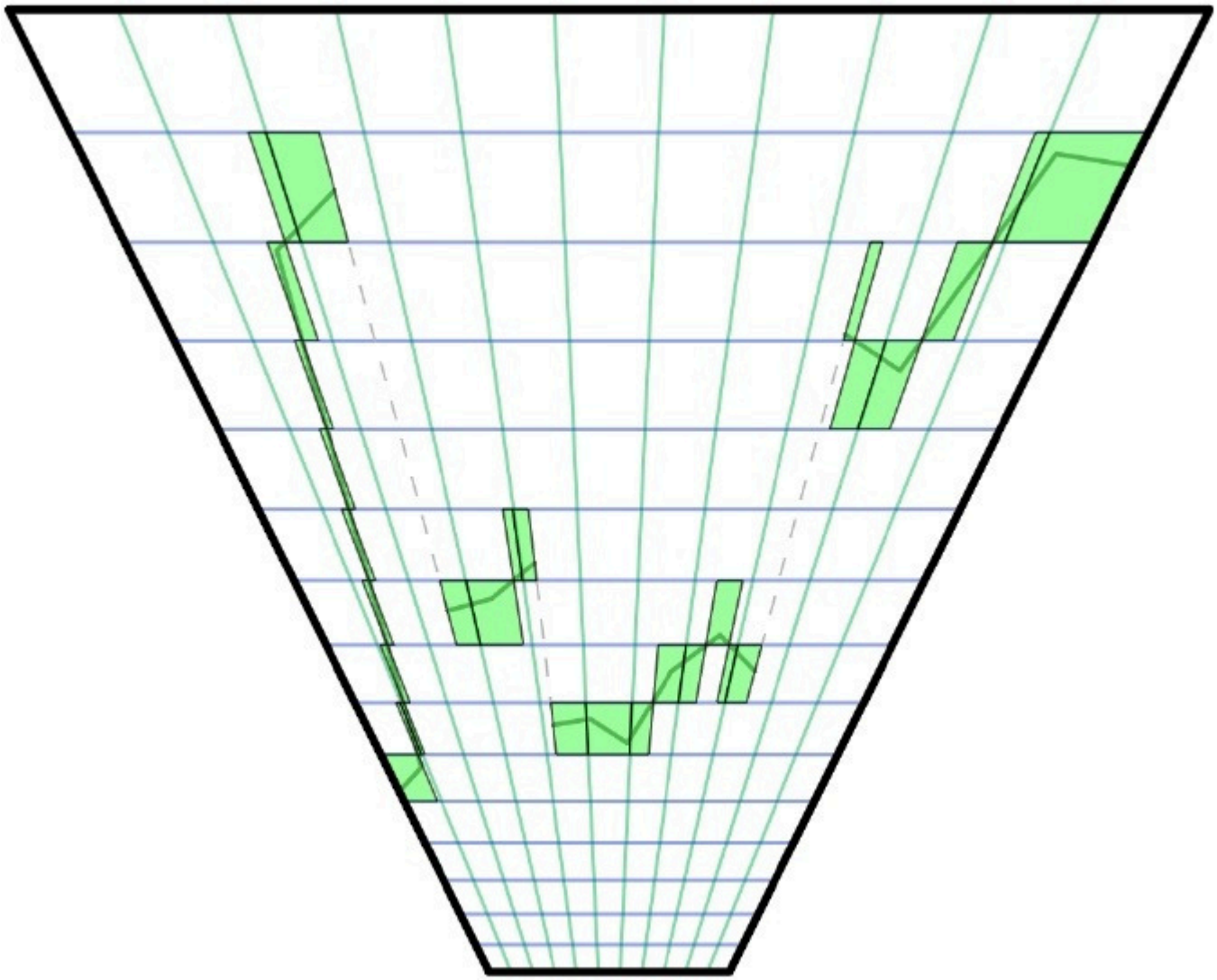
Compared with the immediate mode architecture, PowerVR's TBDR architecture adds a Tiling stage after clipping and before rasterization, and adds an On-Chip Depth Buffer and Color Buffer to access depth and color faster.

The following figure is a schematic diagram of the hardware architecture of the Series7XT series GPU of the PowerVR Rogue family:



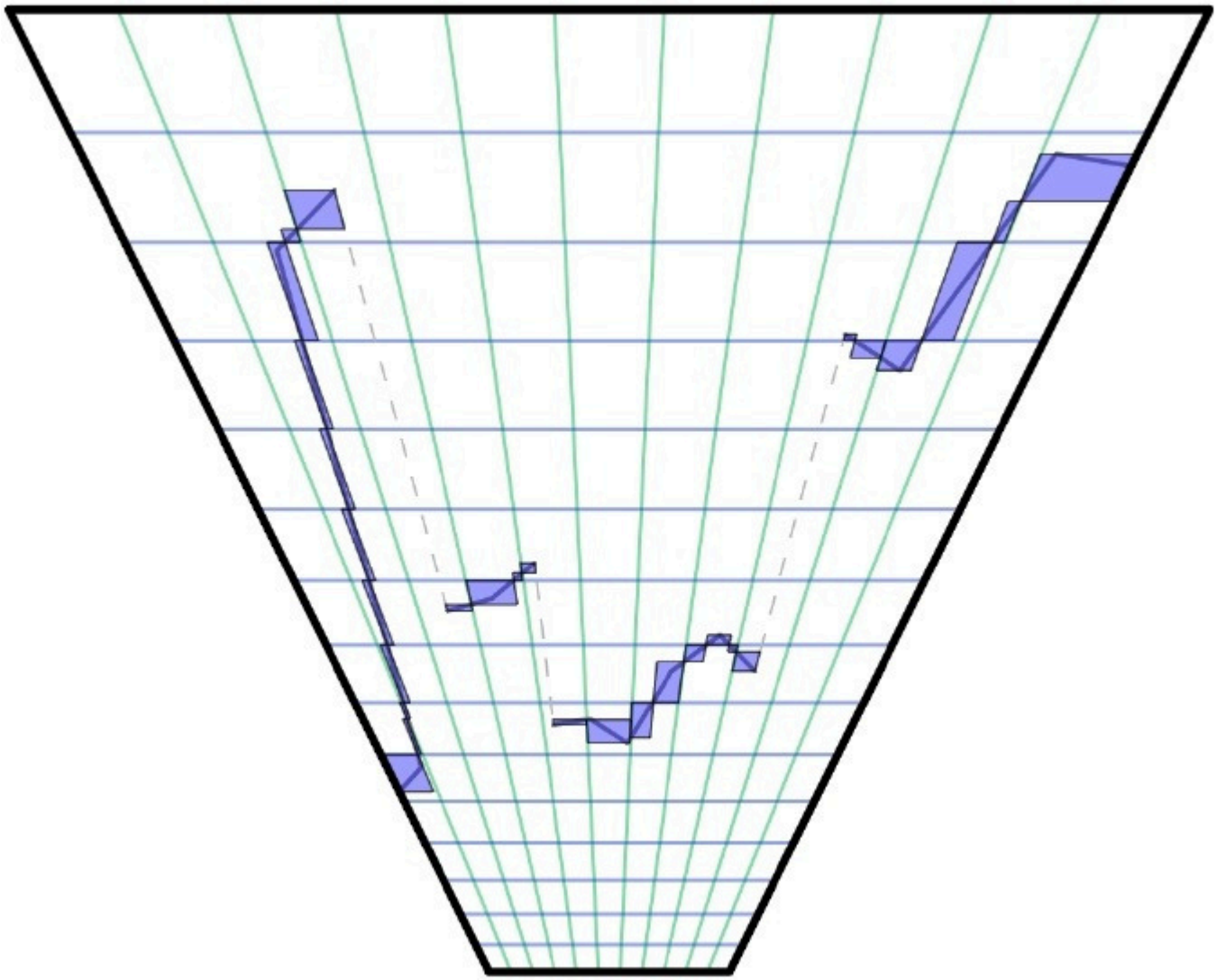
4.2.3.3 Clustered Deferred Rendering

Clustered Deferred Renderingis clustered delayed rendering. The difference from TBDR is that it divides the depth into finer granularity, thus avoiding the problem of reduced light source clipping efficiency caused by TBDR when the depth range jumps greatly (there are no valid pixels in the middle).



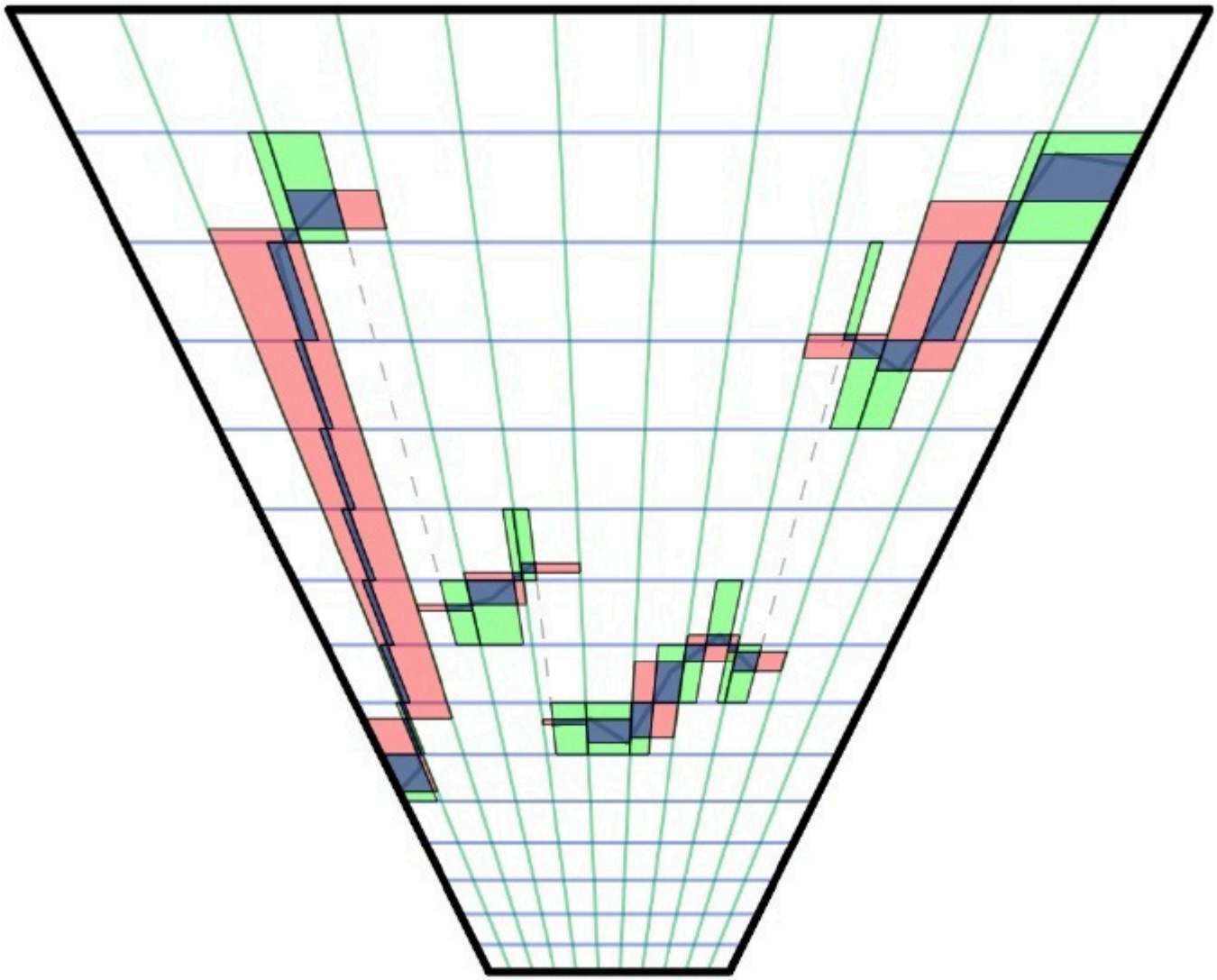
The core idea of Clustered Deferred Rendering is to subdivide the depth into several parts in a certain way, so as to more accurately calculate the bounding box of each cluster, and then more Accurately clip the light source, avoiding the reduction of light source clipping efficiency when the depth is discontinuous.

The clustering method in the figure above is called**implicit clustering**.In fact, there is**an explicit clustering method**that can further refine the depth subdivision and calculate the bounding box of each family with the actual depth range:



Explicit depth clustering locates the bounding box of each cluster more accurately.

The following figure is a comparison of Tiled, Implicit, and Explicit depth partitioning methods:

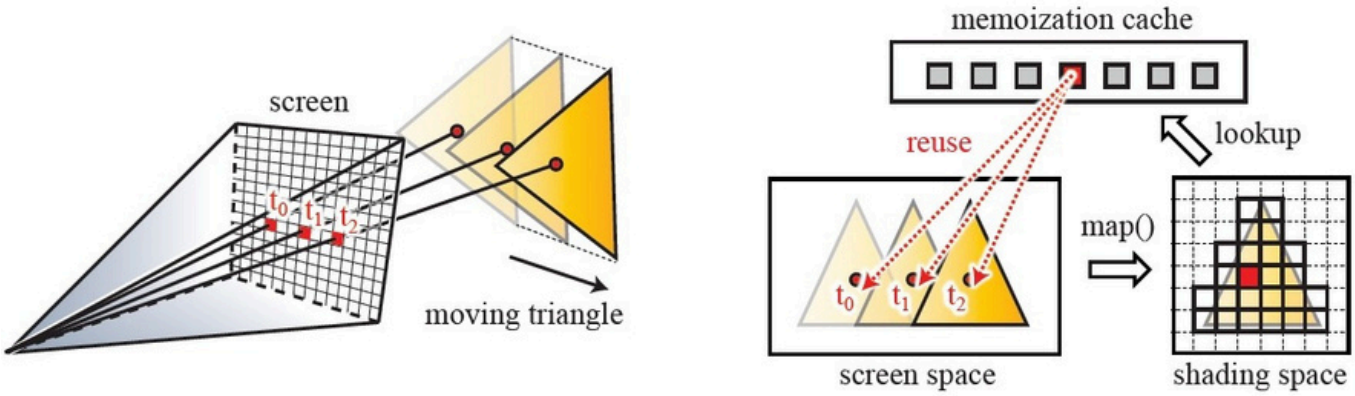


The red one is the Tiled clustering method, the green one is the Implicit clustering method, and the blue one is the Explicit clustering method. It can be seen that the bounding box obtained by the Explicit clustering method is more accurate and smaller, thereby improving the efficiency of light source clipping.

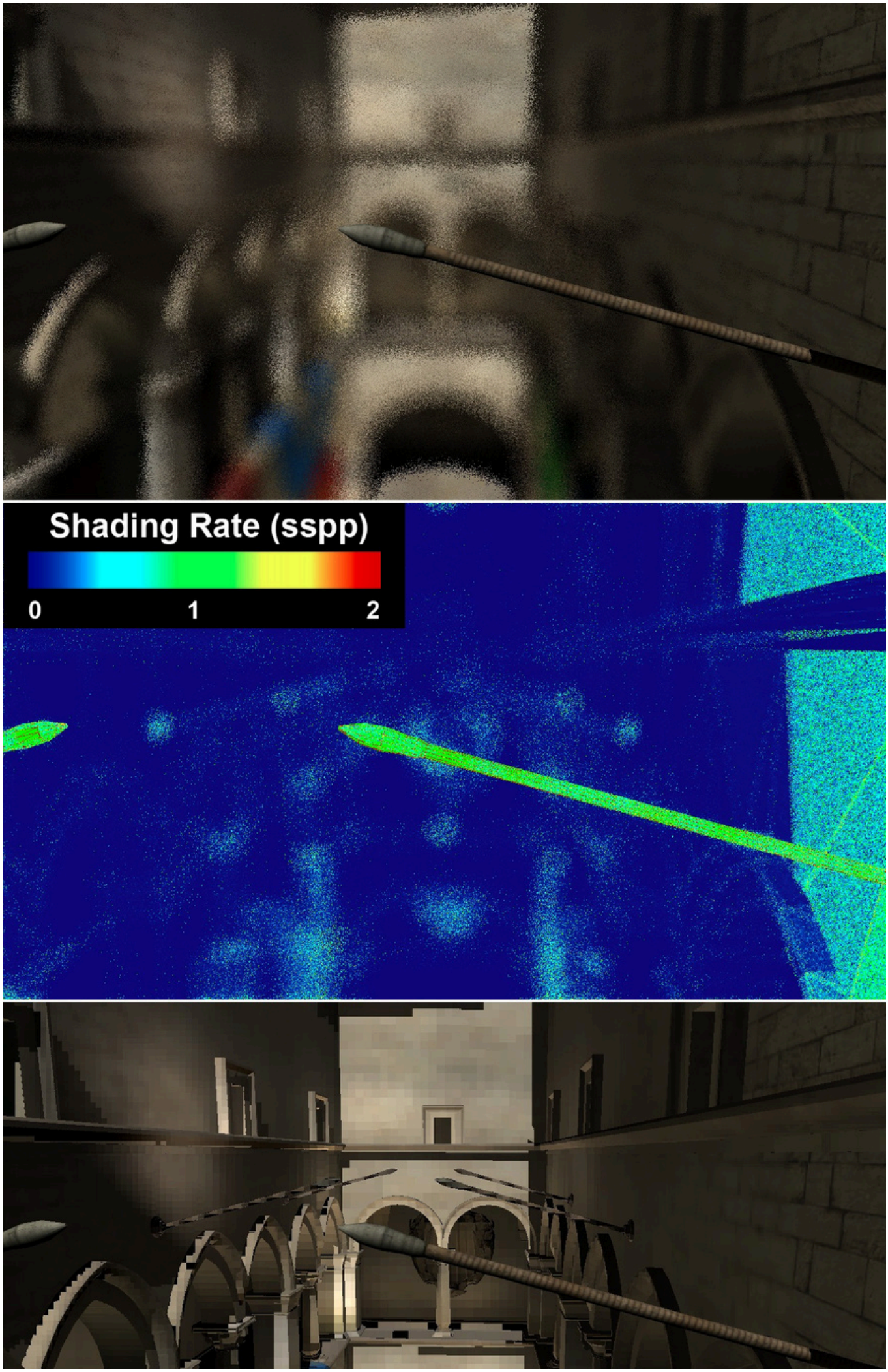
With the Clustered Deferred method, mom no longer has to worry about the computer freezing and dropping frames when rendering the following pictures O_O:

4.2.3.4 Decoupled Deferred Shading

Decoupled Deferred Shadingis an optimized deferred shading technology, first proposed by Gabor Liktor et al. in the paperDecoupled deferred shading for hardware rasterization . Its core idea is to add a compact geometry buffer to store shading sampling data (independent of visibility), and through the memoization cache (below), to compress the amount of calculation, improve the shading reuse rate of stochastic rasterization, and reduce the consumption of AA, thereby improving
shading efficiency and solving the problem that deferred shading is difficult to use MSAA.



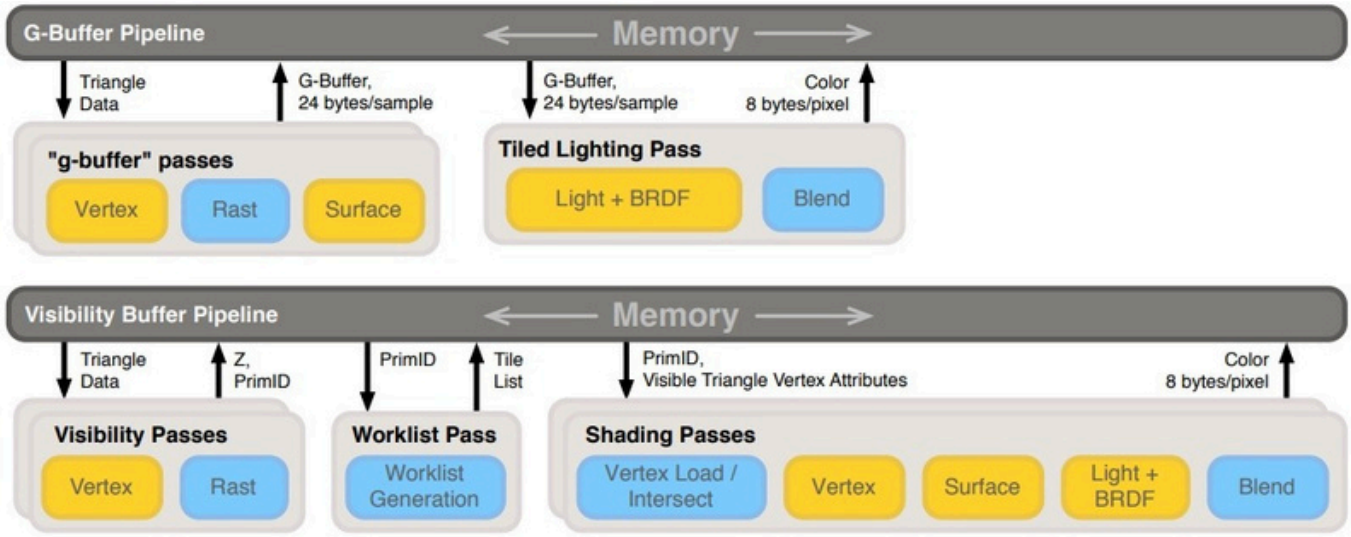
Left: A schematic diagram of the change of the projection point of a point on the surface on the screen over time. Assuming t2-t0 is small enough, their projection points on the screen will be located at the same point, so the previous shading results can be reused. Right: Decoupled Deferred Shading uses the memoization cache buffer to reuse the previous shading results, mapping the coordinates of the object to the screen space. If the UV is the same, the result of the memoization cache buffer is directly taken instead of direct lighting.


Shading Rate (sspp)
0
1
2

An overview of the effects of using Decoupled Deferred Shading. Top: Depth of field rendered with 4x supersampling; Middle: Shading Rate Per Pixel (SSPP); Bottom: Scene captured from the camera.

4.2.3.5 Visibility Buffer

Visibility Bufferis very similar to**Deferred Texturing**,and is a more daring improvement to Deferred Lighting. The core idea is: in order to reduce GBuffer usage (large GBuffer usage means large bandwidth and energy consumption), GBuffer is not rendered, but Visibility Buffer is rendered instead. Only triangles and instance IDs are stored in Visibility Buffer. With these attributes, the vertex attributes and map attributes that are really needed are read from UAV and bindless texture respectively during the lighting calculation stage (shading), and mip-map is calculated automatically based on the difference of UV (see the figure below).



Comparison diagram of GBuffer and Visibility Buffer rendering pipeline. The latter replaces the Gemotry Pass of the former with the Visiblity stage, which only records triangles and instance IDs, and can compress them into a 4-byte buffer, thus greatly reducing the usage of video memory.

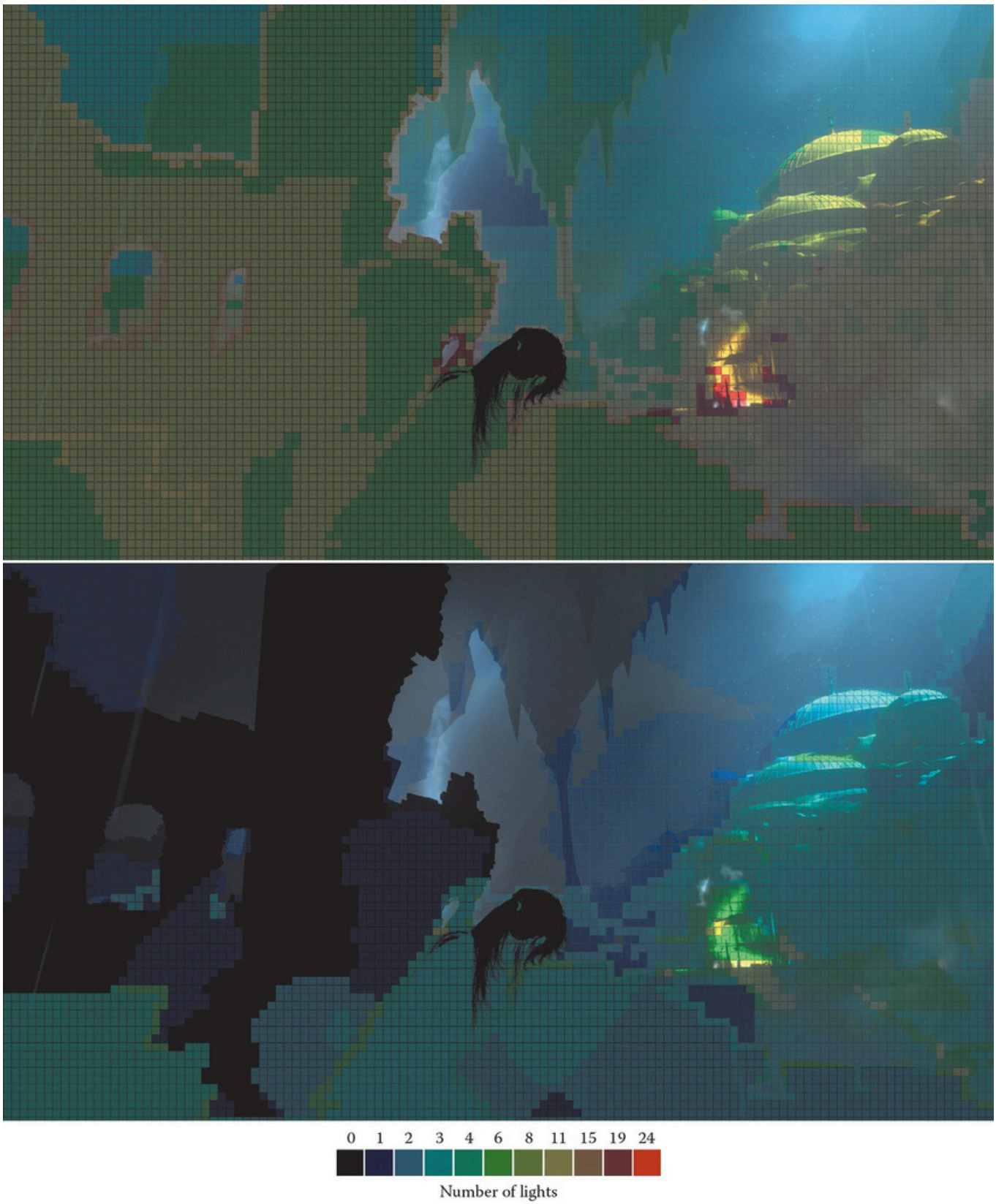
Although this method can reduce the occupancy of the buffer, it requires the support of bindless texture and is not friendly to the GPU cache (the triangle and instance ID jumps between adjacent pixels are large, reducing the spatial locality of the cache)

4.2.3.6 Fine Pruned Tiled Light Lists

Fine Pruned Tiled Light Listsis an optimized Tiled Shading. Unlike traditional tile rendering, it has two passes to calculate the intersection of objects and light sources:

The first pass calculates the AABB of the light source in screen space, and then makes a rough judgment with the AABB of the object;

The second pass performs an accurate intersection with the actual shape of the light source (not the AABB) on a tile-by-tile basis.

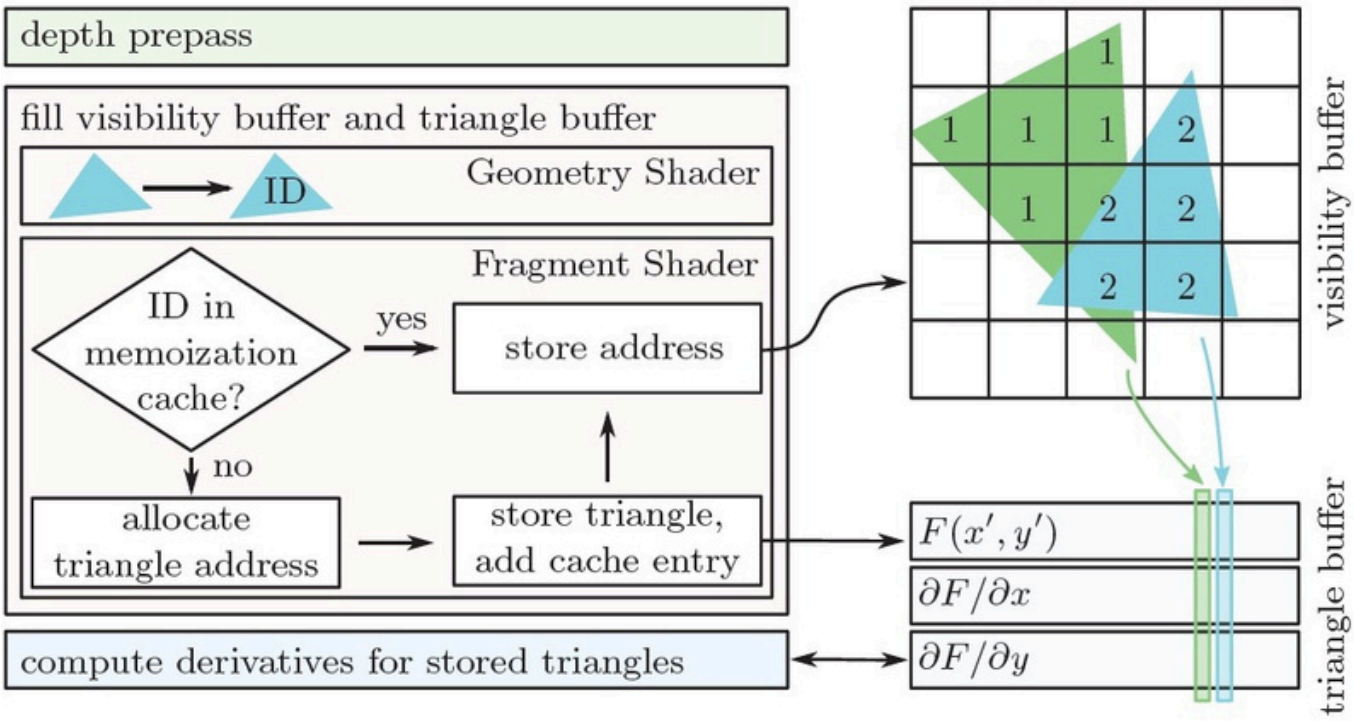


Fine Pruned Tiled Light Lists clipping diagram. The upper picture shows the result of rough clipping, and the lower picture shows the result of precise clipping. It can be seen that the number of rough clipping is generally more than that of precise clipping, but its speed is fast; precise clipping performs more accurate intersection on this basis, thereby further discarding non-intersecting light sources, reducing a large number of light sources and avoiding entering shading calculations.

This Tiled Shading can achieve irregular light sources and can be implemented using Compute Shader. It has been used in the console game "Rise of the Tomb Raider".

4.2.3.7 Deferred Attribute Interpolation Shading

Similar to Visibility Buffer,**Deferred Attribute Interpolation Shading**also solves the problem of high memory usage of traditional GBuffer. It provides a method to save triangle information instead of GBuffer information, and then calculate through interpolation to reduce the memory consumption of Deferred. This method can also use Visibility Buffer to remove redundant triangle information, further reducing the usage of video memory.

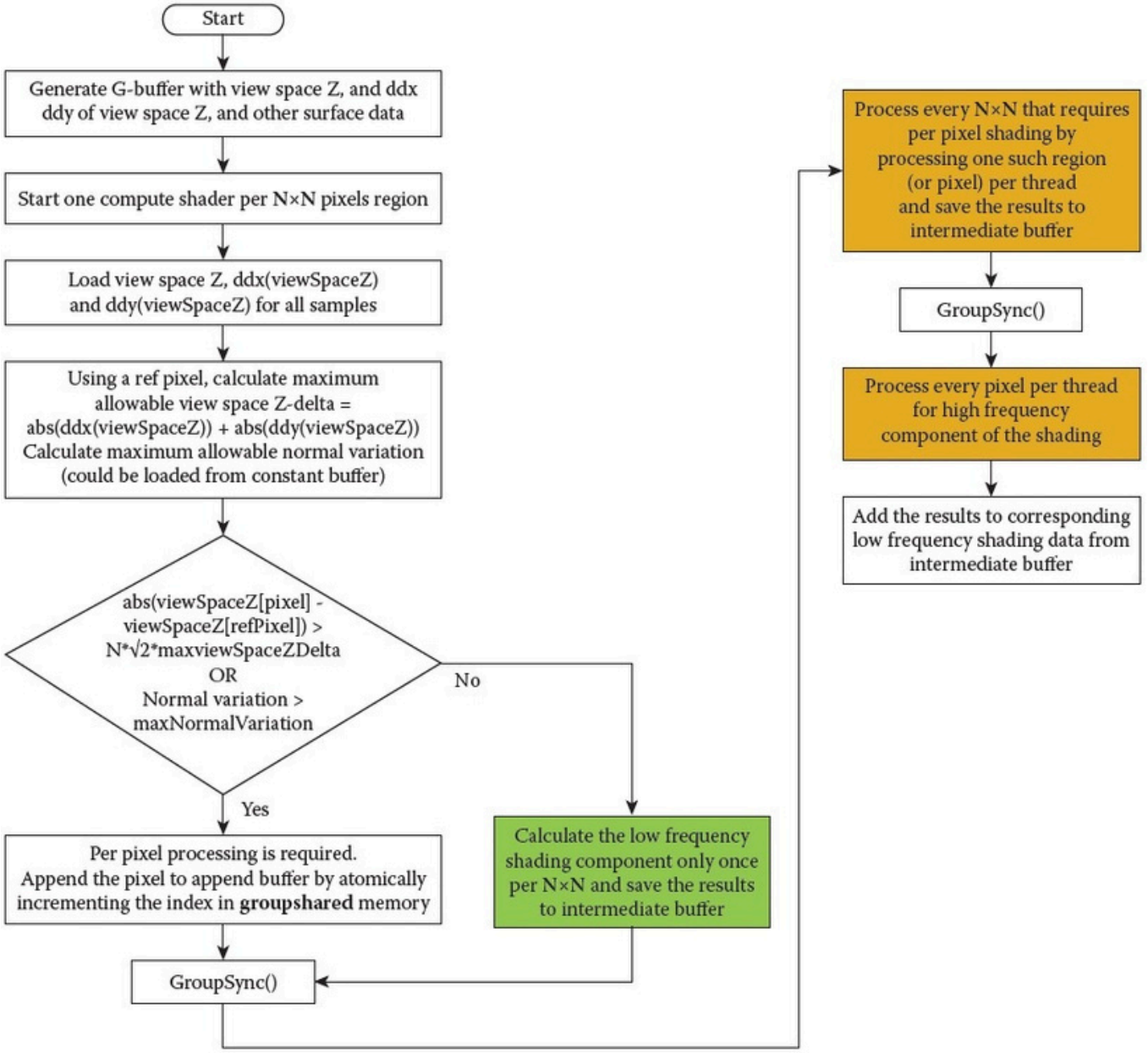


Schematic diagram of the Deferred Attribute Interpolation Shading algorithm. The first pass generates the Visibility Buffer; the second pass uses Visibility to ensure that only one triangle's information is written into the memoization cache for all pixels, where the memoization cache is a mapping buffer that records the triangle id and the actual address; the last pass calculates the screen space partial derivative of each triangle for attribute interpolation.

This method can also calculate low-frequency lighting such as GI separately to reduce shading consumption, and can also enable MSAA well.

4.2.3.8 Deferred Coarse Pixel Shading

Deferred Coarse Pixel Shadingwas proposed to solve the high consumption problem of traditional deferred rendering in the lighting calculation stage of screen space. When generating GBuffer, ddx and ddy are additionally generated, and then Compute Shader is used to find areas with little change in blocks of a certain size , and low-frequency shading is used (several pixels share one calculation, which is similar to variable rate shading), thereby improving shading efficiency and reducing consumption.



Schematic diagram of Deferred Coarse Pixel Shading in Compute Shader rendering mechanism. The key step is to calculate ddx and ddy to find pixels with low changes, reduce shading frequency, and increase reuse rate.

After rendering using this method, the performance is greatly improved when rendering the following two scenes.

Two scenes rendered with Deferred Coarse Pixel Shading. The Power Plant on the left and Sponza on the right.

