Analysis of Unreal Rendering System (12) - Mobile Special Part 1 (UE Mobile Rendering Analysis)

Table of contents

12.1 Overview

All previous chapters explain the UE rendering system based on the deferred rendering pipeline on the PC side, especiallyAnalysis of Unreal Rendering System (04) - Deferred Rendering Pipeline, which elaborates on the process and steps of the deferred rendering pipeline on the PC side.

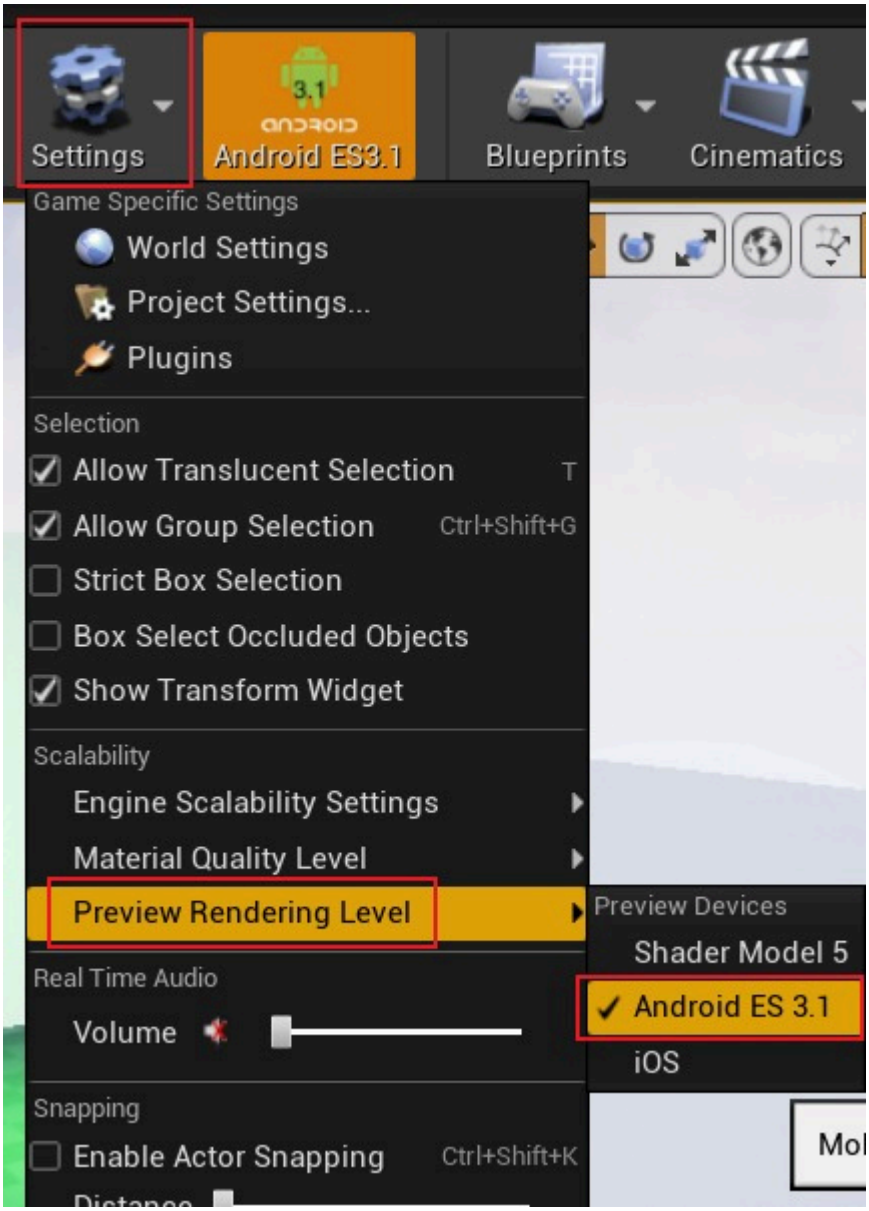
This article will only explain the rendering pipeline of UE's mobile terminal, and will eventually compare the rendering differences between mobile and PC terminals, as well as special optimization

measures. This article mainly explains the following contents of the UE rendering system:

- The main processes and steps of FMobileSceneRenderer.
- Forward and deferred rendering pipelines for mobile.
- Lighting and shadows on mobile.
- The similarities and differences between mobile and PC, as well as the special optimization techniques involved.

It should be pointed out in particular that the UE source code analyzed in this article has been upgraded to **4.27.1**.Students who need to read the source code synchronously should pay attention to the update.

If you want to open the mobile rendering pipeline in the UE editor on PC, you can select the menu as shown below:

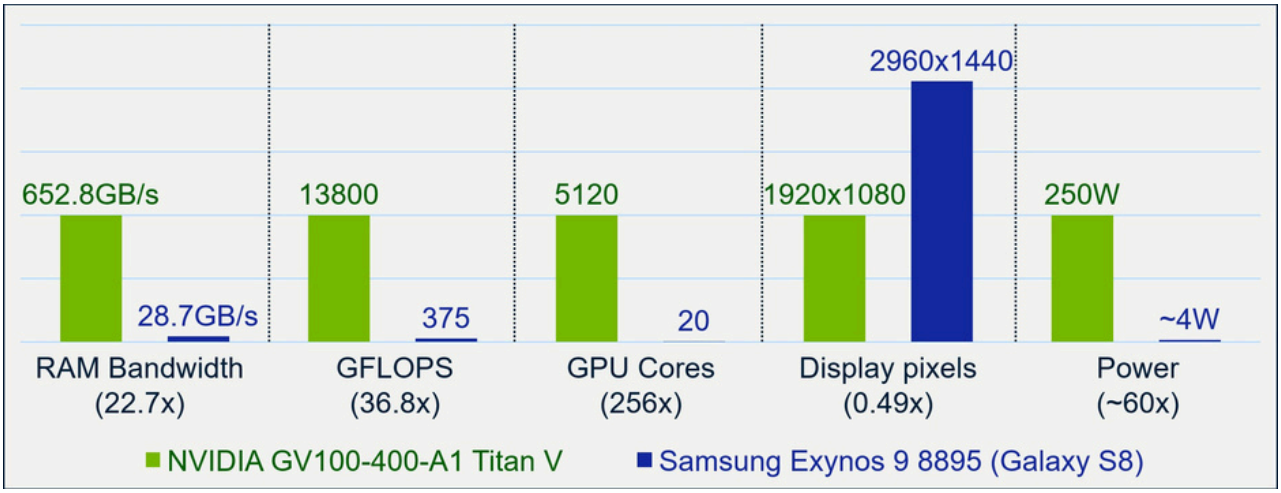


Wait for Shader compilation to complete, and the mobile preview effect will be displayed in the viewport of the UE editor.

12.1.1 Characteristics of mobile devices

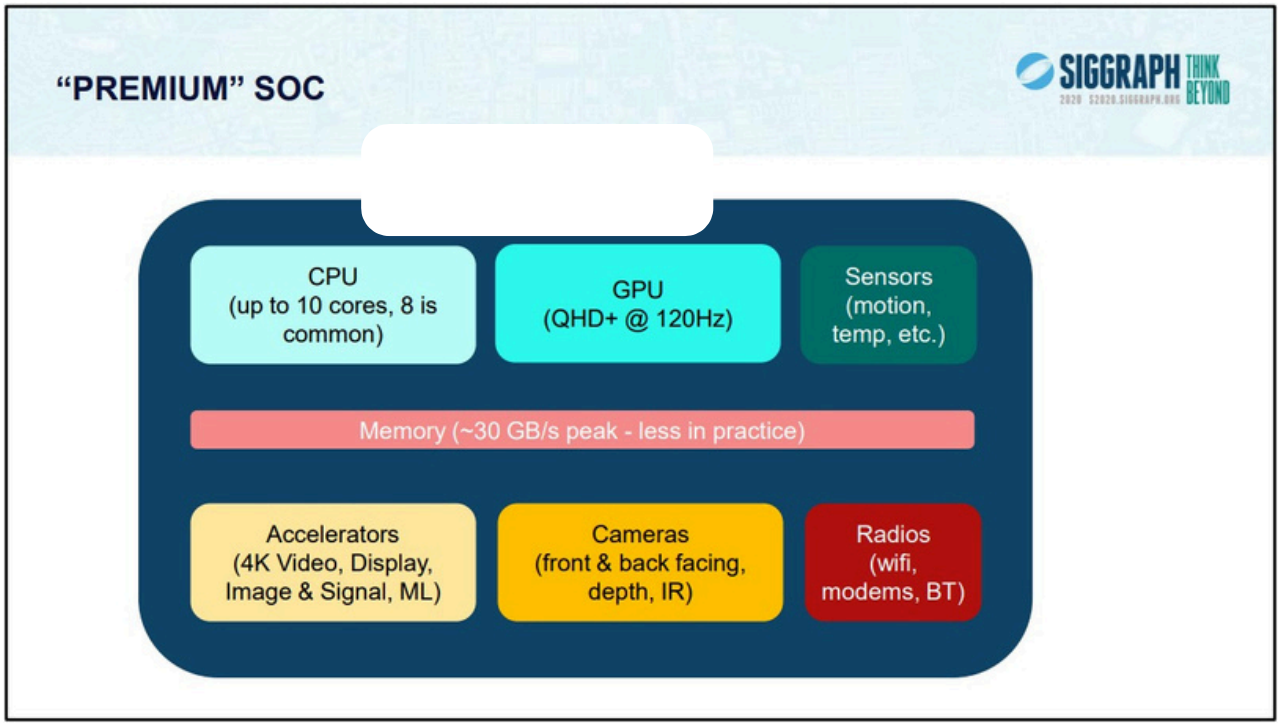
Compared with PC desktop platforms, mobile terminals have significant differences in size, power consumption, hardware performance and many other aspects, as shown in the following:

- **Smaller size.**The portability of mobile terminals requires that the entire device must be lightweight and can be placed in the palm of your hand or in your pocket, so the entire device can only be limited to a very small size. **Limited energy and power.**Limited by battery storage
- technology, the current mainstream lithium battery is generally 10,000 mAh, but the resolution and image quality of mobile devices are getting higher and higher. In order to meet the requirements of long enough battery life and heat dissipation, the power of mobile devices must be strictly controlled, usually within 5W. **Limited heat dissipation methods.**PC devices can usually be equipped with
- cooling fans or even water cooling systems, but mobile devices do not have these active heat dissipation methods and can only rely on heat conduction to dissipate heat. If the heat dissipation is not appropriate, the CPU and GPU will actively reduce the frequency and run at very limited performance to prevent the device components from being damaged due to overheating. **Limited**
- **hardware performance.**The performance of various components of mobile devices (CPU, bandwidth, memory, GPU, etc.) is only a few tenths of that of PC devices.

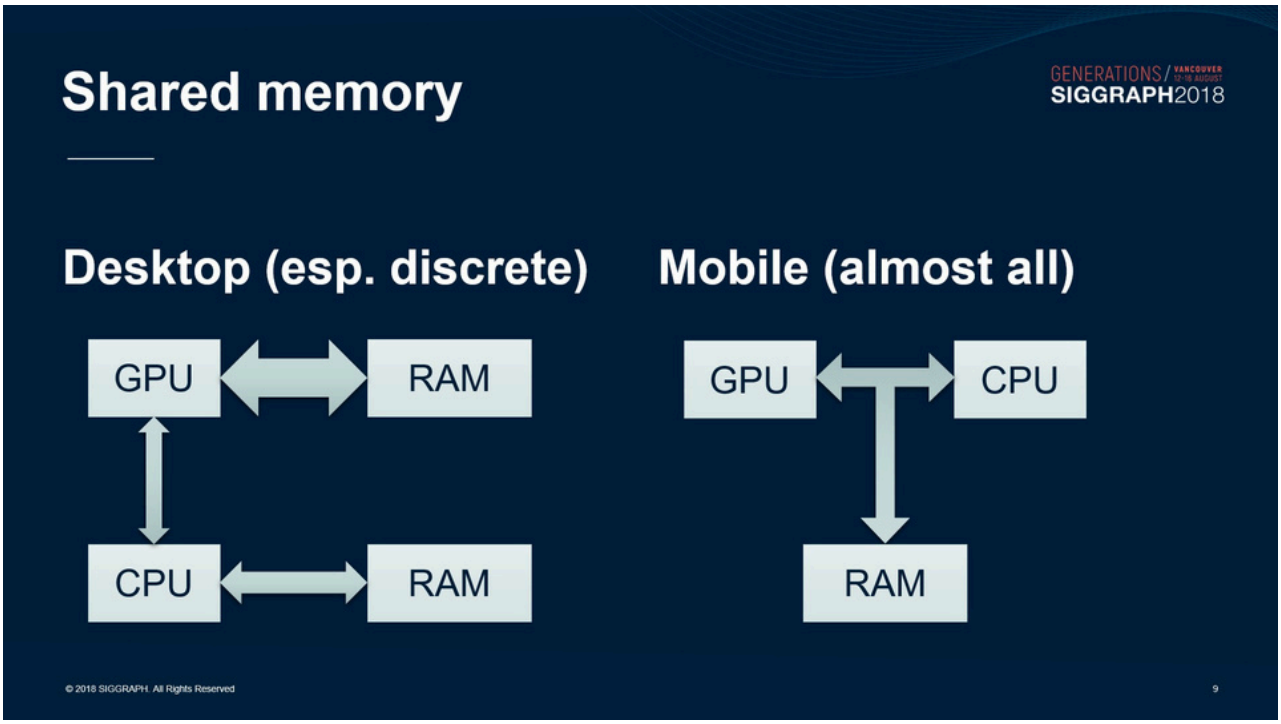


Performance comparison chart of mainstream PC devices (NV GV100-400-A1 Titan V) and mainstream mobile devices (Samsung Exynos 9 8895) in 2018. Many hardware performances of mobile devices are only a few tenths of those of PC devices, but the resolution is close to half of that of PC, which further highlights the challenges and dilemmas of mobile devices.

By 2020, the performance of mainstream mobile devices will be as follows:

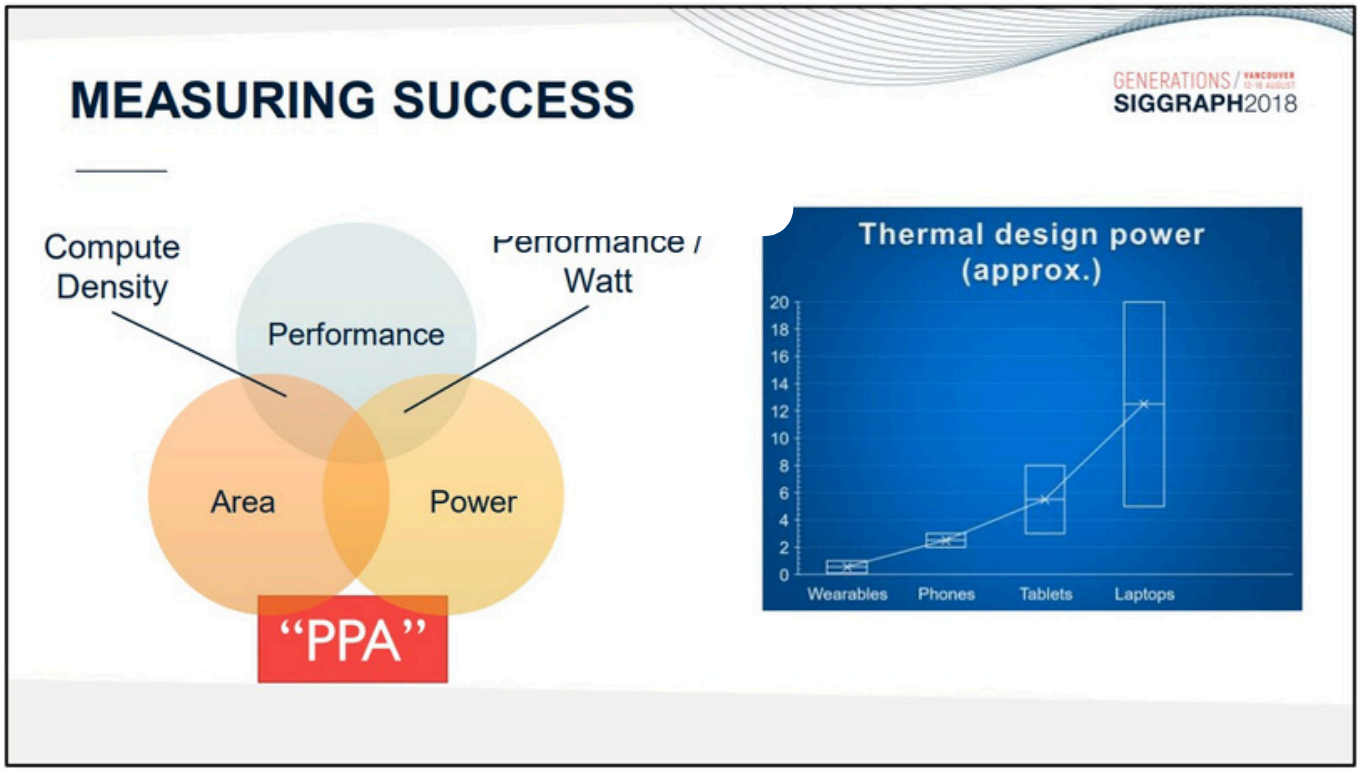


- **Special hardware architectures,**such as the CPU and GPU sharing memory storage devices, are called coupled architectures, and the GPU's TB(D)R architecture, all aim to complete as many operations as possible within low power consumption.



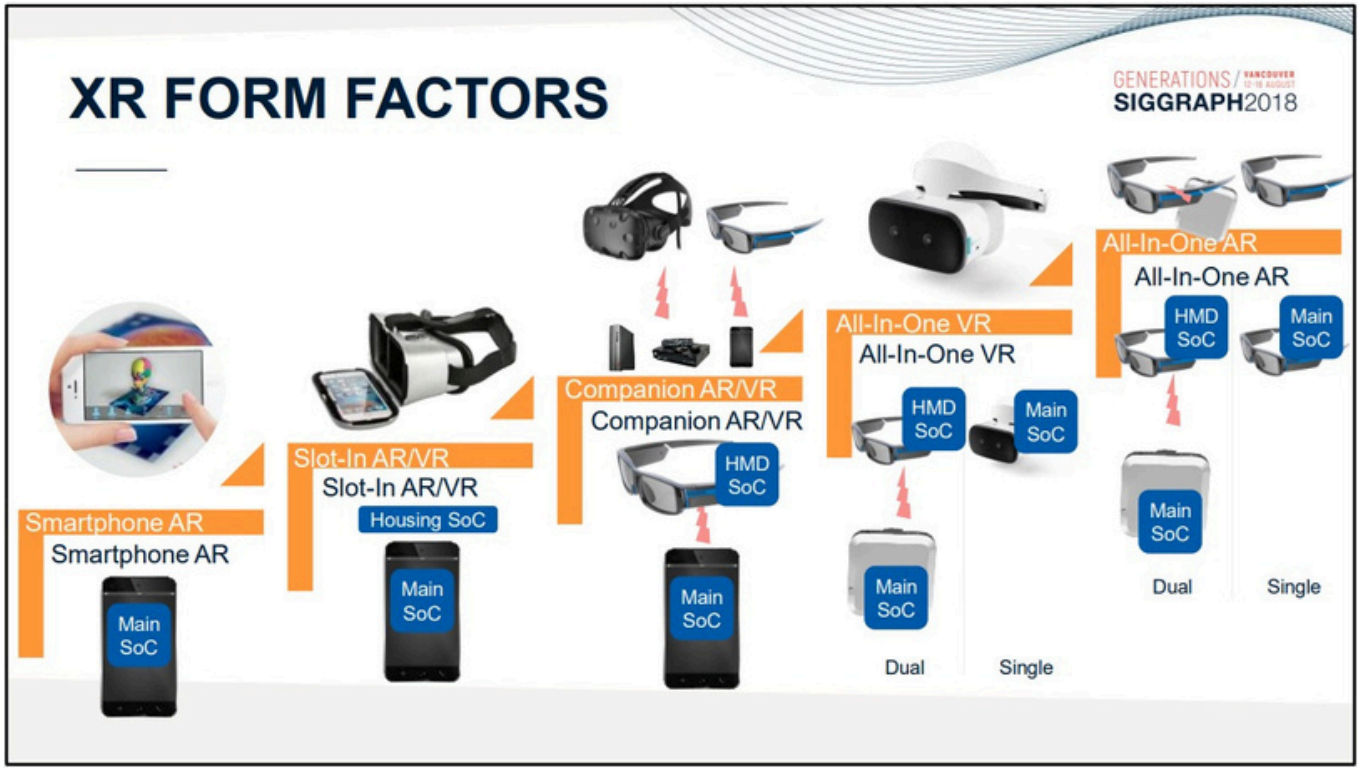
Comparison chart of the decoupled hardware architecture of PC devices and the coupled hardware architecture of mobile devices.

In addition, unlike the CPU and GPU on the PC side, which purely pursue computing performance, there are three indicators to measure the performance of the mobile side: performance**,**power**,**and area**,** commonly known as**PPA**. (Figure below)



There are three basic parameters for measuring mobile devices: Performance, Area, and Power. Compute Density involves performance and area, and energy efficiency involves performance and capacity consumption. The larger the better.

Along with mobile devices, XR devices are also on the rise, which is an important development branch of mobile devices. Currently, there are various XR devices of different sizes, functions, and application scenarios:



Various forms of XR devices.

With the recent explosion of the Metaverse and FaceBook's name change to Meta, coupled with the fact that technology giants such as Apple, Microsoft, NVidia, and Google are stepping up their efforts to develop future immersive experiences, XR devices, as the carrier and entrance that are closest to the Metaverse's vision, have naturally become a new track with great potential for the emergence of giants in the future.

12.2 UE Mobile Rendering Features

This chapter explains the rendering features of UE4.27 on mobile terminals.

12.2.1 Feature Level

UE supports the following graphics APIs on mobile devices:

Feature Level	illustrate
OpenGL ES 3.1	The default feature level of the Android system. You can configure specific material parameters in the project settings (**Project Settings**>**Platforms**> **Android Material Quality - ES31).**
Android Vulkan	A high-end renderer that can be used on certain Android devices. It supports the Vulkan 1.2 API. Vulkan, with its lightweight design concept, is more efficient than OpenGL in most cases.
Metal 2.0	A level of features specific to iOS devices. Material parameters can be configured in**Project Settings**>**Platforms**>**iOS Material Quality .**

On current mainstream Android devices, using Vulkan can achieve better performance because of Vulkan's lightweight design concept, which enables UE and other applications to perform optimization more accurately. The following is a comparison table between Vulkan and OpenGL:

Vulkan	OpenGL
Object-based state, no global state.	Single global state machine.
All state concepts are placed in the command buffer.	State is bound to a single context.
Multi-threaded encoding is possible.	Rendering operations can only be performed sequentially.

Vulkan	OpenGL
Allows precise and explicit control over GPU memory and synchronization.	The memory and synchronization details of the GPU are usually hidden by the driver.
There is no runtime error detection for the driver, but there is a validation layer for developers.	Extensive runtime error detection.

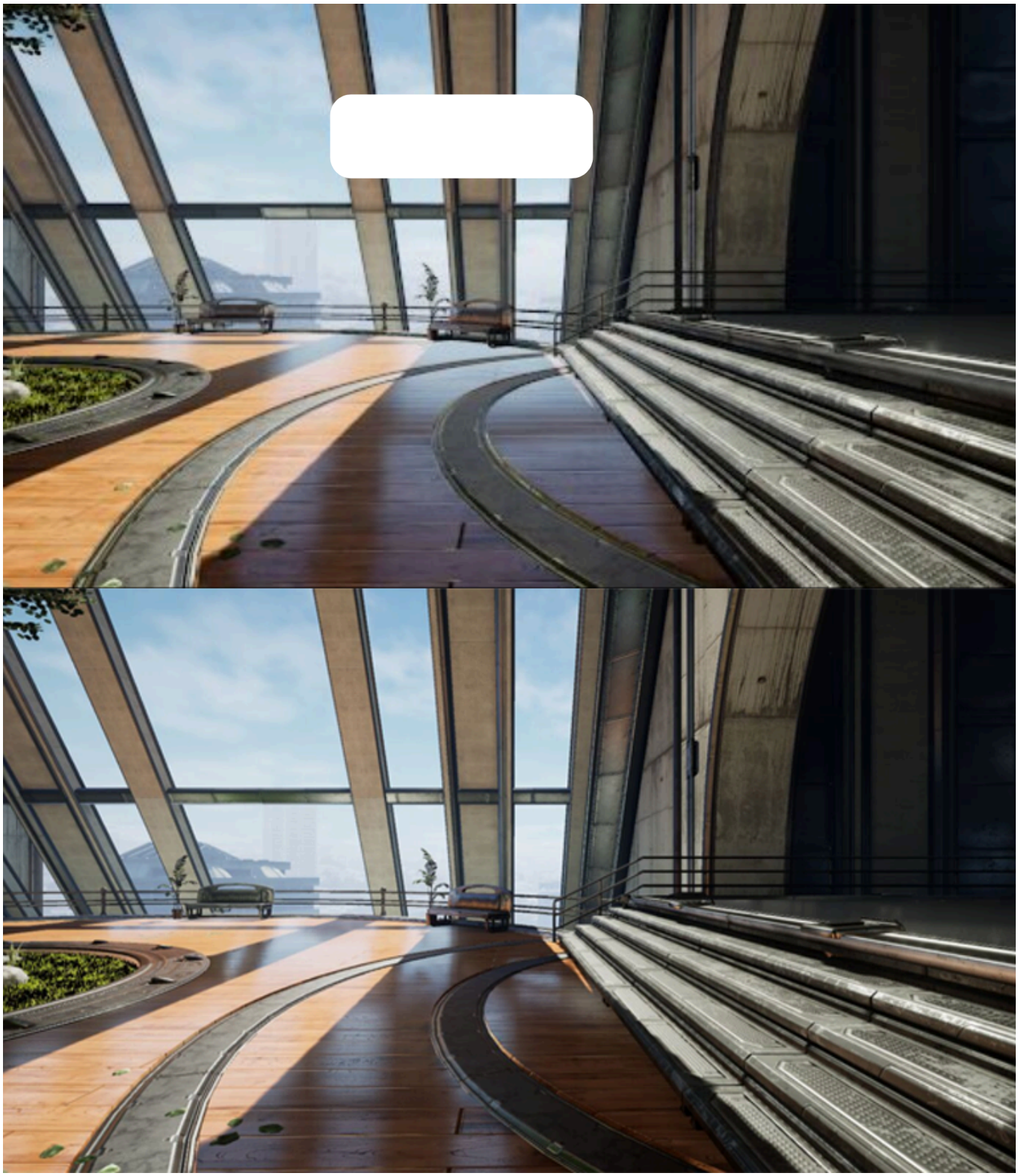
If you are on the Windows platform, the UE editor can also start the OpenGL, Vulkan, and Metal simulators to preview the effects in the editor, but the screen may be different from the actual running device, so you cannot rely entirely on this function .

> Before enabling Vulkan, you need to configure some parameters in the project. For details, see the official documentAndroid Vulkan Mobile Renderer .

In addition, UE removed OpenGL support under Windows in previous versions. Although there is still an OpenGL simulation option in the UE editor, the underlying layer is actually rendered with D3D.

12.2.2 Deferred Shading

UE's Deferred Shading is a feature added in 4.26, which enables developers to achieve more complex lighting effects on mobile devices, such as high-quality reflections, multiple dynamic lighting, decals, and advanced lighting features.



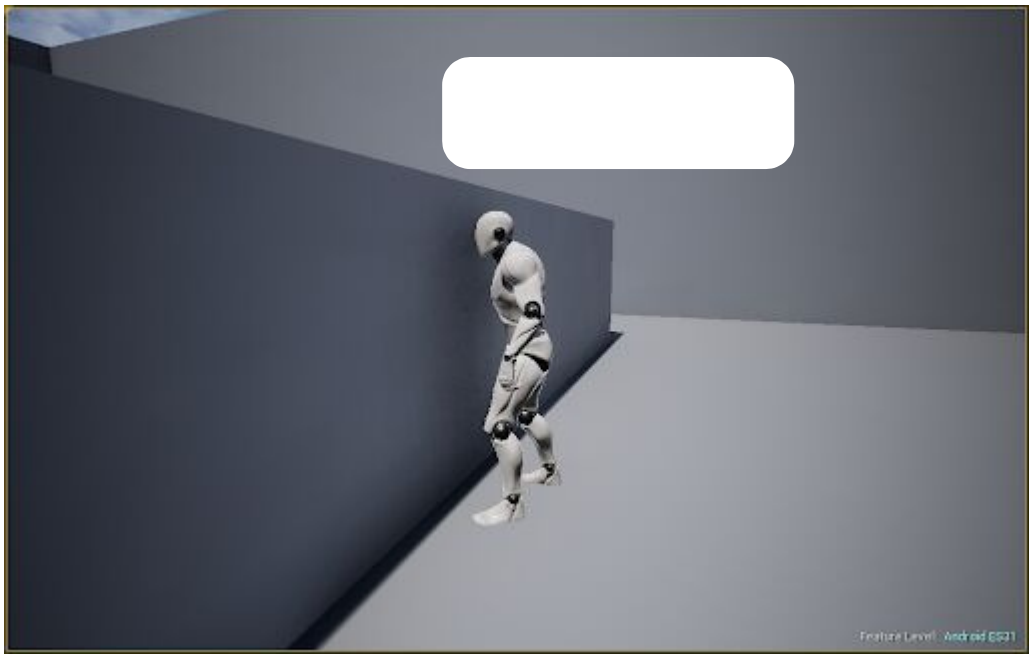
Top: forward rendering; bottom: deferred rendering.

If you want to enable deferred rendering on mobile devices, you need to add a field to **DefaultEngine.ini**in the project configuration directory`r.Mobile.ShadingPath=1`and then restart the editor.

12.2.3 Ground Truth Ambient Occlusion

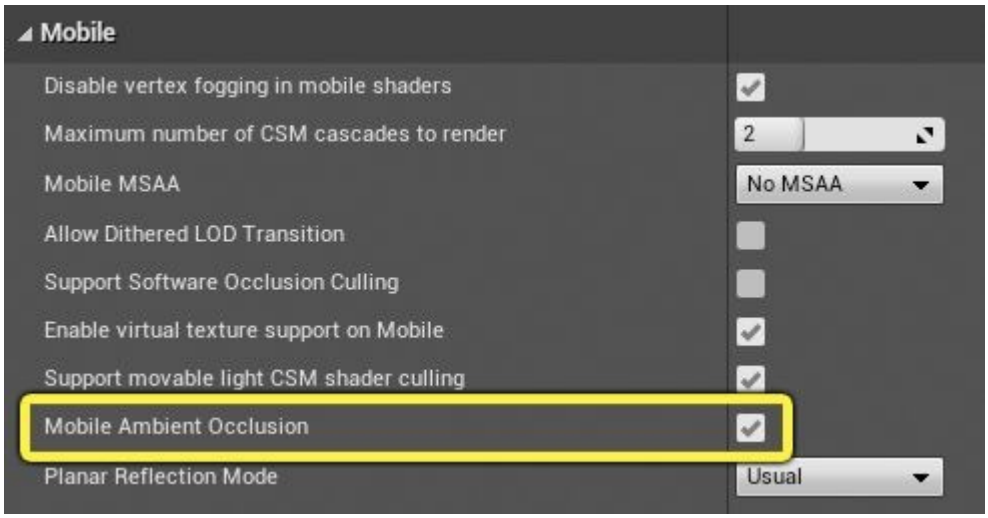
Ground Truth Ambient Occlusion (GTAO)is an ambient occlusion technology close to the real world. It is a compensation for shadows and can block part of the indirect lighting to obtain a good

soft shadow effect.



The GTAO effect is turned on. Note that when the robot approaches the wall, it will leave a soft shadow effect with a gradient on the wall.

To enable GTAO, check the following options:



In addition, GTAO relies on the Mobile HDR option. In order to enable it on the corresponding target device, you also need to add a field to the**[Platform]Scalability.ini**configuration r.Mobile.AmbientOcclusionQuality,and the value needs to be greater than 0, otherwise GTAO will be disabled.

It is worth noting that GTAO has performance issues on Mali devices because their maximum number of Compute Shader threads is less than 1024.

12.2.4 Dynamic Lighting and Shadow

The light source characteristics implemented by UE on the mobile terminal are:

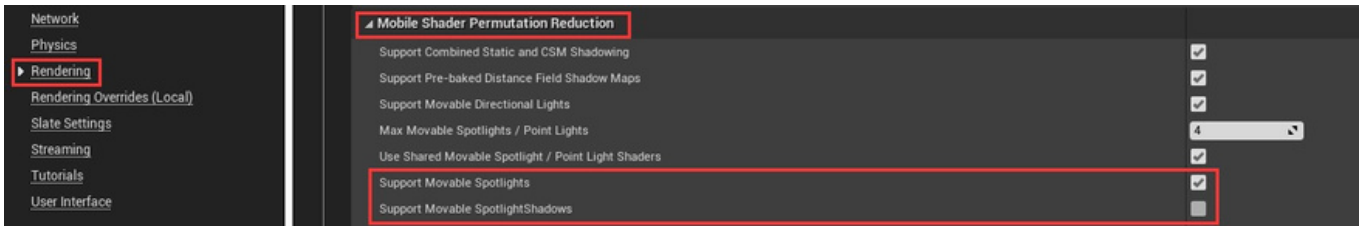
- Linear space HDR lighting.

- Directional light map (taking normals into account).
- Sun (directional light) supports distance field shadows + resolved specular highlights. IBL lighting:
- Each object samples the nearest reflection capturer, without parallax correction.
- Dynamic objects are properly lit and can cast shadows.

The types, quantities, shadows, and other information of dynamic light sources supported by the UE mobile terminal are as follows:

Light source type	Maximum number	shadow	describe
Directional Light	1	CSM	The default CSM level is 2, and a maximum of 4 levels are supported.
Point Light	4	Not supported	Point light shadows require a cube shadow map, and the technology of single-pass rendering cube shadows (OnePassPointLightShadow) requires GS (only available in SM5) to support it.
spotlight	4	support	Disabled by default, needs to be enabled in the project.
Area Light	0	Not supported	Dynamic area lighting effects are not currently supported.

Dynamic spotlights need to be explicitly enabled in the project configuration:



In the pixel shader of the mobile BasePass, the spotlight shadow map shares the same texture sampler as the CSM, and the spotlight shadow and CSM use the same shadow map atlas. The CSM is Guaranteed to have enough space, and the spotlight will be sorted by shadow resolution.

By default, the maximum number of visible shadows is capped at 8, but this can be

r.Mobile.MaxVisibleMovableSpotLightsShadowchanged by changing the value of . The resolution of spotlight shadows is based on the screen size andr.Shadow.TexelsPerPixelSpotlight.

In the forward rendering path, the total number of local lights (point lights and spot lights) cannot exceed 4.

The mobile terminal also supports a special shadow mode,**Modulated Shadows**,which can only be used for stationary parallel lights. The effect of opening modulated shadows is as follows:

Modulated shadows also support changing shadow color and blending ratio:

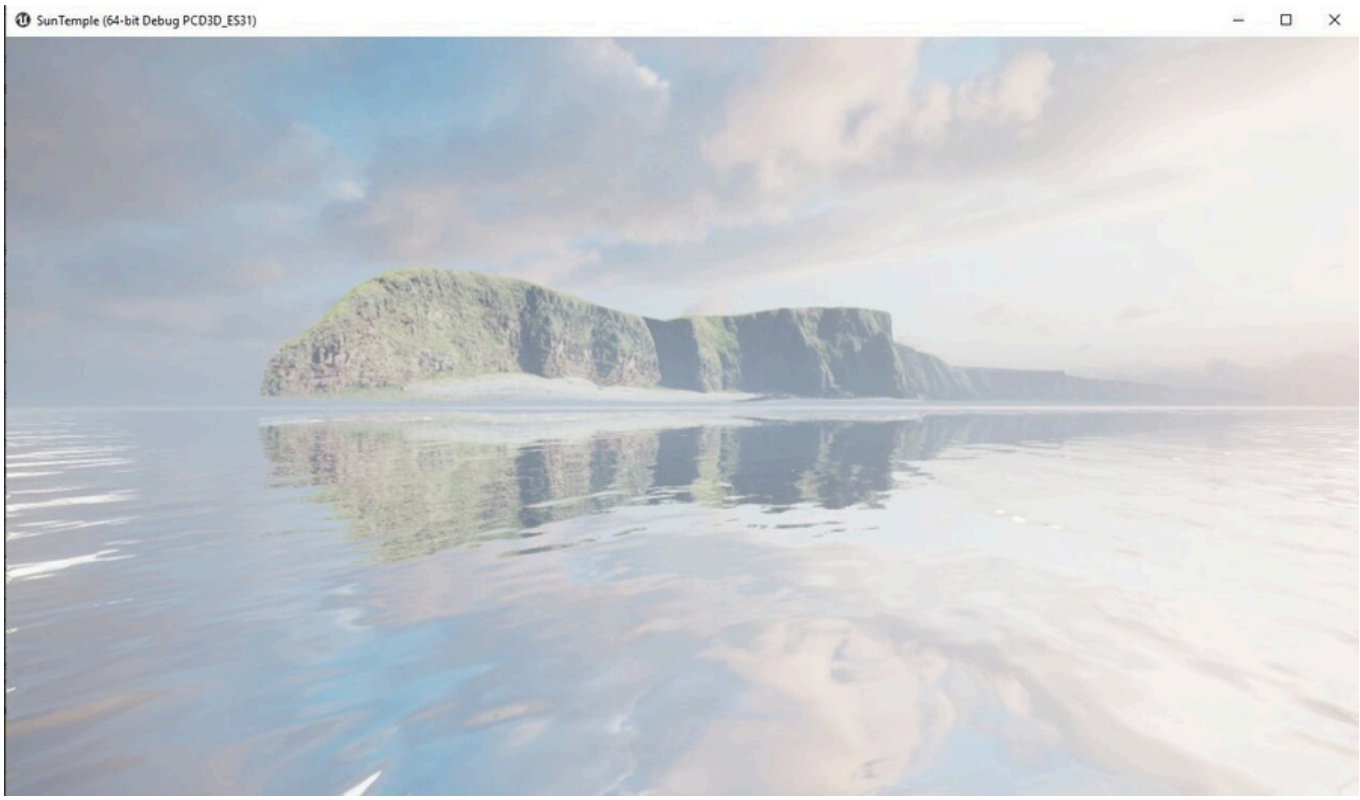
Left: Dynamic shadows; Right: Modulated shadows.

Mobile shadows also support settings for self-shadows, shadow quality level (r.shadowquality), depth offset, and other parameters.

In addition, the mobile terminal uses GGX specular reflection by default. If you want to switch to the traditional specular shading model, you can modify it in the following configuration:

12.2.5 Pixel Projected Reflection

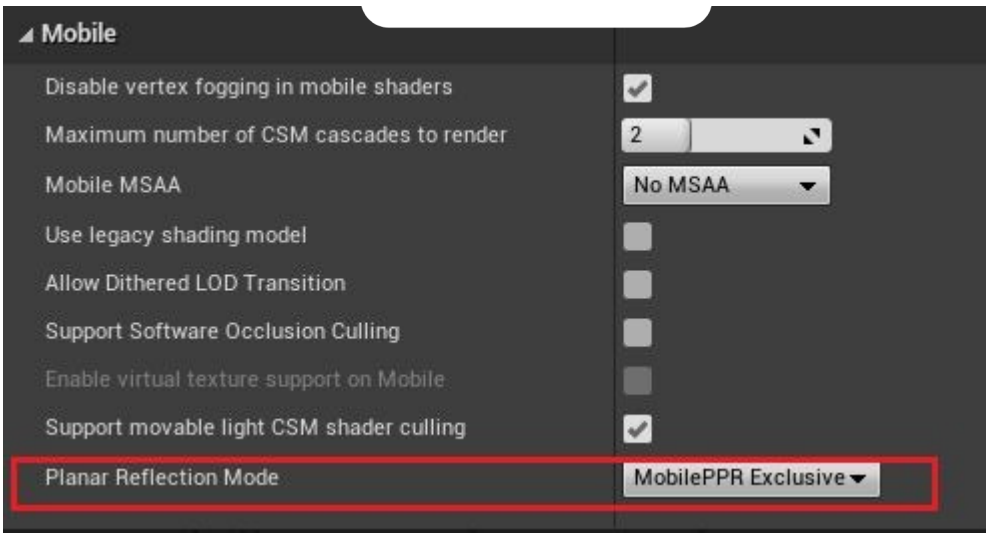
UE has made an optimized version of SSR for mobile terminals, called**Pixel Projected Reflection (PPR)**, which is also the core idea of reusing screen space pixels.



PPR renderings.

In order to enable the PPR effect, the following conditions must be met:

- Turn on the MobileHDR option.
- r.Mobile.PixelProjectedReflectionQualityThe value of is greater than 0.
- Set**Project Settings**>**Mobile**and set the**Planar Reflection Mode**to the correct mode:



Planar Reflection Modehas 3 options:

  - **Usual**:Planar Reflection Actors function the same way on all platforms. **MobilePPR**:Planar
  - Reflection Actors work normally on PC/Console platforms, but use PPR rendering on mobile platforms.
  - **MobilePPRExclusive**:Planar Reflection Actors will only be used for PPR on mobile platforms, leaving room for PC and Console projects to use traditional SSR.

By default, only high-end mobile devices will have this enabled in**[Project]Scalability.ini** r.Mobile.PixelProjectedReflectionQuality.

12.2.6 Mesh Auto-Instancing

The mesh drawing pipeline on PC already supports automatic instancing and merging of meshes, which can greatly improve rendering performance. 4.27 already supports this feature on mobile devices.

If you want to enable it, you need to open DefaultEngine.iniin the project configuration directory and add the following fields:

```
r.Mobile.SupportGPUScene=1
r.Mobile.UseGPUSceneTexture=1
```

Restart the editor and wait for the Shader to be compiled before you can preview the effect.

Since GPUSceneTexture support is required, and the maximum Uniform Buffer of Mali devices is only 64kb, which cannot support a large enough space, Mali devices use textures instead of buffers to store GPUScene data.

But there are some limitations:

- Auto-instancing on mobile devices primarily benefits CPU-intensive projects, not GPUintensive ones. While it's unlikely that enabling auto-instancing will hurt GPU-intensive projects, it's unlikely that you'll see significant performance improvements from using it.
- If a game or app requires a lot of memory,r.Mobile.UseGPUSceneTextureit may be more beneficial to turn it off and use the buffer, as it will not run properly on Mali devices.

> It can also be turned off for Mali devicesr.Mobile.UseGPUSceneTexture,while devices from other GPU manufacturers can be used normally.

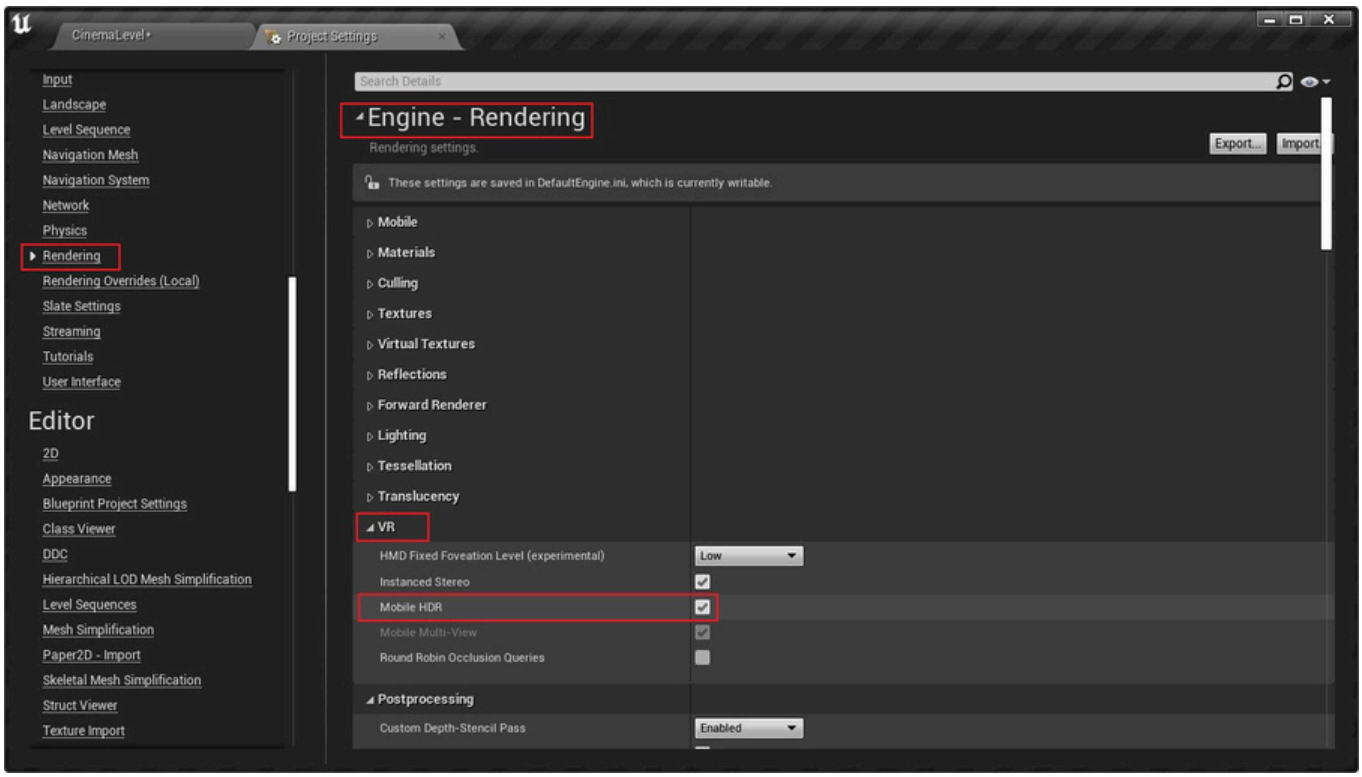
The effectiveness of auto-instancing depends very much on the exact specifications and targeting of your project, and it is recommended to create a build with auto-instancing enabled and profile it to determine if you will see substantial performance gains.

12.2.7 Post Processing

Due to the limitations of mobile devices such as slower dependent texture reads, limited hardware features, special hardware architectures, additional render target parsing, and limited bandwidth, post-processing on mobile devices is more performance-intensive and can even block the rendering pipeline in extreme cases.

Nevertheless, in some games or applications that require high image quality, they still rely heavily on the powerful expressiveness of post-processing to take high quality to a higher level. UE will not restrict developers from using post-processing.

In order to enable post-processing, you must first enable**the MobileHDR**option:



After turning on post-processing, you can set various post-processing effects in**the Post Process Volume .**

The post-processing supported on mobile devices includes Mobile Tonemapper, Color Grading, Lens, Bloom, Dirt Mask, Auto Exposure, Lens Flares, Depth of Field, etc.

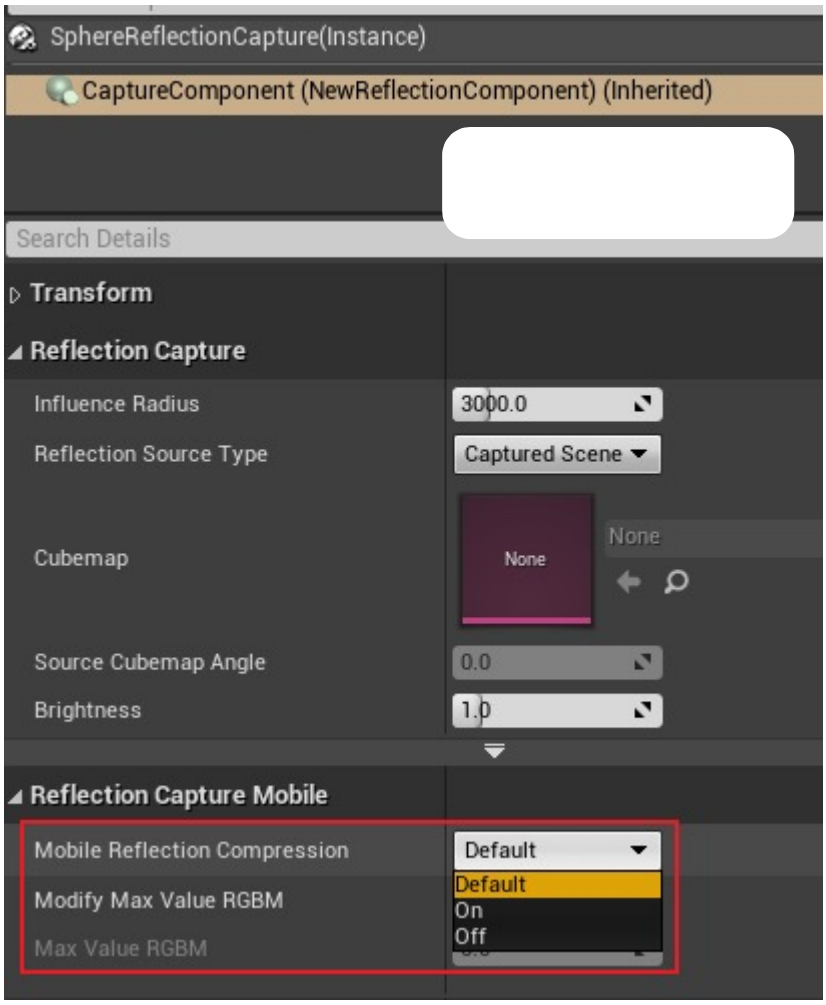
In order to achieve better performance, the official recommendation is to only enable Bloom and TAA on mobile devices.

12.2.8 Other Features and Limitations

- **Reflection Capture Compression**

The mobile terminal supports the compression of the Reflection Capture Component, which can reduce the memory and bandwidth of the Reflection Capture runtime and improve the rendering efficiency. It needs to be enabled in the project configuration:

After it is turned on, ETC2 compression is used by default. In addition, it can also be adjusted for each Reflection Capture Component:



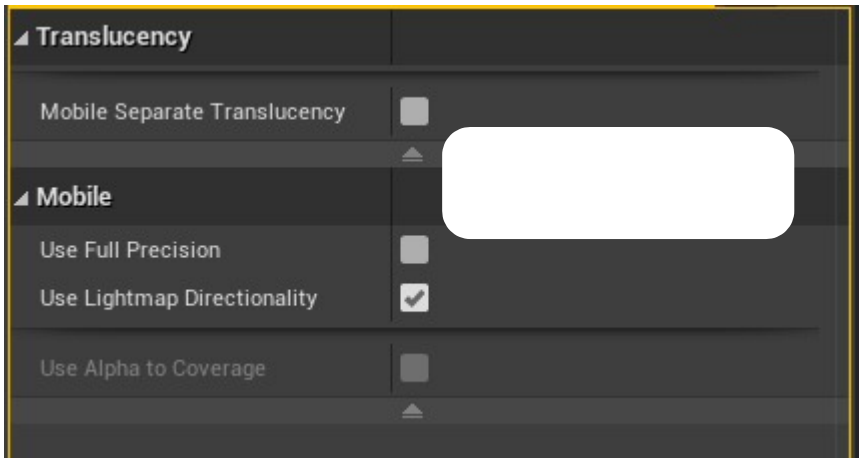
- **Material properties**

Materials on mobile platforms (feature level Open ES 3.1) use the same node-based creation process as other platforms, and the vast majority of nodes are supported on mobile.
The material properties supported on mobile platforms are:**BaseColor**,**Roughness**,**Metallic**, **Specular**, **Normal**,**Emissive**,**Refraction**,but Scene Color expressions, Tessellation inputs, and subsurface scattering shading models are not supported.

There are some limitations on the materials supported on mobile platforms:

- Due to hardware limitations, only 16 texture samplers can be used.
- Only DefaultLit and Unlit shading models are available.
- Custom UVs should be used to avoid relying on texture reads (no texture uv math).
- Translucent and Masked materials are extremely performance-intensive, and it is recommended to use opaque materials as much as possible.
- Depth fade can be used in translucent materials on iOS, but is not supported on platforms where the hardware does not support fetching data from the depth buffer and will incur an unacceptable performance cost.

The Material Properties panel has some special options for mobile devices:



These properties are described below:

- **Mobile Separate Translucency**:Whether to enable separate translucent rendering texture on mobile devices.
- **Use Full Precision**:Whether to use full precision. If not, it can reduce bandwidth usage and energy consumption and improve performance, but there may be artifacts in distant objects:

Left: Full-precision material. Right: Half-precision material with artifacts in the distant sun.

- **Use Lightmap Directionality**:Whether to enable the directionality of the light map. If checked, the direction of the light map and the pixel normal will be considered, but the performance consumption will be increased.
- **Use Alpha to Coverage**:Whether to enable MSAA anti-aliasing for Masked materials. If checked, MSAA will be enabled.
- **Fully Rough**:Whether it is completely rough. If checked, the rendering efficiency of this material will be greatly improved.

In addition, the grid types supported by the mobile terminal are:

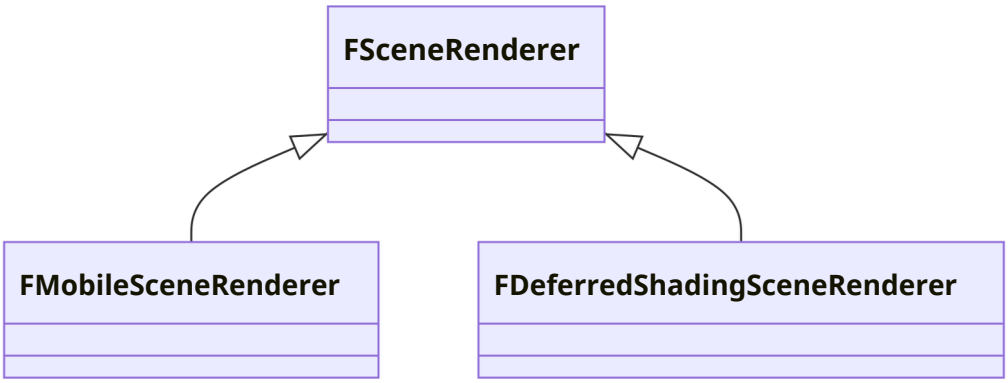
- **Skeletal Mesh**
- **Static Mesh**
- **Landscape**
- **CPU particle sprites, particle meshes**

Other types than the above are not supported. Other restrictions include:

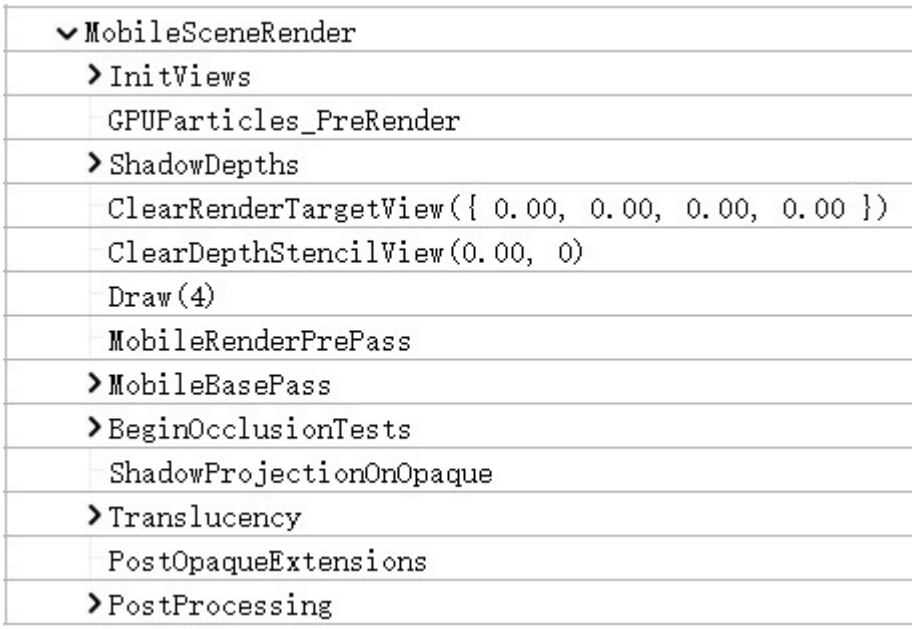
- A single mesh is limited to about 65k at most, since vertex indices are only 16 bits. The
- number of bones in a single Skeletal Mesh must be within 75 due to hardware performance limitations.