The specific data is as follows:

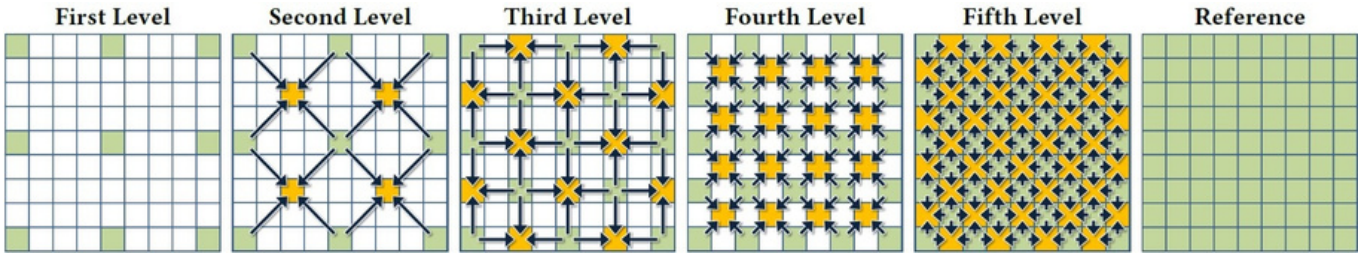
Loop Count	Power Plant			Sponza		
	Pixel (ms)	Coarse Pixel (ms)	Savings (%)	Pixel (ms)	Coarse Pixel (ms)	Savings (%)
0	22.7	11.2	50.7	12.5	9.4	24.9
100	50.3	21.1	58.1	26.8	17.6	34.3
500	87.7	34.9	60.2	43.6	27.7	36.5

The Power Plant scenario can save 50%-60%, while the Sponza scenario is between 25%-37%.

This approach also applies to screen-space post-processing.

4.2.3.9 Deferred Adaptive Compute Shading

The core idea of Deferred Adaptive Compute Shadingis to divide the screen pixels into 5 levels of pixel blocks of different granularity in a certain way, so as to decide whether to interpolate directly from the adjacent levels or recolor them.



Schematic diagram of the five levels of Deferred Adaptive Compute Shading and their interpolation. For each newly added pixel (shown in yellow), the local variance of their frame buffer and GBuffer is estimated by considering the surrounding pixels of the previous level to decide whether to interpolate from adjacent pixels or directly shade.

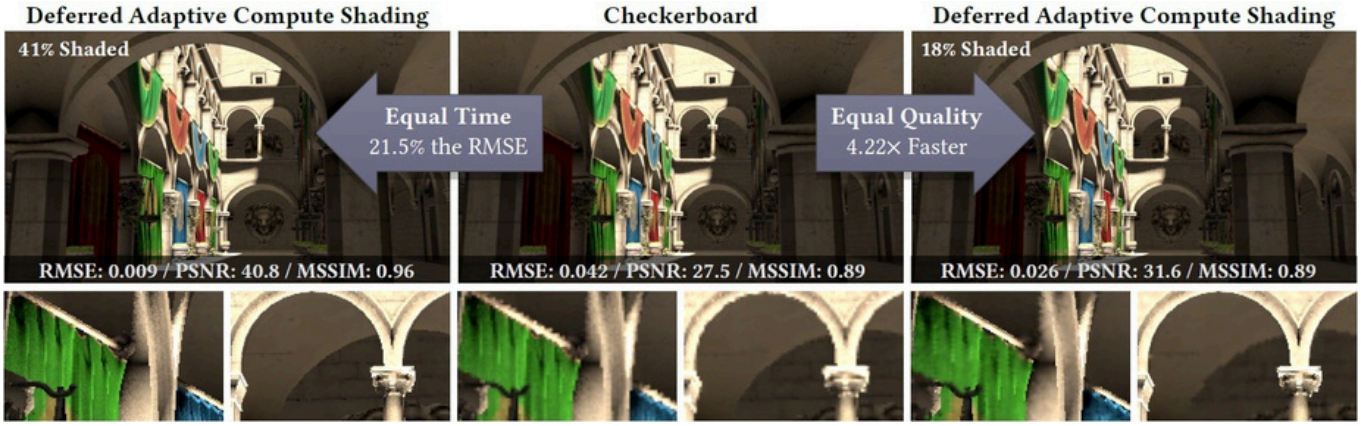
Its algorithm process is as follows:

- The first level renders 1/16 (because 4x4 blocks are taken) of the pixels.
- Traverse Levels 2 to 5 and color each level according to the following steps:
    - Calculate the similarity (Similarity Criterion) of the 4 adjacent pixels of the previous Level. If the
    - Similarity Criterion is less than a certain threshold, it is considered that this pixel is similar enough to the surrounding pixels, and the result of the current pixel is directly interpolated.
    - If not, it means that the difference between this pixel and the surrounding pixels is too large, and the shading calculation is turned on.

This method can achieve good indicators in mean square error (RMSE), peak signal-to-noise ratio (PSNR), and mean structural similarity (MSSIM) when rendering different scenes of UE4. (Figure below)

Scene Name	20% Shading Rate			50% Shading Rate			80% Shading Rate		
	RMSE	PSNR	MSSIM	RMSE	PSNR	MSSIM	RMSE	PSNR	MSSIM
Kite Demo Landscape	0.013434	37.450223	0.917924	0.006175	44.213449	0.976649	0.001992	54.152261	0.996919
Kite Demo Forest	0.014833	36.585668	0.933676	0.007348	42.700667	0.984564	0.003379	49.513014	0.992994
Xoio Berlin Flat	0.014190	37.059629	0.928814	0.006144	44.298021	0.974983	0.003052	50.316580	0.992241
Elvish Citadel	0.015064	36.508011	0.915203	0.007673	42.307247	0.966405	0.002751	51.219504	0.994643
Elemental Ice	0.011527	38.987657	0.948483	0.004451	47.111801	0.988349	0.001417	56.986239	0.998347
Elemental Lava	0.017941	34.989632	0.816162	0.010482	40.192873	0.910682	0.005681	46.208439	0.969872
Elemental Fire	0.007222	42.839877	0.978908	0.003622	49.077806	0.990305	0.002151	53.685692	0.996294

When rendering some scenes of UE4, the DACS method can obtain good image rendering indicators at different shading rates.



When rendering the same scene and screen, compared with the Checkerboard shading method, the mean square error (RMSE) of DACS is only 21.5% of the former in the same time. Under the same image quality (MSSIM), DACS is 4.22 times faster .

4.2.4 Forward rendering and its variants

4.2.4.1 Forward Rendering

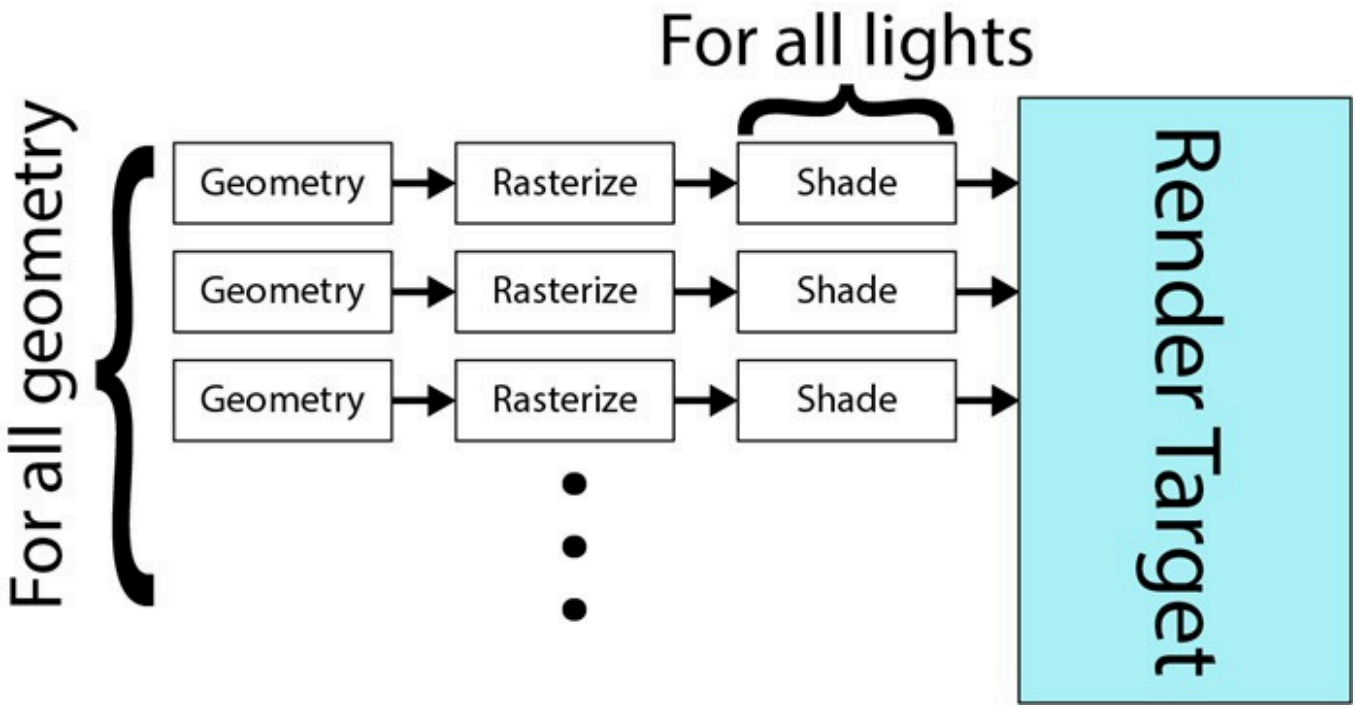
Forward Shadingis the simplest and most widely supported rendering technology. Its implementation idea is to traverse all objects in the scene, call the drawing quality once for each object, and write it to the rendering texture after rasterization through the rendering pipeline.

pseudo code for its implementation is as follows:

```
//TraversalRTAll pixels
foreach(object in ObjectsInScene) {

    color =0;
    //Iterate over all lights and accumulate the lighting calculation results of each light
    into the color. foreach(light in Lights)

        color += CalculateLightColor(light, object);
    }
    //Write color toRT.
```

```
        WriteColorToRenderTarget(color);
}
```

The rendering process diagram is as follows:



Its time complexity is , where:$O(g \cdot f \cdot l)$

- $g$represents the number of objects in the scene.
- $f$represents the number of fragments to be colored.
- $l$represents the number of light sources in the scene.

This simple technology has been naturally supported by hardware since the emergence of GPUs and has a very wide range of applications. Its advantages are simple implementation, no need for multiple pass rendering, no need for MRT support, and perfect enablement of MSAA anti-aliasing.

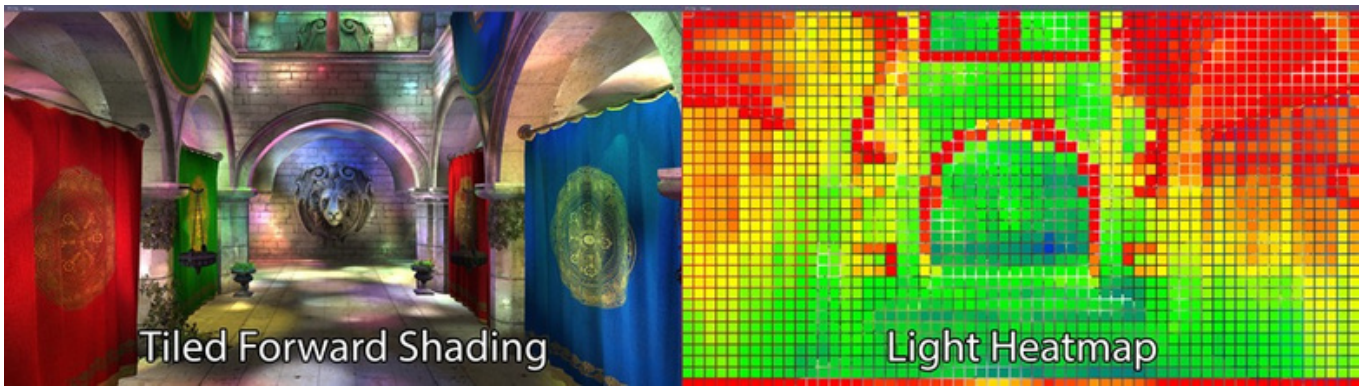
Of course, it also has its shortcomings. It cannot support the rendering of large and complex scenes with a large number of light sources, has a lot of redundant calculations, and suffers from severe over-drawing.

4.2.4.2 Forward Rendering Variants

- **Forward+ Rendering**

Forward+is also known as**Tiled Forward Rendering**.In order to increase the number of forward rendering light sources, it adds a light source culling stage with 3 passes: depth prepass, light culling pass, and shading pass.

The light culling pass is similar to the deferred rendering of tiles. The screen is divided into several tiles, each tile is intersected with the light source, and the valid light sources are written to the tile's light source list to reduce the number of light sources in the shading stage.



Forward+ may generate false positives at the geometric boundary due to the elongation of street cones (which can be improved by separating axis theorem (SAT)).

- **Cluster Forward Rendering**

Cluster Forward Renderingis similar to Cluster Deferred Rendering, which divides the screen space into equal tiles and subdivides them into clusters, so as to cut the light source in a more fine-grained manner. The algorithm is similar, so I will not repeat it here.

- **Volume Tiled Forward Rendering**

Volume Tiled Forward Renderingis a technology that expands on the forward rendering of Tiled and Clusterer. It aims to increase the number of light sources supported by the scene. The authors of the paper believe that scenes with up to 4 million light sources can be rendered in real time at 30FPS.
It consists of the following steps:

1. Initialization phase:

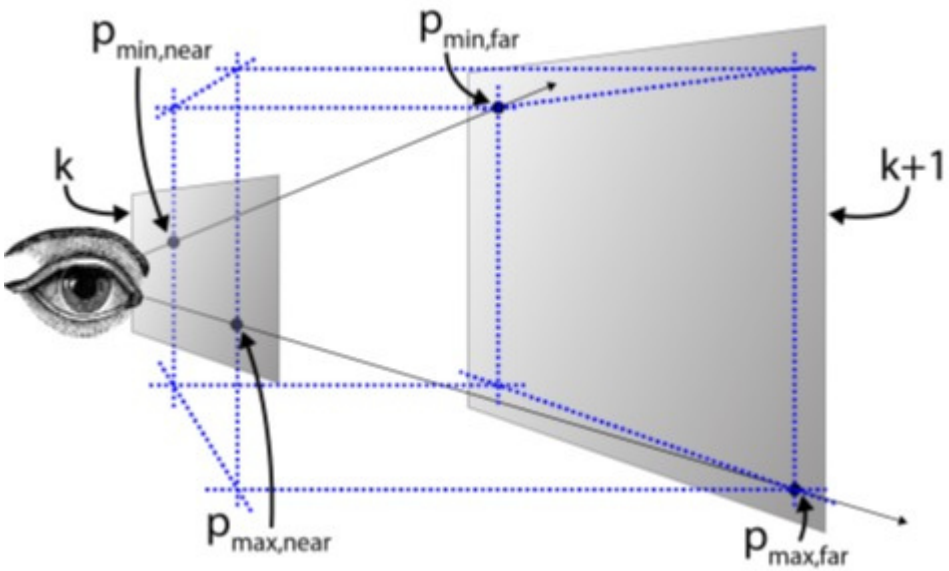
1.1 Calculate the size of the Grid (Volume Tile). Given the Tile size and the screen resolution, you can calculate the number of screen subdivisions($numSS_{(x,y)}$):

$$(Sx,Sy) = \left(\left| \frac{w}{tx} \right|, \left| \frac{h}{ty} \right| \right)$$

The number of subdivisions in the depth direction is:

$$= \left| \frac{\log(Z_{far} - Z_{near}) - SZ}{\log(1 + 2\tan\theta_y)} \right|$$

1.2 Calculate the AABB of each Volume Tile. Combined with the following figure, the AABB boundary of each Tile is calculated as follows:



$$k_{near} = Z_{near}(1 + \frac{2\tan(\theta)}{S_y})^{k}$$

$$k_{far} = Z_{near}(1 + \frac{2\tan(\theta)}{S_y})^{k+1}$$

$$p_{min} = (S_x \cdot i, S_y \cdot j)$$

$$p_{max} = (S_x \cdot (i+1), S_y \cdot (j+1))$$

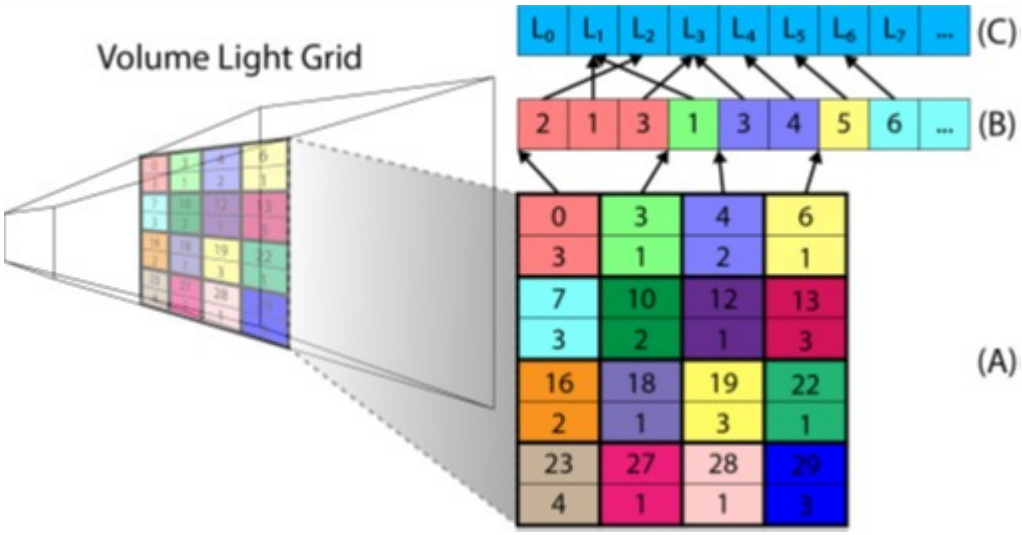
2. Update phase:

2.1 Depth Pre-pass: Only record the depth of non-translucent objects.

2.2 Mark the activated tile.

2.3 Create and compress the Tile list.

2.4 Assign light sources to Tiles. Each thread group executes an activated Volume Tile, intersects the Tile's AABB with all light sources in the scene (BVH can be used to reduce the number of intersections), and records the intersecting light source indexes into the corresponding Tile's light source list (the light source data of each Tile is the starting position of the light source list and the number of light sources):



2.5 Coloring. This stage is no different from the previous method.

Although voxel-based rendering can meet the rendering needs of massive light sources, it also has problems such as an increase in the number of Draw Calls and self-similar voxel tiles (the voxels close to the camera are very small and those far away are quite large).

4.2.5 Rendering Path Summary

4.2.5.1 Deferred Rendering vs Forward Rendering

The comparison between deferred rendering and forward rendering is shown in the following table:

	Forward Rendering	Deferred Rendering
Scene complexity	Simple	complex
Number of light sources supported	few	many
Time Complexity	$O(g \cdot f \cdot l)$	$O(f \cdot l)$
Anti-Aliasing	MSAA, FXAA, SMAA	TAA, SMAA
Video Memory and Bandwidth	Lower	Higher
Number of Passes	few	many
MRT	No requirement	Require

	Forward Rendering	Deferred Rendering
Overdraw	serious	Good avoid
Material Type	many	few
Image Quality	Clear, good anti-aliasing effect	Blurred, anti-aliasing effect compromised
Transparent objects	Opaque objects, Masked objects, Translucent objects	Opaque objects, masked objects, **do not**support semi-transparent objects
Screen resolution	Low	high
Hardware requirements	Low, covers almost 100% of hardware devices	Higher, requires MRT support, requires Shader Model 3.0+
Implementation complexity	Low	high
Post-processing	Unable to support post-processing that requires information such as normals and depth	Support post-processing that requires normal and depth information, such as SSAO, SSR, SSGI, etc.

Here are some additional points:

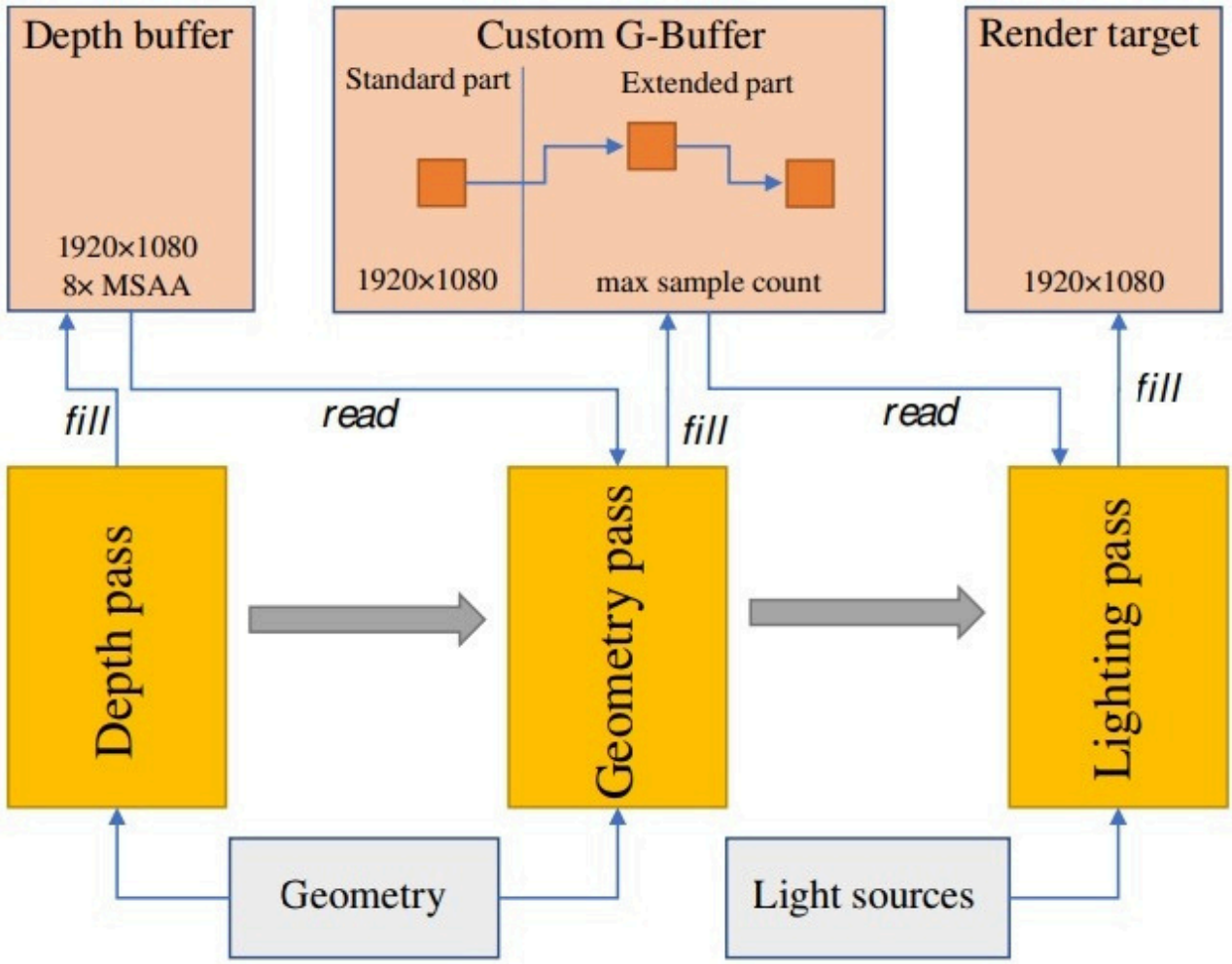
- The reason why deferred rendering does not support MSAA is that RT with MSAA needs to be resolved before performing lighting (the final color is calculated by weight of MSAA RT). Of course, it is also possible to go directly into shading without resolving, but this will lead to a surge in video memory and bandwidth.
- The reason why the delayed rendering picture is blurrier than the forward rendering is that it has undergone two rasterizations: the geometry channel and the lighting channel, which is equivalent to performing two discretizations, resulting in two signal losses, affecting the subsequent reconstruction, and causing the variance of the final picture to increase. In addition, delayed rendering will adopt TAA as an anti-aliasing technology. The core idea of TAA is to convert spatial sampling into temporal sampling, and temporal sampling is often not very accurate and has various small problems, which once again increases the variance of the picture, which may produce ghosting, moiré, flickering, and gray pictures. This is why the game screen made by UE with delayed rendering turned on by default will feel a little opaque.

> The problem of blurred image quality can be alleviated through some post-processing (high contrast retention, tone mapping) and targeted AA technology (SMAA, SSAA, DLSS), but in any case it is difficult to achieve the image quality effect of forward rendering with the same consumption.

- Deferred rendering does not support translucent objects, so it is often necessary to add a forward rendering step in the final stage to specifically render translucent objects.
- At this stage (first half of 2021), deferred rendering is the preferred rendering technology for PCs and consoles, while forward rendering is the preferred rendering technology for mobile devices. As time goes by and hardware develops, I believe that deferred rendering will soon be extended to mobile platforms.
- For material rendering types, deferred rendering can write different shading models as geometric data into the GBuffer to expand the material and lighting shading types, but it is also limited by the bandwidth of the GBuffer. If 1 byte is taken, 256 material shading types can be supported.

> UE uses 4 bits to store Shading Model ID, and can only support 16 shading models at most. I have tried to expand UE's shading types, and found that the other 4 bits are occupied, unless an additional GBuffer is opened to store it. However , it is not costeffective to have an additional GBuffer for more material types.

- Regarding MSAA for deferred rendering, some papers (Multisample anti-aliasing in deferred rendering ,Optimizing multisample anti-aliasing for deferred renderers) explain how to support MSAA, sampling strategies, data structures and bandwidth optimization under the deferred rendering pipeline.



Multisample anti-aliasing in deferred rendering is a diagram of the MSAA algorithm in deferred rendering. Its core idea is that in addition to storing the standard GBuffer, the Geometry Pass also adds additional extended data corresponding to the GBuffer. These extended data only contain the MSAA data of the pixels that need to be multisampled for use in Lighting Pass shading. This can not only meet the requirements of MSAA, but also reduce the occupancy of the GBuffer.

4.2.5.2 Comparison of Deferred Rendering Variants

This section mainly compares deferred rendering and its common variants, see the table below:

	Deferred	Tiled Deferred	Clustered Deferred
Number of light sources	Slightly more	many	a lot of
Bandwidth consumption	Very high	high	Slightly higher
Block method	none	Screen Space	Screen Space + Depth

	Deferred	Tiled Deferred	Clustered Deferred
Implementation complexity	Slightly higher	high	Very high
Applicable scenarios	Scenes with many objects and light sources	Scenes with many objects and local light sources that do not overlap	Scenes with many objects, many local light sources, non-overlapping objects, and large view space distances

4.3 Deferred Shading Scene Renderer

4.3.1 FSceneRenderer

FSceneRendererIt is the parent class of the UE scene renderer, the brain and engine of the UE rendering system, and plays an important role in the entire rendering system. It is mainly used to process and render scenes and generate rendering instructions for the RHI layer. Some of its definitions and parsing are as follows:

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

//Scene Renderer
classFSceneRenderer
{
public:
	FScene* Scene;//The rendered scene
	FSceneViewFamily ViewFamily;//The rendered scene view family (which holds allview).
	TArray<FViewInfo> Views;//Need to be renderedviewExample.

	FMeshElementCollector MeshCollector;//Grid Collector
	FMeshElementCollector RayTracingCollector;//Ray Tracing Grid Collector

	//Visible light source information
	TArray<FVisibleLightInfo,SceneRenderingAllocator> VisibleLightInfos;

	//Shadow related data
	TArray<FParallelMeshDrawCommandPass*,		SceneRenderingAllocator>
	DispatchedShadowDepthPasses;
	FSortedShadowMaps SortedShadowsForShadowDepthPass;

	//Special markings
	bool	bHasRequestedToggleFreeze;
	bool	bUsedPrecomputedVisibility;

	//A list of point lights that use global scene shadows (available throughr.SupportPointLightWholeSceneShadowsswitch)
	TArray<FName, SceneRenderingAllocator> UsedWholeScenePointLightNames;
```

```
        //platformLevelinformation
        ERHIFeatureLevel::Type FeatureLevel;
        EShaderPlatform ShaderPlatform;

        (......)

public:
        FSceneRendererconst        FSceneViewFamily*        InViewFamily,FHitProxyConsumer*
HitProxyConsumer);
        virtual ~FSceneRenderer();

        //FSceneRendererInterface (note that the part is an empty implementation and an abstract interface)

        //Rendering Entry
        virtualvoid Render(FRHICommandListImmediate& RHICmdList) =0; virtualvoid
        RenderHitProxies(FRHICommandListImmediate& RHICmdList) {}

        //Scene Renderer Instance
        staticFSceneRenderer*CreateSceneRenderer(constFSceneViewFamily* InViewFamily, FHitProxyConsumer*
        HitProxyConsumer);
        void PrepareViewRectsForRendering();

#ifWITH_MGPU//manyGPUsupport
        void ComputeViewGPUMasks(FRHIGPUMask        RenderTargetGPUMask);
#endif

        //Update eachviewThe result of the render texture
        void DoCrossGPUTransfers(FRHICommandListImmediate& RHICmdList, FRHIGPUMask
        RenderTargetGPUMask);

        //Occlusion query interface and data
        bool DoOcclusionQueries(ERHIFeatureLevel::Type InFeatureLevel)const;
        void BeginOcclusionTests(FRHICommandListImmediate& RHICmdList,boolbRenderQueries); staticFGraphEventRef

OcclusionSubmittedFence[FOcclusionQueryHelpers::MaxBufferedOcclusionFrames];
        void   FenceOcclusionTests(FRHICommandListImmediate& RHICmdList);
        void   WaitOcclusionTests(FRHICommandListImmediate& RHICmdList);
        bool ShouldDumpMeshDrawCommandInstancingStats()const{return
        bDumpMeshDrawCommandInstancingStats; }

        staticFGlobalBoundShaderState OcclusionTestBoundShaderState; static bool
        ShouldCompositeEditorPrimitives(constFViewInfo& View);

        //Wait for scene renderer to complete and clean up before final deletion
        static void WaitForTasksClearSnapshotsAndDeleteSceneRenderer(FRHICommandListImmediate& RHICmdList,
FSceneRenderer* SceneRenderer,boolbWaitForTasks =true);
        static void
DelayWaitForTasksClearSnapshotsAndDeleteSceneRenderer(FRHICommandListImmediate& RHICmdList,
FSceneRenderer* SceneRenderer);

        //Other interfaces
        staticFIntPointApplyResolutionFraction(...);
        staticFIntPointQuantizeViewRectMin(constFIntPoint& ViewRectMin);
        staticFIntPointGetDesiredInternalBufferSize(constFSceneViewFamily& ViewFamily); static
        ISceneViewFamilyScreenPercentage*ForkScreenPercentageInterface(...);

        staticint32GetRefractionQuality(constFSceneViewFamily& ViewFamily);
```

```
protected:
    (......)

#ifWITH_MGPU//manyGPUsupport
    FRHIGPUMask AllViewsGPUMask;
    FRHIGPUMaskGetGPUMaskForShadow(FProjectedShadowInfo* ProjectedShadowInfo)const;
#endif

    //----Interfaces that can be shared by all renderers----

    // --Rendering process andMeshPassRelated interfaces --
    void    OnStartRender(FRHICommandListImmediate& RHICmdList);
    void    RenderFinish(FRHICommandListImmediate& RHICmdList);
    void SetupMeshPass(FViewInfo& View, FExclusiveDepthStencil::Type
    BasePassDepthStencilAccess,    FViewCommands& ViewCommands);
    void GatherDynamicMeshElements(...);
    void    RenderDistortion(FRHICommandListImmediate& RHICmdList);
    void InitFogConstants();
    bool ShouldRenderTranslucency(ETranslucencyPass::Type TranslucencyPass)const; void
    RenderCustomDepthPassAtLocation(FRHICommandListImmediate& RHICmdList, int32 Location);

    void RenderCustomDepthPass(FRHICommandListImmediate& RHICmdList);
    void RenderPlanarReflection(class FPlanarReflectionSceneProxy* ReflectionSceneProxy); void
        InitSkyAtmosphereForViews(FRHICommandListImmediate& RHICmdList);
    void    RenderSkyAtmosphereLookUpTables(FRHICommandListImmediate& RHICmdList);
    void    RenderSkyAtmosphere(FRHICommandListImmediate& RHICmdList);
    void    RenderSkyAtmosphereEditorNotifications(FRHICommandListImmediate& RHICmdList);

    //---Shadow related interfaces ---
    void InitDynamicShadows(FRHICommandListImmediate& RHICmdList,
FGlobalDynamicIndexBuffer& DynamicIndexBuffer,          FGlobalDynamicVertexBuffer&
DynamicVertexBuffer, FGlobalDynamicReadBuffer&          DynamicReadBuffer);
    bool RenderShadowProjections(FRHICommandListImmediate& RHICmdList,const FLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture, IPooledRenderTarget*
ScreenShadowMaskSubPixelTexture,boolbProjectingForForwardShading, boolbMobileModulatedProjections,const
structFHairStrandsVisibilityViews* InHairVisibilityViews);

    TRefCountPtr<FProjectedShadowInfo> void    GetCachedPreshadow(...);
        CreatePerObjectProjectedShadow(...); SetupInteractionShadows(...);
    void    AddViewDependentWholeSceneShadowsForView(...); AllocateShadowDepthTargets
    void    (FRHICommandListImmediate& RHICmdList);
    void
    void AllocatePerObjectShadowDepthTargets(FRHICommandListImmediate& RHICmdList, ...); void
    AllocateCachedSpotlightShadowDepthTargets(FRHICommandListImmediate& RHICmdList,
. . . ) ;
    void AllocateCSMDepthTargets(FRHICommandListImmediate& RHICmdList, ...); void
    AllocateRSMDepthTargets(FRHICommandListImmediate& RHICmdList, ...);
    void AllocateOnePassPointLightDepthTargets(FRHICommandListImmediate& RHICmdList, ...); void
    AllocateTranslucentShadowDepthTargets(FRHICommandListImmediate& RHICmdList, ...); bool
    CheckForProjectedShadows(constFLightSceneInfo* LightSceneInfo)const; void
        GatherShadowPrimitives(...);
    void    RenderShadowDepthMaps(FRHICommandListImmediate& RHICmdList);
    void    RenderShadowDepthMapAtlases(FRHICommandListImmediate& RHICmdList);
    void CreateWholeSceneProjectedShadow(FLightSceneInfo* LightSceneInfo, ...); void
        UpdatePreshadowCache(FSceneRenderTargets& SceneContext); InitProjectedShadowVisibility
    void    (FRHICommandListImmediate& RHICmdList); GatherShadowDynamicMeshElements
    void    (FGlobalDynamicIndexBuffer& DynamicIndexBuffer,
```

```
FGlobalDynamicVertexBuffer& DynamicVertexBuffer, FGlobalDynamicReadBuffer&
DynamicReadBuffer);

    // --Light source interface--
    static void GetLightNameForDrawEvent(constFLightSceneProxy* LightProxy, FString& LightNameWithLevel);

    static void GatherSimpleLights(constFSceneViewFamily& ViewFamily, ...); static void
    SplitSimpleLightsByView(constFSceneViewFamily& ViewFamily, ...);

    // --Visibility Interface --
    void PreVisibilityFrameSetup(FRHICommandListImmediate& RHICmdList); void
    ComputeViewVisibility(FRHICommandListImmediate& RHICmdList, ...); void
    PostVisibilityFrameSetup(FILCUpdatePrimTaskData& OutILCTaskData);

    // --Other interfaces --
    void GammaCorrectToViewportRenderTarget(FRHICommandList& RHICmdList,constFViewInfo* View,float
    OverrideGamma);
    FRHITexture*GetMultiViewSceneColor(constFSceneRenderTargets& SceneContext)const; void
        UpdatePrimitiveIndirectLightingCacheBuffers();
    bool ShouldRenderSkyAtmosphereEditorNotifications();
    void ResolveSceneColor(FRHICommandList& RHICmdList);

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

FSceneRendererThe game thread isFRendererModule::BeginRenderingViewFamilyresponsible for creating and initializing it, and then passing it to the rendering thread. The rendering thread will call it FSceneRenderer::Render(),and after rendering, it will deleteFSceneRendererthe instance. In other words,FSceneRendererit will be created and destroyed every frame. The schematic code of the specific process is as follows:

```
void UGameEngine::Tick(floatDeltaSeconds,boolbIdleMode ) {

    UGameEngine::RedrawViewports() {

        void FViewport::Draw(boolbShouldPresent) {

            void UGameViewportClient::Draw() {

                //calculateViewFamily,ViewVarious attributes of
                ULocalPlayer::CalcSceneView(); //Sending
                rendering commands
                FRendererModule::BeginRenderingViewFamily(..., FSceneViewFamily*
ViewFamily)
                {
                    FScene*constScene = ViewFamily->Scene->GetRenderScene();

                    World->SendAllEndOfFrameUpdates(); //Creating
                    a scene renderer
                    FSceneRenderer* SceneRenderer =
FSceneRenderer::CreateSceneRenderer(ViewFamily, ...);
                    //Sends scene drawing instructions to the rendering thread.
                    ENQUEUE_RENDER_COMMAND(FDrawSceneCommand)
                    ( [SceneRenderer](FRHICommandListImmediate& {      RHICmdList)
```

```
                                RenderViewFamily_RenderThread(RHICmdList, SceneRenderer) {

                                        (......)
                                        //Call the drawing interface of the scene renderer.
                                        SceneRenderer->Render(RHICmdList); (......)


                                        //waitSceneRendererThe task is completed and then clean up and delete the instance.

FSceneRenderer::WaitForTasksClearSnapshotsAndDeleteSceneRenderer(RHICmdList, SceneRenderer)

                                        {
                                                //Wait for rendering tasks to complete before deleting instances.

RHICmdList.ImmediateFlush(EImmediateFlushType::WaitForOutstandingTasksOnly);
                                                //Waiting for all shadowsPassFinish.
                                                for(int32 PassIndex =0; PassIndex < SceneRenderer-
> DispatchedShadowDepthPasses.Num(); ++PassIndex)
                                                {
                                                        SceneRenderer->DispatchedShadowDepthPasses[PassIndex]-
> WaitForTasksAndEmpty();
                                                }
                                                (......)
                                                //deleteSceneRendererExample.
                                                delete SceneRenderer;
                                                (......)
                                        }
                                }
                                FlushPendingDeleteRHIResources_RenderThread();
                        });
                }
}}}}
```

FSceneRendererHas two subclasses:FMobileSceneRendererandFDeferredShadingSceneRenderer.

FMobileSceneRendererIt is a scene renderer for mobile platforms, which uses the forward rendering process by default.

FDeferredShadingSceneRendererAlthough it is called deferred shading scene renderer, it actually integrates two rendering paths including forward rendering and deferred rendering. It is the default scene renderer for PC and console platforms (the author was also confused by this mysterious name when he first came into contact with it).

The following chapters willFDeferredShadingSceneRendererfocus on the deferred rendering path and explain the process and mechanism of UE's deferred rendering pipeline.

4.3.2 FDeferredShadingSceneRenderer

FDeferredShadingSceneRenderer is the scene renderer of UE on PC and host platforms, which implements the logic of the delayed rendering path. Some of its definitions and declarations are as follows:

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

```
classFDeferredShadingSceneRenderer: public FSceneRenderer {

public:
    //EarlyZRelated
    EDepthDrawingMode EarlyZPassMode;
    bool bEarlyZPassMovable;
    bool bDitheredLODTransitionsUseStencil;
    int32 StencilLODMode =0;

    FComputeFenceRHIRef TranslucencyLightingVolumeClearEndFence;

    //Constructor
    FDeferredShadingSceneRenderer(const FSceneViewFamily* InViewFamily,FHitProxyConsumer*
    HitProxyConsumer);

    //Clean up the interface
    void ClearView(FRHICommandListImmediate& RHICmdList);
    void ClearGBufferAtMaxZ(FRHICommandList& RHICmdList);
    void ClearLPVs(FRHICommandListImmediate& RHICmdList);
    void UpdateLPVs(FRHICommandListImmediate& RHICmdList);

    // PrePassRelated interfaces
    bool RenderPrePass(FRHICommandListImmediate& RHICmdList, TFunctionRef<void
    ()>AfterTasksAreStarted);
    bool RenderPrePassHMD(FRHICommandListImmediate& RHICmdList); void
    RenderPrePassView(FRHICommandList& RHICmdList, ...); bool
    RenderPrePassViewParallel(constFViewInfo& View, ...);
    void PreRenderDitherFill(FRHIAsyncComputeCommandListImmediate& RHICmdList, ...); void
    PreRenderDitherFill(FRHICommandListImmediate& RHICmdList, ...); void RenderPrePassEditorPrimitives
    (FRHICommandList& RHICmdList, ...);


    // basepassinterface
    bool RenderBasePass(FRHICommandListImmediate& RHICmdList, ...); void
    RenderBasePassViewParallel(FViewInfo& View, ...);
    bool RenderBasePassView(FRHICommandListImmediate& RHICmdList, ...);

    // skypassinterface
    void RenderSkyPassViewParallel(FRHICommandListImmediate& ParentCmdList, ...); bool
    RenderSkyPassView(FRHICommandListImmediate& RHICmdList, ...);

    (.....)

    // GBuffer & Texture
    void CopySingleLayerWaterTextures(FRHICommandListImmediate& RHICmdList, ...); static void
    BeginRenderingWaterGBuffer(FRHICommandList& RHICmdList, ...); void FinishWaterGBufferPassAndResolve
    (FRHICommandListImmediate& RHICmdList, ...);


    //Water Rendering
    bool RenderSingleLayerWaterPass(FRHICommandListImmediate& RHICmdList, ...); bool
    RenderSingleLayerWaterPassView(FRHICommandListImmediate& RHICmdList, ...); void
    RenderSingleLayerWaterReflections(FRHICommandListImmediate& RHICmdList, ...);

    //    Rendering process related
    //    Rendering main entry
    virtualvoid Render(FRHICommandListImmediate& RHICmdList) override; void RenderFinish
    (FRHICommandListImmediate& RHICmdList);


    //Other rendering interfaces
```

```
    bool   RenderHzb(FRHICommandListImmediate& RHICmdList);
    void   RenderOcclusion(FRHICommandListImmediate& FinishOcRcHluICsimondList);
    void (FRHICommandListImmediate&                    RHICmdList);
    virtualvoid RenderHitProxies(FRHICommandListImmediate& RHICmdList) override;

private:
    //Static data (used for variouspass
    Initialization) staticFGraphEventRef
TranslucencyTimestampQuerySubmittedFence[FOcclusionQueryHelpers::MaxBufferedOcclusionFrame s +
    1];
    static   FGlobalDynamicIndexBuffer      DynamicIndexBufferForInitViews;
    static   FGlobalDynamicIndexBuffer      DynamicIndexBufferForInitShadows;
    static   FGlobalDynamicVertexBuffer     DynamicVertexBufferForInitViews;
    static   FGlobalDynamicVertexBuffer     DynamicVertexBufferForInitShadows;
    static   TGlobalResource<FGlobalDynamicReadBuffer>      DynamicReadBufferForInitViews;
    static   TGlobalResource<FGlobalDynamicReadBuffer>      DynamicReadBufferForInitShadows;

    //Visibility Interface
    void PreVisibilityFrameSetup(FRHICommandListImmediate& RHICmdList);

    //initializationviewdata
    bool InitViews(FRHICommandListImmediate& RHICmdList, ...);
    void InitViewsPossiblyAfterPrepass(FRHICommandListImmediate& RHICmdList, ...);

    void   SetupSceneReflectionCaptureBuffer(FRHICommandListImmediate& RHICmdList);
    void
UpdateTranslucencyTimersAndSeparateTranslucencyBufferSize(FRHICommandListImmediate& RHICmdList);

    void CreateIndirectCapsuleShadows();

    //Rendering Fog
    bool RenderFog(FRHICommandListImmediate& RHICmdList, ...);
    void RenderViewFog(FRHICommandList& RHICmdList,constFViewInfo& View, ...);

    //Atmosphere, sky, indirect light, distance field, ambient light
    void RenderAtmosphere(FRHICommandListImmediate& RHICmdList, ...); void
        RenderDebugSkyAtmosphere(FRHICommandListImmediate& RHICmdList);
    void   RenderDiffuseIndirectAndAmbientOcclusion(FRHICommandListImmediate&    RHICmdList);
    void   RenderDeferredReflectionsAndSkyLighting(FRHICommandListImmediate&    RHICmdList,
...);
    void   RenderDeferredReflectionsAndSkyLightingHair(FRHICommandListImmediate&    RHICmdList,
...);
    void   RenderDFAOAsIndirectShadowing(FRHICommandListImmediate& RHICmdList,...);
    bool   RenderDistanceFieldLighting(FRHICommandListImmediate& RHICmdList,...);
    void RenderDistanceFieldAOScreenGrid(FRHICommandListImmediate& RHICmdList, ...); void
    RenderMeshDistanceFieldVisualization(FRHICommandListImmediate& RHICmdList, ...);

    //   - - - - TiledLight source interface----
    //   calculateTileModeTileData to trimTileNon-functional light source. Can only be used inforward    Shading,clustered
deferred    Shadingand enableSurface lightingTransparent mode.
    void ComputeLightGrid(FRHICommandListImmediate& RHICmdList, ...); bool
    CanUseTiledDeferred()const;
    bool ShouldUseTiledDeferred(int32 NumTiledDeferredLights)const; bool
        ShouldUseClusteredDeferredShading()const;
    bool AreClusteredLightsInLightGrid()const;
    void AddClusteredDeferredShadingPass(FRHICommandListImmediate& RHICmdList, ...); void
    RenderTiledDeferredLighting(FRHICommandListImmediate& RHICmdList,const
TArray<FSortedLightSceneInfo, SceneRenderingAllocator>& SortedLights, ...);
```

```
    void GatherAndSortLights(FSortedLightSetSceneInfo& OutSortedLights); void RenderLights
    (FRHICommandListImmediate& RHICmdList, ...);
    void RenderLightArrayForOverlapViewmode(FRHICommandListImmediate& RHICmdList, ...); void
    RenderStationaryLightOverlap(FRHICommandListImmediate& RHICmdList);

    //----Rendering light sources (direct light, indirect light, ambient light, static light,
    volumetric light)---- void RenderLight(FRHICommandList& RHICmdList, ...);
    void RenderLightsForHair(FRHICommandListImmediate& RHICmdList, ...); void
    RenderLightForHair(FRHICommandList& RHICmdList, ...);
    void RenderSimpleLightsStandardDeferred(FRHICommandListImmediate& RHICmdList, ...); void
    ClearTranslucentVolumeLighting(FRHICommandListImmediate& RHICmdListViewIndex,
...);
    void InjectAmbientCubemapTranslucentVolumeLighting(FRHICommandList& RHICmdList, ...); void
    ClearTranslucentVolumePerObjectShadowing(FRHICommandList& RHICmdList,constint32 ViewIndex);

    void AccumulateTranslucentVolumeObjectShadowing(FRHICommandList& RHICmdList, ...); void
    InjectTranslucentVolumeLighting(FRHICommandListImmediate& RHICmdList, ...); void
    InjectTranslucentVolumeLightingArray(FRHICommandListImmediate& RHICmdList, ...); void
    InjectSimpleTranslucentVolumeLightingArray(FRHICommandListImmediate& RHICmdList,
...);
    void FilterTranslucentVolumeLighting(FRHICommandListImmediate& RHICmdList, ...);

    // Light Function
    bool RenderLightFunction(FRHICommandListImmediate& RHICmdList, ...);
    bool RenderPreviewShadowsIndicator(FRHICommandListImmediate& RHICmdList, ...); bool
    RenderLightFunctionForMaterial(FRHICommandListImmediate& RHICmdList, ...);

    //transparentpass
    void RenderViewTranslucency(FRHICommandListImmediate& RHICmdList, ...); void
    RenderViewTranslucencyParallel(FRHICommandListImmediate& RHICmdList, ...); void
    BeginTimingSeparateTranslucencyPass(FRHICommandListImmediate& RHICmdList,const FViewInfo& View);

    void EndTimingSeparateTranslucencyPass(FRHICommandListImmediate& RHICmdList,const FViewInfo&
            View);
    void   BeginTimingSeparateTranslucencyModulatePass(FRHICommandListImmediate& RHICmdList,
...);
    void   EndTimingSeparateTranslucencyModulatePass(FRHICommandListImmediate& RHICmdList,
...);
    void   SetupDownsampledTranslucencyViewParameters(FRHICommandListImmediate& RHICmdList,
...);
    void   ConditionalResolveSceneColorForTranslucentMaterials(FRHICommandListImmediate&
RHICmdList,    ...);
    void RenderTranslucency(FRHICommandListImmediate& RHICmdList,bool
bDrawUnderwaterViews =false);
    void RenderTranslucencyInner(FRHICommandListImmediate& RHICmdList, ...);

    //beam
    void RenderLightShaftOcclusion(FRHICommandListImmediate&                RHICmdList,
FLightShaftsOutput& Output);
    void RenderLightShaftBloom(FRHICommandListImmediate& RHICmdList);

    //Speed Buffer
    bool ShouldRenderVelocities()const;
    void RenderVelocities(FRHICommandListImmediate& RHICmdList, ...); void
    RenderVelocitiesInner(FRHICommandListImmediate& RHICmdList, ...);

    //Other rendering interfaces
    bool RenderLightMapDensities(FRHICommandListImmediate& RHICmdList);
```

```
bool   RenderDebugViewMode(FRHICommandListImmediate& RHICmdList);
void   UpdateDownsampledDepthSurface(FRHICommandList& RHICmdList);
void DownsampleDepthSurface(FRHICommandList& RHICmdList, ...); void
CopyStencilToLightingChannelTexture(FRHICommandList& RHICmdList);

//----Shadow rendering related interfaces----
void CreatePerObjectProjectedShadow(...);
bool InjectReflectiveShadowMaps(FRHICommandListImmediate& RHICmdList, ...); bool
RenderCapsuleDirectShadows(FRHICommandListImmediate& RHICmdList, ...)const; void
SetupIndirectCapsuleShadows(FRHICommandListImmediate& RHICmdList, ...)const; void
RenderIndirectCapsuleShadows(FRHICommandListImmediate& RHICmdList, ...)const; void
RenderCapsuleShadowsForMovableSkylight(FRHICommandListImmediate& RHICmdList, ...) const;

bool RenderShadowProjections(FRHICommandListImmediate& RHICmdList, ...);
void RenderForwardShadingShadowProjections(FRHICommandListImmediate& RHICmdList, ...);

//Volumetric Fog
bool   ShouldRenderVolumetricFog()          const;
void   SetupVolumetricFog();
void   RenderLocalLightsForVolumetricFog(...);
void   RenderLightFunctionForVolumetricFog(...);
void   VoxelizeFogVolumePrimitives(...); ComputeVolumetricFog
void   (FRHICommandListImmediate& VisualizeVolumetricLightmap      RHICmdList);
void   (FRHICommandListImmediate&                                  RHICmdList);

//reflection
void RenderStandardDeferredImageBasedReflections(FRHICommandListImmediate&          RHICmdList,
...);
bool HasDeferredPlanarReflections(constFViewInfo& View)const; void
RenderDeferredPlanarReflections(FRDGBuilder& GraphBuilder, ...); bool
ShouldDoReflectionEnvironment()const;

//Distance FieldAOand shadows
bool   ShouldRenderDistanceFieldAO()const; ShouldPrepareForDistanceFieldShadows()
bool   ShouldPrepareForDistanceFieldAO()const; ShouldPrepcoanresFt;orDFInsetIndirectShadow()
bool   ShouldPrepareDistanceFieldScene() ShouldPrepareGlobalDistanceField()
bool   ShouldPrepareHeightFieldScene()const; UpdateGlobalDcoisntastn;ceFieldObjectBuffers
bool   (FRHICommandListImmediate& UpdateGlobalcHoenigsth;tFieldObjectBuffers
bool   (FRHICommandListImmediate& RHICmdList);    const;
bool
void                                                                RHICmdList);
void
void AddOrRemoveSceneHeightFieldPrimitives(boolbSkipAdd =false); void
PrepareDistanceFieldScene(FRHICommandListImmediate& RHICmdList,bool bSplitDispatch);

void   CopySceneCaptureComponentToTarget(FRHICommandListImmediate& RHICmdList);
void   SetupImaginaryReflectionTextureParameters(...);

//Ray Tracing Interface
void   RenderRayTracingDeferredReflections(...);
void RenderRayTracingShadows(...);
void RenderRayTracingStochasticRectLight(FRHICommandListImmediate& RHICmdList, ...); void
CompositeRayTracingSkyLight(FRHICommandListImmediate& RHICmdList, ...); bool
RenderRayTracingGlobalIllumination(FRDGBuilder& GraphBuilder, ...);
void RenderRayTracingGlobalIlluminationBruteForce(FRDGBuilder& GraphBuilder, ...); void
RayTracingGlobalIlluminationCreateGatherPoints(FRDGBuilder& GraphBuilder, ...); void
RenderRayTracingGlobalIlluminationFinalGather(FRDGBuilder& GraphBuilder, ...); void
RenderRayTracingAmbientOcclusion(FRDGBuilder& GraphBuilder, ...);
```

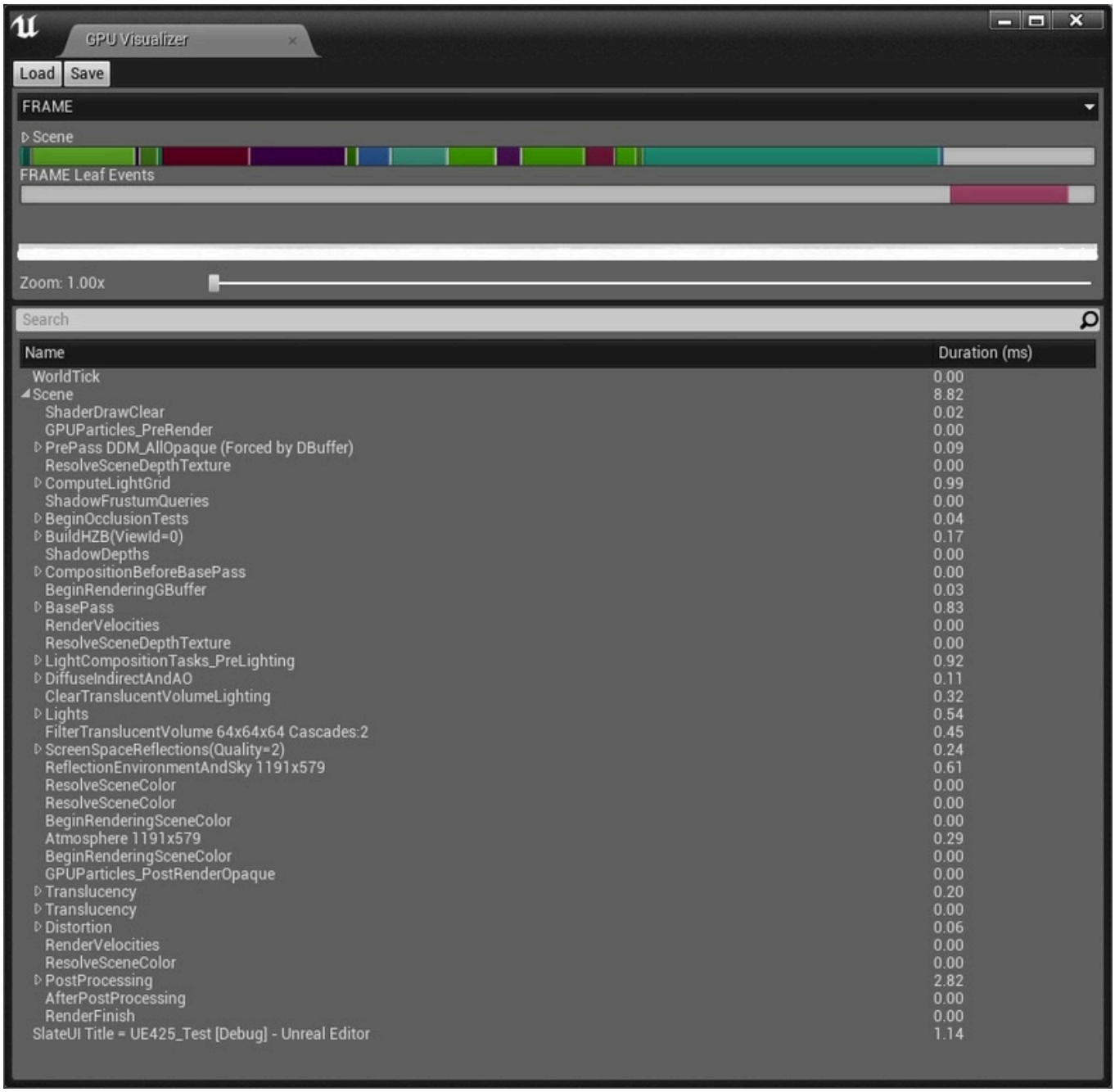
```
    (.....)

    //Whether to enable clustering judgment light source
    boolbClusteredShadingLightsInLightGrid;
};
```

As can be seen above,FDeferredShadingSceneRendererit mainly includes several major types of interfaces such as Mesh Pass, light source, shadow, ray tracing, reflection, visibility, etc. The most important interface is undoubtedlyFDeferredShadingSceneRenderer::Render,which is FDeferredShadingSceneRendererthe main entrance of rendering. The main process and the call of important interfaces all occur directly or indirectly inside it. IfFDeferredShadingSceneRenderer::Render the logic is subdivided, it can be divided into the following main stages:

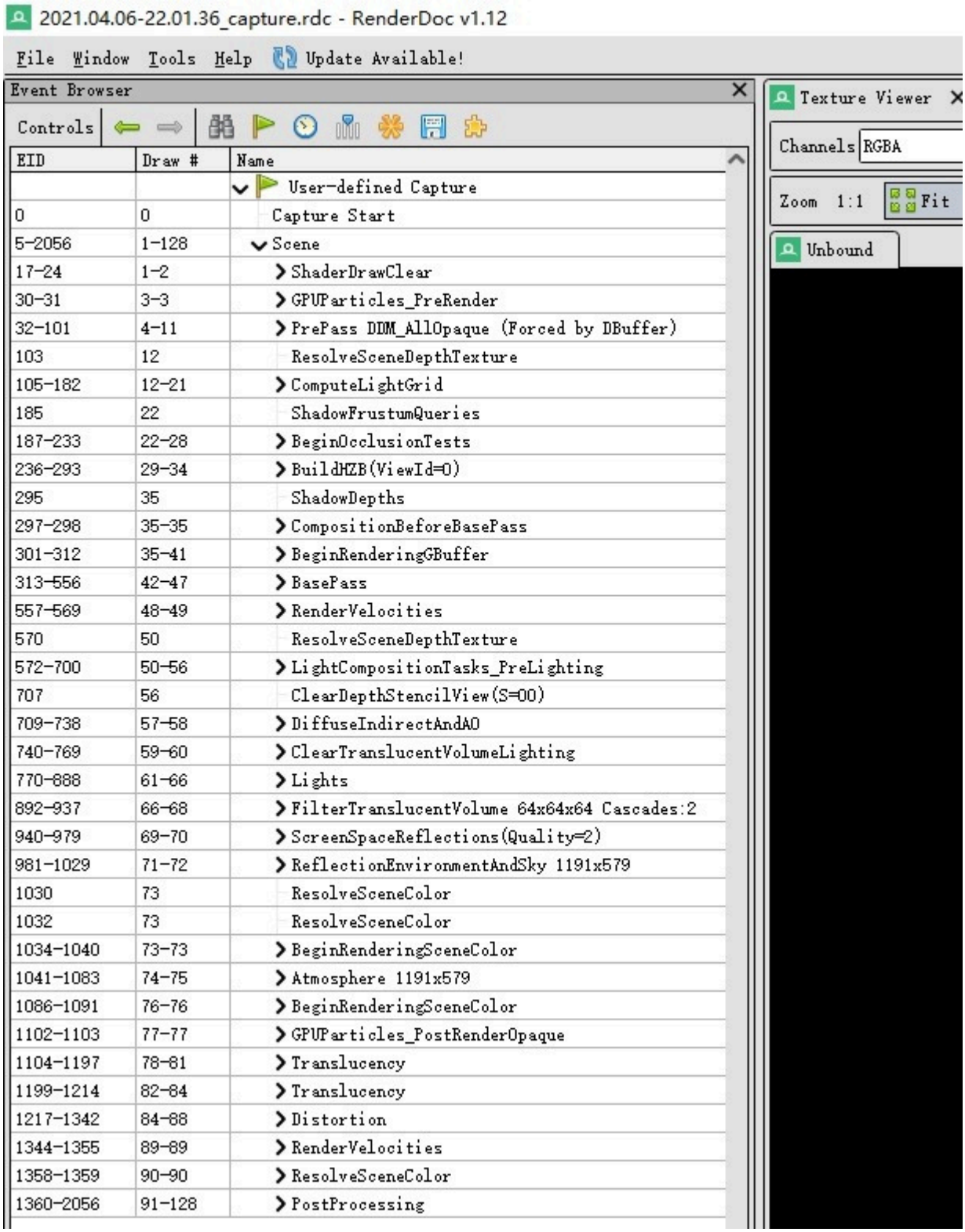
stage	Analysis
FScene::UpdateAllPrimitiveSceneInfos	Update the information of all primitives to the GPU. If GPUScene is enabled, a two-dimensional texture or StructureBuffer will be used to store the primitive information.
FSceneRenderTargets::Allocate	If necessary (resolution change, API trigger), reallocate the scene's render texture to ensure that it is large enough to render the corresponding view.
InitViews	Initializes visibility of primitives using clipping in several ways, sets visible dynamic shadows, and intersects the shadow frustum with the world when necessary (global and per-object shadows).
PrePass / Depth only pass	Early depth pass is used to render the depth of opaque objects. This pass only writes depth but not color. There are three modes for writing depth: disabled, occlusion only, and complete depths, depending on the platform and feature level. It is usually used to establish Hierarchical-Z so that the hardware Early-Z technology can be enabled to improve the rendering efficiency of the Base Pass.
Base pass	**This is the geometry channel mentioned in the previous chapter.**It is used to render the geometry information of opaque objects (Opaque and Masked materials), including normals, depth, color, AO, roughness, metalness, etc. This geometry information will be written into several GBuffers. The contribution of dynamic light sources will not be calculated at this stage, but the contribution of Lightmap and sky light will be calculated.

stage	Analysis
Issue Occlusion Queries / BeginOcclusionTests	Turn on occlusion rendering. The rendered occlusion data of this frame is used for occlusion culling in the**next frame**InitViewsstage. This stage mainly uses the bounding box of the object for rendering, and may also pack the bounding boxes of similar objects to reduce Draw Calls.
Lighting	**This stage is the lighting pass mentioned in the previous chapter**.It is a mixture of standard deferred shading and block deferred shading. The shadow map of the light source with shadows turned on is calculated, and the contribution of each light to the screen space pixel is calculated and accumulated into the Scene Color. In addition, the contribution of the light source to the translucency lighting volumes is also calculated.
Fog	Calculates the effects of fog and atmosphere on opaque surface pixels in screen space.
Translucency	Rendering translucent objects. All translucent objects are drawn one by one from far to near (view space) into the offscreen render texture (offscreen render target, called separate translucent render target in the code), and then a separate pass is used to correctly calculate and mix the lighting results.
Post Processing	Post-processing stage. This includes Bloom, tone mapping, gamma correction, etc. that do not require GBuffer, and SSR, SSAO, SSGI, etc. that require GBuffer. This stage will blend the semi-transparent rendering texture into the final scene color.

The above is just a brief list of some processes rather than all of them. You can use the RenderDoc tool to take screenshots or use the command lineprofilegputo view the detailed rendering process of each frame of UE.



The GPU Visualizer window that pops up after the UE console command line profilegpu is executed can clearly show the rendering steps and duration of each frame of the scene.



A frame of UE captured using the rendering analysis tool RenderDoc.

The following chapters will analyze the rendering process and the optimizations involved one by one in the above order. The analyzed code is mainly concentrated in

FDeferredShadingSceneRenderer::Render FDeferredShadingSceneRenderer::Render the implementation body, following this vine to touch the various melons involved in UE scene rendering.

4.3.3 FScene::UpdateAllPrimitiveSceneInfos

FScene::UpdateAllPrimitiveSceneInfosThe call occurs onFDeferredShadingSceneRenderer::Renderthe first line:

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

void FDeferredShadingSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

    Scene->UpdateAllPrimitiveSceneInfos(RHICmdList,true);

    (.....)
}
```

FScene::UpdateAllPrimitiveSceneInfosThe main function is to delete, add, and update the metadata on the CPU side and synchronize it to the GPU side. There are two ways to store GPU metadata:

- Each primitive has a unique uniform buffer. When you need to access primitive data in the shader, you can get it from the primitive's uniform buffer. This structure is simple and easy to understand, and is compatible with all FeatureLevel devices. However, it will increase the IO of the CPU and GPU and reduce the GPU cache hit rate.
- Use Texture2D or StructuredBuffer GPU Scene, all primitive data is placed here in a regular manner. When you need to access primitive data in the shader, you need to read the data from the corresponding location in the GPU Scene. SM5 support is required, which is difficult to implement and difficult to understand, but it can reduce CPU and GPU IO, improve GPU Cache hit rate, and better support ray tracing and GPU Driven Pipeline.

Although the above access methods are different, they have been encapsulated in the shader. Users do not need to distinguish which form of Buffer it is, just use the following method:

```
GetPrimitiveData(PrimitiveId).xxx;
```

Wherexxxis the name of the primitive attribute. The specific accessible attributes are as follows:

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

structFPrimitiveSceneData {

    float4x4    LocalToWorld;
    float4    InvNonUniformScaleAndDeterminantSign;
    float4    ObjectWorldPositionAndRadius;
    float4x4    WorldToLocal;
    float4x4    PreviousLocalToWorld;
    float4x4    PreviousWorldToLocal;
    float3    ActorWorldPosition;
    float    UseSingleSampleShadowFromStationaryLights;
    float3 ObjectBounds;
    float    LpvBiasMultiplier;
    float    DecalReceiverMask;
    float    PerObjectGBufferData;
    float    UseVolumetricLightmapShadowFromStationaryLights;
    float    DrawsVelocity;
    float4    ObjectOrientation;
```

```
    float4      NonUniformScale;
    float3      LocalObjectBoundsMin;
    uint LightingChannelMask; float3
    LocalObjectBoundsMax; uint
    LightmapDataIndex; float3
    PreSkinnedLocalBoundsMin; int
    SingleCaptureIndex; float3
    PreSkinnedLocalBoundsMax; uint
    OutputVelocity;
    float4 CustomPrimitiveData[NUM_CUSTOM_PRIMITIVE_DATA];
};
```

It can be seen that each primitive has a lot of accessible data and occupies a considerable amount of video memory. Each primitive occupies about 576 bytes. If there are 10,000 primitives in the scene (very common in game scenes), ignoring padding, the total amount of primitive data reaches about 5.5M.

Let's get back to the topic and go back to the C++ layer to look at the definition of GPU Scene:

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

classFGPUScene
{
public:
    //Whether to update all metadata. This is usually used for debugging and may cause performance
    degradation during runtime. boolbUpdateAllPrimitives;

    //The index of the element whose data needs to be updated.
    TArray<int32> PrimitivesToUpdate;
    //All graphics elementsbit,When the corresponding indexbitfor1When it indicates that an update is needed (while
    PrimitivesToUpdatemiddle). TBitArray<> PrimitivesMarkedToUpdate;

    //Storing graphics elementsGPUData structures can beTextureBufferorTexture2D,But only one of them will be created and take effect. The mobile
terminal canMobile.UseGPUSceneTextureConsole variable settings.
    FRWBufferStructured         PrimitiveBuffer;
    FTextureRWBuffer2D        PrimitiveTexture;
    //Uploadedbuffer
    FScatterUploadBuffer          PrimitiveUploadBuffer;
    FScatterUploadBuffer          PrimitiveUploadViewBuffer;