12.3 FMobileSceneRenderer

FMobileSceneRenderer inherits from FSceneRenderer, which is responsible for the scene rendering process on the mobile side, while the PC side isFDeferredShadingSceneRenderer, which also inherits from FSceneRenderer . Their inheritance relationship diagram is as follows:



The previous articles have mentioned FDeferredShadingSceneRenderer, whose rendering process is particularly complex, including complex lighting and rendering steps. In contrast, the logic and steps of FMobileSceneRenderer are much simpler. The following is a screenshot of RenderDoc:



The above mainly includes InitViews, ShadowDepths, PrePass, BasePass, OcclusionTest, ShadowProjectionOnOpaque, Translucency, PostProcessing and other steps. These steps exist on the PC side, but the implementation process may be different. See the analysis in the subsequent chapters.

12.3.1 Renderer Main Process

The main process of the mobile scene renderer also occurs in FMobileSceneRenderer::Render, the code and analysis are as follows:

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

void FMobileSceneRenderer::Render(FRHICommandListImmediate& RHICmdList) {

	//Update primitive scene information.
	Scene->UpdateAllPrimitiveSceneInfos(RHICmdList);

	//Prepare the view's rendering area.
	PrepareViewRectsForRendering(RHICmdList);

	//	Prepare sky and atmosphere data
	if	(ShouldRenderSkyAtmosphere(Scene, ViewFamily.EngineShowFlags))
	{
		for(int32 LightIndex =0; LightIndex < NUM_ATMOSPHERE_LIGHTS; ++LightIndex) {

			if(Scene->AtmosphereLights[LightIndex]) {

				PrepareSunLightProxy(*Scene->GetSkyAtmosphereSceneInfo(), LightIndex,
* Scene->AtmosphereLights[LightIndex]);
			}
		}
	}
	else
	{
		Scene->ResetAtmosphereLightsProperties();
	}

	if(!ViewFamily.EngineShowFlags.Rendering) {

		return;
	}

	//Waiting for occlusion culling test.
	WaitOcclusionTests(RHICmdList);
	FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
	RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

	//Initialize the view, find visible primitives, and prepare for renderingRTand
	buffer etc. InitViews(RHICmdList);

	if(GRHINeedsExtraDeletionLatency || !GRHICommandList.Bypass()) {

		QUICK_SCOPE_CYCLE_COUNTER(STAT_FMobileSceneRenderer_PostInitViewsFlushDel);
```

```
        //may pause occlusion queries, so it is best to letRHIThreads andGPUwork. In addition, when executingRHIWhen threading, this is the only place where pending deletions will be processed.
        FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();

FRHICommandListExecutor::GetImmediateCommandList().ImmediateFlush(EImmediateFlushType::FlushRHIThreadFlushResources);
    }

    GEngine->GetPreRenderDelegate().Broadcast();

    //Commit the global dynamic buffer before rendering starts.
    DynamicIndexBuffer.Commit(); DynamicVertexBuffer.Commit();
    DynamicReadBuffer.Commit();
    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);


    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_SceneSim));

    if(ViewFamily.bLateLatchingEnabled) {

        BeginLateLatching(RHICmdList);
    }

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

    //  Working with Virtual Textures
    if  (bUseVirtualTexturing)
    {
        SCOPED_GPU_STAT(RHICmdList,     VirtualTextureUpdate);

        FVirtualTextureSystem::Get().Update(RHICmdList, FeatureLevel, Scene);

        // Clear virtual texture feedback to default value
        FUnorderedAccessViewRHIRef FeedbackUAV =
        SceneContext.GetVirtualTextureFeedbackUAV();
        RHICmdList.Transition(FRHITransitionInfo(FeedbackUAV,                    ERHIAccess::SRVMask,
        ERHIAccess::UAVMask));
        RHICmdList.ClearUAVUint(FeedbackUAV, FUintVector4(~0u, ~0u, ~0u, ~0u));
        RHICmdList.Transition(FRHITransitionInfo(FeedbackUAV, ERHIAccess::UAVMask,
        ERHIAccess::UAVMask));
        RHICmdList.BeginUAVOverlap(FeedbackUAV);

    }

    //Sorted light source information.
    FSortedLightSetSceneInfo SortedLightSet;

    //  Deferred rendering.
    if  (bDeferredShading)
    {
        //Collect and sort light sources.
        GatherAndSortLights(SortedLightSet);

        int32 NumReflectionCaptures = Views[0].NumBoxReflectionCaptures + Views[0
        ].NumSphereReflectionCaptures;
        boolbCullLightsToGrid = (NumReflectionCaptures >0||
GMobileUseClusteredDeferredShading !=0);
        FRDGBuilderGraphBuilder(RHICmdList); //Calculate the
        light source grid.
```

```
        ComputeLightGrid(GraphBuilder, bCullLightsToGrid, SortedLightSet);
        GraphBuilder.Execute();
}

//Creating the sky/atmosphereLUT.
const boolbShouldRenderSkyAtmosphere = ShouldRenderSkyAtmosphere(Scene,
ViewFamily.EngineShowFlags);
if(bShouldRenderSkyAtmosphere) {

        FRDGBuilderGraphBuilder(RHICmdList);
        RenderSkyAtmosphereLookUpTables(GraphBuilder);
        GraphBuilder.Execute();
}

//Notify the special effects system that the scene is ready for rendering.
if(FXSystem && ViewFamily.EngineShowFlags.Particles) {

        FXSystem->PreRender(RHICmdList,NULL, !Views[0].bIsPlanarReflection); if
        (FGPUSortManager* GPUSortManager = FXSystem->GetGPUSortManager()) {

                GPUSortManager->OnPreRender(RHICmdList);
        }
}
//Poll for occlusion culling requests.
FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Shadows));

//Render shadows.
RenderShadowDepthMaps(RHICmdList);
FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

//Collects a list of views.
TArray<constFViewInfo*> ViewList;
for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

        ViewList.Add(&Views[ViewIndex]);
}

//    Rendering custom depth.
if    (bShouldRenderCustomDepth)
{
        FRDGBuilderGraphBuilder(RHICmdList); FSceneTextureShaderParameters SceneTextures
        CreateSceneTextureShaderParameters(GraphBuilder, Views=[0].GetFeatureLevel(),
        ESceneTextureSetupMode::None);

        RenderCustomDepthPass(GraphBuilder, SceneTextures);
        GraphBuilder.Execute();
}

//    Render DepthPrePass.
if    (bIsFullPrepassEnabled)
{
        //SDFandAONeed completePrePassdepth.

        FRHIRenderPassInfoDepthPrePassRenderPassInfo(
```

```
                SceneContext.GetSceneDepthSurface(),
                EDepthStencilTargetActions::ClearDepthStencil_StoreDepthStencil);

            DepthPrePassRenderPassInfo.NumOcclusionQueries =
            ComputeNumOcclusionQueriesToBatch();
            DepthPrePassRenderPassInfo.bOcclusionQueries =
DepthPrePassRenderPassInfo.NumOcclusionQueries !=0;

            RHICmdList.BeginRenderPass(DepthPrePassRenderPassInfo, TEXT("DepthPrepass"));

            RHICmdList.SetCurrentStat(GET_STATID(STAT_CLM_MobilePrePass));

            //Rendering full depthPrePass.
            RenderPrePass(RHICmdList);

            //Submit occlusion culling.
            RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Occlusion));
            RenderOcclusion(RHICmdList);

            RHICmdList.EndRenderPass();

            //  SDFshadow
            if  (bRequiresDistanceFieldShadowingPass)
            {
                CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderSDFShadowing);
                RenderSDFShadowing(RHICmdList);
            }

            //  HZB.
            if  (bShouldRenderHZB)
            {
                RenderHZB(RHICmdList,       SceneContext.SceneDepthZ);
            }

            //  AO.
            if  (bRequiresAmbientOcclusionPass)
            {
                RenderAmbientOcclusion(RHICmdList,          SceneContext.SceneDepthZ);
            }
        }

        FRHITexture* SceneColor = nullptr;

        //  Deferred rendering.
        if  (bDeferredShading)
        {
            SceneColor = RenderDeferred(RHICmdList, ViewList, SortedLightSet);
        }
        //  Forward rendering.
        else
        {
            SceneColor = RenderForward(RHICmdList, ViewList);
        }

        //  Rendering speed buffer.
        if  (bShouldRenderVelocities)
        {
            FRDGBuilderGraphBuilder(RHICmdList);
```

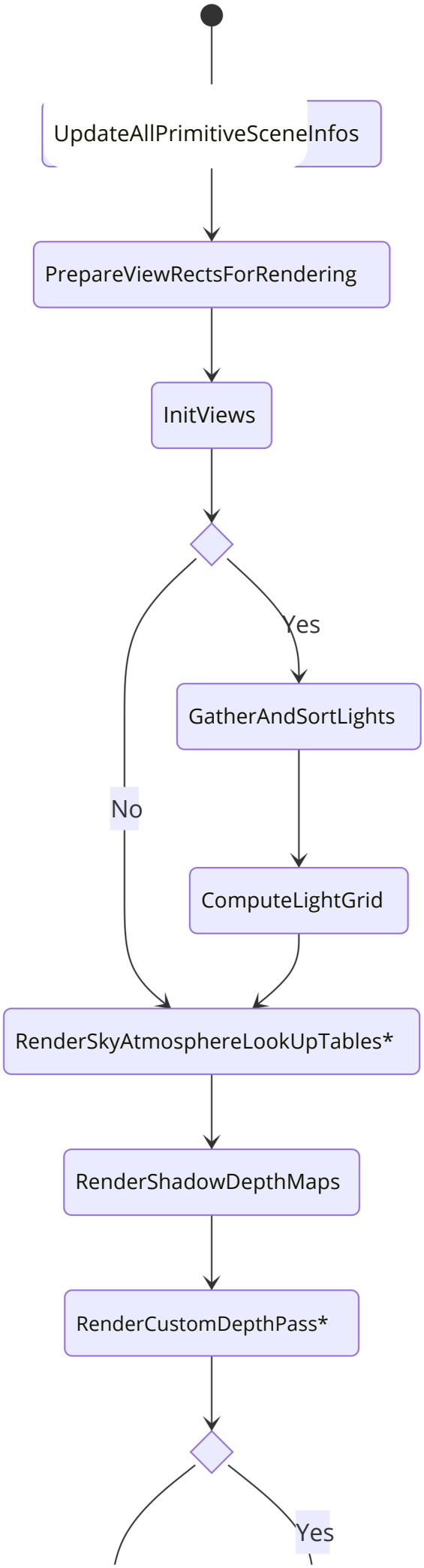
```
            FRDGTextureMSAA SceneDepthTexture = RegisterExternalTextureMSAA(GraphBuilder,
            SceneContext.SceneDepthZ);
            FRDGTextureRef VelocityTexture = TryRegisterExternalTexture(GraphBuilder,
            SceneContext.SceneVelocity);

            if(VelocityTexture != nullptr) {

                AddClearRenderTargetPass(GraphBuilder,              VelocityTexture);
            }

            //Renders a velocity buffer for movable objects.
            AddSetCurrentStatPass(GraphBuilder, GET_STATID(STAT_CLMM_Velocity));
            RenderVelocities(GraphBuilder, SceneDepthTexture.Resolve, VelocityTexture,
FSceneTextureShaderParameters(), EVelocityPass::Opaque,false);
            AddSetCurrentStatPass(GraphBuilder, GET_STATID(STAT_CLMM_AfterVelocity));

            //Velocity buffer for rendering transparent objects.
            AddSetCurrentStatPass(GraphBuilder, GET_STATID(STAT_CLMM_TranslucentVelocity));
            RenderVelocities(GraphBuilder, SceneDepthTexture.Resolve, VelocityTexture,
GetSceneTextureShaderParameters(CreateMobileSceneTextureUniformBuffer(GraphBuilder,
EMobileSceneTextureSetupMode::SceneColor) ), EVelocityPass::Translucent,false);

            GraphBuilder.Execute();
        }

        //Handles the logic after scene rendering.
        {
            FRendererModule& RendererModule = static_cast<FRendererModule&>
            (GetRendererModule());
            FRDGBuilderGraphBuilder(RHICmdList);
            RendererModule.RenderPostOpaqueExtensions(GraphBuilder, Views,                SceneContext);

            if(FXSystem && Views.IsValidIndex(0)) {

                AddUntrackedAccessPass(GraphBuilder,            [this](FRHICommandListImmediate&
RHICmdList)
                {
                    check(RHICmdList.IsOutsideRenderPass());

                    FXSystem->PostRenderOpaque(
                        RHICmdList,
                        Views[0].ViewUniformBuffer, nullptr,

                        nullptr,
                        Views[0].AllowGPUParticleUpdate()
                    ) ;
                    if(FGPUSortManager* GPUSortManager = FXSystem->GetGPUSortManager()) {

                        GPUSortManager->OnPostRenderOpaque(RHICmdList);
                    }
                });
            }
            GraphBuilder.Execute();
        }

        //  Flush/commit the command buffer.
        if  (bSubmitOffscreenRendering)
```

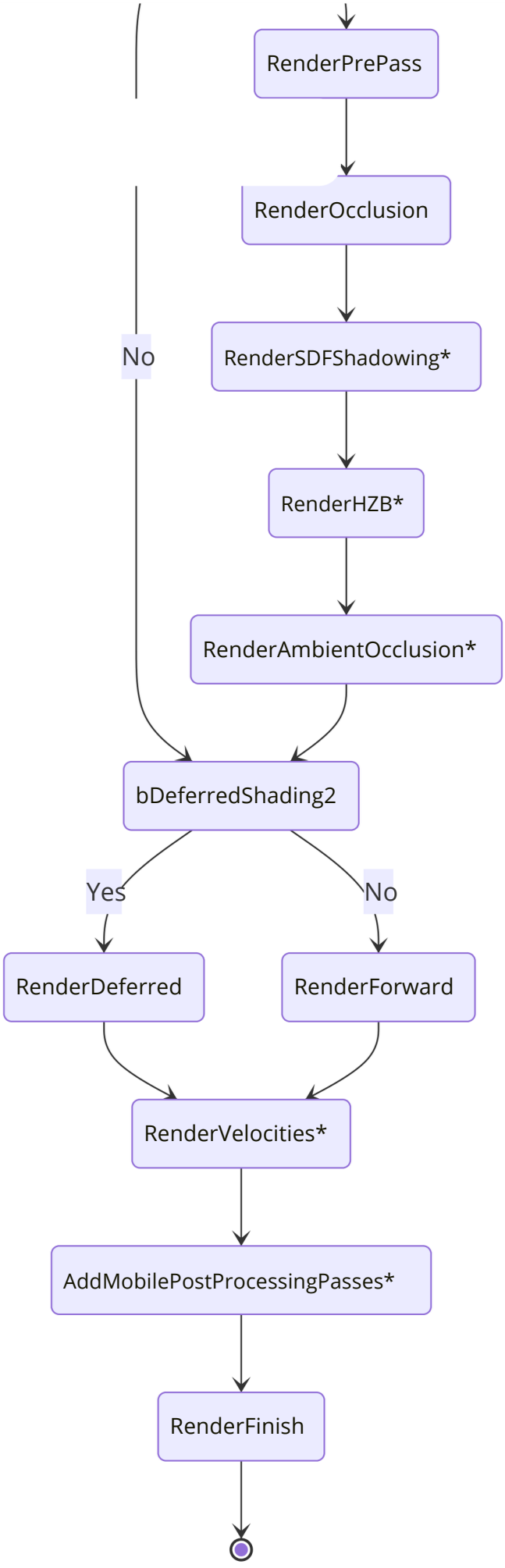
```
    {
        RHICmdList.SubmitCommandsHint();
        RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
    }

    //Convert scene colors toSRV,For subsequent reading. if(!
    bGammaSpace || bRenderToSceneColor) {

        RHICmdList.Transition(FRHITransitionInfo(SceneColor,                ERHIAccess::Unknown,
        ERHIAccess::SRVMask));
    }

    if(bDeferredShading)
    {
        //Releases the original reference on the scene render target.
        SceneContext.AdjustGBufferRefCount(RHICmdList,          - 1);
    }

    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Post));

    //  Working with virtual textures.
    if  (bUseVirtualTexturing)
    {
        SCOPED_GPU_STAT(RHICmdList,     VirtualTextureUpdate);

        // No pass after this should make VT page requests
        RHICmdList.EndUAVOverlap(SceneContext.VirtualTextureFeedbackUAV);
        RHICmdList.Transition(FRHITransitionInfo(SceneContext.VirtualTextureFeedbackUAV, ERHIAccess::UAVMask,
        ERHIAccess::SRVMask));

        TArray<FIntRect, TInlineAllocator<4>> ViewRects;
        ViewRects.AddUninitialized(Views.Num());
        for(int32 ViewIndex =0; ViewIndex < Views.Num(); ++ViewIndex) {

            ViewRects[ViewIndex] = Views[ViewIndex].ViewRect;
        }

        FVirtualTextureFeedbackBufferDesc Desc;
        Desc.Init2D(SceneContext.GetBufferSizeXY(),          ViewRects,
        SceneContext.GetVirtualTextureFeedbackScale());

        SubmitVirtualTextureFeedbackBuffer(RHICmdList,
SceneContext.VirtualTextureFeedback, Desc);
    }

    FMemMarkMark(FMemStack::Get()); FRDGBuider
    GraphBuilder(RHICmdList);

    FRDGTextureRef ViewFamilyTexture = TryCreateViewFamilyTexture(GraphBuilder, ViewFamily);


    //  Parsing the scene
    if  (ViewFamily.bResolveScene)
    {
        if(!bGammaSpace || bRenderToSceneColor) {

            //Complete rendering of each view or full stereo buffer (if enabled) {
```

```
                RDG_EVENT_SCOPE(GraphBuilder,"PostProcessing");
                SCOPE_CYCLE_COUNTER(STAT_FinishRenderViewTargetTime);

                TArray<TRDGUniformBufferRef<FMobileSceneTextureUniformParameters>,
TInlineAllocator<1, SceneRenderingAllocator>> MobileSceneTexturesPerView;
                MobileSceneTexturesPerView.SetNumZeroed(Views.Num());

                constautoSetupMobileSceneTexturesPerView = [&]() {

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

                        EMobileSceneTextureSetupMode SetupMode =
EMobileSceneTextureSetupMode::SceneColor;
                        if(Views[ViewIndex].bCustomDepthStencilValid) {

                            SetupMode |= EMobileSceneTextureSetupMode::CustomDepth;
                        }

                        if(bShouldRenderVelocities) {

                            SetupMode |= EMobileSceneTextureSetupMode::SceneVelocity;
                        }

                        MobileSceneTexturesPerView[ViewIndex] =
CreateMobileSceneTextureUniformBuffer(GraphBuilder, SetupMode);
                    }
                };

                SetupMobileSceneTexturesPerView();

                FMobilePostProcessingInputs PostProcessingInputs;
                PostProcessingInputs.ViewFamilyTexture = ViewFamilyTexture;

                //Rendering post-processing effects.
                for(int32 ViewIndex =0; ViewIndex < Views.Num(); ViewIndex++) {

                    RDG_EVENT_SCOPE_CONDITIONAL(GraphBuilder, Views.Num() >1,"View%d",
ViewIndex);
                    PostProcessingInputs.SceneTextures =
MobileSceneTexturesPerView[ViewIndex];
                    AddMobilePostProcessingPasses(GraphBuilder, Views[ViewIndex],
PostProcessingInputs, NumMSAASamples >1);
                }
            }
        }
    }

    GEngine->GetPostRenderDelegate().Broadcast();

    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_SceneEnd));

    if(bShouldRenderVelocities) {

        SceneContext.SceneVelocity.SafeRelease();
    }

    if(ViewFamily.bLateLatchingEnabled) {
```

```
            EndLateLatching(RHICmdList, Views[0]);
        }

        RenderFinish(GraphBuilder, ViewFamilyTexture);
        GraphBuilder.Execute();

        //Poll for occlusion culling requests.
        FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();

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

Those who have read the chapteron analyzing the Unreal rendering system (04) - Deferred rendering pipeline should know that the scene rendering process on mobile terminals has been simplified by many steps, which is equivalent to a subset of the scene renderer on PC terminals. Of
Of course, in order to adapt to the GPU hardware architecture unique to mobile terminals, the scene rendering on mobile terminals is also different from that on PC terminals. This will be analyzed in detail later. The main steps and processes of the mobile terminal scene are as follows:


UpdateAllPrimitiveSceneInfos
PrepareViewRectsForRendering
InitViews
Yes
GatherAndSortLights
No
ComputeLightGrid
RenderSkyAtmosphereLookUpTables*
RenderShadowDepthMaps
RenderCustomDepthPass*
Yes



Regarding the above flowchart, there are several points that need to be explained:

- The flowchart nodebDeferredShading andbDeferredShading2are the same variable. They are distinguished here mainly to prevent mermaidsyntax drawing errors.
- Nodes with * are conditional and not necessarily executed steps.

UE4.26 added a deferred rendering pipeline for mobile devices, so there are forward rendering branches RenderForwardand deferred rendering branches in the above codeRenderDeferred,and they both return the rendering result SceneColor.

The mobile terminal also supports rendering features such as primitive GPU scenes, SDF shadows, AO, sky atmosphere, virtual textures, and occlusion culling.

Since UE4.26, the rendering system has widely usedthe RDG system , and the mobile scene renderer is no exception. In the above code, a total of several FRDGBuilder instances are declared, which are used to calculate the light grid, render the sky atmosphere LUT, custom depth, speed buffer, render post events, post-processing, etc. They are relatively independent functional modules or rendering stages.

12.3.2 RenderForward

RenderForwardThe branch responsible for forward rendering in the mobile scene renderer has the Following code and analysis:

```
FRHITexture*FMobileSceneRenderer::RenderForward(FRHICommandListImmediate& RHICmdList, constTArrayView<
constFViewInfo*> ViewList) {

	constFViewInfo& View = *ViewList[0];
	FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);

	FRHITexture* SceneColor = nullptr; FRHITexture*
	SceneColorResolve = nullptr; FRHITexture* SceneDepth
	= nullptr;
	ERenderTargetActions ColorTargetAction = ERenderTargetActions::Clear_Store;
	EDepthStencilTargetActions DepthTargetAction =
EDepthStencilTargetActions::ClearDepthStencil_DontStoreDepthStencil;

	//Whether to enable mobile terminalMSAA.
	boolbMobileMSAA = NumMSAASamples >1&& SceneContext.GetSceneColorSurface()-
> GetNumSamples() >1;

	//Whether to enable multi-view mode on mobile terminals.
	static constautoCVarMobileMultiView = IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT(
"vr.MobileMultiView"));
	const boolbIsMultiViewApplication = (CVarMobileMultiView && CVarMobileMultiView-
> GetValueOnAnyThread() !=0);

	// gammaRendering branch of space.
	if(bGammaSpace && !bRenderToSceneColor) {

		// If turned onMSAA,FromSceneContextGet the rendered texture (including scene color and resolved
		if texture) (bMobileMSAA)
		{
			SceneColor = SceneContext.GetSceneColorSurface();
			SceneColorResolve	=ViewFamily.RenderTarget->GetRenderTargetTexture();
			ColorTargetAction	= ERenderTargetActions::Clear_Resolve;
			RHICmdList.Transition(FRHITransitionInfo(SceneColorResolve,
ERHIAccess::Unknown, ERHIAccess::RTV | ERHIAccess::ResolveDst));
		}
```

```
            //NonMSAA, get the render texture from the view family.
            else
            {
                SceneColor = ViewFamily.RenderTarget->GetRenderTargetTexture();
                RHICmdList.Transition(FRHITransitionInfo(SceneColor, ERHIAccess::Unknown,
ERHIAccess::RTV));
            }
            SceneDepth = SceneContext.GetSceneDepthSurface();
        }
        //Linear space or render to a scene texture.
        else
        {
            SceneColor = SceneContext.GetSceneColorSurface(); if
            (bMobileMSAA)
            {
                SceneColorResolve       =SceneContext.GetSceneColorTexture();
                ColorTargetAction       = ERenderTargetActions::Clear_Resolve;
                RHICmdList.Transition(FRHITransitionInfo(SceneColorResolve,
ERHIAccess::Unknown, ERHIAccess::RTV | ERHIAccess::ResolveDst));
            }
            else
            {
                SceneColorResolve       =  nullptr;
                ColorTargetAction       =  ERenderTargetActions::Clear_Store;
            }

            SceneDepth = SceneContext.GetSceneDepthSurface();

            if(bRequiresMultiPass) {

                // store targets after opaque so translucency render pass can be restarted ColorTargetAction
                                    = ERenderTargetActions::Clear_Store;
                DepthTargetAction       =
EDepthStencilTargetActions::ClearDepthStencil_StoreDepthStencil;
            }

            if(bKeepDepthContent)
            {
                // store depth if post-processing/capture needs it
                DepthTargetAction =
EDepthStencilTargetActions::ClearDepthStencil_StoreDepthStencil;
            }
        }

        //    prepassThe depth texture state of the .
        if    (bIsFullPrepassEnabled)
        {
            ERenderTargetActions DepthTarget            =
MakeRenderTargetActions(ERenderTargetLoadAction::ELoad,
GetStoreAction(GetDepthActions(DepthTargetAction)));
            ERenderTargetActions StencilTarget =
MakeRenderTargetActions(ERenderTargetLoadAction::ELoad,
GetStoreAction(GetStencilActions(DepthTargetAction)));
            DepthTargetAction = MakeDepthStencilTargetActions(DepthTarget, StencilTarget);
        }

        FRHITexture* ShadingRateTexture = nullptr;
```

```
        if(!View.bIsSceneCapture && !View.bIsReflectionCapture) {

            TRefCountPtr<IPooledRenderTarget> ShadingRateTarget =
            GVRSImageManager.GetMobileVariableRateShadingImage(ViewFamily);
            if(ShadingRateTarget.IsValid()) {

                ShadingRateTexture = ShadingRateTarget-
> GetRenderTargetItem().ShaderResourceTexture; }

        }

        //Scene color renderingPassinformation. FRHIRenderPassInfo
        SceneColorRenderPassInfo(
            SceneColor,
            ColorTargetAction,
            SceneColorResolve,
            SceneDepth,
            DepthTargetAction,
            nullptr,// we never resolve scene depth on mobile
            ShadingRateTexture,
            VRSRB_Sum,
            FExclusiveDepthStencil::DepthWrite_StencilWrite
        ) ;
        SceneColorRenderPassInfo.SubpassHint = ESubpassHint::DepthReadSubpass; if(!
        bIsFullPrepassEnabled) {

            SceneColorRenderPassInfo.NumOcclusionQueries =
            ComputeNumOcclusionQueriesToBatch();
            SceneColorRenderPassInfo.bOcclusionQueries =
SceneColorRenderPassInfo.NumOcclusionQueries !=0;
        }
        //If the scene color is not multi-view, but the application is, you need to render the multi-view to the shader as a
        single view. SceneColorRenderPassInfo.MultiViewCount = View.bIsMobileMultiViewEnabled?2:
(bIsMultiViewApplication ?1:0);

        //Start rendering scene colors.
        RHICmdList.BeginRenderPass(SceneColorRenderPassInfo, TEXT("SceneColorRendering"));

        if(GIsEditor && !View.bIsSceneCapture) {

            DrawClearQuad(RHICmdList, Views[0].BackgroundColor);
        }

        if(!bIsFullPrepassEnabled) {

            RHICmdList.SetCurrentStat(GET_STATID(STAT_CLM_MobilePrePass)); //Render Depth
            pre-pass
            RenderPrePass(RHICmdList);
        }

        RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Opaque)); //RenderingBasePass:
        Opaque andmaskedobject. RenderMobileBasePass(RHICmdList, ViewList);
        RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);



        //Rendering debug mode.
#if!(UE_BUILD_SHIPPING || UE_BUILD_TEST) if
  (ViewFamily.UseDebugViewPS())
```

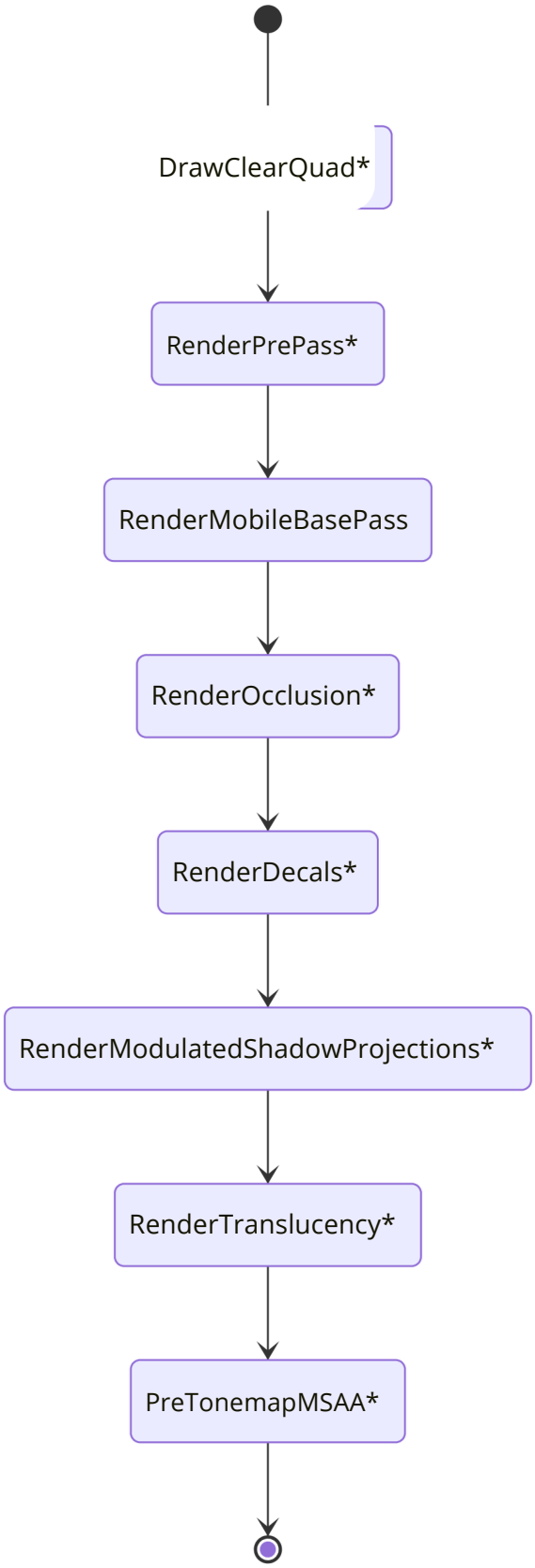
```
        {
            // Here we use the base pass depth result to get z culling for opaque and masque. // The color needs to be
            cleared at this point since shader complexity renders in additive.

            DrawClearQuad(RHICmdList, FLinearColor::Black); RenderMobileDebugView(RHICmdList,
            ViewList); RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

        }
#endif// !(UE_BUILD_SHIPPING || UE_BUILD_TEST)

        const boolbAdrenoOcclusionMode =
        CVarMobileAdrenoOcclusionMode.GetValueOnRenderThread()          !=   0;
        if(!bIsFullPrepassEnabled) {

            //    Occlusion Culling
            if    (!bAdrenoOcclusionMode)
            {
                //Submit occlusion culling
                RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Occlusion));
                RenderOcclusion(RHICmdList);
                RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
            }
        }

        //Post-event, handles plugin rendering.
        {
            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(ViewExtensionPostRenderBasePass);

QUICK_SCOPE_CYCLE_COUNTER(STAT_FMobileSceneRenderer_ViewExtensionPostRenderBasePass);
            for(int32 ViewExt =0; ViewExt < ViewFamily.ViewExtensions.Num(); ++ViewExt) {

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

                    ViewFamily.ViewExtensions[ViewExt]-
> PostRenderBasePass_RenderThread(RHICmdList, Views[ViewIndex]);
                }
            }
        }

        //If you need to render reflections of transparent objects or pixel projections, you need to splitpass.
        if(bRequiresMultiPass || bRequiresPixelProjectedPlanarRelfetionPass) {

            RHICmdList.EndRenderPass();
        }

        RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Translucency));

        //If necessary, re-enable the transparent rendering pass.
        if(bRequiresMultiPass || bRequiresPixelProjectedPlanarRelfetionPass) {

            check(RHICmdList.IsOutsideRenderPass());

            //If the current hardware does not support reading and writing the same depth buffer,
            the scene depth is copied. ConditionalResolveSceneDepth(RHICmdList, View);

            if(bRequiresPixelProjectedPlanarRelfectionPass) {

                constFPlanarReflectionSceneProxy* PlanarReflectionSceneProxy = Scene? Scene-
```

```
> GetForwardPassGlobalPlanarReflection() : nullptr;
                RenderPixelProjectedReflection(RHICmdList, SceneContext,
PlanarReflectionSceneProxy);

                FRHITransitionInfo TranslucentRenderPassTransitions[] =
                { FRHITransitionInfo(SceneColor,        ERHIAccess::SRVMask,    ERHIAccess::RTV),
                FRHITransitionInfo(SceneDepth, };       ERHIAccess::SRVMask,    ERHIAccess::DSVWrite)

                RHICmdList.Transition(MakeArrayView(TranslucentRenderPassTransitions,
UE_ARRAY_COUNT(TranslucentRenderPassTransitions)));
        }

        DepthTargetAction =
EDepthStencilTargetActions::LoadDepthStencil_DontStoreDepthStencil;
        FExclusiveDepthStencil::Type ExclusiveDepthStencil =
        FExclusiveDepthStencil::DepthRead_StencilRead;
        if(bModulatedShadowsInUse) {

                ExclusiveDepthStencil = FExclusiveDepthStencil::DepthRead_StencilWrite;
        }

        //Opaque grids used for mobile pixel projection reflections must have depth written to depthRT,Because the mesh is rendered only once (if the quality level is lower than
or equal toBestPerformance).
        if(IsMobilePixelProjectedReflectionEnabled(View.GetShaderPlatform())
                && GetMobilePixelProjectedReflectionQuality() ==
EMobilePixelProjectedReflectionQuality::BestPerformance)
        {
                ExclusiveDepthStencil = FExclusiveDepthStencil::DepthWrite_StencilWrite;
        }

        if(bKeepDepthContent && !bMobileMSAA) {

                DepthTargetAction =
EDepthStencilTargetActions::LoadDepthStencil_StoreDepthStencil;
        }

#ifPLATFORM_HOLOLENS
        if(bShouldRenderDepthToTranslucency) {

                ExclusiveDepthStencil = FExclusiveDepthStencil::DepthWrite_StencilWrite;
        }
#endif

        //Transparent object renderingPass.
        FRHIRenderPassInfo      TranslucentRenderPassInfo(
                SceneColor,
                SceneColorResolve ? ERenderTargetActions::Load_Resolve :
ERenderTargetActions::Load_Store,
                SceneColorResolve,
                SceneDepth,
                DepthTargetAction,
                nullptr,
                ShadingRateTexture,
                VRSRB_Sum,
                ExclusiveDepthStencil

        ) ;
        TranslucentRenderPassInfo.NumOcclusionQueries           =0;
        TranslucentRenderPassInfo.bOcclusionQueries =           false;
```

```
            TranslucentRenderPassInfo.SubpassHint = ESubpassHint::DepthReadSubpass;

            //Start rendering semi-transparent objects.
            RHICmdList.BeginRenderPass(TranslucentRenderPassInfo, TEXT(
"SceneColorTranslucencyRendering"));
        }

        //The scene depth is read-only and can be obtained.
        RHICmdList.NextSubpass();

        if(!View.bIsPlanarReflection) {

            // Rendering decals.
            if (ViewFamily.EngineShowFlags.Decals)
            {
                CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderDecals);
                RenderDecals(RHICmdList);
            }

            // Rendering modulated shadow casting.
            if (ViewFamily.EngineShowFlags.DynamicShadows)
            {
                CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderShadowProjections);
                RenderModulatedShadowProjections(RHICmdList);
            }
        }

        // Draw semi-transparently.
        if (ViewFamily.EngineShowFlags.Translucency)
        {
            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderTranslucency);
            SCOPE_CYCLE_COUNTER(STAT_TranslucencyDrawTime);

            RenderTranslucency(RHICmdList, ViewList);

            FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
            RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
        }

        if(!bIsFullPrepassEnabled) {

            // AdrenoOcclusion culling mode.
            if (bAdrenoOcclusionMode)
            {
                RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Occlusion)); // flush

                RHICmdList.SubmitCommandsHint(); bSubmitOffscreenRendering
                =false;// submit once // Issue occlusion queries

                RenderOcclusion(RHICmdList);
                RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
            }
        }

        // existMSAAPrecompute the tone map before parsing (only iniOS
        if efficient) (!bGammaSpace)
        {
            PreTonemapMSAA(RHICmdList);
```

```
        }

        //End scene color rendering.
        RHICmdList.EndRenderPass();

        //Optimize the parsed texture that returns the scene color (turned onMSAAOnly then).
        returnSceneColorResolve ? SceneColorResolve : SceneColor;
}
```

The main steps of forward rendering on mobile devices are similar to those on PC devices, which include rendering PrePass, BasePass, special rendering (decals, AO, occlusion culling, etc.), and translucent objects in sequence. Their flowcharts are as follows:



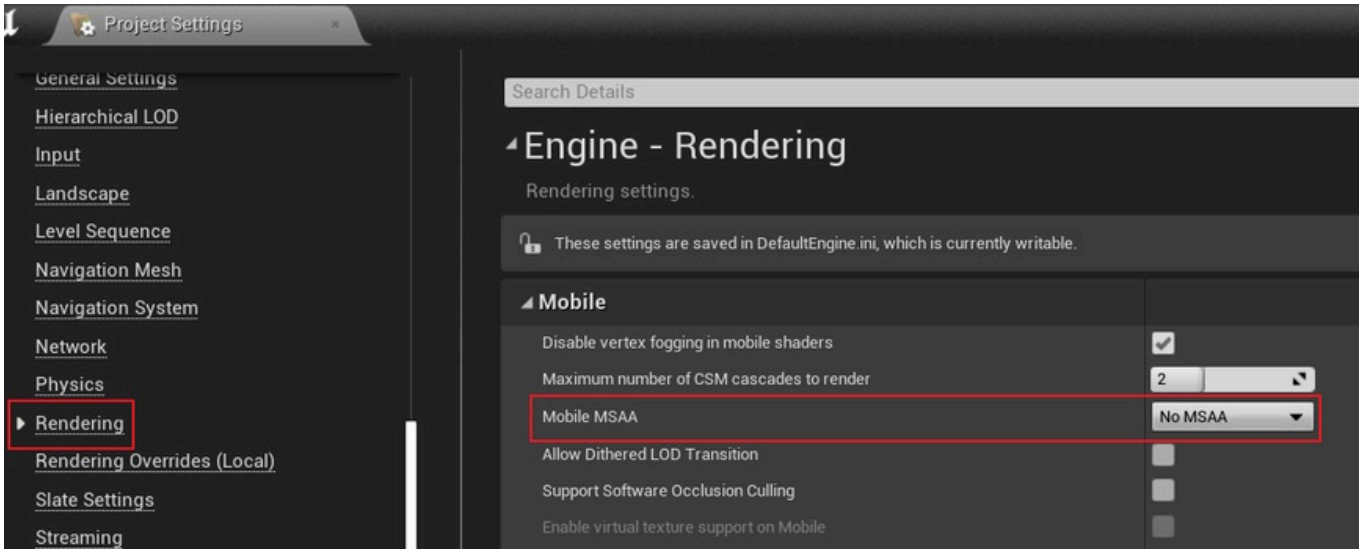
Among them, occlusion culling is related to GPU manufacturers. For example, Qualcomm Adreno series GPU chips require it between Flush rendering instructions and Switch FBO:

> Render Opaque -> Render Translucent -> Flush ->**Render Queries**->Switch FBO

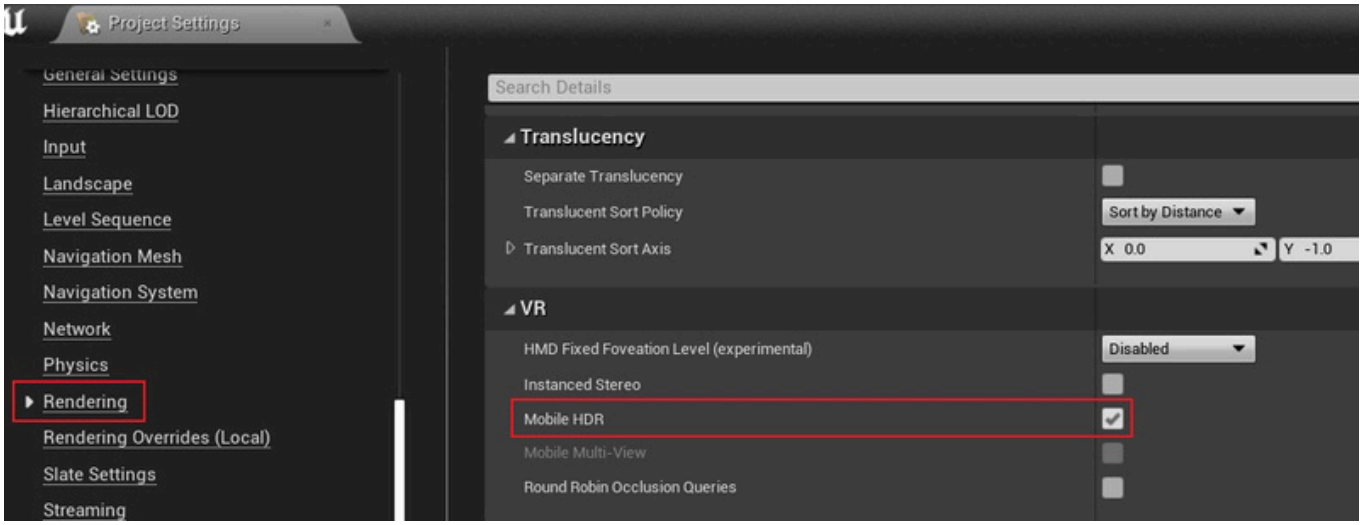
Then UE also complies with the special requirements of the Adreno series chips and makes special treatment for its occlusion culling.

> Adreno series chips support TBDR architecture Bin and ordinary Direct mixed mode rendering, and will automatically switch to Direct mode during occlusion query to reduce the overhead of occlusion query. If the query is not submitted between the Flush rendering instruction and the Switch FBO, the entire rendering pipeline will be stuck, causing rendering performance to degrade.

MSAA is the preferred anti-aliasing for UE's forward rendering on mobile devices due to its natural hardware support and good balance between effect and efficiency. Therefore, there are a lot of logic for processing MSAA in the above code, including color and depth textures and their resource status.
If MSAA is turned on, the default is toRHICmdList.EndRenderPass()parse the scene color (while writing the data on the chip blocks back to the system video memory), thereby obtaining an antialiasing texture. MSAA on mobile devices is not turned on by default, but can be set in the following interface:



Forward rendering supports two color space modes: Gamma space and HDR (linear space). If it is linear space, tone mapping and other steps are required in the post-rendering process. The default is HDR, which can be changed in the project configuration:

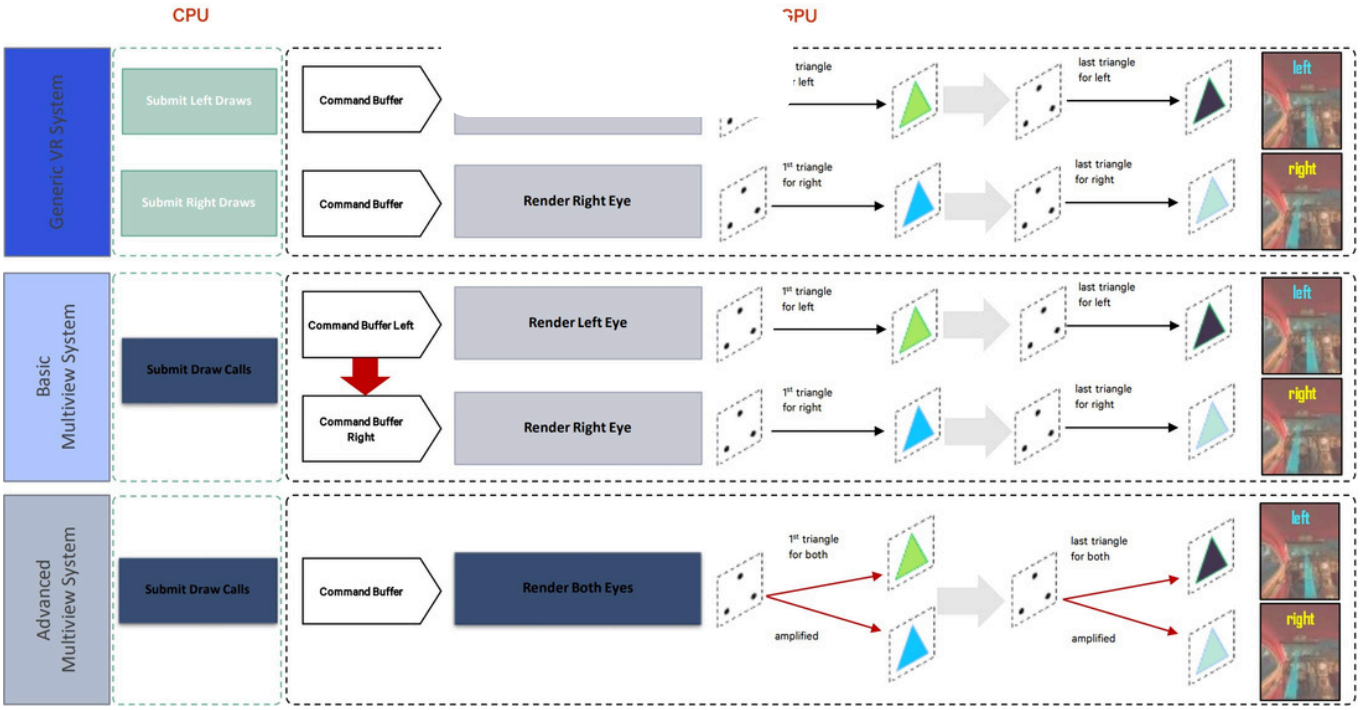


The bRequiresMultiPass in the above code indicates whether a dedicated rendering Pass is required to draw translucent objects. Its value is determined by the following code:

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

bool FMobileSceneRenderer::RequiresMultiPass(FRHICommandListImmediate& RHICmdList,const FViewInfo& View)
const
{
	// Vulkan uses subpasses
	if(IsVulkanPlatform(ShaderPlatform)) {

		returnfalse;
	}

	// All iOS support frame_buffer_fetch if
	(IsMetalMobilePlatform(ShaderPlatform)) {

		returnfalse;
	}

	if(IsMobileDeferredShadingEnabled(ShaderPlatform)) {

		//TODO:add GL support return
		true;
	}

	// Some Androids support frame_buffer_fetch
	if(IsAndroidOpenGLESPlatform(ShaderPlatform) && (GSupportsShaderFramebufferFetch ||
	GSupportsShaderDepthStencilFetch))
	{
		returnfalse;
	}

	// Always render reflection capture in single pass if
	(View.bIsPlanarReflection || View.bIsSceneCapture) {

		returnfalse;
	}

	// Always render LDR in single pass if(!
	IsMobileHDR())
	{
		returnfalse;
	}

	// MSAA depth can't be sampled or resolved, unless we are on PC (no vulkan) if(NumMSAASamples >1
	&& !IsSimulatedPlatform(ShaderPlatform)) {

		returnfalse;
	}

	returntrue;
}
```

Similar but different in meaning are the bIsMultiViewApplication and bIsMobileMultiViewEnabled flags, which indicate whether multi-view rendering is enabled and the number of multi-views. Only used for VR, vr.MobileMultiViewdetermined by factors such as console variables and graphics API.

MultiView is used for XR to optimize the situation of rendering twice, and it has two modes: Basic and Advanced:



Comparison chart of MultiView for optimizing VR rendering. Top: Rendering without MultiView mode, each eye submits drawing commands separately; Middle: Basic MultiView mode, reuses submitted commands, and copies an extra Command List at the GPU layer; Bottom: Advanced MultiView mode, which can reuse DC, Command List, and geometry information.

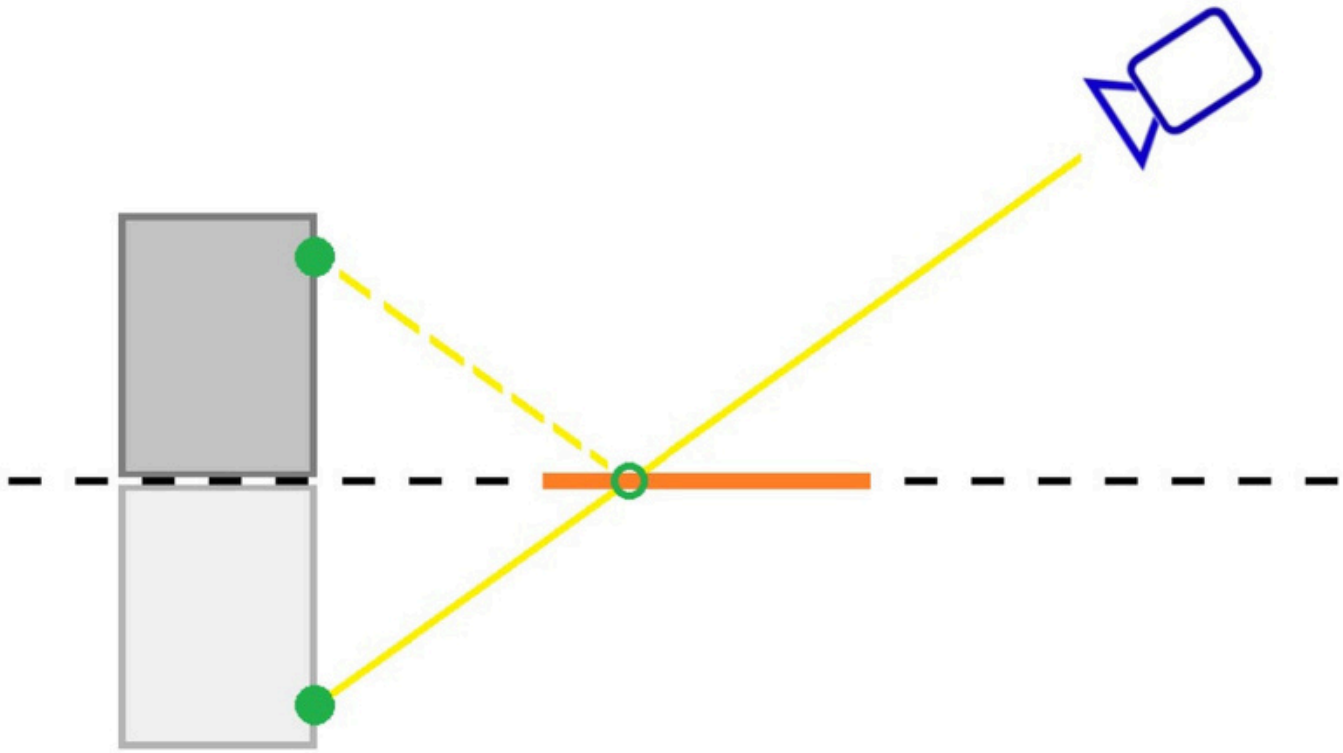
bKeepDepthContent indicates whether to keep the depth content. The code to determine it is:

```
bKeepDepthContent =
        bRequiresMultiPass        ||
        bForceDepthResolve        ||
        bRequiresPixelProjectedPlanarRelfetionPass                    ||
        bSeparateTranslucencyActive ||
        Views[0].bIsReflectionCapture || (bDeferredShading &&
        bPostProcessUsesSceneDepth) || bShouldRenderVelocities ||


        bIsFullPrepassEnabled;