    //Light Map
    FGrowOnlySpanAllocator              LightmapDataAllocator;
    FRWBufferStructured                   LightmapDataBuffer;
    FScatterUploadBuffer                LightmapUploadBuffer;
};
```

Return to the topic at the beginning of this section and continue the analysis FScene::UpdateAllPrimitiveSceneInfosprocess:

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

void FScene::UpdateAllPrimitiveSceneInfos(FRHICommandListImmediate& RHICmdList,bool bAsyncCreateLPIs)

{
```

```
	TArray<FPrimitiveSceneInfo*>
RemovedLocalPrimitiveSceneInfos(RemovedPrimitiveSceneInfos.Array());
	RemovedLocalPrimitiveSceneInfos.Sort(FPrimitiveArraySortKey());

	TArray<FPrimitiveSceneInfo*> AddedLocalPrimitiveSceneInfos
	(AddedPrimitiveSceneInfos.Array());
	AddedLocalPrimitiveSceneInfos.Sort(FPrimitiveArraySortKey());

	TSet<FPrimitiveSceneInfo*> DeletedSceneInfos;
	DeletedSceneInfos.Reserve(RemovedLocalPrimitiveSceneInfos.Num());

	(......)

	//Handling Element Deletion
	{
		(......)

		while	(RemovedLocalPrimitiveSceneInfos.Num())
		{
			//	Find the starting point
			intStartIndex = RemovedLocalPrimitiveSceneInfos.Num() -1;
			SIZE_T InsertProxyHash = RemovedLocalPrimitiveSceneInfos[StartIndex]->Proxy-
> GetTypeHash();

			while(StartIndex >0&& RemovedLocalPrimitiveSceneInfos[StartIndex -1]-
> Proxy->GetTypeHash() == InsertProxyHash)
			{
				StartIndex--;
			}

			(......)

			{
				SCOPED_NAMED_EVENT(FScene_SwapPrimitiveSceneInfos, FColor::Turquoise);

				for(intCheckIndex = StartIndex; CheckIndex <
RemovedLocalPrimitiveSceneInfos.Num(); CheckIndex++)
				{
					intSourceIndex = RemovedLocalPrimitiveSceneInfos[CheckIndex]-
> PackedIndex;

					for(intTypeIndex = BroadIndex; TypeIndex < TypeOffsetTable.Num();
TypeIndex++)
					{
						FTypeOffsetTableEntry& NextEntry = TypeOffsetTable[TypeIndex]; intDestIndex = --
						NextEntry.Offset;//decrement and prepare swap

						//Deletion operation diagram, withPrimitiveSceneProxiesandTypeOffsetTable,Can reduce
Number of moves or swaps to remove an element.
						// example swap chain of removing X //
							PrimitiveSceneProxies[0,0,0,6,X,6,6,6,2,2,2,2,1,1,1,7,4,8]
						//	PrimitiveSceneProxies[0,0,0,6,6,	6,6,6,X,2,2,2,1,1,1,7,4,8]
						//	PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,	X,1,1,1,7,4,8]
						//	PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,X,7,4,	8]
						//	PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,7,X,4,8]
						//	PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,7,4,X,8]
						//	PrimitiveSceneProxies[0,0,0,6,6, 6,6,6,2,2,2,1,1,1,7,4,8,X]
```

```
                                if(DestIndex != SourceIndex) {

                                    checkfSlow(DestIndex > SourceIndex, TEXT("Corrupted Prefix Sum
[%d, %d]"), DestIndex, SourceIndex);
                                    Primitives[DestIndex]->PackedIndex   =          SourceIndex;
                                    Primitives[SourceIndex]->PackedIndex            = DestIndex;

                                    TArraySwapElements(Primitives, DestIndex, SourceIndex);
                                    TArraySwapElements(PrimitiveTransforms, DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveSceneProxies, DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveBounds, DestIndex,          SourceIndex);
                                    TArraySwapElements(PrimitiveFlagsCompact,          DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveVisibilityIds,          DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveOcclusionFlags,          DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveComponentIds, DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveVirtualTextureFlags, DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveVirtualTextureLod, DestIndex,
SourceIndex);
                                    TArraySwapElements(PrimitiveOcclusionBounds, DestIndex,
SourceIndex);
                                    TBitArraySwapElements(PrimitivesNeedingStaticMeshUpdate,
DestIndex, SourceIndex);

                                    AddPrimitiveToUpdateGPU(*this,          SourceIndex);
                                    AddPrimitiveToUpdateGPU(*this,          DestIndex);

                                    SourceIndex = DestIndex;
                                }
                            }
                        }
                    }

                    const intPreviousOffset = BroadIndex >0?TypeOffsetTable[BroadIndex - 0;
1].Offset       :
                    const intCurrentOffset = TypeOffsetTable[BroadIndex].Offset;

                    if(CurrentOffset - PreviousOffset ==0) {

                        // remove empty OffsetTable entries eg //
                            TypeOffsetTable[3,8,12,15,15,17,18]
                        // TypeOffsetTable[3,8,12,15,17,18]
                        TypeOffsetTable.RemoveAt(BroadIndex);
                    }

                    (.....)

                    for(intRemoveIndex = StartIndex; RemoveIndex <
RemovedLocalPrimitiveSceneInfos.Num(); RemoveIndex++)
                    {
                        intSourceIndex = RemovedLocalPrimitiveSceneInfos[RemoveIndex]-
> PackedIndex;
```

```
                check(SourceIndex >= (Primitives.Num() -
RemovedLocalPrimitiveSceneInfos.Num() + StartIndex));
                Primitives.Pop();
                PrimitiveTransforms.Pop();
                PrimitiveSceneProxies.Pop();
                PrimitiveBounds.Pop();
                PrimitiveFlagsCompact.Pop();
                PrimitiveVisibilityIds.Pop();
                PrimitiveOcclusionFlags.Pop();
                PrimitiveComponentIds.Pop();
                PrimitiveVirtualTextureFlags.Pop();
                PrimitiveVirtualTextureLod.Pop();
                PrimitiveOcclusionBounds.Pop();

PrimitivesNeedingStaticMeshUpdate.RemoveAt(PrimitivesNeedingStaticMeshUpdate.Num() -1);
            }

            CheckPrimitiveArrays();

            for(intRemoveIndex = StartIndex; RemoveIndex <
RemovedLocalPrimitiveSceneInfos.Num(); RemoveIndex++)
            {
                FPrimitiveSceneInfo* PrimitiveSceneInfo =
RemovedLocalPrimitiveSceneInfos[RemoveIndex];
                FScopeCycleCounterContext(PrimitiveSceneInfo->Proxy->GetStatId()); int32 PrimitiveIndex
                = PrimitiveSceneInfo->PackedIndex; PrimitiveSceneInfo->PackedIndex = INDEX_NONE;


                if(PrimitiveSceneInfo->Proxy->IsMovable()) {

                    // Remove primitive's motion blur information.
                    VelocityData.RemoveFromScene(PrimitiveSceneInfo-
>PrimitiveComponentId);
                }

                // Unlink the primitive from its shadow parent.
                PrimitiveSceneInfo->UnlinkAttachmentGroup();

                // Unlink the LOD parent info if valid PrimitiveSceneInfo-
                >UnlinkLODParentComponent();

                // Flush virtual textures touched by primitive PrimitiveSceneInfo-
                >FlushRuntimeVirtualTexture();

                // Remove the primitive from the scene.
                PrimitiveSceneInfo->RemoveFromScene(true);

                // Update the primitive that was swapped to this index
                AddPrimitiveToUpdateGPU(*this, PrimitiveIndex);

                DistanceFieldSceneData.RemovePrimitive(PrimitiveSceneInfo);

                DeletedSceneInfos.Add(PrimitiveSceneInfo);
            }
            RemovedLocalPrimitiveSceneInfos.RemoveAt(StartIndex,
RemovedLocalPrimitiveSceneInfos.Num() - StartIndex);
        }
    }
```

```
//Processing primitives increase
{
	CSV_SCOPED_TIMING_STAT_EXCLUSIVE(AddPrimitiveSceneInfos);
	SCOPED_NAMED_EVENT(FScene_AddPrimitiveSceneInfos, FColor::Green);
	SCOPE_CYCLE_COUNTER(STAT_AddScenePrimitiveRenderThreadTime); if
	(AddedLocalPrimitiveSceneInfos.Num()) {

		Primitives.Reserve(Primitives.Num() + AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveTransforms.Reserve(PrimitiveTransforms.Num() +
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveSceneProxies.Reserve(PrimitiveSceneProxies.Num()		+
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveBounds.Reserve(PrimitiveBounds.Num() +
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveFlagsCompact.Reserve(PrimitiveFlagsCompact.Num()		+
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveVisibilityIds.Reserve(PrimitiveVisibilityIds.Num()		+
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveOcclusionFlags.Reserve(PrimitiveOcclusionFlags.Num()		+
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveComponentIds.Reserve(PrimitiveComponentIds.Num() +
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveVirtualTextureFlags.Reserve(PrimitiveVirtualTextureFlags.Num()		+
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveVirtualTextureLod.Reserve(PrimitiveVirtualTextureLod.Num() +
AddedLocalPrimitiveSceneInfos.Num());
		PrimitiveOcclusionBounds.Reserve(PrimitiveOcclusionBounds.Num() +
AddedLocalPrimitiveSceneInfos.Num());

PrimitivesNeedingStaticMeshUpdate.Reserve(PrimitivesNeedingStaticMeshUpdate.Num() +
AddedLocalPrimitiveSceneInfos.Num());
	}

	while(AddedLocalPrimitiveSceneInfos.Num()) {

		intStartIndex = AddedLocalPrimitiveSceneInfos.Num() -1;
		SIZE_T InsertProxyHash = AddedLocalPrimitiveSceneInfos[StartIndex]->Proxy-
> GetTypeHash();

		while(StartIndex >0&& AddedLocalPrimitiveSceneInfos[StartIndex -1]->Proxy-
> GetTypeHash() == InsertProxyHash)
		{
			StartIndex--;
		}

		for(intAddIndex = StartIndex; AddIndex <
AddedLocalPrimitiveSceneInfos.Num();		AddIndex++)
		{
			FPrimitiveSceneInfo*		PrimitiveSceneInfo		=
AddedLocalPrimitiveSceneInfos[AddIndex];
			Primitives.Add(PrimitiveSceneInfo);
			constFMatrix LocalToWorld = PrimitiveSceneInfo->Proxy->GetLocalToWorld();
			PrimitiveTransforms.Add(LocalToWorld);
			PrimitiveSceneProxies.Add(PrimitiveSceneInfo->Proxy);
			PrimitiveBounds.AddUninitialized();
			PrimitiveFlagsCompact.AddUninitialized();
			PrimitiveVisibilityIds.AddUninitialized();
```

```
                PrimitiveOcclusionFlags.AddUninitialized();
                PrimitiveComponentIds.AddUninitialized();
                PrimitiveVirtualTextureFlags.AddUninitialized();
                PrimitiveVirtualTextureLod.AddUninitialized();
                PrimitiveOcclusionBounds.AddUninitialized();
                PrimitivesNeedingStaticMeshUpdate.Add(false);

                const intSourceIndex = PrimitiveSceneProxies.Num() -1; PrimitiveSceneInfo-
                >PackedIndex = SourceIndex;

                AddPrimitiveToUpdateGPU(*this, SourceIndex);
            }

            boolEntryFound =false; int
            BroadIndex =-1;
            //broad phase search for a matching type
            for(BroadIndex = TypeOffsetTable.Num() -1; BroadIndex >=0; BroadIndex--) {

                // example how the prefix sum of the tails could look like //
                    PrimitiveSceneProxies[0,0,0,6,6,6,6,6,2,2,2,2,1,1,1,7,4,8]
                //    TypeOffsetTable[3,8,12,15,16, 17,18]

                if(TypeOffsetTable[BroadIndex].PrimitiveSceneProxyType ==
InsertProxyHash)
                {
                    EntryFound =true; break
                    ;
                }
            }

            //new type encountered
            if(EntryFound ==false) {

                BroadIndex = TypeOffsetTable.Num(); if
                (BroadIndex)
                {
                    FTypeOffsetTableEntry Entry = TypeOffsetTable[BroadIndex -1]; //adding to the end of the
                    list and offset of the tail (will will be
incremented once during the while loop)
                    TypeOffsetTable.Push(FTypeOffsetTableEntry(InsertProxyHash,
Entry.Offset));
                }
                else
                {
                    //starting with an empty list and offset zero (will will be
incremented once during the while loop)
                    TypeOffsetTable.Push(FTypeOffsetTableEntry(InsertProxyHash,0));
                }
            }

            {
                SCOPED_NAMED_EVENT(FScene_SwapPrimitiveSceneInfos, FColor::Turquoise);

                for(intAddIndex = StartIndex; AddIndex <
AddedLocalPrimitiveSceneInfos.Num(); AddIndex++)
                {
                    intSourceIndex = AddedLocalPrimitiveSceneInfos[AddIndex]-
> PackedIndex;
```

```
                        for(intTypeIndex = BroadIndex; TypeIndex < TypeOffsetTable.Num();
TypeIndex++)
                        {
                            FTypeOffsetTableEntry& NextEntry = TypeOffsetTable[TypeIndex]; intDestIndex =
                            NextEntry.Offset++;//prepare swap and increment

                            // example swap chain of inserting a type of 6 at the end //
                                PrimitiveSceneProxies[0,0,0,6,6,6,6,6,2,2,2,2,1,1,1,7,4,8,6]
                            //  PrimitiveSceneProxies[0,0,0,6,    6,6,6,6,6,2,2,2,1,1,1,7,4,8,2]
                            //  PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,    2,2,2,2,1,1,7,4,8,1]
                            //  PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,6,2,2,2,2,1,    1,1,4,8,7]
                            //  PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,6,2,2,2,2,1,1,1,7,8,4]
                            // PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,2,2,2,2,1,1,1,7,4,8]

                            if(DestIndex != SourceIndex) {

                                checkfSlow(SourceIndex > DestIndex, TEXT("Corrupted Prefix Sum
[%d, %d]"), SourceIndex, DestIndex);
                                Primitives[DestIndex]->PackedIndex   =           SourceIndex;
                                Primitives[SourceIndex]->PackedIndex             = DestIndex;

                                TArraySwapElements(Primitives, DestIndex, SourceIndex);
                                TArraySwapElements(PrimitiveTransforms, DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveSceneProxies, DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveBounds, DestIndex,              SourceIndex);
                                TArraySwapElements(PrimitiveFlagsCompact,           DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveVisibilityIds,            DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveOcclusionFlags,           DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveComponentIds, DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveVirtualTextureFlags, DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveVirtualTextureLod, DestIndex,
SourceIndex);
                                TArraySwapElements(PrimitiveOcclusionBounds, DestIndex,
SourceIndex);
                                TBitArraySwapElements(PrimitivesNeedingStaticMeshUpdate,
DestIndex, SourceIndex);

                                AddPrimitiveToUpdateGPU(*this, DestIndex);
                            }
                        }
                    }
                }

                CheckPrimitiveArrays();

                for(intAddIndex = StartIndex; AddIndex <
AddedLocalPrimitiveSceneInfos.Num();          AddIndex++)
                {
                    FPrimitiveSceneInfo*      PrimitiveSceneInfo      =
AddedLocalPrimitiveSceneInfos[AddIndex];
```

```
                FScopeCycleCounterContext(PrimitiveSceneInfo->Proxy->GetStatId()); int32 PrimitiveIndex
                = PrimitiveSceneInfo->PackedIndex;

                // Add the primitive to its shadow parent's linked list of children. // Note: must happen
                before AddToScene because AddToScene depends on
LightingAttachmentRoot
                PrimitiveSceneInfo->LinkAttachmentGroup();
        }



        {
                SCOPED_NAMED_EVENT(FScene_AddPrimitiveSceneInfoToScene,
FColor::Turquoise);
                if(GIsEditor)
                {
                        FPrimitiveSceneInfo::AddToScene(RHICmdList, this,
TArrayView<FPrimitiveSceneInfo*>(&AddedLocalPrimitiveSceneInfos[StartIndex],
AddedLocalPrimitiveSceneInfos.Num() - StartIndex),true);
                }
                else
                {
                        const boolbAddToDrawLists = !
(CVarDoLazyStaticMeshUpdate.GetValueOnRenderThread());
                        if(bAddToDrawLists)
                        {
                                FPrimitiveSceneInfo::AddToScene(RHICmdList, this,
TArrayView<FPrimitiveSceneInfo*>(&AddedLocalPrimitiveSceneInfos[StartIndex],
AddedLocalPrimitiveSceneInfos.Num() - StartIndex),true,true, bAsyncCreateLPIs);
                        }
                        else
                        {
                                FPrimitiveSceneInfo::AddToScene(RHICmdList, this,
TArrayView<FPrimitiveSceneInfo*>(&AddedLocalPrimitiveSceneInfos[StartIndex],
AddedLocalPrimitiveSceneInfos.Num() - StartIndex),true,false, bAsyncCreateLPIs);

                                for(intAddIndex = StartIndex; AddIndex <
AddedLocalPrimitiveSceneInfos.Num(); AddIndex++)
                                {
                                        FPrimitiveSceneInfo* PrimitiveSceneInfo                =
AddedLocalPrimitiveSceneInfos[AddIndex];
                                        PrimitiveSceneInfo->BeginDeferredUpdateStaticMeshes();
                                }
                        }
                }
        }

        for(intAddIndex = StartIndex; AddIndex <
AddedLocalPrimitiveSceneInfos.Num();             AddIndex++)
        {
                FPrimitiveSceneInfo*          PrimitiveSceneInfo          =
AddedLocalPrimitiveSceneInfos[AddIndex];
                int32 PrimitiveIndex = PrimitiveSceneInfo->PackedIndex;

                if(PrimitiveSceneInfo->Proxy->IsMovable() && GetFeatureLevel() >
ERHIFeatureLevel::ES3_1)
                {
                        // We must register the initial LocalToWorld with the velocity state. // In the case of a moving
                        component with MarkRenderStateDirty()
```

```
called every frame, UpdateTransform will never happen.
                        VelocityData.UpdateTransform(PrimitiveSceneInfo,
PrimitiveTransforms[PrimitiveIndex], PrimitiveTransforms[PrimitiveIndex]);
                    }

                    AddPrimitiveToUpdateGPU(*this,          PrimitiveIndex);
                    bPathTracingNeedsInvalidation          = true;

                    DistanceFieldSceneData.AddPrimitive(PrimitiveSceneInfo);

                    // Flush virtual textures touched by primitive PrimitiveSceneInfo-
                    >FlushRuntimeVirtualTexture();

                    // Set LOD parent information if valid PrimitiveSceneInfo-
                    >LinkLODParentComponent();

                    // Update scene LOD tree
                    SceneLODHierarchy.UpdateNodeSceneInfo(PrimitiveSceneInfo-
> PrimitiveComponentId, PrimitiveSceneInfo);
                }
                AddedLocalPrimitiveSceneInfos.RemoveAt(StartIndex,
AddedLocalPrimitiveSceneInfos.Num() - StartIndex);
            }
        }

        //Update the primitive transformation matrix
        {
            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(UpdatePrimitiveTransform);
            SCOPED_NAMED_EVENT(FScene_AddPrimitiveSceneInfos, FColor::Yellow);
            SCOPE_CYCLE_COUNTER(STAT_UpdatePrimitiveTransformRenderThreadTime);

            TArray<FPrimitiveSceneInfo*> TArray<FUPpridmaiteivdeSScceenneeIInnffoos*W> ithStaticDrawListUpdate;
            UpdatedSceneInfosWithStaticDrawListUUppddaattee.RdeSscernveeI(nUfpodsWatiethdoTruatnStsafotircmDrsa.NwuLmist(U))
            UpdatedSceneInfosWithoutStaticDrawListUpdate.Reserve(UpdatedTransforms.Num());


            for(constauto& Transform : UpdatedTransforms) {

                FPrimitiveSceneProxy* PrimitiveSceneProxy = Transform.Key;
                if(DeletedSceneInfos.Contains(PrimitiveSceneProxy->GetPrimitiveSceneInfo())) {

                    continue;
                }
                check(PrimitiveSceneProxy->GetPrimitiveSceneInfo()->PackedIndex !=
INDEX_NONE);

                constFBoxSphereBounds& WorldBounds = Transform.Value.WorldBounds; const
                FBoxSphereBounds& LocalBounds = Transform.Value.LocalBounds; constFMatrix&
                LocalToWorld = Transform.Value.LocalToWorld; constFVector& AttachmentRootPosition =

Transform.Value.AttachmentRootPosition;
                FScopeCycleCounterContext(PrimitiveSceneProxy->GetStatId());

                FPrimitiveSceneInfo* PrimitiveSceneInfo = PrimitiveSceneProxy-
> GetPrimitiveSceneInfo();
                const boolbUpdateStaticDrawLists = !PrimitiveSceneProxy-
> StaticElementsAlwaysUseProxyPrimitiveUniformBuffer();
                if(bUpdateStaticDrawLists)
```

```
                {
                    UpdatedSceneInfosWithStaticDrawListUpdate.Push(PrimitiveSceneInfo);
                }
                else
                {
                    UpdatedSceneInfosWithoutStaticDrawListUpdate.Push(PrimitiveSceneInfo);
                }

                PrimitiveSceneInfo->FlushRuntimeVirtualTexture();

                // Remove the primitive from the scene at its old location
                // (note that the octree update relies on the bounds not being modified yet). PrimitiveSceneInfo-
                >RemoveFromScene(bUpdateStaticDrawLists);

                if(PrimitiveSceneInfo->Proxy->IsMovable() && GetFeatureLevel() >
ERHIFeatureLevel::ES3_1)
                {
                    VelocityData.UpdateTransform(PrimitiveSceneInfo, LocalToWorld,
PrimitiveSceneProxy->GetLocalToWorld());
                }

                // Update the primitive transform. PrimitiveSceneProxy-
                >SetTransform(LocalToWorld,                    WorldBounds,    LocalBounds,
AttachmentRootPosition);
                PrimitiveTransforms[PrimitiveSceneInfo->PackedIndex]                = LocalToWorld;

                if(!RHISupportsVolumeTextures(GetFeatureLevel())
                    && (PrimitiveSceneProxy->IsMovable() || PrimitiveSceneProxy-
> NeedsUnbuiltPreviewLighting() || PrimitiveSceneProxy->GetLightmapType() ==
ELightmapType::ForceVolumetric))
                {
                    PrimitiveSceneInfo->MarkIndirectLightingCacheBufferDirty();
                }

                AddPrimitiveToUpdateGPU(*this, PrimitiveSceneInfo->PackedIndex);

                DistanceFieldSceneData.UpdatePrimitive(PrimitiveSceneInfo);

                // If the primitive has static mesh elements, it should have returned true
from            ShouldRecreateProxyOnUpdateTransform!
                check(!(bUpdateStaticDrawLists            && PrimitiveSceneInfo->StaticMeshes.Num()));
            }

            // Re-add the primitive to the scene with the new transform. if
            (UpdatedSceneInfosWithStaticDrawListUpdate.Num() >0) {

                FPrimitiveSceneInfo::AddToScene(RHICmdList, this,
UpdatedSceneInfosWithStaticDrawListUpdate,true,true, bAsyncCreateLPIs);
            }

            if(UpdatedSceneInfosWithoutStaticDrawListUpdate.Num() >0) {

                FPrimitiveSceneInfo::AddToScene(RHICmdList, this,
UpdatedSceneInfosWithoutStaticDrawListUpdate,false,true, bAsyncCreateLPIs);
                for(FPrimitiveSceneInfo* PrimitiveSceneInfo:
UpdatedSceneInfosWithoutStaticDrawListUpdate)
                {
                    PrimitiveSceneInfo->FlushRuntimeVirtualTexture();
```

```
                }
            }

            if(AsyncCreateLightPrimitiveInteractionsTask &&
AsyncCreateLightPrimitiveInteractionsTask->GetTask().HasPendingPrimitives())
            {
                check(GAsyncCreateLightPrimitiveInteractions);
                AsyncCreateLightPrimitiveInteractionsTask->StartBackgroundTask();
            }

            for(constauto& Transform : OverridenPreviousTransforms) {

                FPrimitiveSceneInfo* PrimitiveSceneInfo = Transform.Key;
                VelocityData.OverridePreviousTransform(PrimitiveSceneInfo-
> PrimitiveComponentId, Transform.Value); }

        }

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

In summary,FScene::UpdateAllPrimitiveSceneInfosthe function is to delete, add primitives, and update all the data of primitives, including transformation matrices, custom data, distance field data, etc.

interestingly, when deleting or adding primitives, the orderedPrimitiveSceneProxiessum is combined TypeOffsetTable,which can reduce the number of moves or exchanges in the process of operating elements, which is quite clever. The exchange process is clearly given in the commented code:

```
// Deleting primitives: The deleted elements are swapped to the end of the same type in
// sequence until the end of the list PrimitiveSceneProxies[0,0,0,6,X,6,6,6,2,2,2,2,1,1,1,7,4,8]
// PrimitiveSceneProxies[0,0,0,6,6,                    6,6,6,X,2,2,2,1,1,1,7,4,8]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,     X,1,1,1,7,4,8]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,X,7,4,          8]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,7,X,4,8]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,1,1,1,7,4,X,8]   PrimitiveSceneProxies[0,0,0,6,6,
// 6,6,6,2,2,2,1,1,1,7,4,8,X]

// Schematic diagram of adding primitives: first place the added elements at the end of the
// list, and then exchange them with the end of the same type one by one.
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,2,2,2,2,1,1,1,7,4,8,6] PrimitiveSceneProxies[0,0,0,6,
// 6,6,6,6,6,2,2,2,1,1,1,7,4,8,2] PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6, 2,2,2,2,1,1,7,4,8,1]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,2,2,2,2,1, 1,1,4,8,7]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,2,2,2,2,1,1,1,7,8,4]
// PrimitiveSceneProxies[0,0,0,6,6,6,6,6,6,6,2,2,2,2,1,1,1,7,4,8]
```

In addition, for all the primitive indexes whose data has been added, deleted or changed, AddPrimitiveToUpdateGPUthey will be added to the list to be updated:

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

void AddPrimitiveToUpdateGPU(FScene& Scene, int32 PrimitiveId)
```

```
{
    if (UseGPUScene(GMaxRHIShaderPlatform, Scene.GetFeatureLevel())
    {
        if(PrimitiveId +1> Scene.GPUScene.PrimitivesMarkedToUpdate.Num()) {

            constint32 NewSize = Align(PrimitiveId +1,64);
            Scene.GPUScene.PrimitivesMarkedToUpdate.Add(0, NewSize -
Scene.GPUScene.PrimitivesMarkedToUpdate.Num());
        }

        // Add the index of the element to be updatedPrimitivesToUpdateandPrimitivesMarkedToUpdate
        if middle (!Scene.GPUScene.PrimitivesMarkedToUpdate[PrimitiveId])
        {
            Scene.GPUScene.PrimitivesToUpdate.Add(PrimitiveId);
            Scene.GPUScene.PrimitivesMarkedToUpdate[PrimitiveId] =true;
        }
    }
}
```

After GPUScene's PrimitivesToUpdate and PrimitivesMarkedToUpdate collect all the primitive indices that need to be updated, they will be synchronized to the GPU afterwards

FDeferredShadingSceneRenderer::Render:InitViews

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

    //renewGPUSceneData Scene-
    >UpdateAllPrimitiveSceneInfos(RHICmdList,true);

    (......)

    //initializationViewData
    bDoInitViewAftersPrepass = InitViews(RHICmdList, BasePassDepthStencilAccess, ILCTaskData,
    UpdateViewCustomDataEvents);

    (......)

    //synchronousCPUEndGPUScenearriveGPU.
    UpdateGPUScene(RHICmdList, *Scene);

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

Let's takeUpdateGPUScenea look at the specific synchronization logic:

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

void UpdateGPUScene(FRHICommandListImmediate& RHICmdList, FScene& Scene) {

    // According to differentGPUSceneThe storage type calls the corresponding
    if interface. (GPUSceneUseTexture2D(Scene.GetShaderPlatform()))
    {
        UpdateGPUSceneInternal<FTextureRWBuffer2D>(RHICmdList, Scene);
    }
    else
    {
```

```
            UpdateGPUSceneInternal<FRWBufferStructured>(RHICmdList, Scene);
        }
}

// ResourceTypeIt's encapsulatedFTextureRWBuffer2DandFRWBufferStructuredTemplate type
template<typename ResourceType>
void UpdateGPUSceneInternal(FRHICommandListImmediate& RHICmdList, FScene& Scene) {

        if(UseGPUScene(GMaxRHIShaderPlatform, Scene.GetFeatureLevel())) {

            //Are all metadata synchronized?
            if(GGPUSceneUploadEveryFrame || Scene.GPUScene.bUpdateAllPrimitives) {

                for(int32 Index : Scene.GPUScene.PrimitivesToUpdate) {

                    Scene.GPUScene.PrimitivesMarkedToUpdate[Index] =false;
                }
                Scene.GPUScene.PrimitivesToUpdate.Reset();

                for(int32 i =0; i < Scene.Primitives.Num(); i++) {

                    Scene.GPUScene.PrimitivesToUpdate.Add(i);
                }

                Scene.GPUScene.bUpdateAllPrimitives =false;
            }

            boolbResizedPrimitiveData =false; bool
            bResizedLightmapData =false;

            //GetGPUImage resources:GPUScene.PrimitiveBufferorGPUScene.PrimitiveTexture ResourceType*
            MirrorResourceGPU = GetMirrorGPU<ResourceType>(Scene); //If the resource size is insufficient, expand it.
            The growth strategy is not less than256,And guarantee upward2ofNSecond power. {

                constuint32 SizeReserve = FMath::RoundUpToPowerOfTwo( FMath::Max(
Scene.Primitives.Num(),256) );
                //If the resource size is insufficient, a new resource will be created and the data from the old resource will be copied to the new resource.
                bResizedPrimitiveData = ResizeResourceIfNeeded(RHICmdList, *MirrorResourceGPU,
SizeReserve *sizeof(FPrimitiveSceneShaderData::Data), TEXT("PrimitiveData"));
            }

            {
                constuint32 SizeReserve = FMath::RoundUpToPowerOfTwo( FMath::Max(
Scene.GPUScene.LightmapDataAllocator.GetMaxSize(),256) );
                bResizedLightmapData = ResizeResourceIfNeeded(RHICmdList,
Scene.GPUScene.LightmapDataBuffer, SizeReserve *sizeof(FLightmapSceneShaderData::Data), TEXT("LightmapData"));

            }

            constint32 NumPrimitiveDataUploads = Scene.GPUScene.PrimitivesToUpdate.Num();

            int32 NumLightmapDataUploads =0;

            if(NumPrimitiveDataUploads >0) {

                //Collect all data that needs to be updatedPrimitiveUploadBufferIn. BecauseBufferThere is a maximum value for the size of So single
UploadedBufferThere is also a size limit. If it exceeds the maximum size, it will be uploaded in batches.
                constint32 MaxPrimitivesUploads =
```

```
GetMaxPrimitivesUpdate(NumPrimitiveDataUploads,
FPrimitiveSceneShaderData::PrimitiveDataStrideInFloat4s);
                for(int32 PrimitiveOffset =0; PrimitiveOffset < NumPrimitiveDataUploads;
PrimitiveOffset += MaxPrimitivesUploads)
                {
                    SCOPED_DRAW_EVENTF(RHICmdList, UpdateGPUScene, TEXT("UpdateGPUScene
PrimitivesToUpdate and Offset = %u %u"), NumPrimitiveDataUploads, PrimitiveOffset);

                    Scene.GPUScene.PrimitiveUploadBuffer.Init(MaxPrimitivesUploads,
sizeof(FPrimitiveSceneShaderData::Data),true, TEXT("PrimitiveUploadBuffer"));

                    //In batches of a single maximum size, fill the metadata intoPrimitiveUploadBuffermiddle.
                    for(int32 IndexUpdate =0; (IndexUpdate < MaxPrimitivesUploads) &&
((IndexUpdate + PrimitiveOffset) < NumPrimitiveDataUploads); ++IndexUpdate)
                    {
                        int32 Index = Scene.GPUScene.PrimitivesToUpdate[IndexUpdate +
PrimitiveOffset];
                        // PrimitivesToUpdate may contain a stale out of bounds index, as we
don't remove update request on primitive removal from scene.
                        if(Index < Scene.PrimitiveSceneProxies.Num()) {

                            FPrimitiveSceneProxy* RESTRICT PrimitiveSceneProxy =
Scene.PrimitiveSceneProxies[Index];
                            NumLightmapDataUploads += PrimitiveSceneProxy-
> GetPrimitiveSceneInfo()->GetNumLightmapDataEntries();

                            FPrimitiveSceneShaderDataPrimitiveSceneData(PrimitiveSceneProxy);
                            Scene.GPUScene.PrimitiveUploadBuffer.Add(Index,
&PrimitiveSceneData.Data[0]);
                        }
                        Scene.GPUScene.PrimitivesMarkedToUpdate[Index] =false;
                    }

                    //  ConversionUAVStatus toCompute,To upload dataGPUCopy
                    if  data. (bResizedPrimitiveData)
                    {
                        RHICmdList.TransitionResource(EResourceTransitionAccess::ERWBarrier,
EResourceTransitionPipeline::EComputeToCompute, MirrorResourceGPU->UAV);
                    }
                    else
                    {
                        RHICmdList.TransitionResource(EResourceTransitionAccess::EWritable,
EResourceTransitionPipeline::EGfxToCompute, MirrorResourceGPU->UAV);
                    }

                    //  Upload data toGPU.
                    {
                        Scene.GPUScene.PrimitiveUploadBuffer.ResourceUploadTo(RHICmdList, true);
 * MirrorResourceGPU,
                     }
                }

                //ConversionUAVStatus toGfx,So that when rendering objectsshaderAvailable.
                RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
EResourceTransitionPipeline::EComputeToGfx, MirrorResourceGPU->UAV);
            }

            //MakeMirrorResourceGPUThe data takes effect.
```

```
            if(GGPUSceneValidatePrimitiveBuffer && (Scene.GPUScene.PrimitiveBuffer.NumBytes > 0||
Scene.GPUScene.PrimitiveTexture.NumBytes >0))
            {
                //UE_LOG(LogRenderer, Warning, TEXT("r.GPUSceneValidatePrimitiveBuffer
enabled, doing slow readback from GPU"));
                uint32 Stride =0;
                FPrimitiveSceneShaderData* InstanceBufferCopy = (FPrimitiveSceneShaderData*)
(FPrimitiveSceneShaderData*)LockResource(*MirrorResourceGPU, Stride);

                constint32 TotalNumberPrimitives = Scene.PrimitiveSceneProxies.Num(); int32
                MaxPrimitivesUploads = GetMaxPrimitivesUpdate(TotalNumberPrimitives,
FPrimitiveSceneShaderData::PrimitiveDataStrideInFloat4s);
                for(int32 IndexOffset =0; IndexOffset < TotalNumberPrimitives; IndexOffset
+ = MaxPrimitivesUploads)
                {
                    for(int32 Index =0; (Index < MaxPrimitivesUploads) && ((Index +
IndexOffset) < TotalNumberPrimitives); ++Index)
                    {
                        FPrimitiveSceneShaderData
PrimitiveSceneData(Scene.PrimitiveSceneProxies[Index + IndexOffset]);

                        for(inti =0; i <
FPrimitiveSceneShaderData::PrimitiveDataStrideInFloat4s;                    i++)
                        {
                            check(PrimitiveSceneData.Data[i]                ==
InstanceBufferCopy[Index].Data[i]);
                        }
                    }
                    InstanceBufferCopy += Stride /sizeof(FPrimitiveSceneShaderData);
                }

                UnlockResourceGPUScene(*MirrorResourceGPU);
            }

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

Regarding the above code, it is worth mentioning that:

- In order to reduce the need to recreate GPU resources due to the size being too small, the GPU Scene resource growth (retention) strategy is: no less than 256, and up to the power of 2 (ie doubled). This strategy isstd::vectorsomewhat similar to .
- UE's default RHI is based on DirectX11. When converting resources, there are only two types: Gfx and ComputeResourceTransitionPipeline,and the Copy type of DirectX12 is ignored:

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

enum classEResourceTransitionPipeline {

    EGfxToCompute,
    EComputeToGfx,
    EGfxToGfx,
```

```
        EComputeToCompute,
};
```

4.3.4 InitViews

InitViewsIt is executed right after the GPU Scene is updated. It handles a lot of important rendering logic: visibility determination, collecting scene graph metadata and tags, creating visible mesh commands, initializing the data required for Pass rendering, etc. The code is as follows :

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

bool FDeferredShadingSceneRenderer::InitViews(FRHICommandListImmediate& RHICmdList,
FExclusiveDepthStencil::Type BasePassDepthStencilAccess,structFILCUpdatePrimTaskData& ILCTaskData,
FGraphEventArray& UpdateViewCustomDataEvents)
{
    //Creates a visibility frame to set up the preparatory phase.
    PreVisibilityFrameSetup(RHICmdList);

    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

    //Special effects system initialization
    {
        if(Scene->FXSystem && Scene->FXSystem->RequiresEarlyViewUniformBuffer() && Views.IsValidIndex(0
        ))
        {
            //Guaranteed firstviewofRHIThe resource has been initialized. Views[0].InitRHIResources();
            Scene->FXSystem->PostInitViews(RHICmdList, Views[0].ViewUniformBuffer,

Views[0].AllowGPUParticleUpdate() && !ViewFamily.EngineShowFlags.HitProxies);
        }
    }

    //Create visibility grid directive.
    FViewVisibleCommandsPerView ViewCommandsPerView;
    ViewCommandsPerView.SetNum(Views.Num());

    //Calculate visibility.
    ComputeViewVisibility(RHICmdList,          BasePassDepthStencilAccess,          ViewCommandsPerView,
DynamicIndexBufferForInitViews,          DynamicVertexBufferForInitViews,
DynamicReadBufferForInitViews);

    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

    //Capsule Shadow
    CreateIndirectCapsuleShadows();
    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

    //  Initializes atmospheric effects.
    if  (ShouldRenderSkyAtmosphere(Scene, ViewFamily.EngineShowFlags))
    {
        InitSkyAtmosphereForViews(RHICmdList);
    }

    //Create a visibility frame setup post stage.
    PostVisibilityFrameSetup(ILCTaskData);
```

```
	RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

	(.....)

	//Is it possible inPrepassAfter initializationview,Depend onGDoInitViewsLightingAfterPrepassdecide, and
GDoInitViewsLightingAfterPrepassYou can also use the console commandr.DoInitViewsLightingAfterPrepassset up.
	boolbDoInitViewAftersPrepass = !!GDoInitViewsLightingAfterPrepass; if(!
	bDoInitViewAftersPrepass) {

		InitViewsPossiblyAfterPrepass(RHICmdList, ILCTaskData,
		UpdateViewCustomDataEvents);
	}

	{
		//Initialize allviewofuniform bufferandRHIresource.
		for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

			FViewInfo& View = Views[ViewIndex];

			if(View.ViewState)
			{
				if(!View.ViewState->ForwardLightingResources) {

					View.ViewState->ForwardLightingResources.Reset(new
FForwardLightingViewResources());
				}

				View.ForwardLightingResources = View.ViewState-
> ForwardLightingResources.Get();
			}
			else
			{
				View.ForwardLightingResourcesStorage.Reset(new
FForwardLightingViewResources());
				View.ForwardLightingResources =
View.ForwardLightingResourcesStorage.Get();
			}

#ifRHI_RAYTRACING
			View.IESLightProfileResource = View.ViewState ? &View.ViewState-
> IESLightProfileResources : nullptr;
#endif
			// Set the pre-exposure before initializing the constant buffers. if(View.ViewState)

			{
				View.ViewState->UpdatePreExposure(View);
			}

			// Initialize the view's RHI resources.
			View.InitRHIResources();
		}
	}

	//Volumetric Fog
	SetupVolumetricFog();

	//Send a start rendering event.
	OnStartRender(RHICmdList);
```

```
    returnbDoInitViewAftersPrepass;
}
```

The above code does not seem to do much logic, but in fact a lot of logic is scattered in some important interfaces above. Let's analyze it firstPreVisibilityFrameSetup:

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

void FDeferredShadingSceneRenderer::PreVisibilityFrameSetup(FRHICommandListImmediate& RHICmdList)

{
    // Possible stencil dither optimization approach
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex]; View.bAllowStencilDither =
        bDitheredLODTransitionsUseStencil;
    }

    FSceneRenderer::PreVisibilityFrameSetup(RHICmdList);
}

void FSceneRenderer::PreVisibilityFrameSetup(FRHICommandListImmediate& RHICmdList) {

    //notifyRHIThe scene has started
    rendering. RHICmdList.BeginScene();

    {
        staticautoCVar = IConsoleManager::Get().FindConsoleVariable(TEXT(
"r.DoLazyStaticMeshUpdate"));
        const boolDoLazyStaticMeshUpdate = (CVar->GetInt() && !GIsEditor);

        //  Deferred static mesh updates.
        if  (DoLazyStaticMeshUpdate)
        {

QUICK_SCOPE_CYCLE_COUNTER(STAT_PreVisibilityFrameSetup_EvictionForLazyStaticMeshUpdate);
            staticint32 RollingRemoveIndex =0; staticint32
            RollingPassShrinkIndex =0;
            if(RollingRemoveIndex >= Scene->Primitives.Num()) {

                RollingRemoveIndex =0;
                RollingPassShrinkIndex++;
                if(RollingPassShrinkIndex >= UE_ARRAY_COUNT(Scene->CachedDrawLists)) {

                    RollingPassShrinkIndex =0;
                }
                // Periodically shrink the SparseArray containing cached mesh draw
commands which we are causing to be regenerated with UpdateStaticMeshes
                Scene->CachedDrawLists[RollingPassShrinkIndex].MeshDrawCommands.Shrink();
            }
            constint32 NumRemovedPerFrame =10; TArray<FPrimitiveSceneInfo*,
            TInlineAllocator<10>> SceneInfos; for(int32 NumRemoved =0; NumRemoved <
            NumRemovedPerFrame &&
RollingRemoveIndex < Scene->Primitives.Num(); NumRemoved++, RollingRemoveIndex++)
            {
```

```
                SceneInfos.Add(Scene->Primitives[RollingRemoveIndex]);
            }
            FPrimitiveSceneInfo::UpdateStaticMeshes(RHICmdList, Scene, SceneInfos,false);
        }
    }

    //ConversionSkin Cache
    RunGPUSkinCacheTransition(RHICmdList, Scene, EGPUSkinCacheTransition::FrameSetup);

    //initializationGroomhair
    if(IsHairStrandsEnable(Scene->GetShaderPlatform()) && Views.Num() >0) {

        constEWorldType::Type WorldType = Views[0].Family->Scene->GetWorld()->WorldType; const
        FShaderDrawDebugData* ShaderDrawData = &Views[0].ShaderDrawData; autoShaderMap =
        GetGlobalShaderMap(FeatureLevel); RunHairStrandsInterpolation(RHICmdList, WorldType,
                                                                      Scene->GetGPUSkinCache(),
ShaderDrawData, ShaderMap, EHairStrandsInterpolationType::SimulationStrands, nullptr);
    }

    //Special Effects System
    if(Scene->FXSystem && Views.IsValidIndex(0)) {

        Scene->FXSystem->PreInitViews(RHICmdList, !          Views[0].AllowGPUParticleUpdate()          &&
        ViewFamily.EngineShowFlags.HitProxies);
    }

    (.....)