//bringMSAAThe depth is never preserved.
bKeepDepthContent = (NumMSAASamples >1?false: bKeepDepthContent);
```

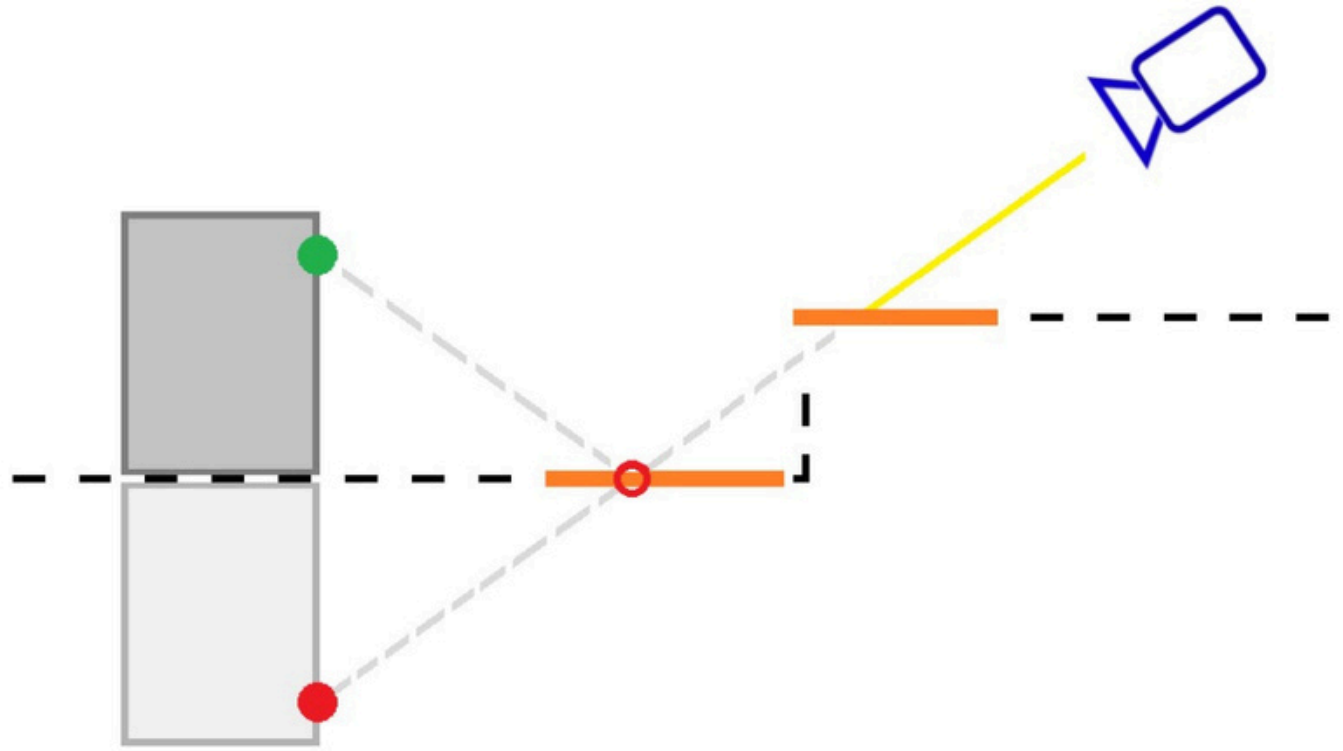
The above code also reveals a special rendering method of planar reflection on mobile terminals: **Pixel Projected Reflection (PPR)**.Its implementation principle is similar to SSR, but it requires less data, only scene color, depth buffer and reflection area. Its core steps are:

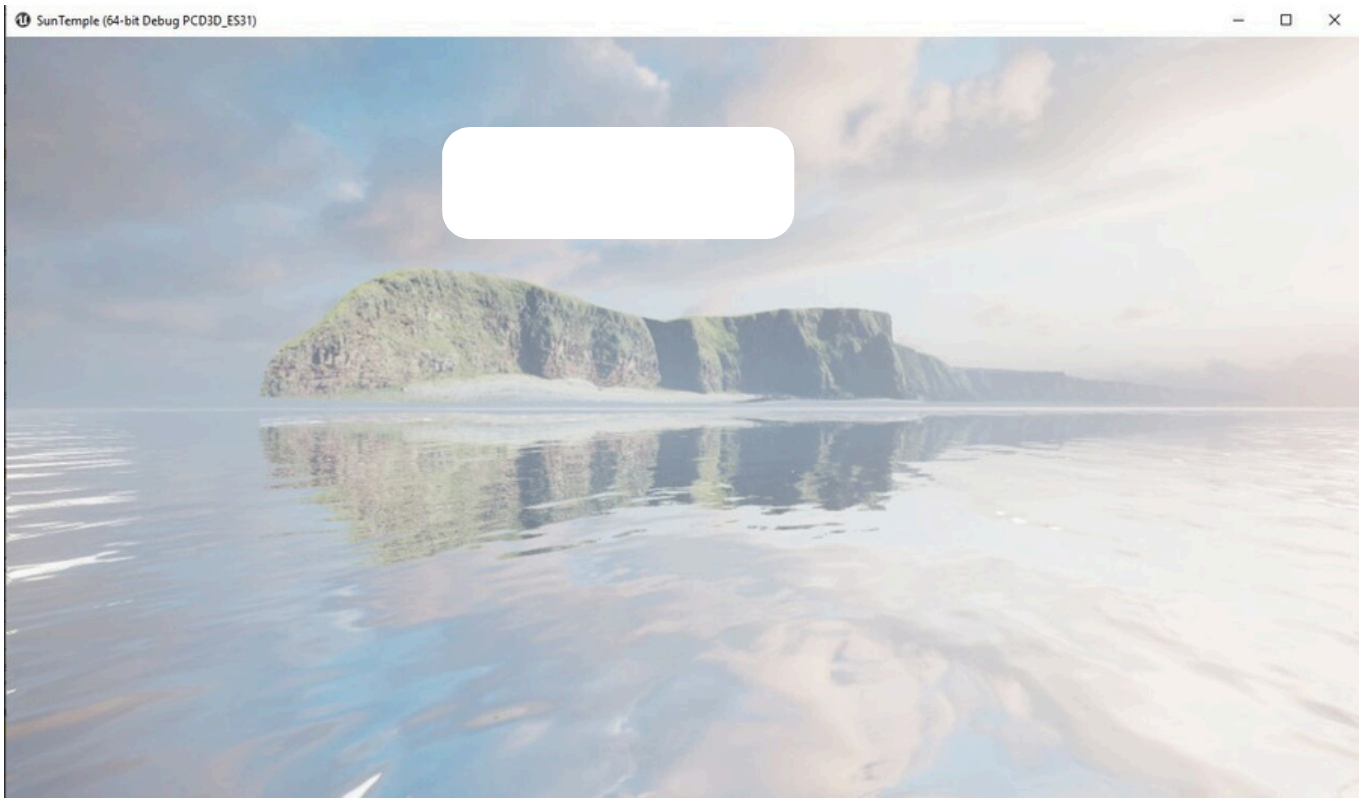
- Calculates the mirror image position of all pixels of the scene color at the reflection plane.
- Tests if the reflection of a pixel is within the reflection area.
    - The ray is cast to the mirrored pixel location.
    - Tests whether the intersection point is within the reflection area.

- If an intersection is found, calculate the mirrored position of the pixel on the screen. Write
- the color of the mirrored pixel at the intersection point.



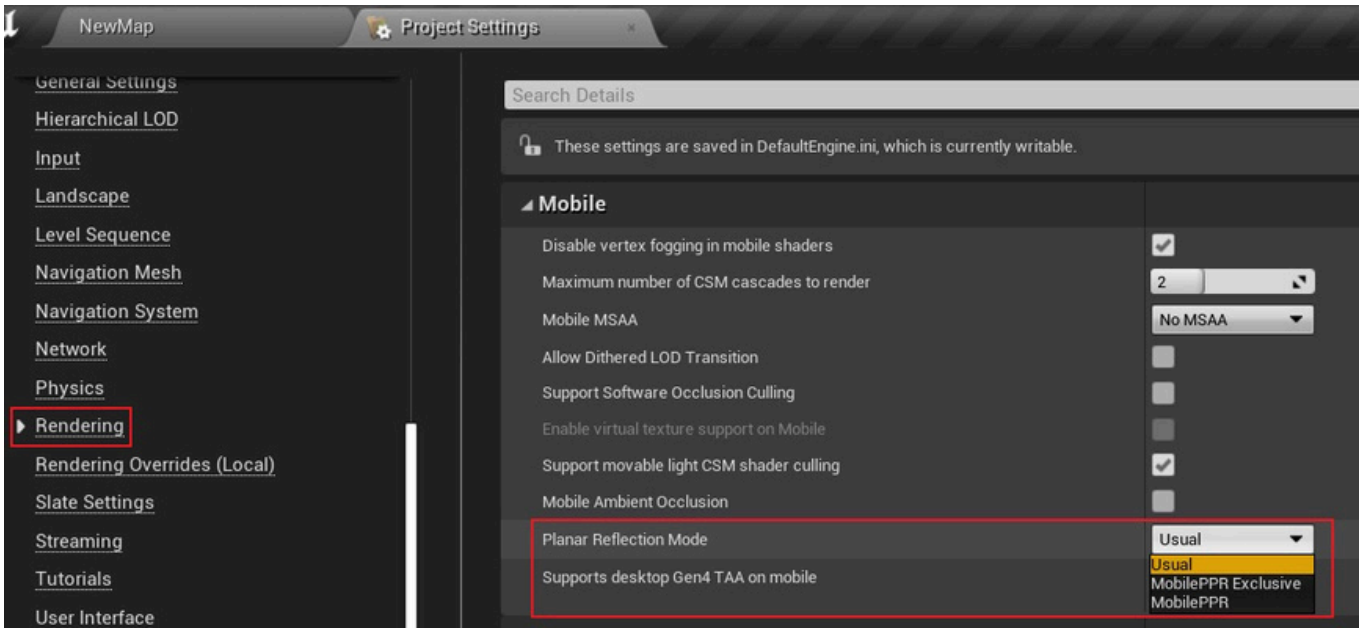
- If the intersection point in the reflection area is blocked by other objects, the reflection at this position is removed.





PPR effects at a glance.

PPR can be set in the project configuration:



12.3.3 RenderDeferred

UE4.26 added a deferred rendering branch to the mobile rendering pipeline, and made improvements and optimizations in 4.27. Whether the deferred shading feature is enabled on the mobile terminal is determined by the following code:

```
// Engine\Source\Runtime\RenderCore\Private\RenderUtils.cpp

bool IsMobileDeferredShadingEnabled(constFStaticShaderPlatform Platform) {
```

```
    //  DisableOpenGLDelayed coloring.
    if (IsOpenGLPlatform(Platform))
    {
        // needs MRT framebuffer fetch or PLS returnfalse;

    }

    //Console variables"r.Mobile.ShadingPath"To1. staticauto* MobileShadingPathCvar =
    IConsoleManager::Get().FindTConsoleVariableDataInt(TEXT("r.Mobile.ShadingPath"));

    returnMobileShadingPathCvar->GetValueOnAnyThread() ==1;
}
```

Simply put, it is a non-OpenGL graphics API and the console variable r.Mobile.ShadingPath is set to 1.

> r.Mobile.ShadingPathThe value cannot be set dynamically in the editor. You can only enable it by adding the following field in the project root directory/Config/DefaultEngine.ini: [/Script/Engine.RendererSettings]
> r.Mobile.ShadingPath=1
> After adding the above fields, restart the UE editor and wait for the shader compilation to complete to preview the delayed shading effect on the mobile terminal.

FMobileSceneRenderer::RenderDeferredThe following is the code and analysis of the delayed rendering branch:

```
FRHITexture*FMobileSceneRenderer::RenderDeferred(FRHICommandListImmediate& RHICmdList, constTArrayView<
constFViewInfo*> ViewList,constFSortedLightSetSceneInfo& SortedLightSet)

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

    //PrepareGBuffer.
    FRHITexture* ColorTargets[4] = {
        SceneContext.GetSceneColorSurface(),
        SceneContext.GetGBufferATexture().GetReference(),
        SceneContext.GetGBufferBTexture().GetReference(),
        SceneContext.GetGBufferCTexture().GetReference()
    };

    //RHIDo you need toGBufferSave toGPUin system memory and rendered in a separate render
    pass. ERenderTargetActions GBufferAction = bRequiresMultiPass ?
ERenderTargetActions::Clear_Store : ERenderTargetActions::Clear_DontStore;
    EDepthStencilTargetActions DepthAction = bKeepDepthContent ?
EDepthStencilTargetActions::ClearDepthStencil_StoreDepthStencil :
EDepthStencilTargetActions::ClearDepthStencil_DontStoreDepthStencil;

    //RTofload/storeaction.
    ERenderTargetActions ColorTargetsAction[4] = {ERenderTargetActions::Clear_Store, GBufferAction,
GBufferAction, GBufferAction};
    if(bIsFullPrepassEnabled) {

        ERenderTargetActions DepthTarget =
        MakeRenderTargetActions(ERenderTargetLoadAction::ELoad,
```

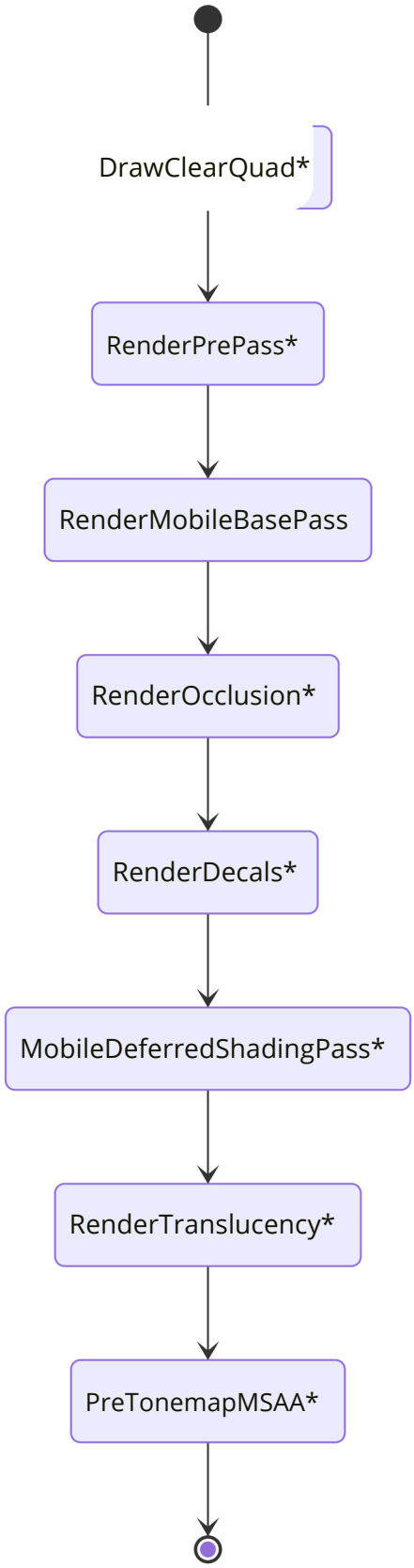
```
GetStoreAction(GetDepthActions(DepthAction)));
            ERenderTargetActions StencilTarget =
MakeRenderTargetActions(ERenderTargetLoadAction::ELoad,
GetStoreAction(GetStencilActions(DepthAction)));
            DepthAction = MakeDepthStencilTargetActions(DepthTarget, StencilTarget);
        }

        FRHIRenderPassInfo BasePassInfo = FRHIRenderPassInfo(); int32
        ColorTargetIndex =0;
        for(; ColorTargetIndex < UE_ARRAY_COUNT(ColorTargets); ++ColorTargetIndex) {

            BasePassInfo.ColorRenderTargets[ColorTargetIndex].RenderTarget                    =
            ColorTargets[ColorTargetIndex];
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].ResolveTarget                     = nullptr;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].ArraySlice                  = - 1;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].MipIndex                 = 0;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].Action =
ColorTargetsAction[Color TargetIndex];
        }

        if(MobileRequiresSceneDepthAux(ShaderPlatform)) {

            BasePassInfo.ColorRenderTargets[ColorTargetIndex].RenderTarget = SceneContext.SceneDepthAux-
            >GetRenderTargetItem().ShaderResourceTexture.GetReference();
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].ResolveTarget                     = nullptr;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].ArraySlice                  = - 1;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].MipIndex                 =0;
            BasePassInfo.ColorRenderTargets[ColorTargetIndex].Action = ColorTargetIndex+G+B;ufferAction;

        }

        BasePassInfo.DepthStencilRenderTarget.DepthStencilTarget        =
        SceneContext.GetSceneDepthSurface();
        BasePassInfo.DepthStencilRenderTarget.ResolveTarget  =  nullptr;
        BasePassInfo.DepthStencilRenderTarget.Action   =   DepthAction;
        BasePassInfo.DepthStencilRenderTarget.ExclusiveDepthStencil                    =
        FExclusiveDepthStencil::DepthWrite_StencilWrite;

        BasePassInfo.SubpassHint = ESubpassHint::DeferredShadingSubpass; if(!
        bIsFullPrepassEnabled) {

            BasePassInfo.NumOcclusionQueries = ComputeNumOcclusionQueriesToBatch();
            BasePassInfo.bOcclusionQueries = BasePassInfo.NumOcclusionQueries !=0;
        }
        BasePassInfo.ShadingRateTexture            = nullptr;
        BasePassInfo.bIsMSAA =false;
        BasePassInfo.MultiViewCount =          0;

        RHICmdList.BeginRenderPass(BasePassInfo, TEXT("BasePassRendering"));

        if(GIsEditor && !Views[0].bIsSceneCapture) {

            DrawClearQuad(RHICmdList, Views[0].BackgroundColor);
        }

        //   depthPrePass
        if   (!bIsFullPrepassEnabled)
        {
```

```
        RHICmdList.SetCurrentStat(GET_STATID(STAT_CLM_MobilePrePass)); // Depth pre-
        pass
        RenderPrePass(RHICmdList);
    }

    // BasePass:Opaque and hollow objects.
    RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Opaque));
    RenderMobileBasePass(RHICmdList, ViewList);
    RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);

    //  Occlusion culling.
    if  (!bIsFullPrepassEnabled)
    {
        // Issue occlusion queries
        RHICmdList.SetCurrentStat(GET_STATID(STAT_CLMM_Occlusion));
        RenderOcclusion(RHICmdList);
        RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
    }

    //  VeryPassmodel
    if  (!bRequiresMultiPass)
    {
        //Next childPass: SSceneColor + GBufferWrite,SceneDepthRead-only.
        RHICmdList.NextSubpass();

        //  Rendering decals.
        if  (ViewFamily.EngineShowFlags.Decals)
        {
            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderDecals);
            RenderDecals(RHICmdList);
        }

        //Next childPass: SceneColorWrite,SceneDepthRead-only
        RHICmdList.NextSubpass();

        //Delayed lighting shading.
        MobileDeferredShadingPass(RHICmdList, *Scene, ViewList, SortedLightSet);

        //  Draw semi-transparently.
        if  (ViewFamily.EngineShowFlags.Translucency)
        {
            CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderTranslucency);
            SCOPE_CYCLE_COUNTER(STAT_TranslucencyDrawTime);
            RenderTranslucency(RHICmdList, ViewList);
            FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
            RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
        }

        //End RenderingPass.
        RHICmdList.EndRenderPass();
    }
    //  manyPassmodel(PCDevice emulation for mobile devices).
    else
    {
        //  Endpass.
        RHICmdList.NextSubpass();
        RHICmdList.NextSubpass();
        RHICmdList.EndRenderPass();
```

```
        // SceneColor + GBuffer write, SceneDepth is read only {

            for(int32 Index =0; Index < UE_ARRAY_COUNT(ColorTargets); ++Index) {

                BasePassInfo.ColorRenderTargets[Index].Action =
ERenderTargetActions::Load_Store;
            }
            BasePassInfo.DepthStencilRenderTarget.Action =
EDepthStencilTargetActions::LoadDepthStencil_StoreDepthStencil;
            BasePassInfo.DepthStencilRenderTarget.ExclusiveDepthStencil                =
FExclusiveDepthStencil::DepthRead_StencilRead;
            BasePassInfo.SubpassHint = ESubpassHint::None;
            BasePassInfo.NumOcclusionQueries          =0;
            BasePassInfo.bOcclusionQueries =          false;

            RHICmdList.BeginRenderPass(BasePassInfo, TEXT("AfterBasePass"));

            //  Rendering decals.
            if  (ViewFamily.EngineShowFlags.Decals)
            {
                CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderDecals);
                RenderDecals(RHICmdList);
            }

            RHICmdList.EndRenderPass();
        }

        // SceneColor write, SceneDepth is read only {

            FRHIRenderPassInfoShadingPassInfo(
                SceneContext.GetSceneColorSurface(),
                ERenderTargetActions::Load_Store, nullptr,

                SceneContext.GetSceneDepthSurface(),
                EDepthStencilTargetActions::LoadDepthStencil_StoreDepthStencil, nullptr,

                nullptr,
                VRSRB_Passthrough,
                FExclusiveDepthStencil::DepthRead_StencilWrite
            ) ;
            ShadingPassInfo.NumOcclusionQueries          =0;
            ShadingPassInfo.bOcclusionQueries =          false;

            RHICmdList.BeginRenderPass(ShadingPassInfo, TEXT("MobileShadingPass"));

            //Delayed lighting shading.
            MobileDeferredShadingPass(RHICmdList, *Scene, ViewList, SortedLightSet);

            //  Draw semi-transparently.
            if  (ViewFamily.EngineShowFlags.Translucency)
            {
                CSV_SCOPED_TIMING_STAT_EXCLUSIVE(RenderTranslucency);
                SCOPE_CYCLE_COUNTER(STAT_TranslucencyDrawTime);
                RenderTranslucency(RHICmdList, ViewList);
                FRHICommandListExecutor::GetImmediateCommandList().PollOcclusionQueries();
                RHICmdList.ImmediateFlush(EImmediateFlushType::DispatchToRHIThread);
            }
```

```
                RHICmdList.EndRenderPass();
            }
        }

        returnColorTargets[0];
}
```

As can be seen above, the deferred rendering pipeline on mobile devices is similar to that on PCs. It renders BasePass first, obtains GBuffer geometry information, and then performs lighting calculations. Their flowcharts are as follows:



Of course, there are also differences from PC. The most obvious one is that the mobile terminal uses SubPass rendering adapted to the TB(D)R architecture, so that when the mobile terminal renders PrePass depth, BasePass, and lighting calculations, the scene color, depth, GBuffer and other information are always in the On-Chip buffer, improving rendering efficiency and reducing device energy consumption.

12.3.3.1 MobileDeferredShadingPass

The process of deferred rendering lighting is MobileDeferredShadingPass undertaken by:

```
void MobileDeferredShadingPass(
	FRHICommandListImmediate& RHICmdList, const
	FScene& Scene,
	constTArrayView<constFViewInfo*> PassViews, const
	FSortedLightSetSceneInfo &SortedLightSet)
{
	SCOPED_DRAW_EVENT(RHICmdList, MobileDeferredShading);

	constFViewInfo& View0 = *PassViews[0];

	FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList); //createUniform
	Buffer. FUniformBufferRHIRef PassUniformBuffer =
	CreateMobileSceneTextureUniformBuffer(RHICmdList);

	FUniformBufferStaticBindingsGlobalUniformBuffers(PassUniformBuffer);
	SCOPED_UNIFORM_BUFFER_GLOBAL_BINDINGS(RHICmdList, GlobalUniformBuffers); //Set the
	viewport.
	RHICmdList.SetViewport(View0.ViewRect.Min.X, View0.ViewRect.Min.Y,0.0f, View0.ViewRect.Max.X,
View0.ViewRect.Max.Y,1.0f);

	//The default material for lighting.
	FCachedLightMaterial DefaultMaterial;
	DefaultMaterial.MaterialProxy = UMaterial::GetDefaultMaterial(MD_LightFunction)-
> GetRenderProxy();
	DefaultMaterial.Material = DefaultMaterial.MaterialProxy-
> GetMaterialNoFallback(ERHIFeatureLevel::ES3_1);
check(DefaultMaterial.Material);

	//Draws a parallel light.
	RenderDirectLight(RHICmdList, Scene, View0, DefaultMaterial);

	if(GMobileUseClusteredDeferredShading ==0) {

		//Rendering non-clustered simple lights.
		RenderSimpleLights(RHICmdList, Scene, PassViews, SortedLightSet, DefaultMaterial);
	}

	//Rendering non-clustered local lights.
	int32 NumLights = SortedLightSet.SortedLights.Num(); int32
	StandardDeferredStart = SortedLightSet.SimpleLightsEnd; if
	(GMobileUseClusteredDeferredShading !=0) {

		StandardDeferredStart = SortedLightSet.ClusteredSupportedEnd;
	}

	//Renders local lights.
	for(int32 LightIdx = StandardDeferredStart; LightIdx < NumLights; ++LightIdx) {