    //Set motion blur parameters (includingTAAProcessing)
    for(int32 ViewIndex =0;ViewIndex < Views.Num();ViewIndex++) {

        FViewInfo& View = Views[ViewIndex]; FSceneViewState*
        ViewState = View.ViewState;

        check(View.VerifyMembersChecks());

        // Once per render increment the occlusion frame counter. if(ViewState)

        {
            ViewState->OcclusionFrameCounter++;
        }

        // HighResScreenshot should get best results so we don't do the occlusion optimization based on
the former frame
        externboolGIsHighResScreenshot;
        const boolbIsHitTesting = ViewFamily.EngineShowFlags.HitProxies;
        // Don't test occlusion queries in collision viewmode as they can be bigger then the rendering bounds.

        const boolbCollisionView = ViewFamily.EngineShowFlags.CollisionVisibility ||
        ViewFamily.EngineShowFlags.CollisionPawn;
        if(GIsHighResScreenshot || !DoOcclusionQueries(FeatureLevel) || bIsHitTesting || bCollisionView)

        {
            View.bDisableQuerySubmissions        =true;
            View.bIgnoreExistingQueries =        true;
        }
        FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);
```

```
            // set up the screen area for occlusion
            floatNumPossiblePixels = SceneContext.UseDownsizedOcclusionQueries() &&
            IsValidRef(SceneContext.GetSmallDepthSurface()) ?
                  (float)View.ViewRect.Width() /
SceneContext.GetSmallColorDepthDownsampleFactor() * (float)View.ViewRect.Height() /
SceneContext.GetSmallColorDepthDownsampleFactor() :
                  View.ViewRect.Width() * View.ViewRect.Height(); View.OneOverNumPossiblePixels = NumPossiblePixels >
            0.0?1.0f/NumPossiblePixels
:0.0f;

            // Still need no jitter to be set for temporal feedback on SSR (it is enabled even when temporal AA is off).

            check(View.TemporalJitterPixels.X            ==  0.0f);
            check(View.TemporalJitterPixels.Y            ==  0.0f);

            // Cache the projection matrix b
            // Cache the projection matrix before AA is applied
            View.ViewMatrices.SaveProjectionNoAAMatrix();

            if(ViewState)
            {
                  check(View.bStatePrevViewInfoIsReadOnly); View.bStatePrevViewInfoIsReadOnly =
                  ViewFamily.bWorldIsPaused ||
ViewFamily.EngineShowFlags.HitProxies || bFreezeTemporalHistories;

                  ViewState->SetupDistanceFieldTemporalOffset(ViewFamily);

                  if(!View.bStatePrevViewInfoIsReadOnly && !bFreezeTemporalSequences) {

                        ViewState->FrameIndex++;
                  }

                  if(View.OverrideFrameIndexValue.IsSet()) {

                        ViewState->FrameIndex = View.OverrideFrameIndexValue.GetValue();
                  }
            }

            // Subpixel jitter for temporal AA
            int32 CVarTemporalAASamplesValue = CVarTemporalAASamples.GetValueOnRenderThread();

            boolbTemporalUpsampling = View.PrimaryScreenPercentageMethod ==
            EPrimaryScreenPercentageMethod::TemporalUpscale;

            // Apply a sub pixel offset to the view.
            if(View.AntiAliasingMethod == AAM_TemporalAA && ViewState && (CVarTemporalAASamplesValue >0||
bTemporalUpsampling) && View.bAllowTemporalJitter)
            {
                  floatEffectivePrimaryResolutionFraction =float(View.ViewRect.Width()) /
float(View.GetSecondaryViewRectSize().X);

                  // Compute number of TAA samples.
                  int32 TemporalAASamples = CVarTemporalAASamplesValue; {

                        if(Scene->GetFeatureLevel() < ERHIFeatureLevel::SM5) {

                              // Only support 2 samples for mobile temporal AA.
                              TemporalAASamples =2;
```

```
                }
                else    if    (bTemporalUpsampling)
                {
                        // When doing TAA upsample with screen percentage < 100%, we need
extra temporal samples to have a
                        // constant temporal sample density for final output pixels to avoid
output pixel aligned converging issues.
                        TemporalAASamples =float(TemporalAASamples) * FMath::Max(1.f,1.f/
(EffectivePrimaryResolutionFraction * EffectivePrimaryResolutionFraction));
                }
                else if(CVarTemporalAASamplesValue ==5) {

                        TemporalAASamples =4;
                }

                TemporalAASamples = FMath::Clamp(TemporalAASamples,1,255);
        }

        // Compute the new sample index in the temporal sequence. int32
        TemporalSampleIndex                             = ViewState->TemporalAASampleIndex +1;
        if(TemporalSampleIndex >= TemporalAASamples || View.bCameraCut) {

                TemporalSampleIndex =0;
        }

        // Updates view state.
        if(!View.bStatePrevViewInfoIsReadOnly && !bFreezeTemporalSequences) {

                ViewState->TemporalAASampleIndex ViewState-          =  TemporalSampleIndex;
                >TemporalAASampleIndexUnclamped                      =  ViewState -
> TemporalAASampleIndexUnclamped+1;
        }

                //According to differentTAASampling strategies and parameters, select and calculate the corresponding
                sampling parameters. floatSampleX, SampleY;
        if(Scene->GetFeatureLevel() < ERHIFeatureLevel::SM5) {

                floatSamplesX[] = {-8.0f/16.0f,0.0/16.0f}; floatSamplesY[] = {/* - */0.0f/
                16.0f,8.0/16.0f}; check(TemporalAASamples ==
                UE_ARRAY_COUNT(SamplesX)); SampleX =
                SamplesX[ TemporalSampleIndex ];
                SampleY = SamplesY[ TemporalSampleIndex ];
        }
        else if(View.PrimaryScreenPercentageMethod ==
EPrimaryScreenPercentageMethod::TemporalUpscale)
        {
                // Uniformly distribute temporal jittering in [-.5; .5], because there is
no longer any alignment of input and output pixels.
                SampleX = Halton(TemporalSampleIndex +1,2) -0.5f; SampleY =
                Halton(TemporalSampleIndex +1,3) -0.5f;

                View.MaterialTextureMipBias = -(FMath::Max(-
FMath::Log2(EffectivePrimaryResolutionFraction),0.0f) ) +
CVarMinAutomaticViewMipBiasOffset.GetValueOnRenderThread();
                View.MaterialTextureMipBias = FMath::Max(View.MaterialTextureMipBias,
CVarMinAutomaticViewMipBias.GetValueOnRenderThread());
        }
        else if(CVarTemporalAASamplesValue ==2)
```

```
                {
                    // 2xMSAA
                    // Pattern docs: http://msdn.microsoft.com/en-
us/library/windows/desktop/ff476218(v=vs.85).aspx
                    //       N.
                    //       . S
                    floatSamplesX[] = {-4.0f/16.0f,4.0/16.0f}; floatSamplesY[] = {-4.0f/16.0f,
                    4.0/16.0f}; check(TemporalAASamples == UE_ARRAY_COUNT(SamplesX));
                    SampleX = SamplesX[ TemporalSampleIndex ];

                    SampleY = SamplesY[ TemporalSampleIndex ];
                }
                else if(CVarTemporalAASamplesValue ==3) {

                    //    3xMSAA
                    //       A..
                    //       . . B
                    //       . C.
                    // Rolling circle pattern (A,B,C). floatSamplesX[] = {-2.0f/3.0f, float
                    SamplesY[] = {-2.0f/3.0f, check(TemporalAASam2.p0l/e3s. 0=f0,0/3.0f};
                    UE_ARRAY_COUNT(SamplesX)); SampleX =       0.0/3.0f,          2.0/3.0f};
                    SamplesX[ TemporalSampleIndex ];

                    SampleY = SamplesY[ TemporalSampleIndex ];
                }
                else if(CVarTemporalAASamplesValue ==4) {

                    // 4xMSAA
                    // Pattern docs: http://msdn.microsoft.com/en-
us/library/windows/desktop/ff476218(v=vs.85).aspx
                    //       . N..
                    //       . . . E
                    //       W...
                    //       . . S.
                    // Rolling circle pattern (N,E,S,W).
                    floatSamplesX[] = {-2.0f/16.0f,6.0/16.0f,2.0/16.0f,-6.0/16.0f}; floatSamplesY[] = {-6.0f/16.0f,-2.0/
                    16.0f,6.0/16.0f,2.0/16.0f}; check(TemporalAASamples == UE_ARRAY_COUNT(SamplesX));

                    SampleX = SamplesX[ TemporalSampleIndex ]; SampleY
                    = SamplesY[ TemporalSampleIndex ];
                }
                else if(CVarTemporalAASamplesValue ==5) {

                    // Compressed 4 sample pattern on same vertical and horizontal line (less
temporal flicker).
                    // Compressed 1/2 works better than correct 2/3 (reduced temporal
flicker).
                    //       . N .
                    //       W.E
                    //       . S .
                    // Rolling circle pattern (N,E,S,W).
                    floatSamplesX[] = {0.0f/2.0f,1.0/2.0f,0.0/2.0f,-1.0/2.0f}; floatSamplesY[] = {-1.0f/2.0f,0.0/2.0f,
                    1.0/2.0f,0.0/2.0f}; check(TemporalAASamples == UE_ARRAY_COUNT(SamplesX));

                    SampleX = SamplesX[ TemporalSampleIndex ]; SampleY
                    = SamplesY[ TemporalSampleIndex ];
                }
                else
```

```
                {
                    floatu1 = Halton( TemporalSampleIndex +1,2); floatu2 =
                    Halton( TemporalSampleIndex +1,3);

                    // Generates samples in normal distribution // exp( x^2 /
                    Sigma^2 )

                    staticautoCVar =
IConsoleManager::Get().FindConsoleVariable(TEXT("r.TemporalAAFilterSize"));
                    floatFilterSize = CVar->GetFloat();

                    // Scale distribution to set non-unit variance // Variance =
                    Sigma^2
                    floatSigma =0.47f* FilterSize;

                    // Window to [-0.5, 0.5] output
                    // Without windowing we could generate samples far away on the infinite
tails.
                    floatOutWindow =0.5f;
                    floatInWindow = FMath::Exp(-0.5* FMath::Square( OutWindow / Sigma ) );

                    // Box-Muller transform floatTheta =
                    2.0f*PI*u2;
                    floatr = Sigma * FMath::Sqrt(-2.0f* FMath::Loge( (1.0f-u1) * InWindow
+ u1   )  ) ;

                    SampleX = r * FMath::Cos( Theta ); SampleY = r
                    * FMath::Sin( Theta );
                }

                View.TemporalJitterSequenceLength          = TemporalAASamples;
                View.TemporalJitterIndex =          TemporalSampleIndex;
                View.TemporalJitterPixels.X           =SampleX;
                View.TemporalJitterPixels.Y           = SampleY;

                View.ViewMatrices.HackAddTemporalAAProjectionJitter(FVector2D(SampleX *2.0f/
View.ViewRect.Width(), SampleY *-2.0f/ View.ViewRect.Height()));
            }

            // Setup a new FPreviousViewInfo from current frame infos. FPreviousViewInfo
            NewPrevViewInfo; {

                NewPrevViewInfo.ViewMatrices = View.ViewMatrices;
            }

            //initializationview state
            if( ViewState ) {

                // update previous frame matrices in case world origin was rebased on this
frame
                if(!View.OffsetThisFrame.IsZero()) {

                    ViewState -
> PrevFrameViewInfo.ViewMatrices.ApplyWorldOffset(View.OriginOffsetThisFrame);
                }

                // determine if we are initializing or we should reset the persistent state const floatDeltaTime =
                View.Family->CurrentRealTime - ViewState-
```

```
> LastRenderTime;
		const boolbFirstFrameOrTimeWasReset = DeltaTime <-0.0001f|| ViewState-
> LastRenderTime <0.0001f;
		const boolbIsLargeCameraMovement = IsLargeCameraMovement(
			View,
			ViewState->PrevFrameViewInfo.ViewMatrices.GetViewMatrix(), ViewState-
			>PrevFrameViewInfo.ViewMatrices.GetViewOrigin(),
			45.0f,10000.0f);
		const boolbResetCamera = (bFirstFrameOrTimeWasReset || View.bCameraCut ||
bIsLargeCameraMovement || View.bForceCameraVisibilityReset);

		(.....)

		if(bResetCamera)
		{
			View.PrevViewInfo = NewPrevViewInfo;

			// PT: If the motion blur shader is the last shader in the post-processing
chain then it is the one that is
			//		adjusting for the viewport offset. So it is always required and we
can't just disable the work the
			//		shader does. The correct fix would be to disable the effect when
we don't need it and to mark properly
			//		the uber-postprocessing effect as the last effect in the chain.

			View.bPrevTransformsReset =true;
		}
		else
		{
			View.PrevViewInfo = ViewState->PrevFrameViewInfo;
		}

		// Replace previous view info of the view state with this frame, clearing out
references over render target.
		if(!View.bStatePrevViewInfoIsReadOnly) {

			ViewState->PrevFrameViewInfo = NewPrevViewInfo;
		}

		// If the view has a previous view transform, then overwrite the previous view
info for the _current_ frame.
		if(View.PreviousViewTransform.IsSet()) {

			// Note that we must ensure this transform ends up in ViewState-
> PrevFrameViewInfo else it will be used to calculate the next frame's motion vectors as well


View.PrevViewInfo.ViewMatrices.UpdateViewMatrix(View.PreviousViewTransform-
> GetTranslation(), View.PreviousViewTransform->GetRotation().Rotator());
		}

		// detect conditions where we should reset occlusion queries if
		(bFirstFrameOrTimeWasReset ||
			ViewState->LastRenderTime + GEngine->PrimitiveProbablyVisibleTime <
View.Family->CurrentRealTime ||
			View.bCameraCut		||
			View.bForceCameraVisibilityReset		||
			IsLargeCameraMovement(
```

```
                                View,
                                ViewState->PrevViewMatrixForOcclusionQuery, ViewState-
                                >PrevViewOriginForOcclusionQuery, GEngine-
                                >CameraRotationThreshold, GEngine-
>CameraTranslationThreshold))
                        {
                                View.bIgnoreExistingQueries =true;
                                View.bDisableDistanceBasedFadeTransitions =true;
                        }

                        // Turn on/off round-robin occlusion querying in the ViewState static constauto
                        CVarRROCC =
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("vr.RoundRobinOcclusion"));
                        const boolbEnableRoundRobin = CVarRROCC ? (CVarRROCC->GetValueOnAnyThread()
!=false) :false;
                        if(bEnableRoundRobin != ViewState->IsRoundRobinEnabled()) {

                                ViewState->UpdateRoundRobin(bEnableRoundRobin);
                                View.bIgnoreExistingQueries =true;
                        }

                        ViewState->PrevViewMatrixForOcclusionQuery              =
View.ViewMatrices.GetViewMatrix();
                        ViewState->PrevViewOriginForOcclusionQuery              =
View.ViewMatrices.GetViewOrigin();

                        (.....)

                        // we don't use DeltaTime as it can be 0 (in editor) and is computed by
subtracting floats (losses precision over time)
                        // Clamp DeltaWorldTime to reasonable values for the purposes of motion blur,
things like TimeDilation can make it very small
                        // Offline renders always control the timestep for the view and always need
the timescales calculated.
                        if(!ViewFamily.bWorldIsPaused || View.bIsOfflineRender) {

                                ViewState->UpdateMotionBlurTimeScale(View);
                        }


                        ViewState->PrevFrameNumber = ViewState->PendingPrevFrameNumber; ViewState-
                        >PendingPrevFrameNumber = View.Family->FrameNumber;

                        // This finishes the update of view state ViewState-
                        >UpdateLastRenderTime(*View.Family);

                        ViewState->UpdateTemporalLODTransition(View);
                }
                else
                {
                        // Without a viewstate, we just assume that camera has not moved. View.PrevViewInfo =
                        NewPrevViewInfo;
                }
        }

        //Set global dither parameters and transitionsuniform buffer.
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {
```

```
            FViewInfo& View = Views[ViewIndex];

            FDitherUniformShaderParameters DitherUniformShaderParameters;
            DitherUniformShaderParameters.LODFactor = View.GetTemporalLODTransition();
            View.DitherFadeOutUniformBuffer =
FDitherUniformBufferRef::CreateUniformBufferImmediate(DitherUniformShaderParameters,
UniformBuffer_SingleFrame);

            DitherUniformShaderParameters.LODFactor = View.GetTemporalLODTransition() -1.0f;
            View.DitherFadeInUniformBuffer =
FDitherUniformBufferRef::CreateUniformBufferImmediate(DitherUniformShaderParameters,
UniformBuffer_SingleFrame);
        }
}
```

From this, we can see thatPreVisibilityFrameSetupa lot of initialization work has been done, such as static grid, Groom, SkinCache, special effects, TAA, ViewState, etc. Then continue the analysis ComputeViewVisibility:

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

void FSceneRenderer::ComputeViewVisibility(FRHICommandListImmediate&                    RHICmdList,
FExclusiveDepthStencil::Type BasePassDepthStencilAccess, ViewCommandsPeFrVViieewV, isibleCommandsPerView&
FGlobalDynamicIndexBuffer&                                                DynamicIndexBuffer,
FGlobalDynamicVertexBuffer& DynamicVertexBuffer,                FGlobalDynamicReadBuffer&
DynamicReadBuffer)
{
    //Assigns a list of visible light source information.
    if(Scene->Lights.GetMaxIndex() >0) {

        VisibleLightInfos.AddZeroed(Scene->Lights.GetMaxIndex());
    }

    int32 NumPrimitives = Scene->Primitives.Num(); float
    CurrentRealTime = ViewFamily.CurrentRealTime;

    FPrimitiveViewMasks HasDynamicMeshElementsMasks;
    HasDynamicMeshElementsMasks.AddZeroed(NumPrimitives);

    FPrimitiveViewMasks HasViewCustomDataMasks;
    HasViewCustomDataMasks.AddZeroed(NumPrimitives);

    FPrimitiveViewMasks HasDynamicEditorMeshElementsMasks;

    if(GIsEditor)
    {
        HasDynamicEditorMeshElementsMasks.AddZeroed(NumPrimitives);
    }

    const boolbIsInstancedStereo = (Views.Num() >0) ? (Views[0].IsInstancedStereoPass() || Views[0
].bIsMobileMultiViewEnabled):false;
    UpdateReflectionSceneData(Scene);

    //Updated static meshes to not check visibility.
    {
        Scene->ConditionalMarkStaticMeshElementsForUpdate();
```

```
            TArray<FPrimitiveSceneInfo*> UpdatedSceneInfos; for
            (TSet<FPrimitiveSceneInfo*>::TIterator It(Scene-
> PrimitivesNeedingStaticMeshUpdateWithoutVisibilityCheck); It; { + + It)

                FPrimitiveSceneInfo* Primitive = *It; if(Primitive-
                >NeedsUpdateStaticMeshes()) {

                    UpdatedSceneInfos.Add(Primitive);
                }
            }
            if(UpdatedSceneInfos.Num() >0) {

                FPrimitiveSceneInfo::UpdateStaticMeshes(RHICmdList, Scene, UpdatedSceneInfos);
            }
            Scene->PrimitivesNeedingStaticMeshUpdateWithoutVisibilityCheck.Reset();
        }

        //Initialize allviewof data.
        uint8 ViewBit =0x1;
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex, ViewBit <<=1) {

            STAT(NumProcessedPrimitives += NumPrimitives);

            FViewInfo& View = Views[ViewIndex];
            FViewCommands& ViewCommands = ViewCommandsPerView[ViewIndex];
            FSceneViewState* ViewState = (FSceneViewState*)View.State;

            // Allocate the view's visibility maps. View.PrimitiveVisibilityMap.Init(false,Scene-
            >Primitives.Num()); // we don't initialized as we overwrite the whole array (in
            GatherDynamicMeshElements)

            View.DynamicMeshEndIndices.SetNumUninitialized(Scene->Primitives.Num());
            View.PrimitiveDefinitelyUnoccludedMap.Init(false,Scene->Primitives.Num());
            View.PotentiallyFadingPrimitiveMap.Init(false,Scene->Primitives.Num());
            View.PrimitiveFadeUniformBuffers.AddZeroed(Scene->Primitives.Num());
            View.PrimitiveFadeUniformBufferMap.Init(false, Scene->Primitives.Num());
            View.StaticMeshVisibilityMap.Init(false,Scene->StaticMeshes.GetMaxIndex());
            View.StaticMeshFadeOutDitheredLODMap.Init(false,Scene-
> StaticMeshes.GetMaxIndex());
            View.StaticMeshFadeInDitheredLODMap.Init(false,Scene->StaticMeshes.GetMaxIndex());
            View.StaticMeshBatchVisibility.AddZeroed(Scene-
> StaticMeshBatchVisibility.GetMaxIndex()); View.PrimitivesLODMask.Init(FLODMask(), Scene-
>Primitives.Num());

            View.PrimitivesCustomData.Init(nullptr, Scene->Primitives.Num());

            // We must reserve to prevent realloc otherwise it will cause memory leak if we In Parallel
Execute
            const boolWillExecuteInParallel = FApp::ShouldUseThreadingForPerformance() &&
CVarParallelInitViews.GetValueOnRenderThread() >0;
            View.PrimitiveCustomDataMemStack.Reserve(WillExecuteInParallel ?
FMath::CeilToInt((((float)View.PrimitiveVisibilityMap.Num() (float /
)FRelevancePrimSet<int32>::MaxInputPrims)) +1:1);

            View.AllocateCustomDataMemStack();

            View.VisibleLightInfos.Empty(Scene->Lights.GetMaxIndex());
```

```
            View.DirtyIndirectLightingCacheBufferPrimitives.Reserve(Scene->Primitives.Num());

            //Create light source information.
            for(int32 LightIndex =0;LightIndex < Scene->Lights.GetMaxIndex();LightIndex++) {

                if( LightIndex+2< Scene->Lights.GetMaxIndex() ) {

                    if(LightIndex >2) {

                        FLUSH_CACHE_LINE(&View.VisibleLightInfos(LightIndex-2));
                    }
                }
                new(View.VisibleLightInfos) FVisibleLightViewInfo();
            }

            View.PrimitiveViewRelevanceMap.Empty(Scene->Primitives.Num());
            View.PrimitiveViewRelevanceMap.AddZeroed(Scene->Primitives.Num());

            const boolbIsParent = ViewState && ViewState->IsViewParent(); if( bIsParent ) {


                ViewState->ParentPrimitives.Empty();
            }

            if(ViewState)
            {
                //Get and decompress the occlusion data of the previous frame.
                View.PrecomputedVisibilityData              =  ViewState->GetPrecomputedVisibilityData(View,
Scene);
            }
            else
            {
                View.PrecomputedVisibilityData              =  NULL;
            }

            if(View.PrecomputedVisibilityData) {

                bUsedPrecomputedVisibility =true;
            }

            boolbNeedsFrustumCulling =true;

#if!(UE_BUILD_SHIPPING || UE_BUILD_TEST) if
            ( ViewState ) {

                //Freeze Visibility
                if(ViewState->bIsFrozen) {

                    bNeedsFrustumCulling =false;
                    for(FSceneBitArray::FIterator BitIt(View.PrimitiveVisibilityMap); BitIt;
+ + BitIt)
                    {
                        if(ViewState->FrozenPrimitives.Contains(Scene-
> PrimitiveComponentIds[BitIt.GetIndex()]))
                        {
                            BitIt.GetValue() =true;
                        }
```

```
                }
            }
        }
#endif

        // Frustum clipping.
        if (bNeedsFrustumCulling)
        {
            // Update HLOD transition/visibility states to allow use during distance
culling
            FLODSceneTree& HLODTree = Scene->SceneLODHierarchy; if
            (HLODTree.IsActive()) {

                QUICK_SCOPE_CYCLE_COUNTER(STAT_ViewVisibilityTime_HLODUpdate);
                HLODTree.UpdateVisibilityStates(View);
            }
            else
            {
                HLODTree.ClearVisibilityState(View);
            }

            int32 NumCulledPrimitivesForView;
            const boolbUseFastIntersect = (View.ViewFrustum.PermutedPlanes.Num() ==8) &&
CVarUseFastIntersect.GetValueOnRenderThread();
            if(View.CustomVisibilityQuery && View.CustomVisibilityQuery->Prepare()) {

                if(CVarAlsoUseSphereForFrustumCull.GetValueOnRenderThread()) {

                    NumCulledPrimitivesForView = bUseFastIntersect? FrustumCull<true,
true,true>(Scene, View) : FrustumCull<true,true,false>(Scene, View);
                }
                else
                {
                    NumCulledPrimitivesForView = bUseFastIntersect? FrustumCull<true,
false,true>(Scene, View) : FrustumCull<true,false,false>(Scene, View);
                }
            }
            else
            {
                if(CVarAlsoUseSphereForFrustumCull.GetValueOnRenderThread()) {

                    NumCulledPrimitivesForView = bUseFastIntersect? FrustumCull<false,
true,true>(Scene, View) : FrustumCull<false,true,false>(Scene, View);
                }
                else
                {
                    NumCulledPrimitivesForView = bUseFastIntersect? FrustumCull<false,
false,true>(Scene, View) : FrustumCull<false,false,false>(Scene, View);
                }
            }
            STAT(NumCulledPrimitives   +=        NumCulledPrimitivesForView); View);
            UpdatePrimitiveFading(Scene,
        }

        // Handling hidden objects.
        if (View.HiddenPrimitives.Num())
        {
            for(FSceneSetBitIterator BitIt(View.PrimitiveVisibilityMap); BitIt; ++BitIt)
```

```
                {
                    if(View.HiddenPrimitives.Contains(Scene-
> PrimitiveComponentIds[BitIt.GetIndex()]))
                    {
                        View.PrimitiveVisibilityMap.AccessCorrespondingBit(BitIt)                    =  false;
                    }
                }
            }

            (......)

            //  Handling static scenes.
            if  (View.bStaticSceneOnly)
            {
                for(FSceneSetBitIterator BitIt(View.PrimitiveVisibilityMap); BitIt; ++BitIt) {

                    // Reflection captures should only capture objects that won't move, since
reflection captures won't update at runtime
                    if(!Scene->Primitives[BitIt.GetIndex()]->Proxy->HasStaticLighting()) {

                        View.PrimitiveVisibilityMap.AccessCorrespondingBit(BitIt) =false;
                    }
                }
            }

            (......)

            //  For non-wireframe mode, only occlusion culling is required.
            if  (!View.Family->EngineShowFlags.Wireframe)
            {
                int32 NumOccludedPrimitivesInView = OcclusionCull(RHICmdList, Scene, View,
DynamicVertexBuffer);
                STAT(NumOccludedPrimitives += NumOccludedPrimitivesInView);
            }

            //Handles visibility determination for static models.
            {
                TArray<FPrimitiveSceneInfo*> AddedSceneInfos;
                for(TConstDualSetBitIterator<SceneRenderingBitArrayAllocator,
FDefaultBitArrayAllocator> BitIt(View.PrimitiveVisibilityMap, Scene-
> PrimitivesNeedingStaticMeshUpdate); BitIt; ++BitIt)
                {
                    int32 PrimitiveIndex = BitIt.GetIndex(); AddedSceneInfos.Add(Scene-
                    >Primitives[PrimitiveIndex]);
                }

                if(AddedSceneInfos.Num() >0) {

                    FPrimitiveSceneInfo::UpdateStaticMeshes(RHICmdList,                    Scene,
AddedSceneInfos);
                }
            }

            (......)
        }

        (......)
```

```
        //Collect AllviewDynamic grid elements. The previous article has been analyzed in detail. {

            GatherDynamicMeshElements(Views, Scene, ViewFamily, DynamicIndexBuffer,
DynamicVertexBuffer, DynamicReadBuffer,
                    HasDynamicMeshElementsMasks, HasDynamicEditorMeshElementsMasks,
HasViewCustomDataMasks, MeshCollector);
        }

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

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

                continue;
            }

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

ComputeViewVisibilityThe most important function is to handle the visibility of primitives, including frustum clipping, occlusion culling, collecting dynamic mesh information, creating light source information, etc. Let's continue with a rough analysis of the process of pre-calculating visibility:

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

constuint8*FSceneViewState::GetPrecomputedVisibilityData(FViewInfo& View,constFScene* Scene)

{
        constuint8* PrecomputedVisibilityData =NULL;
        if(Scene->PrecomputedVisibilityHandler && GAllowPrecomputedVisibility && View.Family-
> EngineShowFlags.PrecomputedVisibility) {

            constFPrecomputedVisibilityHandler& Handler = *Scene-
> PrecomputedVisibilityHandler;
            FViewElementPDIVisibilityCellsPDI(&View, nullptr, nullptr);

            //Draws the occlusion culling grid for debugging.
            if((GShowPrecomputedVisibilityCells || View.Family-
> EngineShowFlags.PrecomputedVisibilityCells) && !GShowRelevantPrecomputedVisibilityCells) {

                for(int32 BucketIndex =0; BucketIndex <
Handler.PrecomputedVisibilityCellBuckets.Num(); BucketIndex++)
                {
                    for(int32 CellIndex =0; CellIndex <
Handler.PrecomputedVisibilityCellBuckets[BucketIndex].Cells.Num(); CellIndex++)
                    {
                        constFPrecomputedVisibilityCell& CurrentCell =
Handler.PrecomputedVisibilityCellBuckets[BucketIndex].Cells[CellIndex];
                        // Construct the cell's bounds
                        constFBoxCellBounds(CurrentCell.Min, CurrentCell.Min +
FVector(Handler.PrecomputedVisibilityCellSizeXY, Handler.PrecomputedVisibilityCellSizeXY,
Handler.PrecomputedVisibilityCellSizeZ));
                        if(View.ViewFrustum.IntersectBox(CellBounds.GetCenter(),
```

```
CellBounds.GetExtent()
                        {
                            DrawWireBox(&VisibilityCellsPDI, CellBounds, FColor(50,50,255),
SDPG_World);
                        }
                    }
                }
            }

            //Calculate hash values and bucket indexes to reduce search time.
            const floatFloatOffsetX = (View.ViewMatrices.GetViewOrigin().X -
            Handler.PrecomputedVisibilityCellBucketOriginXY.X) /
Handler.PrecomputedVisibilityCellSizeXY;
            // FMath::TruncToInt rounds toward 0, we want to always round down constint32 BucketIndexX =
            FMath::Abs((FMath::TruncToInt(FloatOffsetX) - (FloatOffsetX <0.0f?1:0)) /
Handler.PrecomputedVisibilityCellBucketSizeXY % Handler.PrecomputedVisibilityNumCellBuckets);

            const floatFloatOffsetY = (View.ViewMatrices.GetViewOrigin().Y -
            Handler.PrecomputedVisibilityCellBucketOriginXY.Y) /
Handler.PrecomputedVisibilityCellSizeXY;
            constint32 BucketIndexY = FMath::Abs((FMath::TruncToInt(FloatOffsetY) - (FloatOffsetY <0.0f?1:0
)) / Handler.PrecomputedVisibilityCellBucketSizeXY % Handler.PrecomputedVisibilityNumCellBuckets);

            constint32 PrecomputedVisibilityBucketIndex = BucketIndexY *
Handler.PrecomputedVisibilityCellBucketSizeXY + BucketIndexX;

            //Draw the bounding box corresponding to the visible object.
            constFPrecomputedVisibilityBucket& CurrentBucket =
            Handler.PrecomputedVisibilityCellBuckets[PrecomputedVisibilityBucketIndex];
            for(int32 CellIndex =0; CellIndex < CurrentBucket.Cells.Num(); CellIndex++) {

                constFPrecomputedVisibilityCell& CurrentCell =
CurrentBucket.Cells[CellIndex];
                //createcellThe bounding box of .
                constFBoxCellBounds(CurrentCell.Min, CurrentCell.Min +
FVector(Handler.PrecomputedVisibilityCellSizeXY, Handler.PrecomputedVisibilityCellSizeXY,
Handler.PrecomputedVisibilityCellSizeZ));
                // Check if ViewOrigin is inside the current cell
                if(CellBounds.IsInside(View.ViewMatrices.GetViewOrigin())) {

                    //  Checks whether cached data can be reused.
                    if  (CachedVisibilityChunk
                        && CachedVisibilityHandlerId == Scene->PrecomputedVisibilityHandler-
> GetId()
                        && CachedVisibilityBucketIndex == PrecomputedVisibilityBucketIndex &&
                        CachedVisibilityChunkIndex == CurrentCell.ChunkIndex)
                    {
                        PrecomputedVisibilityData = &(*CachedVisibilityChunk)
[CurrentCell.DataOffset];
                    }
                    else
                    {
                        constFCompressedVisibilityChunk& CompressedChunk =
Handler.PrecomputedVisibilityCellBuckets[PrecomputedVisibilityBucketIndex].CellDataChunks[CurrentCell.ChunkIndex];

                        CachedVisibilityBucketIndex      = PrecomputedVisibilityBucketIndex;
                        CachedVisibilityChunkIndex     =  CurrentCell.ChunkIndex;
                        CachedVisibilityHandlerId      = Scene->PrecomputedVisibilityHandler-
```

```
> GetId();

                        //  Decompress occlusion data.
                        if  (CompressedChunk.bCompressed)
                        {
                            // Decompress the needed visibility data chunk
                            DecompressedVisibilityChunk.Reset();

DecompressedVisibilityChunk.AddUninitialized(CompressedChunk.UncompressedSize);
                            verify(FCompression::UncompressMemory(
                                NAME_Zlib,
                                DecompressedVisibilityChunk.GetData(),
                                CompressedChunk.UncompressedSize,
                                CompressedChunk.Data.GetData(),
                                CompressedChunk.Data.Num()));
                            CachedVisibilityChunk          = &DecompressedVisibilityChunk;
                        }
                        else
                        {
                            CachedVisibilityChunk          =  &CompressedChunk.Data;
                        }

                        // Return a pointer to the cell containing ViewOrigin's decompressed
visibility       data
                        PrecomputedVisibilityData = &(*CachedVisibilityChunk)
[CurrentCell.DataOffset];
                    }

                    (......)
                }

                (......)
    }

    returnPrecomputedVisibilityData;
}
```

From the code, we know that visibility judgment can draw some debugging information, such as the size of the bounding box of each object actually used for culling, and can also freeze the culling results to observe the efficiency of occlusion.

Since visibility determination includes pre-calculation, frustum clipping, occlusion culling, etc., occlusion culling alone involves many knowledge points (drawing, acquisition, data compression and decompression, storage structure, multi-threaded transmission, frame-to-frame interaction, visibility determination, HLOD, etc.), only a rough analysis of visibility determination is made here, and more detailed mechanisms and processes will be analyzed in depth in the subsequent rendering
optimization topic.

Next, continueInitViewsto analyzePostVisibilityFrameSetup:

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

void FSceneRenderer::PostVisibilityFrameSetup(FILCUpdatePrimTaskData& OutILCTaskData) {
```

```
//Handling decal sorting and adjustment historyRT.
{
    QUICK_SCOPE_CYCLE_COUNTER(STAT_PostVisibilityFrameSetup_Sort); for(int32
    ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];

        View.MeshDecalBatches.Sort();

        if(View.State)
        {
            ((FSceneViewState*)View.State)->TrimHistoryRenderTargets(Scene);
        }
    }
}

(.....)

//  Handles light visibility.
{
    QUICK_SCOPE_CYCLE_COUNTER(STAT_PostVisibilityFrameSetup_Light_Visibility); Traverse all light sources
//  and combineviewThe same light source may be visible in someviewVisible but otherviewnot visible).
for(TSparseArray<FLightSceneInfoCompact>::TConstIterator                    LightIt(Scene-
>Lights);LightIt;++LightIt) {

    constFLightSceneInfoCompact& LightSceneInfoCompact = *LightIt;
    constFLightSceneInfo* LightSceneInfo = LightSceneInfoCompact.LightSceneInfo;

    //Use eachviewThe camera frustum clips the light source.
    for(int32 ViewIndex =0;ViewIndex < Views.Num();ViewIndex++) {

        constFLightSceneProxy* Proxy = LightSceneInfo->Proxy; FViewInfo& View
        = Views[ViewIndex];
        FVisibleLightViewInfo& VisibleLightViewInfo =
View.VisibleLightInfos[LightIt.GetIndex()];
        //Parallel directional light does not need clipping, only local light sources need clipping
        if(Proxy->GetLightType() == LightType_Point ||
            Proxy->GetLightType() == LightType_Spot || Proxy-
            >GetLightType() == LightType_Rect )
        {
            FSphereconst& BoundingSphere = Proxy->GetBoundingSphere(); //
                judgeviewWhether the viewing frustum intersects with the light source bounding box.
            if  (View.ViewFrustum.IntersectSphere(BoundingSphere.Center,
BoundingSphere.W))
            {
                //  The perspective frustum needs to be corrected for the maximum distance to remove light sources that are
                if  too far from the viewpoint. (View.IsPerspectiveProjection())
                {
                    FSphere Bounds = Proxy->GetBoundingSphere(); float
                    DistanceSquared = (Bounds.Center -
View.ViewMatrices.GetViewOrigin()).SizeSquared();
                    floatMaxDistSquared = Proxy->GetMaxDrawDistance() * Proxy-
> GetMaxDrawDistance() * GLightMaxDrawDistanceScale * GLightMaxDrawDistanceScale;
                    //Taking into account the radius of the light source, the viewLODfactors, minimum light source screen radius, etc. to determine whether the final light source needs
To draw, in order to cull distant light sources that occupy a small part of the screen.
                    const boolbDrawLight = (FMath::Square(FMath::Min(0.0002f,
GMinScreenRadiusForLights / Bounds.W) * View.LODDistanceFactor) * DistanceSquared <1.0f)
                                                    && (MaxDistSquared ==0||
```

```
DistanceSquared < MaxDistSquared);

                                VisibleLightViewInfo.bInViewFrustum             =  bDrawLight;
                            }
                            else
                            {
                                VisibleLightViewInfo.bInViewFrustum             =  true;
                            }
                        }
                    }
                    else
                    {
                        //    Parallel directional light must be visible.
                        VisibleLightViewInfo.bInViewFrustum  =true;

                        (......)
                    }

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

To summarize,PostVisibilityFrameSetupthe main work of the shader is to use the view cone to clip light sources, preventing light sources that are out of sight or have a small screen share or no light intensity from entering the shader calculation. In addition, it also handles decal sorting, adjusting the
RT and fog effects of the previous frame, light beams, etc.

AtInitViewsthe end, the RHI resources of each view will be initialized and the RHICommandList will be notified of the rendering start event. Let's take a look at it firstFViewInfo::InitRHIResources:

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

void FViewInfo::InitRHIResources() {

        FBox VolumeBounds[TVC_MAX];

        //Creating and setting cached viewsUniform Buffer.
        CachedViewUniformShaderParameters = MakeUnique<FViewUniformShaderParameters>();

        FSceneRenderTargets& SceneContext =
        FSceneRenderTargets::Get(FRHICommandListExecutor::GetImmediateCommandList());

        SetupUniformBufferParameters(
                SceneContext,
                VolumeBounds,
                TVC_MAX,
                * CachedViewUniformShaderParameters);

        //Creating and setting the viewUniform
        Buffer. ViewUniformBuffer =
TUniformBufferRef<FViewUniformShaderParameters>::CreateUniformBufferImmediate(*CachedViewU
niformShaderParameters, UniformBuffer_SingleFrame);

        constint32 TranslucencyLightingVolumeDim = GetTranslucencyLightingVolumeDim();

        //Reset cacheUniform Buffer.
```

```
    FScene* Scene = Family->Scene ? Family->Scene->GetRenderScene() : nullptr; if(Scene)

    {
        Scene->UniformBuffers.InvalidateCachedView();
    }

    //Initialize transparent volume lighting parameters.
    for(int32 CascadeIndex =0; CascadeIndex < TVC_MAX; CascadeIndex++) {

        TranslucencyLightingVolumeMin[CascadeIndex] = VolumeBounds[CascadeIndex].Min;
        TranslucencyVolumeVoxelSize[CascadeIndex] = (VolumeBounds[CascadeIndex].Max.X -
VolumeBounds[CascadeIndex].Min.X) / TranslucencyLightingVolumeDim;
        TranslucencyLightingVolumeSize[CascadeIndex] = VolumeBounds[CascadeIndex].Max -
        VolumeBounds[CascadeIndex].Min;
    }
}
```

Continue parsingInitViewsthe endOnStartRender:

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

void FSceneRenderer::OnStartRender(FRHICommandListImmediate& RHICmdList) {

    //ScenarioMRTInitialization of .
    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

    FVisualizeTexturePresent::OnStartRender(Views[0]);
    CompositionGraph_OnStartFrame();
    SceneContext.bScreenSpaceAOIsValid          =false;
    SceneContext.bCustomDepthIsValid =          false;

    //notifyViewStateinitialization. for
    (FViewInfo& View : Views) {

        if(View.ViewState)
        {
            View.ViewState->OnStartRender(View,             ViewFamily);
        }
    }
}

// Engine\Source\Runtime\Renderer\Private\ScenePrivate.h

void FSceneViewState::OnStartRender(FViewInfo& View, FSceneViewFamily& ViewFamily) {

    //Initializes and sets up the light transfer volume.
    if(!(View.FinalPostProcessSettings.IndirectLightingColor *
    View.FinalPostProcessSettings.IndirectLightingIntensity).IsAlmostBlack())
    {
        SetupLightPropagationVolume(View, ViewFamily);
    }
    //Assign software-level scene occlusion culling (for low-end machines that do not support hardware occlusion
    culling) ConditionallyAllocateSceneSoftwareOcclusion(View.GetFeatureLevel());
}
```

In order to help those who don't look closely at code analysis to better understandInitViewsthe process, the following briefly summarizes its main steps and processes:

- **PreVisibilityFrameSetup**:Visibility determination preprocessing stage, mainly initializing and setting static meshes, Groom, SkinCache, special effects, TAA, ViewState, etc.
- Initialize the special effects system (FXSystem).
- **ComputeViewVisibility**:Calculates view-related visibility, performs frustum clipping, occlusion culling, collects dynamic mesh information, creates light source information, etc.
    - FPrimitiveSceneInfo::UpdateStaticMeshes: Updates static mesh data.
    - ViewState::GetPrecomputedVisibilityData: Gets precomputed visibility data.
    - FrustumCull: Frustum culling.
    - **ComputeAndMarkRelevanceForViewParallel**:Computes and marks relevance data for view parallel processing.
    - **GatherDynamicMeshElements**:Collects the dynamic visible elements of the view, which has been analyzed in the previous article.
    - **SetupMeshPass**:Sets the data of the mesh Pass and converts FMeshBatch into FMeshDrawCommand, which has been analyzed in the previous article.

- CreateIndirectCapsuleShadows: Create capsule shadows. UpdateSkyIrradianceGpuBuffer:
- Updates the GPU data of sky volume environment lighting. InitSkyAtmosphereForViews:
- Initializes the atmospheric effect.
- **PostVisibilityFrameSetup**:Post-processing stage of visibility determination, using the view cone to clip light sources, process decal sorting, adjust the RT and fog effects, light beams, etc. of the previous frame.
- View.InitRHIResources: Initializes some RHI resources of the view.
- SetupVolumetricFog: Initializes and sets up volumetric fog.
- OnStartRender: Notify RHI that rendering has been started to initialize view-related data and resources.

The steps in bold above are the ones that I think are more important and need to be paid attention to.

4.3.5 PrePass

PrePass is also called early depth pass, depth only pass, early-Z pass, and is used to render the depth of opaque objects. This pass only writes depth but not color. There are three modes for writing depth: disabled, occlusion only, and complete depths, depending on different platforms and feature levels.

PrePass can be initiated by DBuffer or triggered by Forward Shading. It is usually used to establish Hierarchical-Z so that the hardware Early-Z technology can be enabled. It can also be used for occlusion culling to improve the rendering efficiency of Base Pass.

Next, let's get into the topic and analyze its code. First, let's look at PrePass FDeferredShadingSceneRenderer::Render:

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

	Scene->UpdateAllPrimitiveSceneInfos(RHICmdList,true);

	(.....)

	InitViews(...);

	(.....)

	UpdateGPUScene(RHICmdList,	* Scene);

	(.....)

	//Determine whether it is necessaryPrePass.
	const boolbNeedsPrePass = NeedsPrePass(this);

	// The Z-prepass

	(.....)

	if(bNeedsPrePass)
	{
		//Draw scene depth, build depth buffer and layersZbuffer(HiZ).
		bDepthWasCleared = RenderPrePass(RHICmdList, AfterTasksAreStarted);
	}

	(.....)

	// Z-Prepass End
}
```

As can be seen from the above code, Prepass isUpdateGPUSceneexecuted later and is not necessarily executed, which is determined by the following conditions:

```
staticFORCEINLINEbool NeedsPrePass(constFDeferredShadingSceneRenderer* Renderer) {

	return(RHIHasTiledGPU(Renderer->ViewFamily.GetShaderPlatform()) ==false) &&
		(Renderer->EarlyZPassMode != DDM_None || Renderer->bEarlyZPassMovable !=0);
}
```

To enable PrePass, the following two conditions must be met:

- Non-hardware Tiled GPUs. Modern mobile GPUs usually come with Tiled built-in, and use a TBDR architecture. Early-Z has been done at the GPU layer, so there is no need for explicit drawing.

- A valid EarlyZPassMode is specified or the renderer's bEarlyZPassMovable is non-zero.

Then enterRenderPrePassthe analysis of the main logic:

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

bool FDeferredShadingSceneRenderer::RenderPrePass(FRHICommandListImmediate& RHICmdList, TFunctionRef<
void()> AfterTasksAreStarted) {

	boolbDepthWasCleared =false;

	(.....)

	boolbDidPrePre =false;
	FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

	boolbParallel = GRHICommandList.UseParallelAlgorithms() &&
	CVarParallelPrePass.GetValueOnRenderThread();

	//	Non-parallel mode.
	if	(!bParallel)
	{
		AfterTasksAreStarted();
		bDepthWasCleared =	PreRenderPrePass(RHICmdList);
		bDidPrePre =true;
		SceneContext.BeginRenderingPrePass(RHICmdList,		false);
	}
	else	//Parallel Mode
	{
		//	Allocate a depth buffer.
		SceneContext.GetSceneDepthSurface();
	}

	// Draw a depth pass to avoid overdraw in the other passes. if(EarlyZPassMode !
	= DDM_None) {

		const boolbWaitForTasks = bParallel &&
(CVarRHICmdFlushRenderThreadTasksPrePass.GetValueOnRenderThread() >0||
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0);
		FScopedCommandListWaitForTasksFlusher(bWaitForTasks, RHICmdList);

		//EachviewDraw the depth buffer once.
		for(int32 ViewIndex =0;ViewIndex < Views.Num();ViewIndex++) {

			constFViewInfo& View = Views[ViewIndex];

			//Create and set upUniform Buffer. TUniformBufferRef<FSceneTexturesUniformParameters>
			PassUniformBuffer; CreateDepthPassUniformBuffer(RHICmdList, View, PassUniformBuffer);


			//Handles rendering state.
			FMeshPassProcessorRenderStateDrawRenderState(View, PassUniformBuffer);

			SetupDepthPassState(DrawRenderState);

			if(View.ShouldRenderView()) {

				Scene->UniformBuffers.UpdateViewUniformBuffer(View);

				//	Parallel Mode
				if	(bParallel)
```

```
                {
                    check(RHICmdList.IsOutsideRenderPass());
                    bDepthWasCleared = RenderPrePassViewParallel(View, RHICmdList,
DrawRenderState, AfterTasksAreStarted, !bDidPrePre) || bDepthWasCleared;
                    bDidPrePre =true;
                }
                else
                {
                    RenderPrePassView(RHICmdList, View, DrawRenderState);
                }
            }

            // Parallel rendering has self contained renderpasses so we need a new one for primitives.
editor
            if(bParallel)
            {
                SceneContext.BeginRenderingPrePass(RHICmdList,          false);
            }
            RenderPrePassEditorPrimitives(RHICmdList,                   View, DrawRenderState,
EarlyZPassMode,     true);
            if   (bParallel)
            {
                RHICmdList.EndRenderPass();
            }
        }
    }

    (......)

    if(bParallel)
    {
        // In parallel mode there will be no renderpass here. Need to restart.
        SceneContext.BeginRenderingPrePass(RHICmdList,false);
    }

    (......)

    SceneContext.FinishRenderingPrePass(RHICmdList);

    returnbDepthWasCleared;
}
```

The drawing process of PrePass is similar to the FMeshProcessor and Pass drawing analyzed in the previous article, so I will not repeat the analysis here. However, here we can focus on the rendering status of PrePass:

```
void SetupDepthPassState(FMeshPassProcessorRenderState& DrawRenderState) {

    //Disables writing of colors, enables depth testing and writing, depth comparison function is
    closer or equal. DrawRenderState.SetBlendState(TStaticBlendState<CW_NONE>::GetRHI());
    DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<true,
    CF_DepthNearOrEqual>::GetRHI());
}
```

In addition, when drawing depth, since there is no need to write color, the material used when rendering the object should definitely not be the material of the object itself, but some simple material. To verify the conjecture, take a look atFDepthPassMeshProcessorthe material used by Depth Pass:

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

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

{
	(……)

	if(bDraw)
	{
		(……)

		//GetSurfaceThe default material for the material domain, used as depthPass
		Rendering material. constFMaterialRenderProxy& DefaultProxy =
* UMaterial::GetDefaultMaterial(MD_Surface)->GetRenderProxy();
		constFMaterial& DefaultMaterial = *DefaultProxy.GetMaterial(FeatureLevel); Process<true
		>(MeshBatch, BatchElementMask, StaticMeshId, BlendMode, PrimitiveSceneProxy, DefaultProxy,
DefaultMaterial, MeshFillMode, MeshCullMode);

		(……)
	}
}
```

Pay attention to the key code above, you can see that the material used by the depth pass is UMaterial::GetDefaultMaterial(MD_Surface)obtained from , and continue to track it:

```
// Engine\Source\Runtime\Engine\Private\Materials\Material.cpp

UMaterial*UMaterial::GetDefaultMaterial(EMaterialDomain Domain) {

	InitDefaultMaterials();

	UMaterial* Default = GDefaultMaterials[Domain]; returnDefault;

}

void UMaterialInterface::InitDefaultMaterials() {

	static boolbInitialized =false; if(!bInitialized)

	{
		(……)

		for(int32 Domain =0; Domain < MD_MAX; ++Domain) {

			if(GDefaultMaterials[Domain] == nullptr) {

				FString ResolvedPath =
ResolveIniObjectsReference(GDefaultMaterialNames[Domain]);
```

```
                        GDefaultMaterials[Domain] = FindObject<UMaterial>(nullptr, *ResolvedPath); if
                        (GDefaultMaterials[Domain] == nullptr) {

                            GDefaultMaterials[Domain] = LoadObject<UMaterial>(nullptr,
* ResolvedPath, nullptr, LOAD_DisableDependencyPreloading, nullptr);
                        }
                        if(GDefaultMaterials[Domain]) {

                            GDefaultMaterials[Domain]->AddToRoot();
                        }
                    }
                }

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

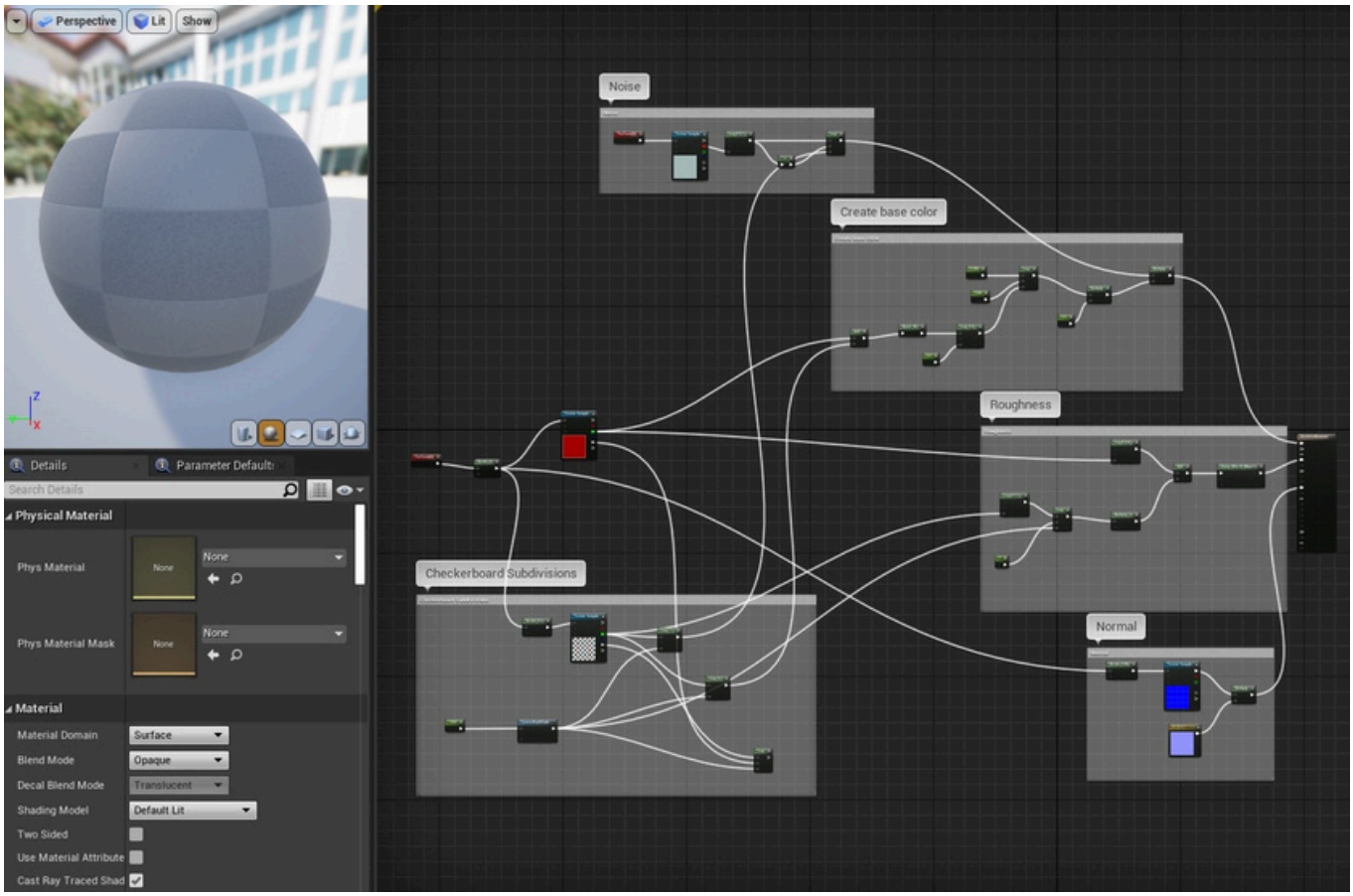
As can be seen above, the default material of the material system isGDefaultMaterialNamesindicated by going to its declaration:

```
static constTCHAR* GDefaultMaterialNames[MD_MAX] = {

    //Surface
    TEXT("engine-ini:/Script/Engine.Engine.DefaultMaterialName"), // Deferred Decal

    TEXT("engine-ini:/Script/Engine.Engine.DefaultDeferredDecalMaterialName"), // Light Function

    TEXT("engine-ini:/Script/Engine.Engine.DefaultLightFunctionMaterialName"), // Volume

    //@todo - get a real MD_Volume default material
    TEXT("engine-ini:/Script/Engine.Engine.DefaultMaterialName"), // Post Process

    TEXT("engine-ini:/Script/Engine.Engine.DefaultPostProcessMaterialName"), // User Interface

    TEXT("engine-ini:/Script/Engine.Engine.DefaultMaterialName"), //Virtual Texture

    TEXT("engine-ini:/Script/Engine.Engine.DefaultMaterialName"),
};
```

It turns out that it can be specified in the engine.ini configuration file. I was lazy and directly got the final material path and name of the depth pass through VS breakpoint debugging:

```
ResolvedPath =L"/Engine/EngineMaterials/WorldGridMaterial.WorldGridMaterial"
```

Through the UE editor, find WorldGridMaterial and open it:



People familiar with UE will be surprised: Isn't this the picture when the material is wrong? Yes, the depth pass uses this material, but the color is not written, so people don't notice its existence.

It is worth mentioning that the Shading Model used by WorldGridMaterial is Default Lit, and there are also redundant nodes in the material. If you want to make extreme optimizations, it is recommended to change this material in the configuration file, delete the redundant material nodes ,

and change to Unlit mode to minimize shader instructions and improve rendering efficiency.

It is also worth mentioning that you can specify the depth drawing mode when drawing the depth pass:

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

enumEDepthDrawingMode
{
	//Do not draw depth
	DDM_None 			= 0,
	//Draw onlyOpaqueMaterial (excludingMaskedMaterial
	DDM_NonMaskedOnly 		=1,
	// OpaqueandMaskedMaterial, But not including closedbUseAsOccluderof objects.
	DDM_AllOccluders = 		2,
	//All opaque object mode, all objects need to be drawn, and every pixel must matchBase PassDepth.
	DDM_AllOpaque 			=3,
	//onlyMaskedmodel.

	DDM_MaskedOnly = 4,
};
```

How to determine the depth drawing mode is determined by the following interface:

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

void FScene::UpdateEarlyZPassMode() {

    DefaultBasePassDepthStencilAccess = FExclusiveDepthStencil::DepthWrite_StencilWrite; EarlyZPassMode =
    DDM_NonMaskedOnly;//By default only drawOpaqueMaterial. bEarlyZPassMovable =false;



    //Depth strategy for deferred rendering pipeline
    if(GetShadingPath(GetFeatureLevel()) == EShadingPath::Deferred) {

        //Overridden by command line, or specified in project settings.
        {
            constint32 CVarValue = CVarEarlyZPass.GetValueOnAnyThread();

                switch(CVarValue)
                {
                case0:EarlyZPassMode = DDM_None;break; case1: EarlyZPassMode
                = DDM_NonMaskedOnly;break; case2: EarlyZPassMode =
                DDM_AllOccluders;break;
                case3:break;            // Note: 3 indicates "default behavior" and does not override
specify     an
                }
        }

        constEShaderPlatform ShaderPlatform =
        GetFeatureLevelShaderPlatform(FeatureLevel);
        if(ShouldForceFullDepthPass(ShaderPlatform)) {

            // DBufferStickers and TemplatesLODDither forces
            all modes. EarlyZPassMode = DDM_AllOpaque;
            bEarlyZPassMovable =true;
        }

        if(EarlyZPassMode == DDM_AllOpaque
            && CVarBasePassWriteDepthEvenWithFullPrepass.GetValueOnAnyThread() ==0)
        {
            DefaultBasePassDepthStencilAccess =
FExclusiveDepthStencil::DepthRead_StencilWrite;
        }
    }

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

4.3.6 BasePass

UE's BasePass is the geometry channel in deferred rendering, which is used to render the geometry information of opaque objects, including normals, depth, color, AO, roughness, metalness, etc. These geometry information will be written into several GBuffers.

The entrance of BasePassFDeferredShadingSceneRenderer::Renderand its adjacent important interfaces are as follows:

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

    (……)

    //Draws scene depth.
    bDepthWasCleared = RenderPrePass(RHICmdList, AfterTasksAreStarted);

    (……)

    //Depth texture resolution (only if enabledMSAAwill be executed).
    AddResolveSceneDepthPass(GraphBuilder, Views, SceneDepthTexture);

    //Light source gridding, commonly used inClusterQuickly remove light sources in lighting calculations. (The specific process will be analyzed in the subsequent
    rendering optimization topic) FSortedLightSetSceneInfo SortedLightSet; {

        GatherLightsAndComputeLightGrid(GraphBuilder, bComputeLightGrid, SortedLightSet);
    }

    (……)

    //Assignment of scenesGBuffer,For storageBase PassThe geometric information of . {

        SceneContext.PreallocGBufferTargets();
        SceneContext.AllocGBufferTargets(RHICmdList);
    }

    (……)

    //RenderingBase Pass.
    RenderBasePass(RHICmdList, BasePassDepthStencilAccess,
ForwardScreenSpaceShadowMask.GetReference(), bDoParallelBasePass,                bRenderLightmapDensity);

    (……)
}
```

Direct access below`RenderBasePass`:

```
bool FDeferredShadingSceneRenderer::RenderBasePass(FRHICommandListImmediate& RHICmdList,
FExclusiveDepthStencil::Type BasePassDepthStencilAccess, IPooledRenderTarget* ForwardScreenSpaceShadowMask,
boolbParallelBasePass,boolbRenderLightmapDensity) {

    (……)

    {
        FExclusiveDepthStencil::Type BasePassDepthStencilAccess_NoDepthWrite                =
FExclusiveDepthStencil::Type(BasePassDepthStencilAccess ~FExclusiveDepth&Stencil::DepthWrite);

        //  Parallel Mode
        if  (bParallelBasePass)
        {
            FScopedCommandListWaitForTasks
Flusher(CVarRHICmdFlushRenderThreadTasksBasePass.GetValueOnRenderThread() >0 | |
CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0, RHICmdList);
            //Traverse allviewRenderingBase Pass
            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {
```

```
                FViewInfo& View = Views[ViewIndex];

                // Uniform Buffer
                TUniformBufferRef<FOpaqueBasePassUniformParameters>          BasePassUniformBuffer;
                CreateOpaqueBasePassUniformBuffer(RHICmdList, View,
ForwardScreenSpaceShadowMask, nullptr, nullptr, nullptr, BasePassUniformBuffer);

                // Render State
                FMeshPassProcessorRenderStateDrawRenderState(View,
BasePassUniformBuffer);

                SetupBasePassState(BasePassDepthStencilAccess,
ViewFamily.EngineShowFlags.ShaderComplexity, DrawRenderState);

                const boolbShouldRenderView = View.ShouldRenderView(); if
                (bShouldRenderView)
                {
                    Scene->UniformBuffers.UpdateViewUniformBuffer(View);

                    //Perform rendering.
                    RenderBasePassViewParallel(View,          RHICmdList,
BasePassDepthStencilAccess, DrawRenderState);
                }

                FSceneRenderTargets::Get(RHICmdList).BeginRenderingGBuffer(RHICmdList,
ERenderTargetLoadAction::ELoad, ERenderTargetLoadAction::ELoad, BasePassDepthStencilAccess, this-
>ViewFamily.EngineShowFlags.ShaderComplexity);
                RenderEditorPrimitives(RHICmdList, View, BasePassDepthStencilAccess,
DrawRenderState, bDirty);
                RHICmdList.EndRenderPass();

                (......)
            }

            bDirty =true;// assume dirty since we are not going to wait
        }
        else    //Non-parallel mode
        {
            (......)
        }
    }

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

The rendering logic of Base Pass is very similar to that of Pre Pass, so we will not go into detail. Next, we will focus on the rendering state and materials used when rendering Base Pass. The following is the rendering state:

```
void SetupBasePassState(FExclusiveDepthStencil::Type BasePassDepthStencilAccess,const boolbShaderComplexity,
FMeshPassProcessorRenderState& DrawRenderState) {

    DrawRenderState.SetDepthStencilAccess(BasePassDepthStencilAccess);

    (......)
```

```
    {
        //allGBufferMixing is turned on. static constautoCVar =
        IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.BasePassOutputsVelocityDebug")) ;


        if(CVar && CVar->GetValueOnRenderThread() ==2) {

            DrawRenderState.SetBlendState(TStaticBlendStateWriteMask<CW_RGBA,                CW_RGBA,
CW_RGBA, CW_RGBA, CW_RGBA, CW_RGBA, CW_NONE>::GetRHI());
        }
        else
        {
            DrawRenderState.SetBlendState(TStaticBlendStateWriteMask<CW_RGBA,                CW_RGBA,
CW_RGBA,    CW_RGBA>::GetRHI());
        }

        //Deep writing and testing are enabled, and the comparison function isNearOrEqual.
        if(DrawRenderState.GetDepthStencilAccess() & FExclusiveDepthStencil::DepthWrite) {

            DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<true,
CF_DepthNearOrEqual>::GetRHI());
        }
        else
        {
            DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<false,
CF_DepthNearOrEqual>::GetRHI());
        }
    }
}
```

Let's takeFBasePassMeshProcessora look at the materials used in the Base Pass:

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

{
    if(MeshBatch.bUseForMaterial) {

        constFMaterialRenderProxy* FallbackMaterialRenderProxyPtr = nullptr; constFMaterial&
        Material = MeshBatch.MaterialRenderProxy-
> GetMaterialWithFallback(FeatureLevel, FallbackMaterialRenderProxyPtr);

    (.....)
}
```

From this, we can see that Base Pass normally uses the material collected by FMeshBatch, that is, the material of the mesh itself. The only difference is that the lighting calculation is not enabled in the shader, similar to the Unlit mode. To summarize the drawing nesting relationship of BasePass, the pseudo code is as follows:

```
foreach(scene in      scenes)
{
    foreach(view    in   views)
    {
        foreach(mesh    in meshes)
```

```
            {
                DrawMesh(...)       //    Executed once per render callBasePassVertexShaderandBasePassPixelShaderof
Code.
            }
        }
}
```

BasePassVertexShaderAndBasePassPixelShaderis the specific shader logic, which is not involved in this article and will be explained in detail in the next article.

4.3.7 LightingPass

UE's LightingPass is the lighting channel mentioned in the previous chapter. This stage calculates the shadow map of the light source with shadows turned on, and also calculates the contribution of each light to the screen space pixel and accumulates it in the Scene Color. In addition , it also calculates the contribution of the light source to the translucency lighting volumes.

The entrance of Lighting PassFDeferredShadingSceneRenderer::Renderand its adjacent important interfaces are as follows:

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

    (......)

    //RenderingBase Pass.
    RenderBasePass(RHICmdList, BasePassDepthStencilAccess,
ForwardScreenSpaceShadowMask.GetReference(), bDoParallelBasePass,                bRenderLightmapDensity);

    (......)

    SceneContext.FinishGBufferPassAndResolve(RHICmdList,                BasePassDepthStencilAccess);

    (......)