		constFSortedLightSceneInfo& SortedLight = SortedLightSet.SortedLights[LightIdx]; constFLightSceneInfo&
		LightSceneInfo = *SortedLight.LightSceneInfo; RenderLocalLight(RHICmdList, Scene, View0, LightSceneInfo,
		DefaultMaterial);
	}
}
```

Let's continue to analyze the interfaces for rendering different types of light sources:

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


//Rendering Parallel Lights
static void RenderDirectLight(FRHICommandListImmediate& RHICmdList,constFScene& Scene, constFViewInfo& View,
constFCachedLightMaterial& DefaultLightMaterial) {

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

	//Find the first parallel light.
	FLightSceneInfo* DirectionalLight = nullptr;
	for(int32 ChannelIdx =0; ChannelIdx < UE_ARRAY_COUNT(Scene.MobileDirectionalLights) !DirectionalLight;
&& ChannelIdx++)
	{
		DirectionalLight = Scene.MobileDirectionalLights[ChannelIdx];
	}

	//Rendering status.
	FGraphicsPipelineStateInitializer GraphicsPSOInit;
	RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit); //Increase
	the self-luminescence toSceneColor.
	GraphicsPSOInit.BlendState = TStaticBlendState<CW_RGB, BO_Add, BF_One,
	BF_One>::GetRHI();
	GraphicsPSOInit.RasterizerState = TStaticRasterizerState<>::GetRHI(); //Only draw the default
	lighting model (MSM_DefaultLit)Pixels.
	uint8 StencilRef = GET_STENCIL_MOBILE_SM_MASK(MSM_DefaultLit);
	GraphicsPSOInit.DepthStencilState =		TStaticDepthStencilState< false,
	CF_Always,
	true, CF_Equal, SO_Keep, SO_Keep, SO_Keep, false, CF_Always,
	SO_Keep, SO_Keep, SO_Keep, GET_STENCIL_MOBILE_SM_MASK(
	0x7),0x00>::GetRHI();
// 4 bits for shading models

	//deal withVS.
	TShaderMapRef<FPostProcessVS>VertexShader(View.ShaderMap);

	constFMaterialRenderProxy* LightFunctionMaterialProxy = nullptr; if(View.Family-
	>EngineShowFlags.LightFunctions && DirectionalLight) {

		LightFunctionMaterialProxy = DirectionalLight->Proxy->GetLightFunctionMaterial();
	}
	FMobileDirectLightFunctionPS::FPermutationDomain PermutationVector =
FMobileDirectLightFunctionPS::BuildPermutationVector(View, DirectionalLight != nullptr);
	FCachedLightMaterial LightMaterial; TShaderRef<FMobileDirectLightFunctionPS>
	PixelShader; GetLightMaterial(DefaultLightMaterial, LightFunctionMaterialProxy,
	PermutationVector.ToDimensionValueId(), LightMaterial, PixelShader);



	GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI		=
	GFilterVertexDeclaration.VertexDeclarationRHI;
	GraphicsPSOInit.BoundShaderState.VertexShaderRHI		= VertexShader.GetVertexShader();
	GraphicsPSOInit.BoundShaderState.PixelShaderRHI		=	PixelShader.GetPixelShader();
	GraphicsPSOInit.PrimitiveType = PT_TriangleList;
	SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);


	//deal withPS.
```

```
        FMobileDirectLightFunctionPS::FParameters PassParameters;
        PassParameters.Forward = View.ForwardLightingResources->ForwardLightDataUniformBuffer;
        PassParameters.MobileDirectionalLight =
Scene.UniformBuffers.MobileDirectionalLightUniformBuffers[1];
        PassParameters.ReflectionCaptureData =
        Scene.UniformBuffers.ReflectionCaptureUniformBuffer;
        FReflectionUniformParameters ReflectionUniformParameters;
        SetupReflectionUniformParameters(View,                ReflectionUniformParameters);
        PassParameters.ReflectionsParameters =
        CreateUniformBufferImmediate(ReflectionUniformParameters,            UniformBuffer_SingleDraw);
        PassParameters.LightFunctionParameters = FVector4(1.0f,1.0f,0.0f,0.0f); if(DirectionalLight)

        {
                const boolbUseMovableLight = DirectionalLight && !DirectionalLight->Proxy-
> HasStaticShadowing();
                PassParameters.LightFunctionParameters2 = FVector(DirectionalLight->Proxy-
> GetLightFunctionFadeDistance(), DirectionalLight->Proxy-
> GetLightFunctionDisabledBrightness(), bUseMovableLight?1.0f:0.0f); constFVector Scale =
                DirectionalLight->Proxy->GetLightFunctionScale();
                // Switch x and z so that z of the user specified scale affects the distance along the light direction

                constFVector InverseScale = FVector(1.f/ Scale.Z,1.f/ Scale.Y,1.f/ Scale.X); PassParameters.WorldToLight =
                DirectionalLight->Proxy->GetWorldToLight() * FScaleMatrix(FVector(InverseScale));

        }
        FMobileDirectLightFunctionPS::SetParameters(RHICmdList, PixelShader, View,
LightMaterial.MaterialProxy, *LightMaterial.Material, PassParameters);

        RHICmdList.SetStencilRef(StencilRef);

        constFIntPoint TargetSize = SceneContext.GetBufferSizeXY();

        //Draw with a full-screen rectangle.
        DrawRectangle(
                RHICmdList,
                0,0,
                View.ViewRect.Width(),          View.ViewRect.Height(),
                View.ViewRect.Min.X, View.ViewRect.Min.Y, View.ViewRect.Width(),
                View.ViewRect.Height(), FIntPoint(View.ViewRect.Width(),
                View.ViewRect.Height()) ,TargetSize,

                VertexShader);
}

//Renders simple lights in non-clustered mode.
static void RenderSimpleLights(
        FRHICommandListImmediate& RHICmdList, const
        FScene& Scene,
        constTArrayView<constFViewInfo*> PassViews, const
                FSortedLightSetSceneInfo &SortedLightSet,
        const   FCachedLightMaterial& DefaultMaterial)
{
        constFSimpleLightArray& SimpleLights = SortedLightSet.SimpleLights; constint32 NumViews
        = PassViews.Num(); constFViewInfo& View0 = *PassViews[0];


        //deal withVS.
        TShaderMapRef<TDeferredLightVS<true>> VertexShader(View0.ShaderMap);
```

```
TShaderRef<FMobileRadialLightFunctionPS> PixelShaders[2]; {

        constFMaterialShaderMap* MaterialShaderMap = DefaultMaterial.Material-
> GetRenderingThreadShaderMap(); FMobileRadialLightFunctionPS::FPermutationDomain PermutationVector;
        PermutationVector.Set<FMobileRadialLightFunctionPS::FSpotLightDim>(false);
        PermutationVector.Set<FMobileRadialLightFunctionPS::FIESProfileDim>(false);
        PermutationVector.Set<FMobileRadialLightFunctionPS::FInverseSquaredDim>(false);PixelShaders[0]    =
        MaterialShaderMap->GetShader<FMobileRadialLightFunctionPS> (PermutationVector);


        PermutationVector.Set<FMobileRadialLightFunctionPS::FInverseSquaredDim>(true);PixelShaders[1] =
        MaterialShaderMap->GetShader<FMobileRadialLightFunctionPS> (PermutationVector);

    }

    //set upPSO.
    FGraphicsPipelineStateInitializer GraphicsPSOLight[2]; {

        SetupSimpleLightPSO(RHICmdList, View0, VertexShader, PixelShaders[0], GraphicsPSOLight[0
        ]);
        SetupSimpleLightPSO(RHICmdList, View0, VertexShader, PixelShaders[1], GraphicsPSOLight[1
        ]);
    }

    //Set up the stencil buffer.
    FGraphicsPipelineStateInitializer GraphicsPSOLightMask; {

        RHICmdList.ApplyCachedRenderTargets(GraphicsPSOLightMask);  GraphicsPSOLightMask.PrimitiveType  =
        PT_TriangleList;  GraphicsPSOLightMask.BlendState  =  TStaticBlendStateWriteMask<CW_NONE,  CW_NONE,
        CW_NONE, CW_NONE, CW_NONE, CW_NONE, CW_NONE, CW_NONE>::GetRHI() ;

        GraphicsPSOLightMask.RasterizerState = View0.bReverseCulling ? TStaticRasterizerState<FM_Solid,
CM_CCW>::GetRHI() : TStaticRasterizerState<FM_Solid, CM_CW>::GetRHI();

        // set stencil to 1 where depth test fails GraphicsPSOLightMask.DepthStencilState =
        TStaticDepthStencilState<
            false, CF_DepthNearOrEqual,
            true, CF_Always, SO_Keep, SO_Replace, SO_Keep, false, CF_Always,
            SO_Keep, SO_Keep, SO_Keep, 0x00, STENCIL_SANDBOX_MASK>::GetRHI();
            GraphicsPSOLightMask.BoundShaderState.VertexDeclarationRHI
            GetVertexDeclarationFVector4();                                                    =

        GraphicsPSOLightMask.BoundShaderState.VertexShaderRHI                =
        VertexShader.GetVertexShader();
        GraphicsPSOLightMask.BoundShaderState.PixelShaderRHI                 =  nullptr;
    }

    //Traverse the list of all simple light sources and perform coloring calculations.
    for(int32 LightIndex =0; LightIndex < SimpleLights.InstanceData.Num(); LightIndex++) {

        constFSimpleLightEntry& SimpleLight = SimpleLights.InstanceData[LightIndex]; for(int32 ViewIndex =0;
        ViewIndex < NumViews; ViewIndex++) {

            constFViewInfo& View = *PassViews[ViewIndex]; const
            FSimpleLightPerViewEntry& SimpleLightPerViewData =
SimpleLights.GetViewDependentData(LightIndex, ViewIndex, NumViews);
            constFSphereLightBounds(SimpleLightPerViewData.Position,
SimpleLight.Radius);
```

```
                if(NumViews >1) {

                    // set viewports only we we have more than one // otherwise it is set at the start of the
                    pass RHICmdList.SetViewport(View.ViewRect.Min.X, View.ViewRect.Min.Y,
                                                                                                0.0f,
View.ViewRect.Max.X, View.ViewRect.Max.Y,1.0f);
                }

                //Renders the light mask.
                SetGraphicsPipelineState(RHICmdList, GraphicsPSOLightMask); VertexShader-
                >SetSimpleLightParameters(RHICmdList, View, LightBounds); RHICmdList.SetStencilRef(1);
                StencilingGeometry::DrawSphere(RHICmdList);



                //Rendering light sources.
                FMobileRadialLightFunctionPS::FParameters PassParameters; FDeferredLightUniformStruct
                DeferredLightUniformsValue; SetupSimpleDeferredLightParameters(SimpleLight,
                SimpleLightPerViewData,
DeferredLightUniformsValue);
                PassParameters.DeferredLightUniforms =
TUniformBufferRef<FDeferredLightUniformStruct>::CreateUniformBufferImmediate(DeferredLight UniformsValue,
EUniformBufferUsage::UniformBuffer_SingleFrame);
                PassParameters.IESTexture = GWhiteTexture->TextureRHI;
                PassParameters.IESTextureSampler = GWhiteTexture->SamplerStateRHI; if
                (SimpleLight.Exponent ==0) {

                    SetGraphicsPipelineState(RHICmdList, GraphicsPSOLight[1]);
                    FMobileRadialLightFunctionPS::SetParameters(RHICmdList, PixelShaders[1],
View, DefaultMaterial.MaterialProxy, *DefaultMaterial.Material, PassParameters);
                }
                else
                {
                    SetGraphicsPipelineState(RHICmdList, GraphicsPSOLight[0]);
                    FMobileRadialLightFunctionPS::SetParameters(RHICmdList, PixelShaders[0],
View, DefaultMaterial.MaterialProxy, *DefaultMaterial.Material, PassParameters);
                }
                VertexShader->SetSimpleLightParameters(RHICmdList, View, LightBounds);

                //Only draw the default lighting model (MSM_DefaultLit)Pixels.
                uint8 StencilRef = GET_STENCIL_MOBILE_SM_MASK(MSM_DefaultLit);
                RHICmdList.SetStencilRef(StencilRef);

                //Render lights (point and spot) as spheres to quickly cull pixels outside of the light's
                influence. StencilingGeometry::DrawSphere(RHICmdList);
            }
        }
}

//Renders local lights.
static void RenderLocalLight(
        FRHICommandListImmediate&       RHICmdList,
        const FScene& Scene,
        const FViewInfo& View,
        const FLightSceneInfo& LightSceneInfo, FCachedLightMaterial&
        const DefaultLightMaterial)
{
        if(!LightSceneInfo.ShouldRenderLight(View))
```

```
{
    return;
}

//Ignore non-local lights (lights other than light sources and spotlights).
constuint8 LightType = LightSceneInfo.Proxy->GetLightType(); const bool
bIsSpotLight = LightType == LightType_Spot; const boolbIsPointLight = LightType
== LightType_Point; if(!bIsSpotLight && !bIsPointLight) {


    return;
}

//Draw a light source template.
if(GMobileUseLightStencilCulling !=0) {

    RenderLocalLight_StencilMask(RHICmdList, Scene, View, LightSceneInfo);
}

//deal withIESillumination.
boolbUseIESTexture =false;
FTexture* IESTextureResource = GWhiteTexture;
if(View.Family->EngineShowFlags.TexturedLightProfiles && LightSceneInfo.Proxy-
> GetIESTextureResource()) {

    IESTextureResource = LightSceneInfo.Proxy->GetIESTextureResource(); bUseIESTexture =
    true;
}

FGraphicsPipelineStateInitializer GraphicsPSOInit;
RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit); 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();

//Set the light rasterization and depth state.
if(GMobileUseLightStencilCulling !=0) {

    SetLocalLightRasterizerAndDepthState_StencilMask(GraphicsPSOInit,                View);
}
else
{
    SetLocalLightRasterizerAndDepthState(GraphicsPSOInit, View, LightBounds);
}

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

constFMaterialRenderProxy* LightFunctionMaterialProxy = nullptr; if(View.Family-
>EngineShowFlags.LightFunctions) {

    LightFunctionMaterialProxy = LightSceneInfo.Proxy->GetLightFunctionMaterial();
}
FMobileRadialLightFunctionPS::FPermutationDomain PermutationVector;
PermutationVector.Set<FMobileRadialLightFunctionPS::FSpotLightDim>(bIsSpotLight);
PermutationVector.Set<FMobileRadialLightFunctionPS::FInverseSquaredDim> (LightSceneInfo.Proxy-
>IsInverseSquared());
```

```
        PermutationVector.Set<FMobileRadialLightFunctionPS::FIESProfileDim>(bUseIESTexture); FCachedLightMaterial
        LightMaterial;
        TShaderRef<FMobileRadialLightFunctionPS> PixelShader;
        GetLightMaterial(DefaultLightMaterial, LightFunctionMaterialProxy,
PermutationVector.ToDimensionValueId(), LightMaterial, PixelShader);

        GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                =
        GetVertexDeclarationFVector4();
        GraphicsPSOInit.BoundShaderState.VertexShaderRHI                 = VertexShader.GetVertexShader();
        GraphicsPSOInit.BoundShaderState.PixelShaderRHI                  =    PixelShader.GetPixelShader();
        SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit);

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

        //set upPS.
        FMobileRadialLightFunctionPS::FParameters PassParamePtearsss.PDaerfaemrredteLrisg;htUniforms =
        TUniformBufferRef<FDeferredLightUniformStruct>::CreateUniformBufferImmediate(GetDeferredLightParameters(View,
LightSceneInfo), EUniformBufferUsage::UniformBuffer_SingleFrame);

        PassParameters.IESTexture = IESTextureResource->TextureRHI; PassParameters.IESTextureSampler =
        IESTextureResource->SamplerStateRHI; const floatTanOuterAngle = bIsSpotLight ?
        FMath::Tan(LightSceneInfo.Proxy-
> GetOuterConeAngle()) :1.0f;
        PassParameters.LightFunctionParameters = FVector4(TanOuterAngle,1.0f /*ShadowFadeFraction*/,
bIsSpotLight ?1.0f:0.0f, bIsPointLight ?1.0f:0.0f);
        PassParameters.LightFunctionParameters2 = FVector(LightSceneInfo.Proxy-
> GetLightFunctionFadeDistance(), LightSceneInfo.Proxy-
> GetLightFunctionDisabledBrightness(),                 0.0f);
        constFVector Scale = LightSceneInfo.Proxy->GetLightFunctionScale();
        // Switch x and z so that z of the user specified scale affects the distance along the light direction

        constFVector InverseScale = FVector(1.f/ Scale.Z,1.f/ Scale.Y,1.f/ Scale.X); PassParameters.WorldToLight =
        LightSceneInfo.Proxy->GetWorldToLight() * FScaleMatrix(FVector(InverseScale));

        FMobileRadialLightFunctionPS::SetParameters(RHICmdList, PixelShader, View,
LightMaterial.MaterialProxy, *LightMaterial.Material, PassParameters);

        //Only draw the default lighting model (MSM_DefaultLit)Pixels.
        uint8 StencilRef = GET_STENCIL_MOBILE_SM_MASK(MSM_DefaultLit);
        RHICmdList.SetStencilRef(StencilRef);

        //Point lights are drawn using spheres.
        if(LightType == LightType_Point) {

                StencilingGeometry::DrawSphere(RHICmdList);
        }
        //Spotlights are drawn with cones.
        else// LightType_Spot
        {
                StencilingGeometry::DrawCone(RHICmdList);
        }
}
```

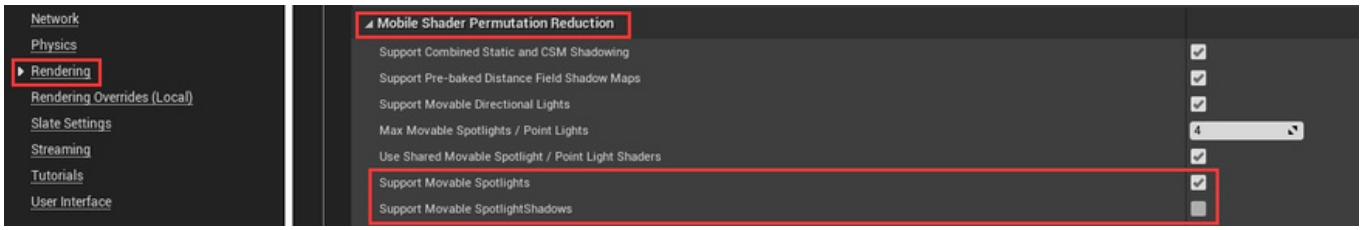
When drawing light sources, there are three steps according to the light source type: parallel light, non-clustered simple light, and local light sources (point light and spotlight). It should be noted that the mobile terminal only supports the calculation of the default lighting model (MSM_DefaultLit),

and other advanced lighting models (hair, subsurface scattering, varnish, eyes, cloth, etc.) are not supported yet.

When drawing parallel light, at most one can be drawn, using full-screen rectangle drawing, and supporting several levels of CSM shadows.

When drawing non-clustered simple light sources, whether point light sources or spotlights, they are drawn using spheres and do not support shadows.

When drawing local light sources, it is much more complicated. First draw the local light source template buffer, then set the rasterization and depth state, and finally draw the light source. Point light sources are drawn with spheres, which do not support shadows; spotlights are drawn with cones, which can support shadows. By default, spotlights do not support dynamic light and shadow calculations, which need to be enabled in the project configuration:

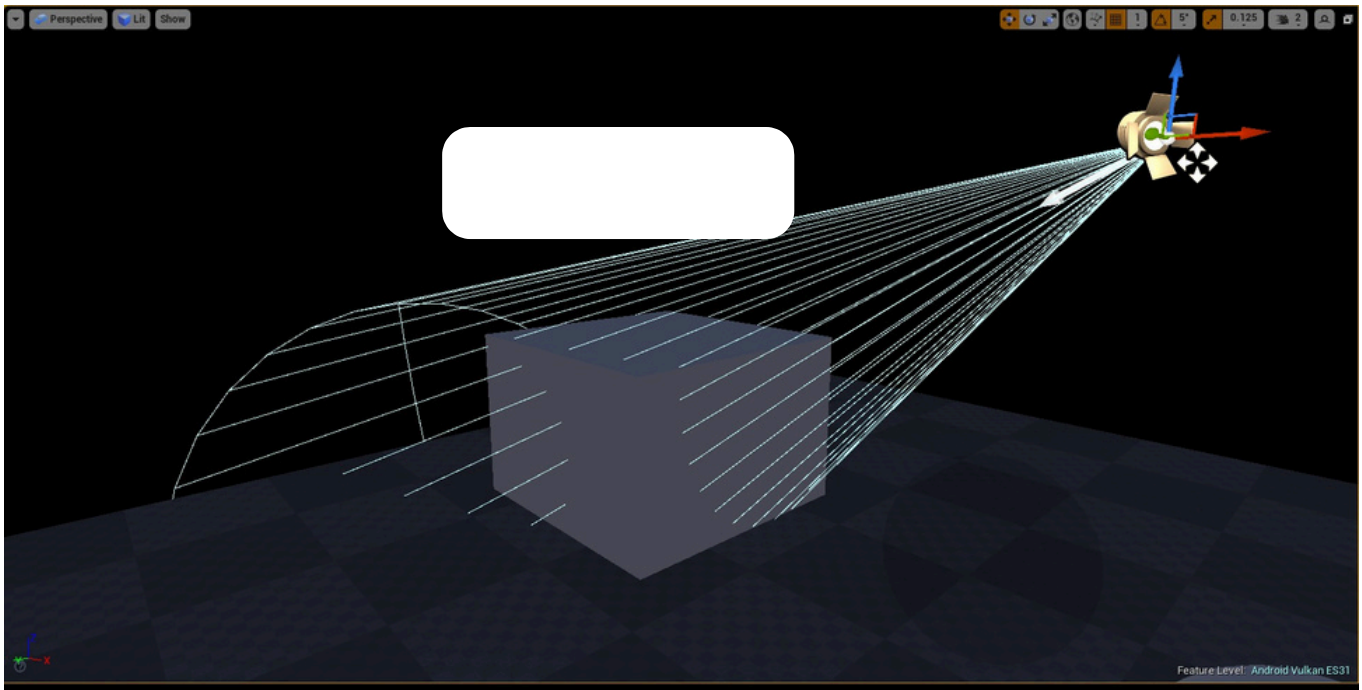


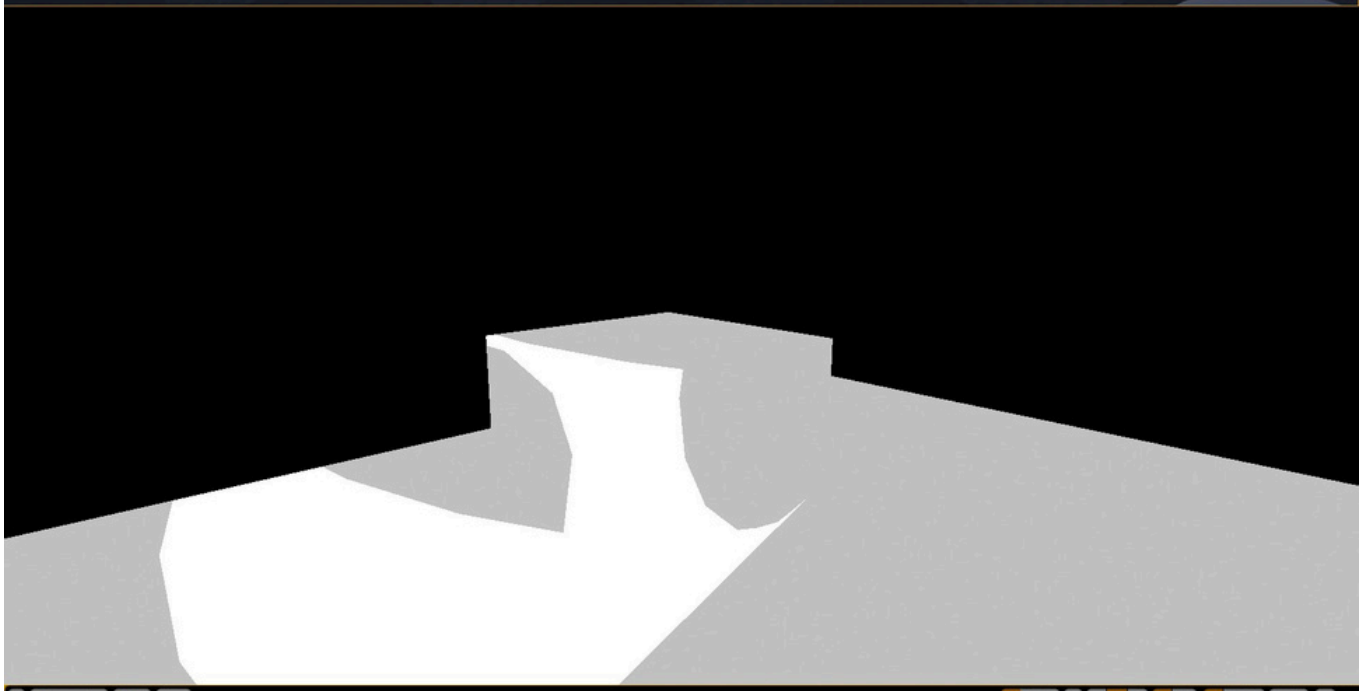
In addition, whether to enable the template to cull pixels that do not intersect with the light source is determined by GMobileUseLightStencilCulling, which isr.Mobile.UseLightStencilCullingdetermined by , and the default value is 1 (ie enabled). The template buffer code for rendering the light source is as follows:

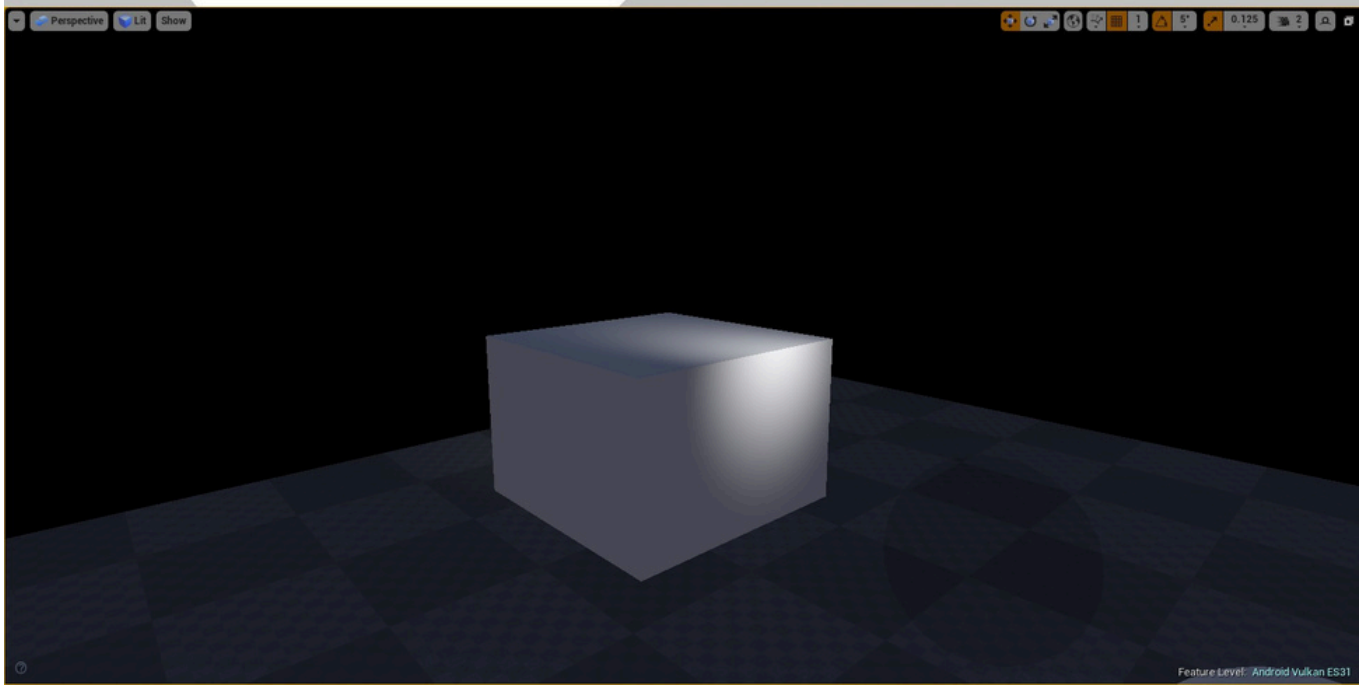
```
static void RenderLocalLight_StencilMask(FRHICommandListImmediate& RHICmdList,const FScene& Scene,const
FViewInfo& View,constFLightSceneInfo& LightSceneInfo) {

	constuint8 LightType = LightSceneInfo.Proxy->GetLightType();

	FGraphicsPipelineStateInitializer GraphicsPSOInit; //Application cache is
	goodRT(color/depth, etc.).
	RHICmdList.ApplyCachedRenderTargets(GraphicsPSOInit);
	GraphicsPSOInit.PrimitiveType = PT_TriangleList; //Disable AllRTof write
	operations.
	GraphicsPSOInit.BlendState = TStaticBlendStateWriteMask<CW_NONE, CW_NONE, CW_NONE, CW_NONE,
CW_NONE, CW_NONE, CW_NONE, CW_NONE>::GetRHI();
	GraphicsPSOInit.RasterizerState = View.bReverseCulling ? TStaticRasterizerState<FM_Solid,
CM_CCW>::GetRHI() : TStaticRasterizerState<FM_Solid, CM_CW>::GetRHI();

	//If the depth test fails, the stencil buffer value is written to1.
	GraphicsPSOInit.DepthStencilState = TStaticDepthStencilState<
		false, CF_DepthNearOrEqual,
		true, CF_Always, SO_Keep, SO_Replace, SO_Keep, false,
		CF_Always, SO_Keep, SO_Keep, SO_Keep, 0x00,

		//Note: only writePassDedicated sandbox (SANBOX)Bit, that is, the index of the template buffer is0
		The position. STENCIL_SANDBOX_MASK>::GetRHI();

	//Drawing of light source templateVSyesTDeferredLightVS.
	TShaderMapRef<TDeferredLightVS<true> > VertexShader(View.ShaderMap);
```

```
    GraphicsPSOInit.BoundShaderState.VertexDeclarationRHI                    =
    GetVertexDeclarationFVector4();
    GraphicsPSOInit.BoundShaderState.VertexShaderRHI //PSIs          =  VertexShader.GetVertexShader();
    empty.
    GraphicsPSOInit.BoundShaderState.PixelShaderRHI                  =  nullptr;

    SetGraphicsPipelineState(RHICmdList, GraphicsPSOInit); VertexShader-
    >SetParameters(RHICmdList, View, &LightSceneInfo); //The template value is1.

    RHICmdList.SetStencilRef(1);

    //Draw with different shapes according to different light sources.
    if(LightType == LightType_Point) {

        StencilingGeometry::DrawSphere(RHICmdList);
    }
    else// LightType_Spot
    {
        StencilingGeometry::DrawCone(RHICmdList);
    }
}
```

Each local light source first draws the Mask within the light source range, and then calculates the lighting of the pixels that pass the Stencil test (Early-Z). The specific analysis process is taken as an example of the spotlight in the figure below:


Perspective
Lit
Show
Feature Level: Android Vulkan ES31




Perspective
Lit
Show
Feature Level: Android Vulkan ES31

Top: A spotlight waiting to be rendered in the scene; Middle: The stencil mask (white area) drawn using the stencil pass marks the pixels in screen space that overlap with the spotlight shape and are closer in depth; Bottom: The effect after lighting calculation for valid pixels.