    //    Rendering lighting for the deferred pipeline.
    if    (bRenderDeferredLighting)
    {
        //Rendering indirect diffuse andAO.
        RenderDiffuseIndirectAndAmbientOcclusion(RHICmdList);

        //Renders indirect capsule shadows. Will modify fromBase PassThe output scene color with indirect lighting
        added. RenderIndirectCapsuleShadows(
            RHICmdList,
            SceneContext.GetSceneColorSurface(), SceneContext.bScreenSpaceAOIsValid ?
            SceneContext.ScreenSpaceAO-
> GetRenderTargetItem().TargetableTexture:NULL);

        TRefCountPtr<IPooledRenderTarget> DynamicBentNormalAO; //Rendering
        Distance FieldsAO.
        RenderDFAOAsIndirectShadowing(RHICmdList, SceneContext.SceneVelocity,
        DynamicBentNormalAO);

        //Clean up transparent lighting voxels before lighting overlay.
```

```
        if((GbEnableAsyncComputeTranslucencyLightingVolumeClear &&
GSupportsEfficientAsyncCompute) ==false)
        {
            for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex) {

                FViewInfo& View = Views[ViewIndex];
                ClearTranslucentVolumeLighting(RHICmdList, ViewIndex);
            }
        }

        //Start rendering the light source.
        {
            RenderLights(RHICmdList, SortedLightSet, HairDatas);
        }

        //Added transparent voxel lighting for environment cubemap.
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex) {

            FViewInfo& View = Views[ViewIndex]; InjectAmbientCubemapTranslucentVolumeLighting(RHICmdList,
            Views[ViewIndex],
ViewIndex);
        }

        //Filter transparent voxel lighting.
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex) {

            FViewInfo& View = Views[ViewIndex];

            FilterTranslucentVolumeLighting(RHICmdList, View, ViewIndex);
        }

        //Light combination preparation stage, such asLPV (Light Propagation Volumes).
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex) {

            FViewInfo& View = Views[ViewIndex];

            if(IsLpvIndirectPassRequired(View)) {

                GCompositionLighting.ProcessLpvIndirect(RHICmdList,                View);
            }
        }

        //Renders diffuse skylight reflections and reflections only on opaque
        objects. RenderDeferredReflectionsAndSkyLighting(RHICmdList,          DynamicBentNormalAO,
SceneContext.SceneVelocity, HairDatas);

        DynamicBentNormalAO =NULL;

        //Analyze the scene color, because the followingSSSNeed scene colorRT
        becomeSRV. ResolveSceneColor(RHICmdList);

        //calculateSSSEffect.
        ComputeSubsurfaceShim(RHICmdList, Views);

        (.....)
    }
```

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

The rendering logic of Lighting Pass is complex and includes indirect shadows, indirect AO, transparent volumetric lighting, light source calculation, LPV, sky light, SSS, etc. However, the core logic of lighting calculation is in the following codeRenderLights,which will be focused on below.
Other parts will not be covered in this article.RenderLights

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

void FDeferredShadingSceneRenderer::RenderLights(FRHICommandListImmediate& RHICmdList,
FSortedLightSetSceneInfo &SortedLightSet,constFHairStrandsDatas* HairDatas) {

    (.....)

    boolbStencilBufferDirty =false; already              // The stencil buffer should've been cleared to 0


    constFSimpleLightArray &SimpleLights = SortedLightSet.SimpleLights; const
    TArray<FSortedLightSceneInfo, SceneRenderingAllocator> &SortedLights =
    SortedLightSet.SortedLights;
    //The starting index of the light source with shadows.
    constint32 AttenuationLightStart = SortedLightSet.AttenuationLightStart; constint32
    SimpleLightsEnd = SortedLightSet.SimpleLightsEnd;


    //Direct Lighting
    {
        SCOPED_DRAW_EVENT(RHICmdList, DirectLighting);

        FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

        (.....)

        //No shadow lighting
        if(ViewFamily.EngineShowFlags.DirectLighting) {

            SCOPED_DRAW_EVENT(RHICmdList, NonShadowLights);

            //The starting index of the light source without shadows.
            int32 StandardDeferredStart = SortedLightSet.SimpleLightsEnd; bool
            bRenderSimpleLightsStandardDeferred =
SortedLightSet.SimpleLights.InstanceData.Num() >0;

            //Clustered delayed lighting.
            if(ShouldUseClusteredDeferredShading() && AreClusteredLightsInLightGrid()) {

                StandardDeferredStart = SortedLightSet.ClusteredSupportedEnd;
                bRenderSimpleLightsStandardDeferred =false; //Added clustered delayed
                renderingPass.
                AddClusteredDeferredShadingPass(RHICmdList, SortedLightSet);
            }
            //Tiled deferred lighting.
            else if(CanUseTiledDeferred()) {

                (.....)
```

```
                    if    (ShouldUseTiledDeferred(SortedLightSet.TiledSupportedEnd) &&
!bAnyViewIsStereo)
                    {
StandardDeferredStart = SortedLightSet.TiledSupportedEnd;
bRenderSimpleLightsStandardDeferred =false; //Rendering tiled deferred
lighting.
RenderTiledDeferredLighting(RHICmdList, SortedLights,
SortedLightSet.SimpleLightsEnd, SortedLightSet.TiledSupportedEnd, SimpleLights);
                    }
                }

                //    Simple lighting.
                if    (bRenderSimpleLightsStandardDeferred)
                {
                    SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite);
                    //Rendering simple lighting.
                    RenderSimpleLightsStandardDeferred(RHICmdList,
SortedLightSet.SimpleLights);
                    SceneContext.FinishRenderingSceneColor(RHICmdList);
                }

                //    Standard deferred lighting.
                if    (!bUseHairLighting)
                {
                    SCOPED_DRAW_EVENT(RHICmdList,       StandardDeferredLighting);

                    SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite,true);

                    for(int32 LightIndex = StandardDeferredStart; LightIndex <
AttenuationLightStart; LightIndex++)
                    {
                        constFSortedLightSceneInfo& SortedLightInfo =
SortedLights[LightIndex];
                        constFLightSceneInfo*constLightSceneInfo =
SortedLightInfo.LightSceneInfo;

                        //Renders shadowless lighting.
                        RenderLight(RHICmdList, LightSceneInfo, nullptr, nullptr,false,
false);
                    }

                    SceneContext.FinishRenderingSceneColor(RHICmdList);
                }

                (......)
            }

            (......)

            //Lighting with shadows
            {
                SCOPED_DRAW_EVENT(RHICmdList, ShadowedLights);

                constint32 DenoiserMode = CVarShadowUseDenoiser.GetValueOnRenderThread();
```

```
            constIScreenSpaceDenoiser* DefaultDenoiser =
IScreenSpaceDenoiser::GetDefaultDenoiser();
            constIScreenSpaceDenoiser* DenoiserToUse = DenoiserMode ==1?
DefaultDenoiser : GScreenSpaceDenoiser;

            TArray<TRefCountPtr<IPooledRenderTarget>>            PreprocessedShadowMaskTextures;
            TArray<TRefCountPtr<IPooledRenderTarget>>
PreprocessedShadowMaskSubPixelTextures;

            constint32 MaxDenoisingBatchSize =
FMath::Clamp(CVarMaxShadowDenoisingBatchSize.GetValueOnRenderThread(),                1,
IScreenSpaceDenoiser::kMaxBatchSize);
            constint32 MaxRTShadowBatchSize =
CVarMaxShadowRayTracingBatchSize.GetValueOnRenderThread();
            const boolbDoShadowDenoisingBatching = DenoiserMode !=0&&
MaxDenoisingBatchSize >1;

            (.....)

            boolbDirectLighting = ViewFamily.EngineShowFlags.DirectLighting; bool
            bShadowMaskReadable =false; TRefCountPtr<IPooledRenderTarget>
            TRefCountPtr<IPooledRenderTarget>          ScreenShadowMaskTexture;
                                                       ScreenShadowMaskSubPixelTexture;

            //Renders shadowed lights and light function lights.
            for(int32 LightIndex = AttenuationLightStart; LightIndex <
SortedLights.Num(); LightIndex++)
            {
                constFSortedLightSceneInfo& SortedLightInfo = SortedLights[LightIndex]; constFLightSceneInfo&
                LightSceneInfo = *SortedLightInfo.LightSceneInfo;

                // Note: Skip shadow mask generation for rect light if direct illumination
is    computed
                //                stochastically (rather than analytically + shadow mask)
                const boolbDrawShadows = SortedLightInfo.SortKey.Fields.bShadowed &&
!ShouldRenderRayTracingStochasticRectLight(LightSceneInfo);
                boolbDrawLightFunction = SortedLightInfo.SortKey.Fields.bLightFunction; bool
                bDrawPreviewIndicator =
ViewFamily.EngineShowFlags.PreviewShadowsIndicator && !
LightSceneInfo.IsPrecomputedLightingValid() && LightSceneInfo.Proxy-
> HasStaticShadowing();
                boolbInjectedTranslucentVolume =false; bool
                bUsedShadowMaskTexture =false;
                const boolbDrawHairShadow = bDrawShadows && bUseHairLighting; const bool
                bUseHairDeepShadow = bDrawShadows && bUseHairLighting &&
LightSceneInfo.Proxy->CastsHairStrandsDeepShadow();

                FScopeCycleCounterContext(LightSceneInfo.Proxy->GetStatId());

                if((bDrawShadows || bDrawLightFunction || bDrawPreviewIndicator) &&
!ScreenShadowMaskTexture.IsValid())
                {
                    SceneContext.AllocateScreenShadowMask(RHICmdList,
ScreenShadowMaskTexture);
                    bShadowMaskReadable =false; if
                    (bUseHairLighting)
                    {
```

```
                        SceneContext.AllocateScreenShadowMask(RHICmdList,
ScreenShadowMaskSubPixelTexture,true);
                    }
                }

                FString LightNameWithLevel;
                GetLightNameForDrawEvent(LightSceneInfo.Proxy,          LightNameWithLevel);
                SCOPED_DRAW_EVENTF(RHICmdList, EventLightPass,          * LightNameWithLevel);

                if(bDrawShadows)
                {
                    INC_DWORD_STAT(STAT_NumShadowedLights);

                    constFLightOcclusionType OcclusionType =
GetLightOcclusionType(*LightSceneInfo.Proxy);

                    (......)

                    //Processing shadow maps.
                    else// (OcclusionType == FOcclusionType::Shadowmap) {

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

                            constFViewInfo& View = Views[ViewIndex];
                            View.HeightfieldLightingViewInfo.ClearShadowing(View,
RHICmdList,     LightSceneInfo);
                        }

                        //Clean up shadow map.
                        autoClearShadowMask = [&](TRefCountPtr<IPooledRenderTarget>&
InScreenShadowMaskTexture)
                        {
                            // Clear light attenuation for local lights with a quad
covering their extents
                            const boolbClearLightScreenExtentsOnly =
CVarAllowClearLightSceneExtentsOnly.GetValueOnRenderThread() &&
SortedLightInfo.SortKey.Fields.LightType != LightType_Directional;
                            // All shadows render with min blending
                            boolbClearToWhite = !bClearLightScreenExtentsOnly;

                            FRHIRenderPassInfoRPInfo(InScreenShadowMaskTexture-
> GetRenderTargetItem().TargetableTexture, ERenderTargetActions::Load_Store);
                            RPInfo.DepthStencilRenderTarget.Action =
MakeDepthStencilTargetActions(ERenderTargetActions::Load_DontStore,
ERenderTargetActions::Load_Store);
                            RPInfo.DepthStencilRenderTarget.DepthStencilTarget =
SceneContext.GetSceneDepthSurface();
                            RPInfo.DepthStencilRenderTarget.ExclusiveDepthStencil                =
FExclusiveDepthStencil::DepthRead_StencilWrite;
                            if(bClearToWhite)
                            {
                                RPInfo.ColorRenderTargets[0].Action =
ERenderTargetActions::Clear_Store;
                            }

                            TransitionRenderPassTargets(RHICmdList, RPInfo);
                            RHICmdList.BeginRenderPass(RPInfo,
TEXT("ClearScreenShadowMask"));
```

```
                                if(bClearLightScreenExtentsOnly) {

                                    SCOPED_DRAW_EVENT(RHICmdList, ClearQuad);

                                    for(int32 ViewIndex =0; ViewIndex < Views.Num();
ViewIndex++)
                                    {
                                        constFViewInfo& View = Views[ViewIndex]; FIntRect
                                        ScissorRect;

                                        if(!LightSceneInfo.Proxy->GetScissorRect(ScissorRect,
View,    View.ViewRect)
                                        {
                                            ScissorRect = View.ViewRect;
                                        }

                                        if(ScissorRect.Min.X < ScissorRect.Max.X &&
ScissorRect.Min.Y        <  ScissorRect.Max.Y)
                                        {
                                            RHICmdList.SetViewport(ScissorRect.Min.X,
ScissorRect.Min.Y,0.0f, ScissorRect.Max.X, ScissorRect.Max.Y,1.0f);
                                            DrawClearQuad(RHICmdList,true, FLinearColor(1,1,
1,1),false,0,false,0);
                                        }
                                        else
                                        {
                                            LightSceneInfo.Proxy->GetScissorRect(ScissorRect,
View, View.ViewRect);
                                        }
                                    }
                                }
                                RHICmdList.EndRenderPass();
                            };

                            ClearShadowMask(ScreenShadowMaskTexture); if
                            (ScreenShadowMaskSubPixelTexture) {

                                ClearShadowMask(ScreenShadowMaskSubPixelTexture);
                            }

                            RenderShadowProjections(RHICmdList, &LightSceneInfo,
ScreenShadowMaskTexture, ScreenShadowMaskSubPixelTexture, HairDatas,
bInjectedTranslucentVolume);
                        }

                        bUsedShadowMaskTexture =true;
                    }

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

                        constFViewInfo& View = Views[ViewIndex];
                        View.HeightfieldLightingViewInfo.ComputeLighting(View, RHICmdList,
LightSceneInfo);
                    }

                    // Processing lighting functions (light
                    if function). (bDirectLighting)
                    {
```

```
                if(bDrawLightFunction) {

                    const boolbLightFunctionRendered =
RenderLightFunction(RHICmdList, &LightSceneInfo, ScreenShadowMaskTexture, bDrawShadows, false);

                    bUsedShadowMaskTexture |= bLightFunctionRendered;
                }

                (......)
            }

            if(bUsedShadowMaskTexture) {

                check(ScreenShadowMaskTexture);
                RHICmdList.CopyToResolveTarget(ScreenShadowMaskTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskTexture-
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                if(ScreenShadowMaskSubPixelTexture) {

                    RHICmdList.CopyToResolveTarget(ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().TargetableTexture, ScreenShadowMaskSubPixelTexture-
> GetRenderTargetItem().ShaderResourceTexture, FResolveParams(FResolveRect()));
                }

                if(!bShadowMaskReadable) {


RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskTexture->GetRenderTargetItem().ShaderResourceTexture);
                    if(ScreenShadowMaskSubPixelTexture) {


RHICmdList.TransitionResource(EResourceTransitionAccess::EReadable,
ScreenShadowMaskSubPixelTexture->GetRenderTargetItem().ShaderResourceTexture);
                    }
                    bShadowMaskReadable =true;
                }

                GVisualizeTexture.SetCheckPoint(RHICmdList, ScreenShadowMaskTexture); if
                (ScreenShadowMaskSubPixelTexture) {

                    GVisualizeTexture.SetCheckPoint(RHICmdList,
ScreenShadowMaskSubPixelTexture);
                }
            }

            (......)

            //Rendering standard deferred lighting.
            {
                SCOPED_DRAW_EVENT(RHICmdList, StandardDeferredLighting);
                SceneContext.BeginRenderingSceneColor(RHICmdList,
ESimpleRenderTargetMode::EExistingColorAndDepth,
FExclusiveDepthStencil::DepthRead_StencilWrite,true);

                // ScreenShadowMaskTexture might have been created for a previous
light, but only use it if we wrote valid data into it for this light
                IPooledRenderTarget* LightShadowMaskTexture = nullptr;
```

```
                    IPooledRenderTarget* LightShadowMaskSubPixelTexture = nullptr; if
                    (bUsedShadowMaskTexture) {

                        LightShadowMaskTexture = ScreenShadowMaskTexture;
                        LightShadowMaskSubPixelTexture = ScreenShadowMaskSubPixelTexture;
                    }

                    if  (bDirectLighting)
                    {
                        //Renders light sources with shadows.
                        RenderLight(RHICmdList, false,&LightSceneInfo,          LightShadowMaskTexture,
InHairVisibilityViews,          true);
                    }

                    SceneContext.FinishRenderingSceneColor(RHICmdList);

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

From the above code, we can see that when rendering light sources, the light sources without shadows will be drawn first, with the starting indexSortedLightSet.SimpleLightsEnddetermined by variables such as , and then the light sources with shadows will be drawn, with the starting index SortedLightSet.AttenuationLightStartdetermined by . Only light sources without shadows support Tiled and Clustered.

Regardless of whether there is a shadow or not, when rendering the light source, it will eventually call RenderLight(note that it is notRenderLights)to actually perform the lighting calculation of a single light source.RenderLightThe code is as follows:

```
void FDeferredShadingSceneRenderer::RenderLight(FRHICommandList& RHICmdList,const FLightSceneInfo*
LightSceneInfo, IPooledRenderTarget* ScreenShadowMaskTexture,const FHairStrandsVisibilityViews*
InHairVisibilityViews,boolbRenderOverlap,bool bIssueDrawEvent)

{
    FGraphicsPipelineStateInitializer GraphicsPSOInit;
    RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);

    //Set the blending state to Overlay so that the light intensity of all lights is added to the same texture.
    GraphicsPSOInit.BlendState = TStaticBlendState<CW_RGBA, BO_Add, BF_One, BF_One, BO_Add, BF_One,
    BF_One>::GetRHI();

    GraphicsPSOInit.PrimitiveType = PT_TriangleList;

    constFSphere LightBounds = LightSceneInfo->Proxy->GetBoundingSphere(); const bool
    bTransmission = LightSceneInfo->Proxy->Transmission();

    //Convenient for allview,Give this light source to eachviewDraw them all once.
    for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        FViewInfo& View = Views[ViewIndex];
```

```
// Ensure the light is valid for this view if(!LightSceneInfo-
>ShouldRenderLight(View)) {

        continue;
}

boolbUseIESTexture =false;

if(View.Family->EngineShowFlags.TexturedLightProfiles) {

        bUseIESTexture = (LightSceneInfo->Proxy->GetIESTextureResource() !=0);
}

// Set the device viewport for the view.
RHICmdList.SetViewport(View.ViewRect.Min.X,          View.ViewRect.Min.Y,          0.0f,
View.ViewRect.Max.X, View.ViewRect.Max.Y,1.0f);

(.....)

//Draws a parallel directional light.
if(LightSceneInfo->Proxy->GetLightType() == LightType_Directional) {

        // Turn DBT back off GraphicsPSOInit.bDepthBounds =false;
        TShaderMapRef<TDeferredLightVS<false> > VertexShader(View.ShaderMap);


        GraphicsPSOInit.RasterizerState =          TStaticRasterizerState<FM_Solid,
CM_None>::GetRHI();
        GraphicsPSOInit.DepthStencilState          = TStaticDepthStencilState<false,
CF_Always>::GetRHI();

        //  Set the delayed light sourceshaderandpso
        if  parameter. (bRenderOverlap)
        {
            TShaderMapRef<TDeferredLightOverlapPS<false>          >
PixelShader(View.ShaderMap);
            GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GFilterVertexDeclaration.VertexDeclarationRHI;
            GraphicsPSOInit.BoundShaderState.VertexShaderRHI           =
VertexShader.GetVertexShader();
            GraphicsPSOInit.BoundShaderState.PixelShaderRHI            =
PixelShader.GetPixelShader();
            SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
            >SetParameters(RHICmdList, View, LightSceneInfo);
        }
        else
        {
            const boolbAtmospherePerPixelTransmittance = LightSceneInfo->Proxy-
> IsUsedAsAtmosphereSunLight() && ShouldApplyAtmosphereLightPerPixelTransmittance(Scene, View.Family-
>EngineShowFlags);

            FDeferredLightPS::FPermutationDomain PermutationVector;
            PermutationVector.Set< FDeferredLightPS::FSourceShapeDim >(
ELightSourceShape::Directional );
            PermutationVector.Set< FDeferredLightPS::FIESProfileDim >(false); PermutationVector.Set<
            FDeferredLightPS::FInverseSquaredDim >(false); PermutationVector.Set<
            FDeferredLightPS::FVisualizeCullingDim >(
```

```
View.Family->EngineShowFlags.VisualizeLightCulling );
                PermutationVector.Set< FDeferredLightPS::FLightingChannelsDim >(
View.bUsesLightingChannels );
                PermutationVector.Set< FDeferredLightPS::FTransmissionDim >( bTransmission
) ;
                PermutationVector.Set< FDeferredLightPS::FHairLighting>(bHairLighting ?1
:0);
                // Only directional lights are rendered in this path, so we only need to
check if it is used to light the atmosphere
                PermutationVector.Set< FDeferredLightPS::FAtmosphereTransmittance >
(bAtmospherePerPixelTransmittance);

                TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );
                GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GFilterVertexDeclaration.VertexDeclarationRHI;
                GraphicsPSOInit.BoundShaderState.VertexShaderRHI                =
VertexShader.GetVertexShader();
                GraphicsPSOInit.BoundShaderState.PixelShaderRHI                =
PixelShader.GetPixelShader();

                SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                >SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, (bHairLighting) ? &RenderLightParams : nullptr);
        }

        VertexShader->SetParameters(RHICmdList, View, LightSceneInfo);

        //Since it is a parallel light, it will cover all areas of the screen space, so a full-screen rectangle is used for
        drawing. DrawRectangle(
            RHICmdList,
            0,0,
            View.ViewRect.Width(), View.ViewRect.Height(),
            View.ViewRect.Min.X, View.ViewRect.Min.Y,
            View.ViewRect.Width(), View.ViewRect.Height(), View.
            ViewRect.Size(),
            FSceneRenderTargets::Get(RHICmdList).GetBufferSizeXY(), VertexShader,

            EDRF_UseTriangleOptimization);
    }
    else    //  Local light source
    {
        //Whether to enable depth bounding box test (DBT).
        GraphicsPSOInit.bDepthBounds = GSupportsDepthBoundsTest &&
GAllowDepthBoundsTest !=0;

        TShaderMapRef<TDeferredLightVS<true> > VertexShader(View.ShaderMap);

        SetBoundingGeometryRasterizerAndDepthState(GraphicsPSOInit, View,
LightBounds);

        //  Set the delayed light sourceshaderandpso
        if  parameter. (bRenderOverlap)
        {
            TShaderMapRef<TDeferredLightOverlapPS<true>          >  PixelShader(View.ShaderMap);
            GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
GetVertexDeclarationFVector4();
            GraphicsPSOInit.BoundShaderState.VertexShaderRHI =
```

```
VertexShader.GetVertexShader();
                    GraphicsPSOInit.BoundShaderState.PixelShaderRHI                =
PixelShader.GetPixelShader();

                    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                    >SetParameters(RHICmdList, View, LightSceneInfo);
                }
                else
                {
                    FDeferredLightPS::FPermutationDomain PermutatPioenrVmeuctaotri.oSneVt<e ctor;
                    FDeferredLightPS::FSourceShapeDim                                    > (
LightSceneInfo->Proxy->IsRectLight() ? ELightSourceShape::Rect :
ELightSourceShape::Capsule );
                    PermutationVector.Set< FDeferredLightPS::FSourceTextureDim            > (
LightSceneInfo->Proxy->IsRectLight() && LightSceneInfo->Proxy->HasSourceTexture() );
                    PermutationVector.Set< FDeferredLightPS::FIESProfileDim >( bUseIESTexture
) ;
                    PermutationVector.Set< FDeferredLightPS::FInverseSquaredDim >(
LightSceneInfo->Proxy->IsInverseSquared() );
                    PermutationVector.Set< FDeferredLightPS::FVisualizeCullingDim          > (
View.Family->EngineShowFlags.VisualizeLightCulling );
                    PermutationVector.Set< FDeferredLightPS::FLightingChannelsDim          > (
View.bUsesLightingChannels );
                    PermutationVector.Set< FDeferredLightPS::FTransmissionDim >( bTransmission
) ;
                    PermutationVector.Set< FDeferredLightPS::FHairLighting>(bHairLighting ?1
:0);
                    PermutationVector.Set < FDeferredLightPS::FAtmosphereTransmittance >
(false);

                    TShaderMapRef< FDeferredLightPS >PixelShader( View.ShaderMap,
PermutationVector );
                    GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI          =
GetVertexDeclarationFVector4();
                    GraphicsPSOInit.BoundShaderState.VertexShaderRHI               =
VertexShader.GetVertexShader();
                    GraphicsPSOInit.BoundShaderState.PixelShaderRHI                =
PixelShader.GetPixelShader();

                    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); PixelShader-
                    >SetParameters(RHICmdList, View, LightSceneInfo,
ScreenShadowMaskTexture, (bHairLighting) ? &RenderLightParams : nullptr);
                }

                VertexShader->SetParameters(RHICmdList, View, LightSceneInfo);

                // Depth bounding box test (DBT)Only works in light sources with shadows, and can effectively cull pixels outside the
                if depth range. (GraphicsPSOInit.bDepthBounds)
                {
                    //UE4Using the inverse depth (reversed depth),sofar<near. floatNearDepth =1.f; floatFarDepth =0.f;
                    CalculateLightNearFarDepthFromBounds(View,LightBounds,NearDepth,FarDepth);



                    if(NearDepth <= FarDepth) {

                        NearDepth    =  1.0f;
                        FarDepth     =  0.0f;
```

```
                }


                //Set the depth bounding box parameters.
                RHICmdList.SetDepthBounds(FarDepth, NearDepth);
            }

            //Point or area lights are drawn using spheres.
            if(LightSceneInfo->Proxy->GetLightType() == LightType_Point ||
                LightSceneInfo->Proxy->GetLightType() == LightType_Rect )
            {
                StencilingGeometry::DrawSphere(RHICmdList);
            }
            //Spotlights are drawn using cones.
            else if(LightSceneInfo->Proxy->GetLightType() == LightType_Spot) {

                StencilingGeometry::DrawCone(RHICmdList);
            }
        }
    }
}
```

From the above, we can see that both parallel light sources and local light sources are drawn in screen space, and parallel light uses a rectangle covering the entire screen to draw the light, point light sources or area light sources use a sphere to draw the light source, and spotlights use a cone.
The advantage of this is that the pixels outside the light source bounding box can be quickly removed, which speeds up the efficiency of lighting calculation. To summarize the nested relationship in LightingPass, the pseudo code is as follows:

```
foreach(scene in scenes) {

    foreach(light in        lights)
    {
        foreach(view    in views)
        {
            RenderLight()    //Executed once each time a render light is calledDeferredLightVertexShadersand
DeferredLightPixelShadersThe code.
        }
    }
}
```

DeferredLightVertexShadersAndDeferredLightPixelShadersis the specific shader logic, which is not involved in this article and will be explained in detail in the next article.

4.3.8 Translucency

Translucency is the stage of rendering translucent objects. All translucent objects are drawn one by one from far to near in the view space into the off-screen render texture (separate translucent render target), and then a separate pass is used to correctly calculate and mix the lighting results.

The entrance of TranslucencyFDeferredShadingSceneRenderer::Renderand its adjacent important interfaces are as follows:

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

	(.....)

	RenderLights(RHICmdList, SortedLightSet, ...);

	(.....)

	RenderSkyAtmosphere(RHICmdList);

	(.....)

	RenderFog(RHICmdList, LightShaftOutput);

	(.....)

	//Draw semi-transparent objects.
	if(bHasAnyViewsAbovewater && bCanOverlayRayTracingOutput &&
ViewFamily.EngineShowFlags.Translucency &&
!ViewFamily.EngineShowFlags.VisualizeLightCulling && !ViewFamily.UseDebugViewPS())
	{
		(.....)

		{
			//Rendering semi-transparent objects.
			RenderTranslucency(RHICmdList);

			static constautoDisableDistortionCVar=
IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.DisableDistortion"));
			const boolbAllowDistortion = DisableDistortionCVar->GetValueOnAnyThread() !=
1;

			//Render the perturbation texture.
			if(GetRefractionQuality(ViewFamily) >0&& bAllowDistortion) {

				RenderDistortion(RHICmdList);
			}
		}

		// Velocity buffer for rendering semi-transparent objects.
		if (bShouldRenderVelocities)
		{
			RenderVelocities(RHICmdList, SceneContext.SceneVelocity,
EVelocityPass::Translucent, bClearVelocityRT);
		}
	}

	(.....)
}
```

The semi-transparent stage will render semi-transparent colors, disturbance textures (for effects such as refraction), and velocity buffers (for TAA anti-aliasing and post-processing effects). The most important logic for rendering semi-transparent isRenderTranslucency:

```
// Source\Runtime\Renderer\Private\TranslucentRendering.cpp

void FDeferredShadingSceneRenderer::RenderTranslucency(FRHICommandListImmediate& RHICmdList,bool
bDrawUnderwaterViews) {

    TRefCountPtr<IPooledRenderTarget> SceneColorCopy; if(!
    bDrawUnderwaterViews) {

        ConditionalResolveSceneColorForTranslucentMaterials(RHICmdList, SceneColorCopy);
    }

    // Disable UAV cache flushing so we have optimal VT feedback performance.
    RHICmdList.BeginUAVOverlap();

    //  Render translucent objects after depth of field.
    if  (ViewFamily.AllowTranslucencyAfterDOF())
    {
        //The first onePassRenders standard translucent objects.
        RenderTranslucencyInner(RHICmdList, ETranslucencyPass::TPT_StandardTranslucency, SceneColorCopy,
        bDrawUnderwaterViews);
        //The secondPassRenderingDOFSubsequent translucent objects will be stored in a separate translucentRTin for
        later use. RenderTranslucencyInner(RHICmdList, ETranslucencyPass::TPT_TranslucencyAfterDOF, SceneColorCopy,
        bDrawUnderwaterViews);
        //The thirdPassThe translucentRTand the scene color buffer inDOF passThen mix
        it up. RenderTranslucencyInner(RHICmdList,
ETranslucencyPass::TPT_TranslucencyAfterDOFModulate, SceneColorCopy,
bDrawUnderwaterViews);
    }
    else//Normal mode, singlePassThat is, all translucent objects are rendered. {

        RenderTranslucencyInner(RHICmdList, ETranslucencyPass::TPT_AllTranslucency, SceneColorCopy,
        bDrawUnderwaterViews);
    }

    RHICmdList.EndUAVOverlap();
}
```

RenderTranslucencyInnerThe code to actually render the semi-transparent object internally is as follows:

```
// Source\Runtime\Renderer\Private\TranslucentRendering.cpp

void FDeferredShadingSceneRenderer::RenderTranslucencyInner(FRHICommandListImmediate& RHICmdList,
ETranslucencyPass::Type TranslucencyPass, IPooledRenderTarget* SceneColorCopy, boolbDrawUnderwaterViews) {


    if(!ShouldRenderTranslucency(TranslucencyPass)) {

        return;// Early exit if nothing needs to be done.
    }

    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

    //Parallel rendering support.
    const boolbUseParallel = GRHICommandList.UseParallelAlgorithms() &&
    CVarParallelTranslucency.GetValueOnRenderThread();
```

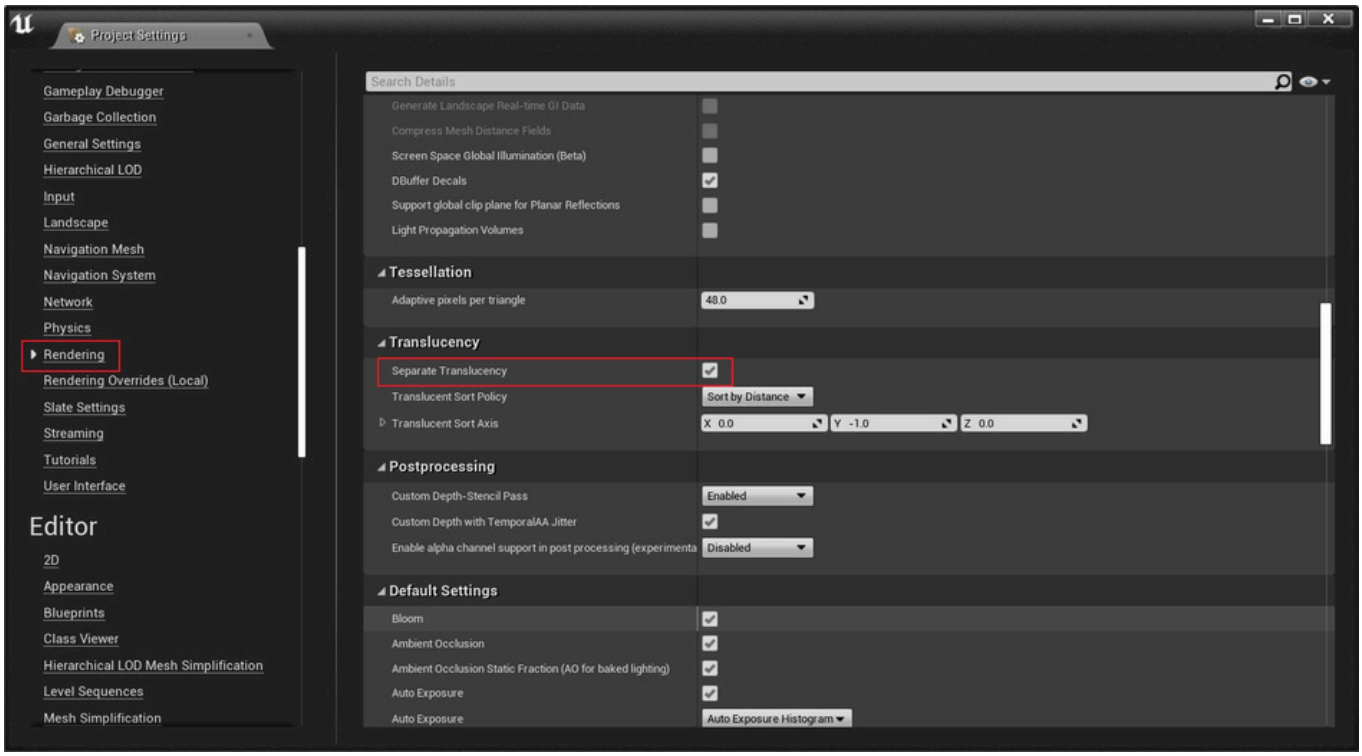
```
if(bUseParallel)
{
    SceneContext.AllocLightAttenuation(RHICmdList);// materials will attempt to get this texture before the deferred command to set it up executes
}
FScopedCommandListWaitForTasksFlusher(bUseParallel && (CVarRHICmdFlushRenderThreadTasksTranslucentPass.GetValueOnRenderThread() >0|| CVarRHICmdFlushRenderThreadTasks.GetValueOnRenderThread() >0), RHICmdList);

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

    FViewInfo& View = Views[ViewIndex];
    if(!View.ShouldRenderView() || (Views[ViewIndex].IsUnderwater() != bDrawUnderwaterViews))
    {
        continue;
    }

    //Update sceneUinform Buffer. Scene->UniformBuffers.UpdateViewUniformBuffer(View);

    TUniformBufferRef<FTranslucentBasePassUniformParameters> //Update BasePassUniformBuffer;
    TranslucencyPassofUinform Buffer.
    CreateTranslucentBasePassUniformBuffer(RHICmdList, View, SceneColorCopy, ESceneTextureSetupMode::All, BasePassUniformBuffer, ViewIndex);
        //Rendering status.
    FMeshPassProcessorRenderStateDrawRenderState(View, BasePassUniformBuffer);

    //RenderingsaparateLine up.
    if(!bDrawUnderwaterViews && RenderInSeparateTranslucency(SceneContext, TranslucencyPass, View.TranslucentPrimCount.DisableOffscreenRendering(TranslucencyPass)))
    {
        FIntPoint ScaledSize;
        floatDownsamplingScale =1.f; SceneContext.GetSeparateTranslucencyDimensions(ScaledSize, DownsamplingScale);

        if(DownsamplingScale <1.f) {

            FViewUniformShaderParameters DownsampledTranslucencyViewParameters;
            SetupDownsampledTranslucencyViewParameters(RHICmdList, View, DownsampledTranslucencyViewParameters);
            Scene->UniformBuffers.UpdateViewUniformBufferImmediate(DownsampledTranslucencyViewParameters);
            DrawRenderState.SetViewUniformBuffer(Scene->UniformBuffers.ViewUniformBuffer);

            (.....)
        }

        //The preparation phase before rendering.
        if(TranslucencyPass == ETranslucencyPass::TPT_TranslucencyAfterDOF) {

            BeginTimingSeparateTranslucencyPass(RHICmdList, View);
            SceneContext.BeginRenderingSeparateTranslucency(RHICmdList, View, * this, NumProcessedViews ==0|| View.Family->bMultiGPUForkAndJoin);
        }
        //Mixed queues.
```

```
		else if(TranslucencyPass == ETranslucencyPass::TPT_TranslucencyAfterDOFModulate)
		{
			BeginTimingSeparateTranslucencyModulatePass(RHICmdList, View);
			SceneContext.BeginRenderingSeparateTranslucencyModulate(RHICmdList, View, * this, NumProcessedViews ==0|| View.Family->bMultiGPUForkAndJoin);
		}
		//Standard queue.
		else
		{
			SceneContext.BeginRenderingSeparateTranslucency(RHICmdList, View, *this, NumProcessedViews ==0|| View.Family->bMultiGPUForkAndJoin);
		}

		// Draw only translucent prims that are in the SeparateTranslucency pass
		DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<false, CF_DepthNearOrEqual>::GetRHI());

		// Actually draw semi-transparent objects.
		if (bUseParallel)
		{
			RHICmdList.EndRenderPass();
			RenderViewTranslucencyParallel(RHICmdList, View, DrawRenderState, TranslucencyPass);
		}
		else
		{
			RenderViewTranslucency(RHICmdList, View, DrawRenderState, TranslucencyPass);
			RHICmdList.EndRenderPass();
		}

		//The final stage after rendering.
		if(TranslucencyPass == ETranslucencyPass::TPT_TranslucencyAfterDOF) {

			SceneContext.ResolveSeparateTranslucency(RHICmdList, View);
			EndTimingSeparateTranslucencyPass(RHICmdList, View);
		}
		else if(TranslucencyPass == ETranslucencyPass::TPT_TranslucencyAfterDOFModulate)
		{
			SceneContext.ResolveSeparateTranslucencyModulate(RHICmdList, View);
			EndTimingSeparateTranslucencyModulatePass(RHICmdList, View);
		}
		else
		{
			SceneContext.ResolveSeparateTranslucency(RHICmdList, View);
		}

		//Upsampling (enlarging) translucent objectsRT.
		 if(TranslucencyPass != ETranslucencyPass::TPT_TranslucencyAfterDOF && TranslucencyPass != ETranslucencyPass::TPT_TranslucencyAfterDOFModulate)
		{
			UpsampleTranslucency(RHICmdList, View,false);
		}
	}
	else//	Standard queue.
	{
```

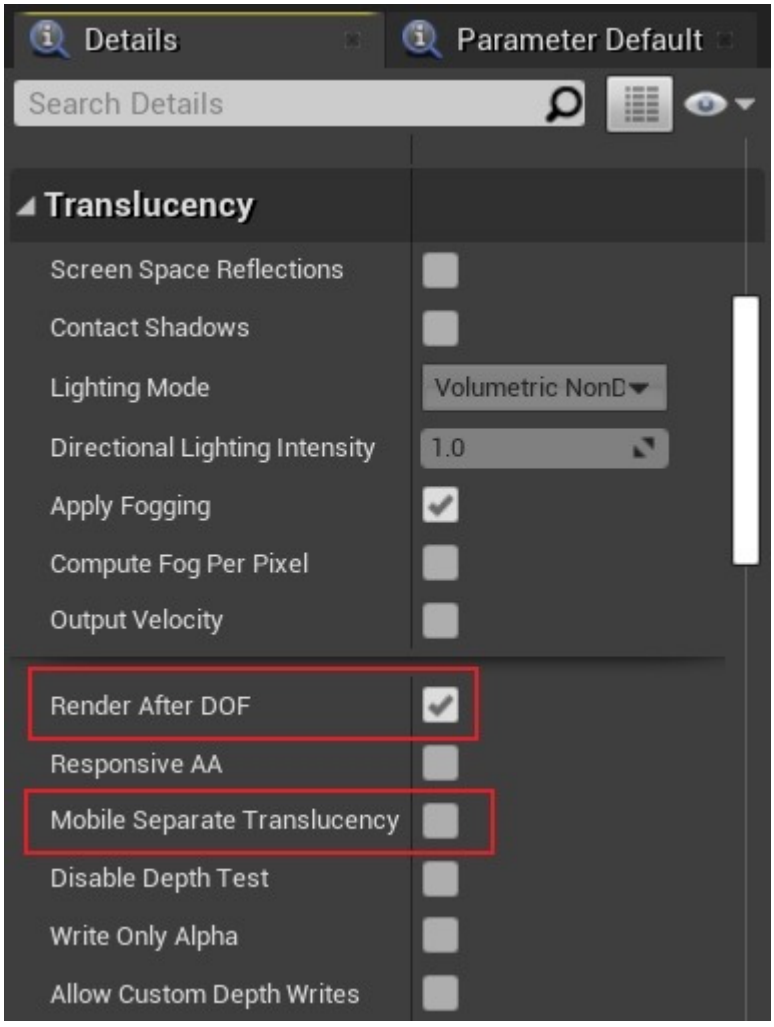
```
                SceneContext.BeginRenderingTranslucency(RHICmdList, View, *this,
NumProcessedViews ==0|| View.Family->bMultiGPUForkAndJoin);
                DrawRenderState.SetDepthStencilState(TStaticDepthStencilState<false,
CF_DepthNearOrEqual>::GetRHI());

                if(bUseParallel && !ViewFamily.UseDebugViewPS()) {

                    RHICmdList.EndRenderPass();
                    RenderViewTranslucencyParallel(RHICmdList, View,          DrawRenderState,
TranslucencyPass);
                }
                else
                {
                    RenderViewTranslucency(RHICmdList, View, DrawRenderState,
TranslucencyPass);
                    RHICmdList.EndRenderPass();
                }

                SceneContext.FinishRenderingTranslucency(RHICmdList);
            }

            // Keep track of number of views not skipped
            NumProcessedViews++;
        }
}
```

The C++ logic and shader logic of semi-transparent rendering are similar to those of Base Pass, except that semi-transparent only processes semi-transparent objects, the Uniform Buffer part is different, the Render State is also different, and the lighting algorithm is also calculated differently. However, their main logic is roughly the same, so I will not analyze it here.