When performing lighting calculations for valid pixels, the DepthStencil state used is as follows:

Depth State

Depth-Stencil State 283

Enabled:	✔
Write:	✘
Func:	Less Equal

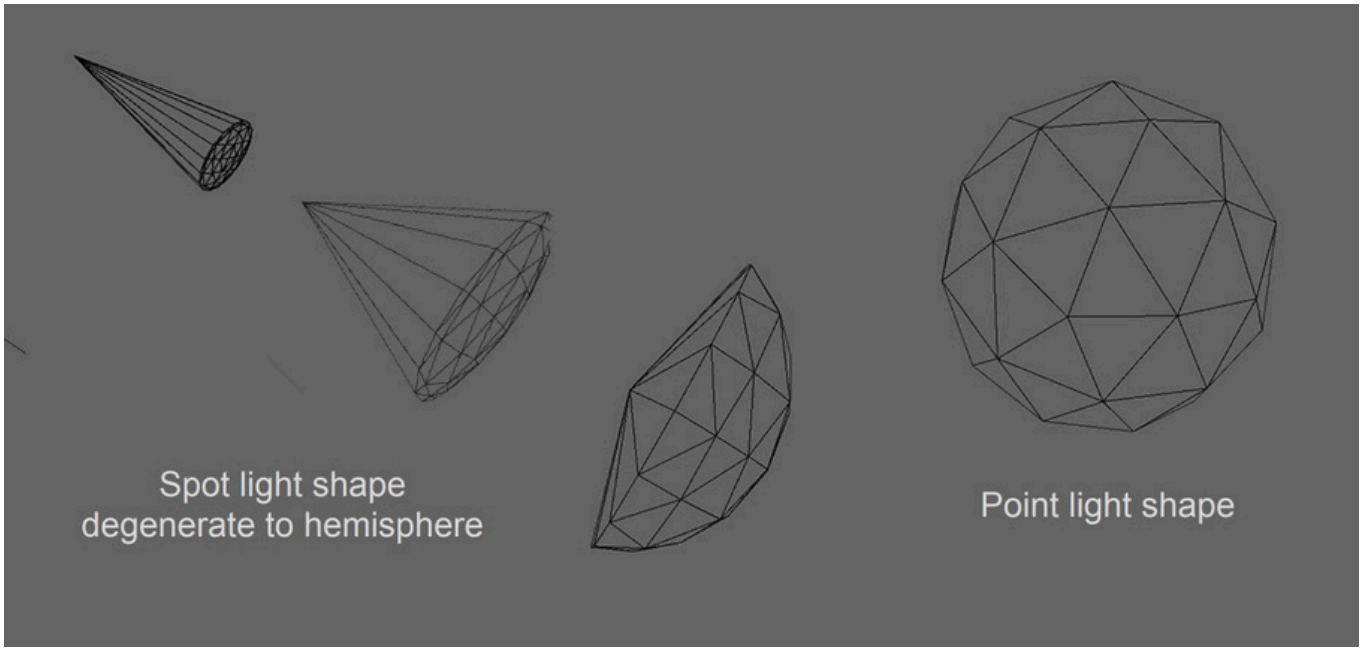
Stencil State

Enabled: ✔ Write Mask: 01 Read Mask: 0F Ref: 02

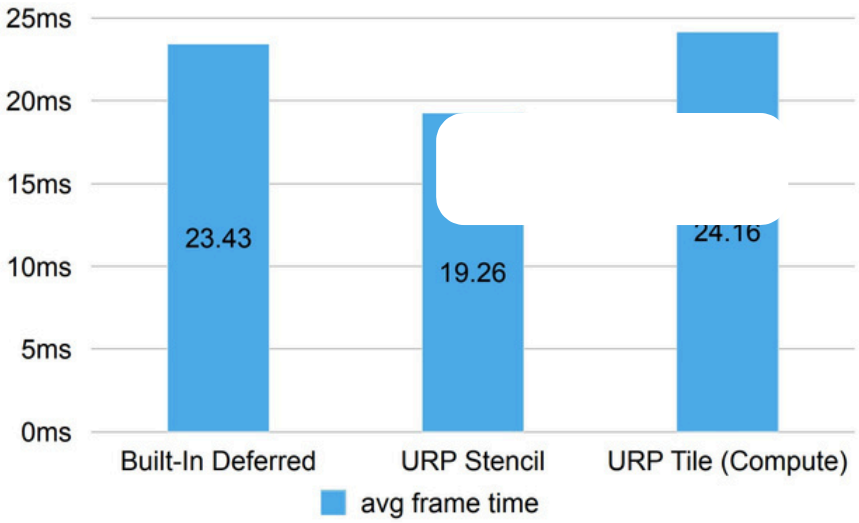
Face	Func	Fail Op	Depth Fail Op	Pass Op
Front	Equal	Keep	Keep	Keep
Back	Equal	Zero	Keep	Zero

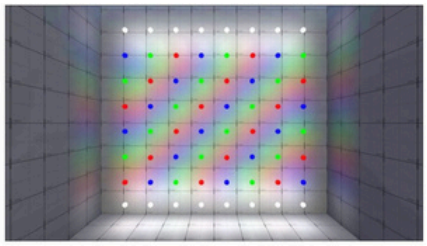
Translated into words, the pixels that perform lighting must be within the light source shape, and the pixels outside the light source shape will be eliminated. The template pass marks the pixels that are closer to the light source shape in depth (pixels outside the light source shape). The light source drawing pass eliminates the pixels marked by the template pass through the template test, and then finds the pixels within the light source shape through the depth test, thereby improving the efficiency of lighting calculation.

This light stencil cullingtechnology on mobile terminals is similar to the template-based lighting calculation mentioned in the Unity speechDeferred Shading in Unity URP at Siggraph 2020 (the idea is the same, but the approach may not be exactly the same). The paper also proposes a geometry simulation that better fits the shape of the light source:



And compared the performance of various light source calculation methods on PC and mobile terminals. The following is a comparison chart of Mali GPU:

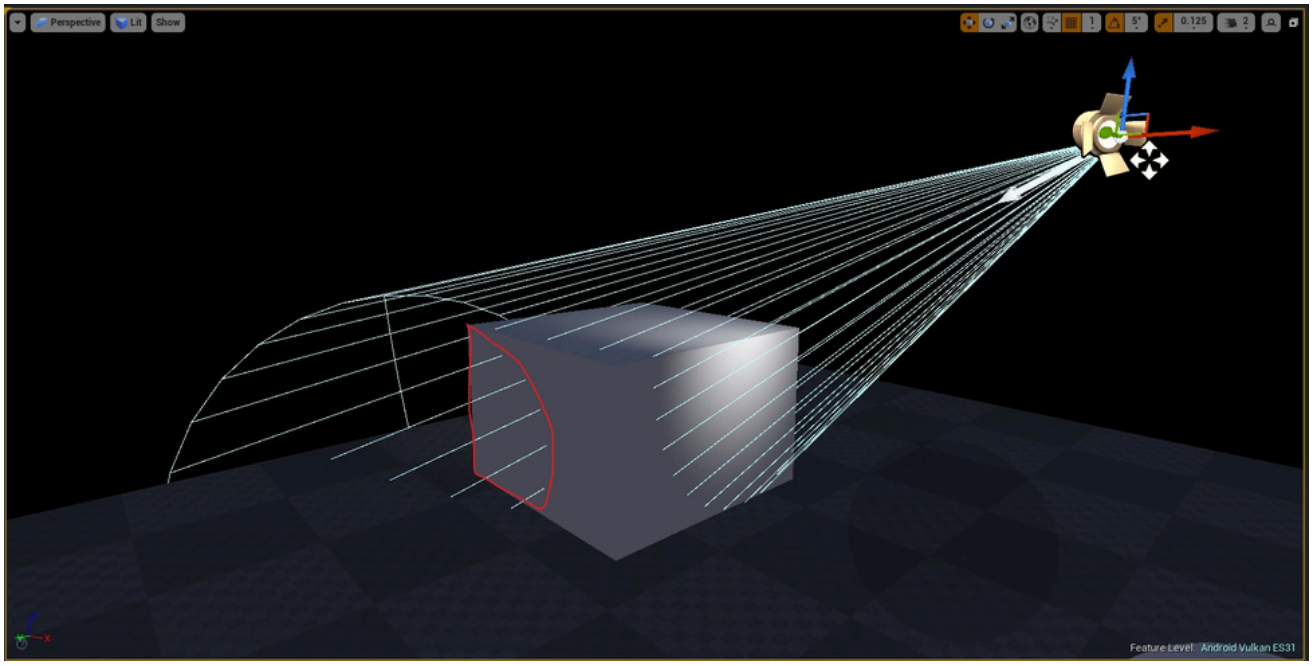




1 main directional light, 64 point lights, baked lighting

Galaxy S9 Mali-G72 MP18
2220x1080

The performance comparison of Mali GPU using different lighting rendering technologies shows that on mobile devices, the template-based clipping lighting algorithm is better than the conventional and block algorithms.

> It is worth mentioning that the light source template cutting technology combined with the GPU's Early-Z technology will greatly improve the lighting rendering performance. The current mainstream mobile GPUs all support Early-Z technology, which also lays the foundation for the application of light source template cutting.
> The light source clipping algorithm currently implemented by UE may still have room for improvement. For example, the pixels facing away from the light source (shown in the red box in the figure below) can actually be ignored. (However, how to quickly and effectively find the pixels facing away from the light source is another problem.)
>
> 
>

12.3.3.2 MobileBasePassShader

This section mainly describes the shaders involved in the mobile BasePass, including VS and PS. Let's look at VS first:

```
// Engine\Shaders\Private\MobileBasePassVertexShader.usf

(.....)

structFMobileShadingBasePassVSToPS {

        FVertexFactoryInterpolantsVSToPS                FactoryInterpolants;
        FMobileBasePassInterpolantsVSToPS float4    BasePassInterpolants;
        Position : SV_POSITION;
};

#define     FMobileShadingBasePassVSOutput FMobileShadingBasePassVSToPS
#define     VertexFactoryGetInterpolants VertexFactoryGetInterpolantsVSToPS

//VSMain entrance.
void Main(
        FVertexFactoryInput          Input
        , out FMobileShadingBasePassVSOutput                Output
#if INSTANCED_STEREO
        , uint InstanceId : SV_InstanceID
        , out uint LayerIndex : SV_RenderTargetArrayIndex
# elif MOBILE_MULTI_VIEW
        , in uint ViewId : SV_ViewID
# endif
        )
{
//Stereoscopic view mode.
#if    INSTANCED_STEREO
        constuint EyeIndex = GetEyeIndex(InstanceId); ResolvedView
        = ResolveView(EyeIndex);
        EyeIndex;
        LayerIndex =
        Output.BasePassInterpolants.MultiViewId Multiple      =float(EyeIndex);
//    view modes.
#elifMOBILE_MULTI_VIEW
        #if COMPILER_GLSL_ES3_1
                const intMultiViewId =int(ViewId); ResolvedView =
                ResolveView(uint(MultiViewId));
                Output.BasePassInterpolants.MultiViewId                  = float(MultiViewId);
        #else
                ResolvedView = ResolveView(ViewId);
                Output.BasePassInterpolants.MultiViewId                  =   float(ViewId);
        #endif
#else
        ResolvedView = ResolveView();
#endif

        //Initialize packed interpolation data.
#if    PACK_INTERPOLANTS
        float4      PackedInterps[NUM_VF_PACKED_INTERPOLANTS];
        UNROLL
        for(inti =0; i < NUM_VF_PACKED_INTERPOLANTS; ++i) {

                PackedInterps[i] =0;
        }
#endif

        //Process vertex factory data.
```

```
        FVertexFactoryIntermediates VFIntermediates = GetVertexFactoryIntermediates(Input); float4
        WorldPositionExcludingWPO = VertexFactoryGetWorldPosition(Input, VFIntermediates);

        float4 WorldPosition = WorldPositionExcludingWPO;

        //Get the vertex data of the material, process coordinates, etc.
        half3x3 TangentToLocal = VertexFactoryGetTangentToLocal(Input, VFIntermediates);
        FMaterialVertexParameters VertexParameters = GetMaterialVertexParameters(Input, VFIntermediates,
WorldPosition.xyz, TangentToLocal);

        half3 WorldPositionOffset = GetMaterialWorldPositionOffset(VertexParameters);

        WorldPosition.xyz += WorldPositionOffset;

        float4 RasterizedWorldPosition = VertexFactoryGetRasterizedWorldPosition(Input, VFIntermediates,
        WorldPosition);
        Output.Position = mul(RasterizedWorldPosition, ResolvedView.TranslatedWorldToClip);
        Output.BasePassInterpolants.PixelPosition = WorldPosition;

#if    USE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS
        Output.BasePassInterpolants.PixelPositionExcludingWPO                =   WorldPositionExcludingWPO.xyz;
#endif

        //Crop surface.
#if USE_PS_CLIP_PLANE
        Output.BasePassInterpolants.OutClipDistance float4(WorldP=osditoiot(nR.exsyoz l-v edView.GlobalClippingPlane,
ResolvedView.PreViewTranslation.xyz,1));
#endif

        //Vertex fog.
#if    USE_VERTEX_FOG
        float4 VertexFog = CalculateHeightFog(WorldPosition.xyz -
        ResolvedView.TranslatedWorldCameraOrigin);

        #ifPROJECT_SUPPORT_SKY_ATMOSPHERE && MATERIAL_IS_SKY==0// Do not apply aerial
Perspective on sky materials
                if(ResolvedView.SkyAtmosphereApplyCameraAerialPerspectiveVolume >0.0f) {

                        const floatOneOverPreExposure = USE_PREEXPOSURE ?
ResolvedView.OneOverPreExposure:1.0f;
                        // Sample the aerial perspective (AP). It is also blended under the VertexFog
parameter.
                        VertexFog = GetAerialPerspectiveLuminanceTransmittanceWithFogOver(
                                ResolvedView.RealTimeReflectionCapture,
ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeSizeAndInvSize,
                                Output.Position, WorldPosition.xyz*CM_TO_SKY_UNIT,
ResolvedView.TranslatedWorldCameraOrigin*CM_TO_SKY_UNIT,
                                View.CameraAerialPerspectiveVolume,
View.CameraAerialPerspectiveVolumeSampler,
                                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthResolutionInv,
                                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthResolution,
                                ResolvedView.SkyAtmosphereAerialPerspectiveStartDepthKm,
                                ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthSliceLengthKm,

ResolvedView.SkyAtmosphereCameraAerialPerspectiveVolumeDepthSliceLengthKmInv,
                                OneOverPreExposure, VertexFog);
                }
        #endif
```

```
    #if    PACK_INTERPOLANTS
        PackedInterps[0]        =  VertexFog;
    #else
        Output.BasePassInterpolants.VertexFog               =  VertexFog;
    #endif    // PACK_INTERPOLANTS
#endif//      USE_VERTEX_FOG

    (......)

    //Get the data to be interpolated.
    Output.FactoryInterpolants = VertexFactoryGetInterpolants(Input, VFIntermediates, VertexParameters);


    Output.BasePassInterpolants.PixelPosition.w = Output.Position.w;

    //Packing interpolation data.
#if    PACK_INTERPOLANTS
    VertexFactoryPackInterpolants(Output.FactoryInterpolants,                    PackedInterps);
#endif// PACK_INTERPOLANTS

#if!OUTPUT_MOBILE_HDR && COMPILER_GLSL_ES3_1
    Output.Position.y *=-1;
#endif
}
```

As can be seen above, view instances are handled differently depending on stereo rendering, multiview, and normal modes. Vertex fog is supported, but it is disabled by default and needs to be enabled in the project configuration.

There is a packed interpolation mode to compress the interpolation consumption and bandwidth between VS and PS. Whether it is enabledPACK_INTERPOLANTSis determined by a macro, which is defined as follows:

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

#definePACK_INTERPOLANTS (USE_VERTEX_FOG && NUM_VF_PACKED_INTERPOLANTS > 0 &&
  (ES3_1_PROFILE))
```

That is to say, the feature of packing interpolation is enabled only when vertex fog is enabled, vertex factory packing interpolation data exists, and the OpenGLES3.1 shading platform is used. Compared with the VS of BasePass on the PC side, the VS on the mobile side has been greatly simplified and can be simply considered as a small subset of the PC side. Let's continue to analyze PS:

```
// Engine\Shaders\Private\MobileBasePassPixelShader.usf

#include"Common.ush"

//Various macro definitions.
#define    MobileSceneTextures      MobileBasePass.SceneTextures
#define    EyeAdaptationStruct      MobileBasePass

(......)
```

```
//A pre-normalized capture of the scene closest to the object being rendered (not supported for fully rough materials)
#if !FULLY_ROUGH
    #ifHQ_REFLECTIONS
    #defineMAX_HQ_REFLECTIONS 3
        TextureCube ReflectionCubemap0;
    SamplerState ReflectionCubemapSampler0;
    TextureCube ReflectionCubemap1;
    SamplerState ReflectionCubemapSampler1;
    TextureCube ReflectionCubemap2;
    SamplerState ReflectionCubemapSampler2;
    // x,y,z - inverted average brightness for 0, 1, 2; w - sky cube texture max mips. float4
            ReflectionAverageBrigtness; ReflectanceMaxValueRGBM;
    float4 ReflectionPositionsAndRadii[MAX_HQ_REFLECTIONS];
    float4 ALLOW_CUBE_REFLECTIONS
        #if
        float4x4 CaptureBoxTransformArray[MAX_HQ_REFLECTIONS];
        float4 CaptureBoxScalesArray[MAX_HQ_REFLECTIONS];
        #endif
    #endif
#endif

//Reflection ball/IBLWaiting for interface.
half4GetPlanarReflection(float3 WorldPosition, half3 WorldNormal, half Roughness); half
MobileComputeMixingWeight(half IndirectIrradiance, half AverageBrightness, half Roughness);

half3GetLookupVectorForBoxCaptureMobile(half3 ReflectionVector, ...); half3GetLookupVectorForSphereCaptureMobile
(half3 ReflectionVector, ...); void GatherSpecularIBL(FMaterialPixelParameters MaterialParameters, ...); void
BlendReflectionCaptures(FMaterialPixelParameters MaterialParameters, ...) half3GetImageBasedReflectionLighting
(FMaterialPixelParameters MaterialParameters, ...);


//Other interfaces.
half3 FrameBufferBlendOp(half4 Source);
bool UseCSM();
void ApplyPixelDepthOffsetForMobileBasePass(inout FMaterialPixelParameters
MaterialParameters, FPixelMaterialInputs PixelMaterialInputs, outfloatOutDepth);

//Accumulated dynamic point lights.
#ifMAX_DYNAMIC_POINT_LIGHTS > 0
void AccumulateLightingOfDynamicPointLight(
    FMaterialPixelParameters MaterialParameters,
    FMobileShadingModelContext ShadingModelContext,
    FGBufferData GBuffer,
    float4 floatLightPositionAndInvRadius, LightColorAndFalloffExponent,
    #ifSUPPORT_SPOTLIGHTS SpotLightDirectionAndSpecularScale,
    W SpotLightAnglesAndSoftTransitionScaleAndLightShadowType,


      FPCFSamplerSettings Settings,
    float4 SpotLightShadowSharpenAndShadowFadeFraction,
    float4 SpotLightShadowmapMinMax,
    float4x4 SpotLightShadowWorldToShadowMatrix,
    # endif
    inout half3 Color)
{
    uint LightShadowType = SpotLightAnglesAndSoftTransitionScaleAndLightShadowType.w; floatFadedShadow =
    1.0f;
```

```
        //Compute spotlight shadows.
#if     SUPPORT_SPOTLIGHTS_SHADOW
        if((LightShadowType & LightShadowType_Shadow) == LightShadowType_Shadow) {



                float4 HomogeneousShadowPosition =
mul(float4(MaterialParameters.AbsoluteWorldPosition,1),
SpotLightShadowWorldToShadowMatrix);
                float2 ShadowUVs = HomogeneousShadowPosition.xy / HomogeneousShadowPosition.w; if
                (all(ShadowUVs >= SpotLightShadowmapMinMax.xy && ShadowUVs <= SpotLightShadowmapMinMax.zw))


                {
                        // Clamp pixel depth in light space for shadowing opaque, because areas of the
shadow depth buffer that weren't rendered to will have been cleared to 1
                        // We want to force the shadow comparison to result in 'unshadowed' in that
case, regardless of whether the pixel being shaded is in front or behind that plane
                        floatLightSpacePixelDepthForOpaque = min(HomogeneousShadowPosition.z,
0.99999f);
                        Settings.SceneDepth = LightSpacePixelDepthForOpaque;
                        Settings.TransitionScale =
SpotLightAnglesAndSoftTransitionScaleAndLightShadowType.z;

                        half Shadow = MobileShadowPCF(ShadowUVs, Settings);

                        Shadow = saturate((Shadow -0.5) *
SpotLightShadowSharpenAndShadowFadeFraction.x +0.5);

                        FadedShadow = lerp(1.0f, Square(Shadow),
SpotLightShadowSharpenAndShadowFadeFraction.y);
                }
        }
#endif

        //Calculate lighting.
        if((LightShadowType & ValidLightType) !=0) {

                float3 ToLight = LightPositionAndInvRadius.xyz -
                MaterialParameters.AbsoluteWorldPosition;
                floatDistanceSqr = dot(ToLight, ToLight); float3 L =
                ToLight * rsqrt(DistanceSqr);
                half3 PointH = normalize(MaterialParameters.CameraVector + L);


                halfPointNoL = max(0, dot(MaterialParameters.WorldNormal, L)); half PointNoH = max(0,
                dot(MaterialParameters.WorldNormal, PointH));


                //Calculates the attenuation of a light source.
                floatAttenuation;
                if(LightColorAndFalloffExponent.w ==0) {

                        // Sphere falloff (technically just 1/d2 but this avoids inf) Attenuation =1/
                        (DistanceSqr +1);

                        floatLightRadiusMask = Square(saturate(1- Square(DistanceSqr *
(LightPositionAndInvRadius.w * LightPositionAndInvRadius.w))));
                        Attenuation *= LightRadiusMask;
                }
                else
```

```
            {
                Attenuation = RadialAttenuation(ToLight * LightPositionAndInvRadius.w,
LightColorAndFalloffExponent.w);
            }

#ifPROJECT_MOBILE_ENABLE_MOVABLE_SPOTLIGHTS
            if((LightShadowType & LightShadowType_SpotLight) == LightShadowType_SpotLight) {

                Attenuation *= SpotAttenuation(L, -SpotLightDirectionAndSpecularScale.xyz,
SpotLightAnglesAndSoftTransitionScaleAndLightShadowType.xy) * FadedShadow;
            }
#endif

            //Accumulate lighting results.
#if !FULLY_ROUGH
            FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelContext, GBuffer, PointNoL,
MaterialParameters.CameraVector, PointH, PointNoH);
            Color += min(65000.0, (Attenuation) * LightColorAndFalloffExponent.rgb * (1.0/PI) * (Lighting.Diffuse +
Lighting.Specular * SpotLightDirectionAndSpecularScale.w));
#else
            Color += (Attenuation * PointNoL) * LightColorAndFalloffExponent.rgb * (1.0/PI)
* ShadingModelContext.DiffuseColor;
#endif
        }
}
#endif

(.....)

//Calculates indirect lighting.
halfComputeIndirect(VTPageTableResult LightmapVTPageTableResult, FVertexFactoryInterpolantsVSToPS
Interpolants, float3 DiffuseDir, FMobileShadingModelContext ShadingModelContext, out half IndirectIrradiance, out
half3 Color)

{
    //To keep IndirectLightingCache conherence with PC, initialize the IndirectIrradiance to zero.

    IndirectIrradiance =0; Color =0;


    //Indirect diffuse reflection.
#if  LQ_TEXTURE_LIGHTMAP
    float2 LightmapUV0,        LightmapUV1;
    uint LightmapDataIndex;
    GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1, LightmapDataIndex);

    half4 LightmapColor = GetLightMapColorLQ(LightmapVTPageTableResult, LightmapUV0, LightmapUV1,
LightmapDataIndex, DiffuseDir);
    Color += LightmapColor.rgb * ShadingModelContext.DiffuseColor *
    View.IndirectLightingColorScale;
    IndirectIrradiance = LightmapColor.a;
#elifCACHED_POINT_INDIRECT_LIGHTING
    #ifMATERIALBLENDING_MASKED || MATERIALBLENDING_SOLID //Apply normals
        to semi-transparent objects.
        FThreeBandSHVectorRGB PointIndirectLighting;
        PointIndirectLighting.R.V0 =
IndirectLightingCache.IndirectLightingSHCoefficients0[0];
        PointIndirectLighting.R.V1 =
```

```
IndirectLightingCache.IndirectLightingSHCoefficients1[0];
            PointIndirectLighting.R.V2 =
IndirectLightingCache.IndirectLightingSHCoefficients2[0];

            PointIndirectLighting.G.V0 =
IndirectLightingCache.IndirectLightingSHCoefficients0[1];
            PointIndirectLighting.G.V1 =
IndirectLightingCache.IndirectLightingSHCoefficients1[1];
            PointIndirectLighting.G.V2 =
IndirectLightingCache.IndirectLightingSHCoefficients2[1];

            PointIndirectLighting.B.V0 =
IndirectLightingCache.IndirectLightingSHCoefficients0[2];
            PointIndirectLighting.B.V1 =
IndirectLightingCache.IndirectLightingSHCoefficients1[2];
            PointIndirectLighting.B.V2 =
IndirectLightingCache.IndirectLightingSHCoefficients2[2];

            FThreeBandSHVector DiffuseTransferSH = CalcDiffuseTransferSH3(DiffuseDir,1);

            //Computes diffuse lighting with normals factored in.
            half3DiffuseGI = max(half3(0,0,0), DotSH3(PointIndirectLighting, DiffuseTransferSH));


            IndirectIrradiance = Luminance(DiffuseGI);
            Color += ShadingModelContext.DiffuseColor * DiffuseGI *
            View.IndirectLightingColorScale;
        #else
            //Translucent uses no direction (Non-directional),Diffuse reflections are packaged inxyz,Already incpuExceptPI
            andSHDiffuse reflection. half3 PointIndirectLighting =
            IndirectLightingCache.IndirectLightingSHSingleCoefficient.rgb;
            half3 DiffuseGI = PointIndirectLighting;

            IndirectIrradiance = Luminance(DiffuseGI);
            Color += ShadingModelContext.DiffuseColor * DiffuseGI *
            View.IndirectLightingColorScale;
        #endif
#endif

        returnIndirectIrradiance;
}

//PSMain entrance.
PIXELSHADER_EARLYDEPTHSTENCIL
void Main(
        FVertexFactoryInterpolantsVSToPS ,          Interpolants
        FMobileBasePassInterpolantsVSToPS               BasePassInterpolants
        , in float4 SvPosition : SV_Position
        OPTIONAL_IsFrontFace
        , out half4 OutColor                : SV_Target0
#if DEFERRED_SHADING_PATH
        , out    half4    OutGBufferA        :   SV_Target1
        , out    half4    OutGBufferB        :   SV_Target2
        , out    half4    OutGBufferC        :   SV_Target3
# endif
#if USE_SCENE_DEPTH_AUX
        , outfloatOutSceneDepthAux : SV_Target4
# endif
```

```
#if	OUTPUT_PIXEL_DEPTH_OFFSET, outfloat
	OutDepth : SV_Depth
# endif
	)
{
# ifMOBILE_MULTI_VIEW
	ResolvedView	=	ResolveView(uint(BasePassInterpolants.MultiViewId));
#else
	ResolvedView	=	ResolveView();
#endif

#if	USE_PS_CLIP_PLANE
	clip(BasePassInterpolants.OutClipDistance);
#endif

	//Unpack packed interpolation data.
#if	PACK_INTERPOLANTS
	float4 PackedInterpolants[NUM_VF_PACKED_INTERPOLANTS];
	VertexFactoryUnpackInterpolants(Interpolants, PackedInterpolants);
#endif

#ifCOMPILER_GLSL_ES3_1 && !OUTPUT_MOBILE_HDR && !MOBILE_EMULATION // LDR
	Mobile needs screen vertical flipped
	SvPosition.y = ResolvedView.BufferSizeAndInvSize.y - SvPosition.y -1;
#endif

	//Gets the pixel properties of a material.
	FMaterialPixelParameters MaterialParameters = GetMaterialPixelParameters(Interpolants, SvPosition);

	FPixelMaterialInputs PixelMaterialInputs; {

		float4 ScreenPosition = SvPositionToResolvedScreenPosition(SvPosition); float3 WorldPosition =
		BasePassInterpolants.PixelPosition.xyz;
		float3 WorldPositionExcludingWPO = BasePassInterpolants.PixelPosition.xyz;
		#if USE_WORLD_POSITION_EXCLUDING_SHADER_OFFSETS
			WorldPositionExcludingWPO = BasePassInterpolants.PixelPositionExcludingWPO;
		#endif
		CalcMaterialParametersEx(MaterialParameters, PixelMaterialInputs, SvPosition, ScreenPosition,
bIsFrontFace, WorldPosition, WorldPositionExcludingWPO);

#ifFORCE_VERTEX_NORMAL
		// Quality level override of material's normal calculation, can be used to avoid normal map reads etc.