In addition, UE4 has two main translucent rendering queues: one is the standard queue and the other is the separate queue. The separate queue needs to be enabled in the project Render settings (enabled by default):



If you want to add a transparent object to the Separate queue, just turn on Render After DOF for the material it uses (it is turned on by default):



Since the project configuration and material tags both enable the separate queue by default, all translucent objects are rendered in the separate queue by default.

4.3.9 PostProcessing

The post-processing stage is alsoFDeferredShadingSceneRenderer::Renderthe last stage. It includes Bloom, tone mapping, gamma correction, etc. that do not require GBuffer, and SSR, SSAO, SSGI, etc. that require GBuffer. This stage will mix the semi-transparent rendering texture into the final scene color .

It isFDeferredShadingSceneRenderer::Renderlocated as follows:

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

    (......)

    RenderTranslucency(RHICmdList, ...);

    (......)

    RenderLightShaftBloom(RHICmdList);

    (......)

    RenderDistanceFieldLighting(RHICmdList,                  . . . ) ;

    (......)

    CopySceneCaptureComponentToTarget(RHICmdList);

    (......)

    //  Post-processing stage.
    if  (ViewFamily.bResolveScene)
    {
        GRenderTargetPool.AddPhaseEvent(TEXT("PostProcessing"));

        //Post-build processingGraphBuilder. FRDGBuilder
        GraphBuilder(RHICmdList);

        FSceneTextureParameters SceneTextures;
        SetupSceneTextureParameters(GraphBuilder, &SceneTextures);

        // Fallback to a black texture if no velocity. if(!
        SceneTextures.SceneVelocityBuffer) {

            SceneTextures.SceneVelocityBuffer =
GSystemTextures.GetBlackDummy(GraphBuilder);
        }

        //Input parameters for postprocessing.
        FPostProcessingInputs PostProcessingInputs; PostProcessingInputs.SceneTextures = &SceneTextures;
        PostProcessingInputs.ViewFamilyTexture = CreateViewFamilyTexture(GraphBuilder, ViewFamily);


        PostProcessingInputs.SceneColor = GraphBuilder.RegisterExternalTexture(SceneContext.GetSceneColor(),
TEXT("SceneColor"));
        PostProcessingInputs.CustomDepth =
```

```
GraphBuilder.TryRegisterExternalTexture(SceneContext.CustomDepth,                    TEXT("CustomDepth"));
            PostProcessingInputs.SeparateTranslucency                =
RegisterExternalTextureWithFallback(GraphBuilder, SceneContext.GSecteSenpeaCroantteeTxrta.SnespluacreantecyTDraunmslmucye()n,
TEXT("SeparateTranslucency"));
            PostProcessingInputs.SeparateModulation = RegisterExternalTextureWithFallback(GraphBuilder,
SceneContext.SeparateTranslucencyModulateRT,
SceneContext.GetSeparateTranslucencyModulateDummy(), TEXT("SeparateModulate"));



            (.....)

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

                    FViewInfo& View = Views[ViewIndex]; //Add post-
                    processing channel.
                    AddPostProcessingPasses(GraphBuilder, View, PostProcessingInputs);
                }
            }

            //Post-release processingRT.
            SceneContext.FreeSeparateTranslucency();
            SceneContext.FreeSeparateTranslucencyModulate();
            SceneContext.SetSceneColor(nullptr);
            SceneContext.AdjustGBufferRefCount(GraphBuilder.RHICmdList,                    - 1);

            //implementGraphBuilder.
            GraphBuilder.Execute();

            GRenderTargetPool.AddPhaseEvent(TEXT("AfterPostprocessing"));

            // End of frame, we don't need it anymore.
            FSceneRenderTargets::Get(RHICmdList).FreeDownsampledTranslucencyDepth();
        }
        else
        {
            // Release the original reference on the scene render targets
            SceneContext.AdjustGBufferRefCount(RHICmdList,-1);
        }

        (.....)
}
```

Post-processing is not used directlyRHICommandList,but isGraphBuilderreplacedGraphBuilderby the dependency rendering graph (RDG), which can automatically cut useless passes, automatically manage the dependencies between passes and the life cycle of resources, as well as optimize resources, PSO, and rendering instructions.

AddPostProcessingPassesIt is to add the dedicated logic of Pass. The code is as follows:

```
// Engine\Source\Runtime\Renderer\Private\PostProcess\PostProcessing.cpp

void AddPostProcessingPasses(FRDGBuilder& GraphBuilder,constFViewInfo& View,const FPostProcessingInputs&
Inputs) {

    Inputs.Validate();
```

```
//Initialize data and tags.

constFIntRect PrimaryViewRect = View.ViewRect;

constFSceneTextureParameters& SceneTextures = *Inputs.SceneTextures; const
FScreenPassRenderTarget ViewFamilyOutput =
FScreenPassRenderTarget::CreateViewFamilyOutput(Inputs.ViewFamilyTexture,                View);
constFScreenPassTextureSceneDepth(SceneTextures.SceneDepthBuffer, PrimaryViewRect); const
FScreenPassTexture PrimaryViewRSeecpta);rateTranslucency(Inputs.SeparateTranslucency,

constFScreenPassTexture          SeparateModulation(Inputs.SeparateModulation,
PrimaryViewRect);
constFScreenPassTextureCustomDepth(Inputs.CustomDepth, PrimaryViewRect);
constFScreenPassTextureVelocity(SceneTextures.SceneVelocityBuffer, PrimaryViewRect); constFScreenPassTexture
BlackDummy(GSystemTextures.GetBlackDummy(GraphBuilder));

// Scene color is updated incrementally through the post process pipeline. FScreenPassTexture
SceneColor(Inputs.SceneColor, PrimaryViewRect);

// Assigned before and after the tonemapper.
FScreenPassTexture      SceneColorBeforeTonemap;
FScreenPassTexture      SceneColorAfterTonemap;

// Unprocessed scene color stores the original input. const
FScreenPassTexture OriginalSceneColor = SceneColor;

// Default the new eye adaptation to the last one in case it's not generated this frame.

constFEyeAdaptationParameters EyeAdaptationParameters =
GetEyeAdaptationParameters(View, ERHIFeatureLevel::SM5);
FRDGTextureRef LastEyeAdaptationTexture = GetEyeAdaptationTexture(GraphBuilder, View); FRDGTextureRef
EyeAdaptationTexture = LastEyeAdaptationTexture;

// Histogram defaults to black because the histogram eye adaptation pass is used for the manual metering mode.

FRDGTextureRef HistogramTexture = BlackDummy.Texture;

constFEngineShowFlags& EngineShowFlags = View.Family->EngineShowFlags; const bool
bVisualizeHDR = EngineShowFlags.VisualizeHDR;
const boolbViewFamilyOutputInHDR = GRHISupportsHDROutput && IsHDREnabled(); const bool
bVisualizeGBufferOverview = IsVisualizeGBufferOverviewEnabled(View); const boolbVisualizeGBufferDumpToFile
= IsVisualizeGBufferDumpToFileEnabled(View); const boolbVisualizeGBufferDumpToPIpe =
IsVisualizeGBufferDumpToPipeEnabled(View); const boolbOutputInHDR = IsPostProcessingOutputInHDR();



constFPaniniProjectionConfigPaniniConfig(View);

//All post-processingPass.
enum classEPass: uint32 {

    MotionBlur,
    Tonemap,
    FXAA,
    PostProcessMaterialAfterTonemapping,
    VisualizeDepthOfField,
    VisualizeStationaryLightOverlap,
    VisualizeLightCulling,
```

```
        SelectionOutline,
        EditorPrimitive,
        VisualizeShadingModels,
        VisualizeGBufferHints,
        VisualizeSubsurface,
        VisualizeGBufferOverview,
        VisualizeHDR,
        PixelInspector,
        HMDDistortion,
        HighResolutionScreenshotMask,
        PrimaryUpscale,
        SecondaryUpscale,
        MAX

    };

    //All post-processingPassThe
    corresponding name. constTCHAR*
    PassNames[] = {
        TEXT("MotionBlur"),
        TEXT("Tonemap"),
        TEXT("FXAA"),
        TEXT("PostProcessMaterial TEXT(    (AfterTonemapping)"),
        "VisualizeDepthOfField"), TEXT(
        "VisualizeStationaryLightOverlap"), TEXT(
        "VisualizeLightCulling"), TEXT("SelectionOutline"),
        TEXT("EditorPrimitive"), TEXT(
        "VisualizeShadingModels"), TEXT(
        "VisualizeGBufferHints"), TEXT("VisualizeSubsurface"),
        TEXT("VisualizeGBufferOverview"), TEXT(
        "VisualizeHDR"), TEXT("PixelInspector"), TEXT(
        "HMDDistortion"),



        TEXT("HighResolutionScreenshotMask"), TEXT(
        "PrimaryUpscale"), TEXT("SecondaryUpscale")

    };

    //Open or close the corresponding post-processing channel according to the mark.
    TOverridePassSequence<EPass>PassSequence(ViewFamilyOutput);
    PassSequence.SetNames(PassNames, UE_ARRAY_COUNT(PassNames));
    PassSequence.SetEnabled(EPass::VisualizeStationaryLightOverlap,
    EngineShowFlags.StationaryLightOverlap);
    PassSequence.SetEnabled(EPass::VisualizeLightCulling,
    EngineShowFlags.VisualizeLightCulling);
    PassSequence.SetEnabled(EPass::SelectionOutline,                false);
    PassSequence.SetEnabled(EPass::EditorPrimitive,                false);

    PassSequence.SetEnabled(EPass::VisualizeShadingModels,
    EngineShowFlags.VisualizeShadingModels);
    PassSequence.SetEnabled(EPass::VisualizeGBufferHints, PassSequence.SeEtnEngainbeleSdh(oEwPFalsasg::sV.iGsBuaulfifzeerSHuib
    EngineShowFlags.VisualizeSSS); PassSequence.SetEnabled(EPass::VisualizeGBufferOverview,
                                                    bVisualizeGBufferOverview          ||
bVisualizeGBufferDumpToFile || bVisualizeGBufferDumpToPIpe);
    PassSequence.SetEnabled(EPass::VisualizeHDR, EngineShowFlags.VisualizeHDR);
    PassSequence.SetEnabled(EPass::PixelInspector,false);
```

```
PassSequence.SetEnabled(EPass::HMDDistortion, EngineShowFlags.StereoRendering &&
EngineShowFlags.HMDDistortion);
PassSequence.SetEnabled(EPass::HighResolutionScreenshotMask,
IsHighResolutionScreenshotMaskEnabled(View));
PassSequence.SetEnabled(EPass::PrimaryUpscale, PaniniConfig.IsEnabled() ||
(View.PrimaryScreenPercentageMethod == EPrimaryScreenPercentageMethod::SpatialUpscale &&
PrimaryViewRect.Size() != View.GetSecondaryViewRectSize()));
PassSequence.SetEnabled(EPass::SecondaryUpscale, View.RequiresSecondaryUpscale());

//deal withMotionBlur, Tonemap, PostProcessMaterialAfterTonemappingWait for post-
processing. if(IsPostProcessingEnabled(View)) {

    //Gets the material input.
    constautoGetPostProcessMaterialInputs = [CustomDepth, SeparateTranslucency, Velocity]
       (FScreenPassTexture InSceneColor)
    {
        FPostProcessMaterialInputs PostProcessMaterialInputs;
        PostProcessMaterialInputs.SetInput(EPostProcessMaterialInput::SceneColor,
InSceneColor);

PostProcessMaterialInputs.SetInput(EPostProcessMaterialInput::SeparateTranslucency, SeparateTranslucency);

        PostProcessMaterialInputs.SetInput(EPostProcessMaterialInput::Velocity,
Velocity);
        PostProcessMaterialInputs.CustomDepthTexture = CustomDepth.Texture; return
        PostProcessMaterialInputs;
    };

    //Handle various types of tags.
    constEStereoscopicPass StereoPass = View.StereoPass;
    const boolbPrimaryView = IStereoRendering::IsAPrimaryView(View);

    (......)

    //Turn special post-processing on or off.
    PassSequence.SetEnabled(EPass::MotionBlur,          bVisualizeMotionBlur      ||
    bMotionBlurEnabled);
    PassSequence.SetEnabled(EPass::Tonemap, bTonemapEnabled);
    PassSequence.SetEnabled(EPass::FXAA, AntiAliasingMethod == AAM_FXAA);
    PassSequence.SetEnabled(EPass::PostProcessMaterialAfterTonemapping,
PostProcessMaterialAfterTonemappingChain.Num() !=0);
    PassSequence.SetEnabled(EPass::VisualizeDepthOfField, bVisualizeDepthOfField); PassSequence.Finalize();


    //Translucent blending into scene colorRTBefore post-processing. {

        constFPostProcessMaterialChain MaterialChain =
GetPostProcessMaterialChain(View, BL_BeforeTranslucency);

        if(MaterialChain.Num()) {

            SceneColor = AddPostProcessMaterialChain(GraphBuilder, View,
GetPostProcessMaterialInputs(SceneColor), MaterialChain);
        }
    }

    (......)
```

```
	//Post-processing before tone mapping.
	{
		constFPostProcessMaterialChain MaterialChain =
GetPostProcessMaterialChain(View, BL_BeforeTonemapping);

		if(MaterialChain.Num()) {

			SceneColor = AddPostProcessMaterialChain(GraphBuilder, View,
GetPostProcessMaterialInputs(SceneColor), MaterialChain);
		}
	}

	FScreenPassTexture HalfResolutionSceneColor;

	// Scene color view rectangle after temporal AA upscale to secondary screen percentage.

	FIntRect SecondaryViewRect = PrimaryViewRect;

	//Temporal Anti-Aliasing (TAA).
	if(AntiAliasingMethod == AAM_TemporalAA) {

		// Whether we allow the temporal AA pass to downsample scene color. It may
choose not to based on internal context,
		// in which case the output half resolution texture will remain null. const bool
		bAllowSceneDownsample =
			IsTemporalAASceneDownsampleAllowed(View) &&
			// We can only merge if the normal downsample pass would happen after.
immediately
			!bMotionBlurEnabled && !bVisualizeMotionBlur &&
			// TemporalAA is only able to match the low quality mode (box filter). GetDownsampleQuality()
			== EDownsampleQuality::Low;

		AddTemporalAAPass(
			GraphBuilder,	SceneTextures,	View,
			bAllowSceneDownsample,
			DownsampleOverrideFormat,
			SceneColor.Texture,
			&SceneColor.Texture,
			&SecondaryViewRect,
			&HalfResolutionSceneColor.Texture,
			&HalfResolutionSceneColor.ViewRect);



	}
	// SSR.
	else if	(ShouldRenderScreenSpaceReflections(View))
	{
		// If we need SSR, and TAA is enabled, then AddTemporalAAPass() has already
handled the scene history.
		// If we need SSR, and TAA is not enable, then we just need to extract the
history.
		if(!View.bStatePrevViewInfoIsReadOnly) {

			check(View.ViewState);
			FTemporalAAHistory& OutputHistory = View.ViewState-
> PrevFrameViewInfo.TemporalAAHistory;
			GraphBuilder.QueueTextureExtraction(SceneColor.Texture,
&OutputHistory.RT[0]);
```

```
        }
    }

    //! SceneColorTexture is now upsampled to the SecondaryViewRect. Use SecondaryViewRect
for input / output.
    SceneColor.ViewRect = SecondaryViewRect;

    //Post-Processing Materials -SSRenter.
    if(View.ViewState && !View.bStatePrevViewInfoIsReadOnly) {

        constFPostProcessMaterialChain MaterialChain =
GetPostProcessMaterialChain(View, BL_SSRInput);

        if(MaterialChain.Num()) {

            // Save off SSR post process output for the next frame. FScreenPassTexture PassOutput =
            AddPostProcessMaterialChain(GraphBuilder,     GetPostProcessMaterialInputs(SceneColor),
View,                   MaterialChain);                 GraphBuilder.QueueTextureExtraction(PassOutput.Texture,
            &View.ViewState-
> PrevFrameViewInfo.CustomSSRInput);
        }
    }

    //  Motion blur.
    if  (PassSequence.IsEnabled(EPass::MotionBlur))
    {
        FMotionBlurInputs PassInputs;
        PassSequence.AcceptOverrideIfLastPass(EPass::MotionBlur,
PassInputs.OverrideOutput);
        PassInputs.SceneColor       = SceneColor;
        PassInputs.SceneDepth       = SceneDepth;
        PassInputs.SceneVelocity        = Velocity;
        PassInputs.Quality      =GetMotionBlurQuality();
        PassInputs.Filter       = GetMotionBlurFilter();

        // Motion blur visualization replaces motion blur when enabled. if
        (bVisualizeMotionBlur) {

            SceneColor = AddVisualizeMotionBlurPass(GraphBuilder, View, PassInputs);
        }
        else
        {
            SceneColor = AddMotionBlurPass(GraphBuilder, View, PassInputs);
        }
    }

    (......)

    FScreenPassTexture Bloom;

    //  Floodlight.
    if  (bBloomEnabled)
    {
        FSceneDownsampleChain     BloomDownsampleChain;

        FBloomInputs PassInputs;
        PassInputs.SceneColor = SceneColor;
```

```
                const boolbBloomThresholdEnabled =
View.FinalPostProcessSettings.BloomThreshold >-1.0f;

                // Reuse the main scene downsample chain if a threshold isn't required for
bloom.
                if(SceneDownsampleChain.IsInitialized() && !bBloomThresholdEnabled) {

                        PassInputs.SceneDownsampleChain = &SceneDownsampleChain;
                }
                else
                {
                        FScreenPassTexture DownsampleInput = HalfResolutionSceneColor;

                        if(bBloomThresholdEnabled) {

                                const floatBloomThreshold =
View.FinalPostProcessSettings.BloomThreshold;

                                FBloomSetupInputs SetupPassInputs; SetupPassInputs.SceneColor =
                                DownsampleInput; SetupPassInputs.EyeAdaptationTexture =
                                EyeAdaptationTexture; SetupPassInputs.Threshold = BloomThreshold;


                                DownsampleInput = AddBloomSetupPass(GraphBuilder, View,
SetupPassInputs);
                        }

                        const boolbLogLumaInAlpha =false;
                        BloomDownsampleChain.Init(GraphBuilder,          View,    EyeAdaptationParameters,
DownsampleInput,     DownsampleQuality, bLogLumaInAlpha);

                        PassInputs.SceneDownsampleChain = &BloomDownsampleChain;
                }

                FBloomOutputs PassOutputs = AddBloomPass(GraphBuilder, View, PassInputs); SceneColor =
                PassOutputs.SceneColor;
                Bloom = PassOutputs.Bloom;

                FScreenPassTexture LensFlares = AddLensFlaresPass(GraphBuilder, View, Bloom,
* PassInputs.SceneDownsampleChain);

                if(LensFlares.IsValid()) {

                        // Lens flares are composited with bloom. Bloom =
                        LensFlares;
                }
        }

        // Tonemapper needs a valid bloom target, even if it's black. if(!Bloom.IsValid())

        {
                Bloom = BlackDummy;
        }

        SceneColorBeforeTonemap = SceneColor;

        //    Tone mapping.
        if    (PassSequence.IsEnabled(EPass::Tonemap))
```

```
	{
		constFPostProcessMaterialChain MaterialChain =
GetPostProcessMaterialChain(View, BL_ReplacingTonemapper);

		if(MaterialChain.Num()) {

			constUMaterialInterface* HighestPriorityMaterial = MaterialChain[0];

			FPostProcessMaterialInputs PassInputs;
			PassSequence.AcceptOverrideIfLastPass(EPass::Tonemap,
PassInputs.OverrideOutput);
			PassInputs.SetInput(EPostProcessMaterialInput::SceneColor,  SceneColor);
			PassInputs.SetInput(EPostProcessMaterialInput::SeparateTranslucency,
SeparateTranslucency);
			PassInputs.SetInput(EPostProcessMaterialInput::CombinedBloom, Bloom);
			PassInputs.CustomDepthTexture = CustomDepth.Texture;

			SceneColor = AddPostProcessMaterialPass(GraphBuilder, View, PassInputs,
HighestPriorityMaterial);
		}
		else
		{
			FRDGTextureRef ColorGradingTexture = nullptr;

			if(bPrimaryView)
			{
				ColorGradingTexture = AddCombineLUTPass(GraphBuilder, View);
			}
			// We can re-use the color grading texture from the primary view. else

			{
				ColorGradingTexture =
GraphBuilder.TryRegisterExternalTexture(View.GetTonemappingLUT());
			}

			FTonemapInputs PassInputs;
			PassSequence.AcceptOverrideIfLastPass(EPass::Tonemap,
PassInputs.OverrideOutput);
			PassInputs.SceneColor = SceneColor;
			PassInputs.Bloom = Bloom;
			PassInputs.EyeAdaptationTexture  =  EyeAdaptationTexture;
			PassInputs.ColorGradingTexture  = ColorGradingTexture;
			PassInputs.bWriteAlphaChannel = AntiAliasingMethod == AAM_FXAA ||
IsPostProcessingWithAlphaChannelSupported();
			PassInputs.bOutputInHDR = bTonemapOutputInHDR;

			SceneColor = AddTonemapPass(GraphBuilder, View, PassInputs);
		}
	}

	SceneColorAfterTonemap = SceneColor;

	//  FXAAAnti-aliasing.
	if  (PassSequence.IsEnabled(EPass::FXAA))
	{
		FFXAAInputs PassInputs;
		PassSequence.AcceptOverrideIfLastPass(EPass::FXAA,  PassInputs.OverrideOutput);
		PassInputs.SceneColor = SceneColor;
```

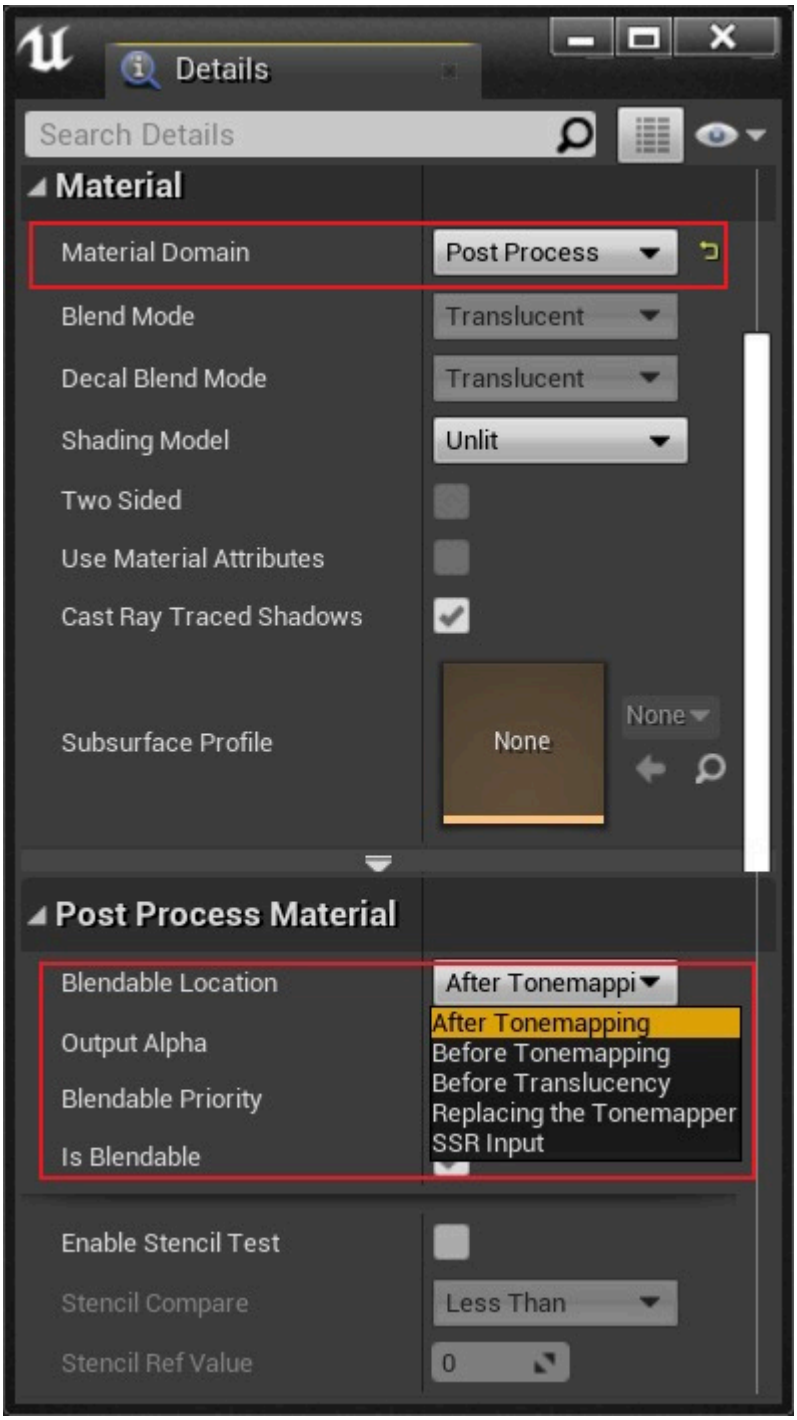
```
                PassInputs.Quality = GetFXAAQuality();

                SceneColor = AddFXAAPass(GraphBuilder, View, PassInputs);
            }

            // Post-processed material after color mapping.
            if (PassSequence.IsEnabled(EPass::PostProcessMaterialAfterTonemapping))
            {
                FPostProcessMaterialInputs PassInputs =
GetPostProcessMaterialInputs(SceneColor);

PassSequence.AcceptOverrideIfLastPass(EPass::PostProcessMaterialAfterTonemapping,
PassInputs.OverrideOutput);
                PassInputs.SetInput(EPostProcessMaterialInput::PreTonemapHDRColor,
SceneColorBeforeTonemap);
                PassInputs.SetInput(EPostProcessMaterialInput::PostTonemapHDRColor,
SceneColorAfterTonemap);

                SceneColor = AddPostProcessMaterialChain(GraphBuilder, View, PassInputs,
PostProcessMaterialAfterTonemappingChain);
            }
        }
        //Combined detached queues for translucencyRTand gamma
        correction. else
        {
            PassSequence.SetEnabled(EPass::MotionBlur,false); PassSequence.SetEnabled(EPass::Tonemap,true);
            PassSequence.SetEnabled(EPass::FXAA,false);
            PassSequence.SetEnabled(EPass::PostProcessMaterialAfterTonemapping,false);
            PassSequence.SetEnabled(EPass::VisualizeDepthOfField,false); PassSequence.Finalize();



            SceneColor.Texture = AddSeparateTranslucencyCompositionPass(GraphBuilder, View, SceneColor.Texture,
SeparateTranslucency.Texture, SeparateModulation.Texture);

            SceneColorBeforeTonemap = SceneColor;

            if(PassSequence.IsEnabled(EPass::Tonemap)) {

                FTonemapInputs PassInputs;
                PassSequence.AcceptOverrideIfLastPass(EPass::Tonemap,
PassInputs.OverrideOutput);
                PassInputs.SceneColor    =    SceneColor;
                PassInputs.bOutputInHDR      = bViewFamilyOutputInHDR; true;
                PassInputs.bGammaOnly =

                SceneColor = AddTonemapPass(GraphBuilder, View, PassInputs);
            }

            SceneColorAfterTonemap = SceneColor;
        }

        (.....)

        // ScenarioRTMain Amplifier.
        if (PassSequence.IsEnabled(EPass::PrimaryUpscale))
        {
            FUpscaleInputs PassInputs;
```

```
            PassSequence.AcceptOverrideIfLastPass(EPass::PrimaryUpscale,
            PassInputs.OverrideOutput);
            PassInputs.SceneColor = SceneColor;
            PassInputs.Method = GetUpscaleMethod();
            PassInputs.Stage = PassSequence.IsEnabled(EPass::SecondaryUpscale) ?
EUpscaleStage::PrimaryToSecondary : EUpscaleStage::PrimaryToOutput;

            // Panini projection is handled by the primary upscale pass.
            PassInputs.PaniniConfig = PaniniConfig;

            SceneColor = AddUpscalePass(GraphBuilder, View, PassInputs);
        }

        //  ScenarioRTSecond zoom in.
        if  (PassSequence.IsEnabled(EPass::SecondaryUpscale))
        {
            FUpscaleInputs PassInputs;
            PassSequence.AcceptOverrideIfLastPass(EPass::SecondaryUpscale,
            PassInputs.OverrideOutput);
            PassInputs.SceneColor = SceneColor;
            PassInputs.Method = View.Family->SecondaryScreenPercentageMethod ==
ESecondaryScreenPercentageMethod::LowerPixelDensitySimulation : EUpscaleMetho? dE:U:NpesacarelesMt;ethod::SmoothStep

            PassInputs.Stage =          EUpscaleStage::SecondaryToOutput;

            SceneColor = AddUpscalePass(GraphBuilder, View, PassInputs);
        }
}
```

All post-processing of UEEPassis specified in , and there are also some custom post-processing materials. Post-processing materials can be divided into the following types according to the rendering stage:

```
enumEBlendableLocation {

        //After tone mapping.
        BL_AfterTonemapping,
        //Before tone mapping.
        BL_BeforeTonemapping,
        //Before translucent combination.
        BL_BeforeTranslucency,
        //Replace tone mapping.
        BL_ReplacingTonemapper, //
        SSRenter.
        BL_SSRInput,



        BL_MAX,
};
```

EBlendableLocationThis can be specified in the Material Editor's Properties panel:



The default blending stage is after tone mapping. To better illustrate how to add a post-processing pass to the renderer, let's take adding tone mapping as an example:

```
// Engine\Source\Runtime\Renderer\Private\PostProcess\PostProcessing.cpp

static FRCPassPostProcessTonemap* AddTonemapper(
    FPostprocessContext& Context,
    const FRenderingCompositeOutputRef& BloomOutputCombined,
    EAutoExposureMethod
    const bool bDoGammaOnly, const bool
    bHDRTonemapperOutput, const bool
    bMetalMSAAHDRDecode, const bool
    bIsMobileDof)

{
    const FViewInfo& View = Context.View;
    const EStereoscopicPass StereoPass = View.StereoPass;
```

```
        FRenderingCompositeOutputRef //      TonemapperCombinedLUTOutputRef;
            LUT Pass.
        if   (IStereoRendering::IsAPrimaryView(View))
        {
              TonemapperCombinedLUTOutputRef = AddCombineLUTPass(Context.Graph);
        }

        const boolbDoEyeAdaptation = IsAutoExposureMethodSupported(View.GetFeatureLevel(),
EyeAdapationMethodId) && EyeAdaptation.IsValid();
        //TowardsContext.GraphRegister aPass,The object type isFRCPassPostProcessTonemap.
        FRCPassPostProcessTonemap* PostProcessTonemap = Context.Graph.RegisterPass(new(FMemStack::Get())
FRCPassPostProcessTonemap(bDoGammaOnly, bDoEyeAdaptation, bHDRTonemapperOutput,
bMetalMSAAHDRDecode, bIsMobileDof));

        //Set the input texture.
        PostProcessTonemap->SetInput(ePId_Input0,          Context.FinalOutput);
        PostProcessTonemap->SetInput(ePId_Input1,          BloomOutputCombined);
        PostProcessTonemap->SetInput(ePId_Input2,          EyeAdaptation);
        PostProcessTonemap->SetInput(ePId_Input3,          TonemapperCombinedLUTOutputRef);

        //Set up the output.
        Context.FinalOutput = FRenderingCompositeOutputRef(PostProcessTonemap);

        returnPostProcessTonemap;
}
```

Since there are too many types of post-processing and many internal mechanisms and details involved, this section only gives you a general understanding of the post-processing rendering process. There will be special chapters later that explain its principles and mechanisms in detail.

4.4 Summary

4.4.1 Deferred Rendering Summary

This article focuses on the deferred rendering pipeline and its variants, explaining its principles, implementations, and variants. It then turns to UEFDeferredShadingSceneRenderer::Renderto explain the main steps and processes of the deferred shading scene renderer, so that people who are new to UE can have a main understanding and impression of its rendering process.

Of course, this article cannot analyze all the details of rendering technology, it just serves as a starting point. More principles, mechanisms and details are waiting for readers to study the UE source code themselves.

4.4.2 The future of deferred rendering

With the development of hardware and software technologies, UE's deferred rendering pipeline will continue to play its application role on the PC platform, and will gradually be supported on mobile

platforms. In the official speech of the last UOD, it is expected that the next version of UE will well support the deferred rendering pipeline of mobile platforms.

In addition, with the development of variant technologies of deferred rendering, and more efficient and high-performance algorithms and engineering technologies will be introduced into commercial game engines, combined with modern lightweight graphics API, RDG, GPU Driven Pipeline and other technologies, it will shine even more brightly.

4.4.3 Thoughts on this article

This article also arranges some small thoughts to help understand and deepen the grasp and understanding of UE's delayed rendering pipeline:

- Please explain the specific process of delayed rendering.
- What are the variants of deferred rendering? What are their characteristics?
- Please briefly describeFDeferredShadingSceneRenderer::Renderthe main steps.
- Implement parallel lighting that is the same as the view's line of sight to simulate the light source bound to the camera.

-
-
-
-
-

References

- Unreal Engine 4 Sources Unreal
- Engine 4 Documentation
- Rendering and Graphics
- Graphics Programming Overview
- Materials
- Unreal Engine 4 Rendering

- Rendering - Schematic Overview
- UE4 Render System Sheet
- Analysis of UE4 rendering module
- The triangle processor and normal vector shader: a VLSI system for high performance graphics Wiki:
- Deferred shading
- LearnOpenGL: Deferred Shading Efficient
- Rasterization on Mobile GPUs Introduction
- to PowerVR for Developers
- A look at the PowerVR graphics architecture: Tile-based rendering
- GDC: Deferred Shading
- Deferred lighting approaches Deferred Shading VS
- Deferred Lighting Clustered Deferred and Forward
- Shading Decoupled deferred shading for hardware
- rasterization
- General techniques for optimizing mobile game performance
- In-depth understanding of GPU hardware architecture and operation mechanism
- Lighting algorithms in game engines
- Light Pre-Pass
- A sort-based deferred shading architecture for decoupled sampling
- Tiled Shading
- Stream compaction for deferred shading
- Deferred Rendering for Current and Future Rendering Pipelines Deferred
- attribute interpolation for memory-efficient deferred shading Deferred
- Adaptive Compute Shading
- Coherent Culling and Shading for Large Molecular Dynamics Visualization
- Volume Tiled Forward Shading
- Efficient Adaptive Deferred Shading with Hardware Scatter Tiles
- Multisample anti-aliasing in deferred rendering
- Improving Real-Time Rendering Quality and Efficiency using Variable Rate Shading on Modern
- Hardware
- Optimizing multisample anti-aliasing for deferred renderers
- HOW UNREAL RENDERS A FRAME
- Unreal Frame Breakdown
- Rendering concept: 5. Rendering path-RenderPath
- Rendering pipeline in game engines
- Practical Clustered Shading Clustered
- Deferred and Forward Shading
- "Gpu Gems", "Gpu Pro" and "Gpu Zen" series reading notes
- The Visibility Buffer: A Cache-Friendly Approach to Deferred Shading GPU
- Pro 7: Fine Pruned Tiled Light Lists
- GPU Pro 7: Deferred Attribute Interpolation Shading
-

- GPU Pro 7: Deferred Coarse Pixel Shading
- BINDLESS TEXTURING FOR DEFERRED RENDERING AND DECALS
- Use Variable Rate Shading (VRS) to Improve the User Experience in Real-Time Game Engines | SIGGRAPH 2019 Technical Sessions
- MSE RMSE PSNR SSIM
- UE4 rendering basics RenderTranslucency

https://github.com/pe7yu