		MaterialParameters.WorldNormal = MaterialParameters.TangentToWorld[2];
		MaterialParameters.ReflectionVector =
ReflectionAboutCustomWorldNormal(MaterialParameters, MaterialParameters.WorldNormal, false);

#endif
	}

	//Pixel depth offset.
#if	OUTPUT_PIXEL_DEPTH_OFFSET
	ApplyPixelDepthOffsetForMobileBasePass(MaterialParameters, OutDepth);	PixelMaterialInputs,

#endif

	// MaskMaterial.
#if	!EARLY_Z_PASS_ONLY_MATERIAL_MASKING
```

```
    //Clip if the blend mode requires it.
    GetMaterialCoverageAndClipping(MaterialParameters,                  PixelMaterialInputs);
#endif

    //Calculate and cacheGBufferdata, preventing the subsequent use
    of textures multiple times. FGBufferData GBuffer = (FGBufferData)0;
    GBuffer.WorldNormal             = MaterialParameters.WorldNormal;
    GBuffer.BaseColor       =GetMaterialBaseColor(PixelMaterialInputs);
    GBuffer.Metallic        =GetMaterialMetallic(PixelMaterialInputs);
    GBuffer.Specular        =GetMaterialSpecular(PixelMaterialInputs);
    GBuffer.Roughness       =GetMaterialRoughness(PixelMaterialInputs);
    GBuffer.ShadingModelID          = GetMaterialShadingModel(PixelMaterialInputs);
    half MaterialAO = GetMaterialAmbientOcclusion(PixelMaterialInputs);

    //ApplicationsAO.
#if APPLY_AO
    half4 GatheredAmbientOcclusion = Texture2DSample(AmbientOcclusionTexture,
AmbientOcclusionSampler, SvPositionToBufferUV(SvPosition));

    MaterialAO *= GatheredAmbientOcclusion.r;
#endif

    GBuffer.GBufferAO = MaterialAO;

    //becauseIEEE 754 (FP16)The minimum standard value that can be expressed is2^-24 = 5.96e-8,The roughness behind this involves
1.0 / Roughness^4So in order to prevent division by zero errors, we need to ensureRoughness^4 >= 5.96e-8,Directly hereClampRoughness
to 0.015625(0.015625^4 = 5.96e-8).
    //In addition, in order to matchPCDeferred rendering on the end (roughness is stored in8bit value), so it is also automatically
    Clamparrive1.0. GBuffer.Roughness = max(0.015625, GetMaterialRoughness(PixelMaterialInputs));

    //Initialize the mobile coloring model contextFMobileShadingModelContext. FMobileShadingModelContext
    ShadingModelContext = (FMobileShadingModelContext)0; ShadingModelContext.Opacity =
    GetMaterialOpacity(PixelMaterialInputs);

    //Thin layer transparency
#if  MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT
    (......)
#endif

    half3 Color =0;

    //Custom data.
    half CustomData0 = GetMaterialCustomData0(MaterialParameters); half CustomData1 =
    GetMaterialCustomData1(MaterialParameters); InitShadingModelContext(ShadingModelContext, GBuffer,
    MaterialParameters.SvPosition, MaterialParameters.CameraVector, CustomData0, CustomData1);

    float3 DiffuseDir = MaterialParameters.WorldNormal;

    //Hair model.
#if  MATERIAL_SHADINGMODEL_HAIR
    (......)
#endif

    //Lightmap virtual texture.
    VTPageTableResult LightmapVTPageTableResult = (VTPageTableResult)0.0f;
#if  LIGHTMAP_VT_ENABLED
    {
        float2 LightmapUV0, LightmapUV1;
```

```
            uint LightmapDataIndex;
            GetLightMapCoordinates(Interpolants, LightmapUV0, LightmapUV1, LightmapDataIndex);
            LightmapVTPageTableResult = LightmapGetVTSampleInfo(LightmapUV0,
LightmapDataIndex, SvPosition.xy);
        }
#endif

#if    LIGHTMAP_VT_ENABLED
        // This must occur after CalcMaterialParameters(), which is required to initialize the feedback mechanism
VT
        // Lightmap request is always the first VT sample in the shader
        StoreVirtualTextureFeedback(MaterialParameters.VirtualTextureFeedback,0,
        LightmapVTPageTableResult.PackedRequest);
#endif


        //Calculates indirect light.
        half    IndirectIrradiance;
        half3    IndirectColor;
        ComputeIndirect(LightmapVTPageTableResult, ShadingMoIdnetelCrpoonltaenDtiffuseDir,
 IndirectIrradiance, IndirectColor);
         Color += IndirectColor;


        //Precomputed shadow maps.
        half Shadow = GetPrimaryPrecomputedShadowMask(LightmapVTPageTableResult,
        Interpolants).r;

#if    DEFERRED_SHADING_PATH
        float4      OutGBufferD;
        float4      OutGBufferE;
        float4      OutGBufferF;
        float4 OutGBufferVelocity =0;

        GBuffer.IndirectIrradiance = IndirectIrradiance;
        GBuffer.PrecomputedShadowFactors.r = Shadow;

        //codingGBufferdata.
        EncodeGBuffer(GBuffer, OutGBufferA, OutGBufferB, OutGBufferC, OutGBufferD, OutGBufferE,
OutGBufferF, OutGBufferVelocity);
#else

#if!MATERIAL_SHADINGMODEL_UNLIT

        //Skylight.
#if    ENABLE_SKY_LIGHT
        half3 SkyDiffuseLighting = GetSkySHDiffuseSimple(MaterialParameters.WorldNormal); half3 DiffuseLookup =
        SkyDiffuseLighting * ResolvedView.SkyLightColor.rgb; IndirectIrradiance += Luminance(DiffuseLookup);

#endif

        Color *= MaterialAO;
        IndirectIrradiance *= MaterialAO;

        floatShadowPositionZ =0;
#ifDIRECTIONAL_LIGHT_CSM && !MATERIAL_SHADINGMODEL_SINGLELAYERWATER //
            CSMshadow.
        if    (UseCSM())
        {
            half ShadowMap = MobileDirectionalLightCSM(MaterialParameters.ScreenPosition.xy,
```

```
MaterialParameters.ScreenPosition.w,                ShadowPositionZ);
    #if   ALLOW_STATIC_LIGHTING
          Shadow = min(ShadowMap, Shadow);
    #else
          Shadow = ShadowMap;
    #endif
    }
#endif/* DIRECTIONAL_LIGHT_CSM */

    //Distance Field Shadows.
#if   APPLY_DISTANCE_FIELD
    if(ShadowPositionZ ==0) {

          Shadow = Texture2DSample(MobileBasePass.ScreenSpaceShadowMaskTexture,
MobileBasePass.ScreenSpaceShadowMaskSampler, SvPositionToBufferUV(SvPosition)).x;
    }
#endif

    halfNoL = max(0, dot(MaterialParameters.WorldNormal,
    MobileDirectionalLight.DirectionalLightDirectionAndShadowTransition.xyz));
    half3 H = normalize(MaterialParameters.CameraVector +
    MobileDirectionalLight.DirectionalLightDirectionAndShadowTransition.xyz);
    half NoH = max(0, dot(MaterialParameters.WorldNormal, H));

    //Parallel light +IBL
#if   FULLY_ROUGH
    Color += (Shadow * NoL) * MobileDirectionalLight.DirectionalLightColor.rgb *
    ShadingModelContext.DiffuseColor;
#else
    FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelContext, GBuffer, NoL,
MaterialParameters.CameraVector, H, NoH);
    // MobileDirectionalLight.DirectionalLightDistanceFadeMADAndSpecularScale.zSave parallel lightSpecularScale.

    Color += (Shadow) * MobileDirectionalLight.DirectionalLightColor.rgb * (Lighting.Diffuse +
Lighting.Specular *
MobileDirectionalLight.DirectionalLightDistanceFadeMADAndSpecularScale.z);

    //   Hair coloring.
#if       !(MATERIAL_SINGLE_SHADINGMODEL       && MATERIAL_SHADINGMODEL_HAIR)
    (.....)
#endif
#endif/* FULLY_ROUGH */

    //Local light source, up to4indivual.
#ifMAX_DYNAMIC_POINT_LIGHTS > 0 && !MATERIAL_SHADINGMODEL_SINGLELAYERWATER

          if(NumDynamicPointLights >0) {


              #if   SUPPORT_SPOTLIGHTS_SHADOW FPCFSamplerSettings Settings;
                    Settings.ShadowDepthTexture = Settings.ShadowDepthTextureSampler
                    Settings.ShadowBufferSize = DynamicSpoDtyLnigahmtSichSapdootwLiBguhftfSehraSdizoew; Texture;
                    Settings.bSubsurface =false; Settings.bTreatMax=D DeypnthamUnicsShpaodtoLwigehdtS =hfadlsoew; Sampler;
                    Settings.DensityMulConstant =0; Settings.ProjectionDepthBiasParameters =0;
```

```
                #endif

                AccumulateLightingOfDynamicPointLight(MaterialParameters, ...);

                if(MAX_DYNAMIC_POINT_LIGHTS >1&& NumDynamicPointLights >1) {

                    AccumulateLightingOfDynamicPointLight(MaterialParameters, ...);

                    if(MAX_DYNAMIC_POINT_LIGHTS >2&& NumDynamicPointLights >2) {

                        AccumulateLightingOfDynamicPointLight(MaterialParameters, ...);

                        if(MAX_DYNAMIC_POINT_LIGHTS >3&& NumDynamicPointLights >3) {

                            AccumulateLightingOfDynamicPointLight(MaterialParameters, ...);
                        }
                    }
                }
            }

#endif

    //Sky light.
#if  ENABLE_SKY_LIGHT
    #ifMATERIAL_TWOSIDED && LQ_TEXTURE_LIGHTMAP if(NoL
    ==0)
    {
    #endif

    #if  MATERIAL_SHADINGMODEL_SINGLELAYERWATER
        ShadingModelContext.WaterDiffuseIndirectLuminance          + =SkyDiffuseLighting;
    #endif
        Color += SkyDiffuseLighting * half3(ResolvedView.SkyLightColor.rgb) *
ShadingModelContext.DiffuseColor * MaterialAO;
    #ifMATERIAL_TWOSIDED && LQ_TEXTURE_LIGHTMAP }

    #endif
#endif

#endif/* !MATERIAL_SHADINGMODEL_UNLIT */

#if  MATERIAL_SHADINGMODEL_SINGLELAYERWATER
    (.....)
#endif// MATERIAL_SHADINGMODEL_SINGLELAYERWATER

#endif//DEFERRED_SHADING_PATH

    //Handles vertex fog.
    half4VertexFog = half4(0,0,0,1); USE_VERTEX_FOG
#i
f   PACK_INTERPOLANTS
#i  VertexFog   =  PackedInterpolants[0];
#else
    VertexFog   =  BasePassInterpolants.VertexFog;
#endif
#endif

    //Self-luminous.
```

```
        half3 Emissive = GetMaterialEmissive(PixelMaterialInputs);
#if    MATERIAL_SHADINGMODEL_THIN_TRANSLUCENT Emissive
                    * = TopMaterialCoverage;
#endif
        Color +=      Emissive;

#if!MATERIAL_SHADINGMODEL_UNLIT && MOBILE_EMULATION
        Color = lerp(Color, ShadingModelContext.DiffuseColor, ResolvedView.UnlitViewmodeMask);
#endif


        //Combines fog color into output color.
        #ifMATERIALBLENDING_ALPHACOMPOSITE || MATERIAL_SHADINGMODEL_SINGLELAYERWATER
            OutColor = half4(Color * VertexFog.a + VertexFog.rgb * ShadingModelContext.Opacity,
            ShadingModelContext.Opacity);
        #elifMATERIALBLENDING_ALPHAHOLDOUT //
            not implemented for holdout
            OutColor = half4(Color * VertexFog.a + VertexFog.rgb *
ShadingModelContext.Opacity, ShadingModelContext.Opacity);
        #elifMATERIALBLENDING_TRANSLUCENT
            OutColor = half4(Color * VertexFog.a + VertexFog.rgb,
            ShadingModelContext.Opacity);
        #elifMATERIALBLENDING_ADDITIVE
            OutColor = half4(Color * (VertexFog.a * ShadingModelContext.Opacity.x),0.0f);
        #elifMATERIALBLENDING_MODULATE
            half3 FoggedColor = lerp(half3(1,1,1), Color, VertexFog.aaa * VertexFog.aaa); OutColor = half4(FoggedColor,
            ShadingModelContext.Opacity);
        #else
            OutColor.rgb = Color * VertexFog.a + VertexFog.rgb;

            #if    !MATERIAL_USE_ALPHA_TO_COVERAGE
                // Scene color alpha is not used yet so we set it to 1 OutColor.a =1.0;


                #if    OUTPUT_MOBILE_HDR
                    // Store depth in FP16 alpha. This depth value can be fetched during
translucency or sampled in post-processing
                    OutColor.a = SvPosition.z;
                #endif
            #else
                half MaterialOpacityMask = GetMaterialMaskInputRaw(PixelMaterialInputs); OutColor.a =
                GetMaterialMask(PixelMaterialInputs) /
max(abs(ddx(MaterialOpacityMask)) +abs(ddy(MaterialOpacityMask)),0.0001f) +0.5f;
            #endif
        #endif

        #if!MATERIALBLENDING_MODULATE && USE_PREEXPOSURE
            OutColor.rgb *= ResolvedView.PreExposure;
        #endif

        #if    MATERIAL_IS_SKY
            OutColor.rgb = min(OutColor.rgb, Max10BitsFloat.xxx *0.5f);
        #endif

#if    USE_SCENE_DEPTH_AUX
        OutSceneDepthAux =      SvPosition.z;
#endif


        //Processing Coloralpha.
```

```
#ifUSE_EDITOR_COMPOSITING && (MOBILE_EMULATION) //
        Editor primitive depth testing OutColor.a
                        =1.0;
        #if MATERIALBLENDING_MASKED
              // some material might have an opacity value OutColor.a =
                                  GetMaterialMaskInputRaw(PixelMaterialInputs);
        #endif
        clip(OutColor.a          - GetMaterialOpacityMaskClipValue());
#else
        #ifOUTPUT_GAMMA_SPACE OutColor.rgb =sqrt
          (OutColor.rgb);
        #endif
#endif

#ifNUM_VIRTUALTEXTURE_SAMPLES || LIGHTMAP_VT_ENABLED
  FinalizeVirtualTextureFeedback(
              MaterialParameters.VirtualTextureFeedback,
              MaterialParameters.SvPosition,
              ShadingModelContext.Opacity,
              View.FrameNumber,
              View.VTFeedbackBuffer
        ) ;
#endif
}
```

The processing of BasePassPS on mobile terminals is relatively complicated and involves many steps, including: decompressing interpolated data, obtaining and calculating material properties, calculating and caching GBuffer, processing or adjusting GBuffer data, calculating lighting (parallel light, local light) for the forward rendering branch, calculating distance fields, CSM and other shadows, calculating sky light, processing static lighting, indirect light and IBL, calculating fog effects, and processing special shading models such as water, hair, and thin layer transparency.

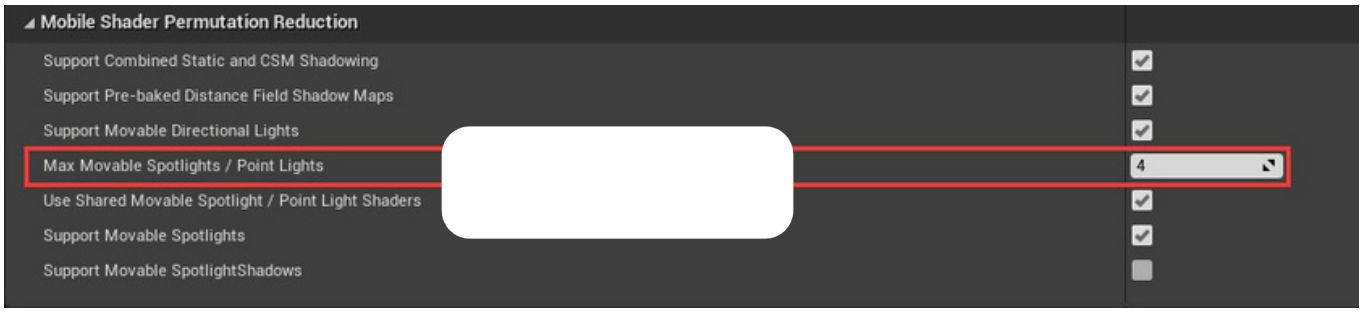
Since the minimum value that can be represented by a standard 16-bit floating-point number (FP16) is , and the subsequent lighting calculation involves the fourth power operation of the roughness ( ), In order to prevent division by zero errors, the roughness needs to be truncated to ( ).

$$\frac{1.0}{2^{\text{twenty four}}} = 5.96 \cdot 10^{8} \quad \frac{1.0}{\text{Roughness}^4} \quad 0.015625 \quad 0.015625^4 = 5.96 \cdot 10^{8}$$

```
GBuffer.Roughness = max(0.015625, GetMaterialRoughness(PixelMaterialInputs));
```

> **This also warns us that when developing rendering features for mobile devices, we need to pay special attention to and control data accuracy. Otherwise, various strange screen anomalies will often occur on low-end devices due to insufficient data accuracy.**

Although there is a lot of code above, it is controlled by many macros, and the code required to actually render a single material may be only a small subset of it. For example, 4 local light sources are supported by default, but if it can be set to 2 or less in the project configuration (see the figure below), the actual number of light source instructions executed is much less.



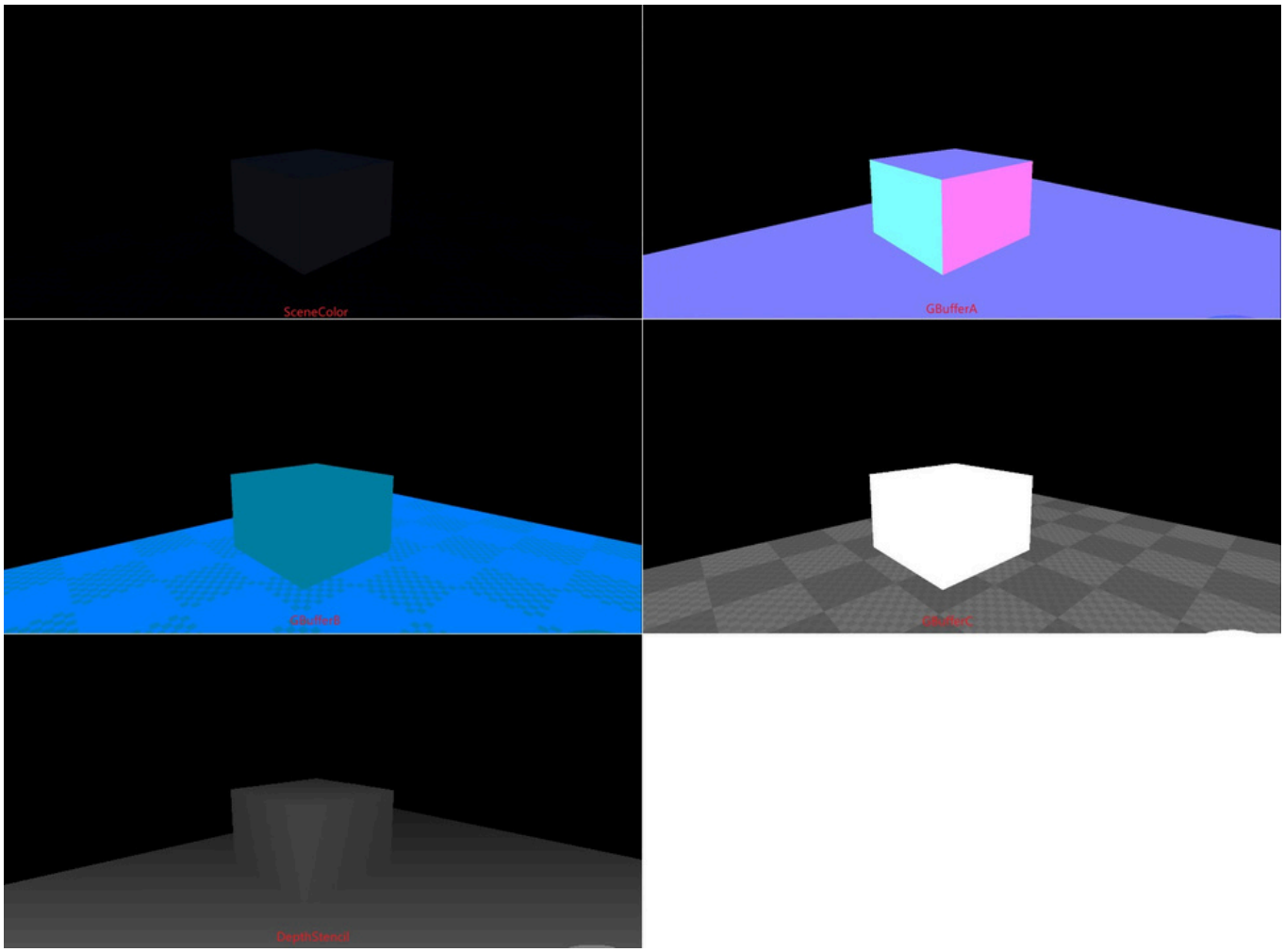
If it is a forward rendering branch, many GBuffer processes will be ignored; if it is a delayed rendering branch, the calculation of parallel light and local light source will be ignored and executed by the shader of the delayed rendering Pass.

EncodeGBufferThe following is an analysis of the important interfaces:

```
void EncodeGBuffer(
	FGBufferData  GBuffer,
	out  float4  outOutGBufferA,
	float4	outOutGBufferB,
	float4	outOutGBufferC,
	float4	outOutGBufferD,
	float4	outOutGBufferE,
	float4		OutGBufferVelocity,
	floatQuantizationBias =0			// -0.5 to 0.5 random float. Used to bias
	quantization.
	)
{
	if(GBuffer.ShadingModelID == SHADINGMODELID_UNLIT) {

		OutGBufferA =0;
		SetGBufferForUnlit(OutGBufferB);
		OutGBufferC	=  0;
		OutGBufferD	=  0;
		OutGBufferE	= 0;
	}
	else
	{
		// GBufferA:Octahedron compressed normals, precomputed shadow factors, per-object data.
#ifMOBILE_DEFERRED_SHADING
		OutGBufferA.rg = UnitVectorToOctahedron( normalize(GBuffer.WorldNormal) ) *0.5f+
0.5f;
		OutGBufferA.b	=  GBuffer.PrecomputedShadowFactors.x;
		OutGBufferA.a	=  GBuffer.PerObjectGBufferData;
#else
		(.....)
#endif

		// GBufferB:Metallicity, Specular, Roughness, Color Model, OtherMask.
		OutGBufferB.r	=  GBuffer.Metallic;
		OutGBufferB.g	=  GBuffer.Specular;
		OutGBufferB.b	=  GBuffer.Roughness;
		OutGBufferB.a	=  EncodeShadingModelIdAndSelectiveOutputMask(GBuffer.ShadingModelID,
GBuffer.SelectiveOutputMask);

		// GBufferC:Basic color,AOor indirect light.
		OutGBufferC.rgb = EncodeBaseColor( GBuffer.BaseColor );
```

```
#ifALLOW_STATIC_LIGHTING
            // No space for AO. Multiply IndirectIrradiance by AO instead of storing. OutGBufferC.a =
            EncodeIndirectIrradiance(GBuffer.IndirectIrradiance * GBuffer.GBufferAO) + QuantizationBias * (
1.0/255.0);
#else
            OutGBufferC.a = GBuffer.GBufferAO;
#endif

            OutGBufferD   =  GBuffer.CustomData;
            OutGBufferE   =  GBuffer.PrecomputedShadowFactors;
      }

#if   WRITES_VELOCITY_TO_GBUFFER
      OutGBufferVelocity      =  GBuffer.Velocity;
#else
      OutGBufferVelocity      =  0;
#endif
}
```

Under the default lighting model (DefaultLit), the output of BasePass has the following textures:



12.3.3.3 MobileDeferredShading

The VS and PC versions of the mobile version of the delayed lighting are the same, both are DeferredLightVertexShaders.usf, but PS is different, using MobileDeferredShading.usf. Since VS is the

same as PC and there is no special operation, it is ignored here. If you are interested, you can read Section5.5.3.1 DeferredLightVertexShader in the fifth article .

Let's analyze the PS code directly:

```
// Engine\Shaders\Private\MobileDeferredShading.usf

(.....)

//Mobile light source data structure.
structFMobileLightData {

    float3    Position;
    float     InvRadius;
    float3    Color;
    float     FalloffExponent;
    float3    Direction;
    float2    SpotAngles;
    float     SourceRadius;
    float     SpecularScale;
    bool bInverseSquared;
    bool bSpotLight;
};

//GetGBufferdata.
void FetchGBuffer(in float2 UV, out float4 GBufferA, out float4 GBufferB, out float4 GBufferC, out float4 GBufferD,
outfloatSceneDepth) {

    // VulkanSon ofpassGet the data.
#ifVULKAN_PROFILE
    GBufferA    =
    GBufferB    VulkanSubpassFetch1();
    GBufferC    =
    GBufferD = 0;ulkanSubpassFetch2();
    SceneDepth=  = ConvertFromDeviceZ(VulkanSubpassDepthFetch());
    //    MetalSon ofpassGet the data.ssFetch3();
#elif     data. METAL_PROFILE
    GBufferA    =  SubpassFetchRGBA_1();
    GBufferB    =  SubpassFetchRGBA_2();
    GBufferC    =  SubpassFetchRGBA_3();
    GBufferD    =  0;
    SceneDepth    = ConvertFromDeviceZ(SubpassFetchR_4());
    //Other platforms(DX,                OpenGL)Son ofpassGet the data.
#else
    GBufferA = Texture2DSampleLevel(MobileSceneTextures.GBufferATexture,
MobileSceneTextures.GBufferATextureSampler, UV,0);
    GBufferB = Texture2DSampleLevel(MobileSceneTextures.GBufferBTexture,
MobileSceneTextures.GBufferBTextureSampler, UV,0);
    GBufferC = Texture2DSampleLevel(MobileSceneTextures.GBufferCTexture,
    MobileSceneTextures.GBufferCTextureSampler,          UV,0);
    GBufferD =      0;
    SceneDepth =
ConvertFromDeviceZ(Texture2DSampleLevel(MobileSceneTextures.SceneDepthTexture,
MobileSceneTextures.SceneDepthTextureSampler, UV,0).r);
#endif
}
```

```
//UnzipGBufferdata.
FGBufferData      DecodeGBufferMobile(
        float4      InGBufferA,
        float4      InGBufferB,
        float4      InGBufferC,
        float4      InGBufferD)
{
        FGBufferData      GBuffer;
        GBuffer.WorldNormal = OctahedronToUnitVector(InGBufferA.xy *2.0f-1.0f);
        GBuffer.PrecomputedShadowFactors          = InGBufferA.z;
        GBuffer.PerObjectGBufferData          = InGBufferA.a;
        GBuffer.Metallic              = InGBufferB.r;
        GBuffer.Specular              = InGBufferB.g;
        GBuffer.Roughness             =max(0.015625, InGBufferB.b);
        // Note: must match GetShadingModelId standalone function logic
        // Also Note: SimpleElementPixelShader directly sets SV_Target2 ( GBufferB ) to indicate unlit.

        // An update there will be required if this layout changes. GBuffer.ShadingModelID =
        DecodeShadingModelId(InGBufferB.a); GBuffer.SelectiveOutputMask =
        DecodeSelectiveOutputMask(InGBufferB.a); GBuffer.BaseColor =
        DecodeBaseColor(InGBufferC.rgb);
#if     ALLOW_STATIC_LIGHTING
        GBuffer.GBufferAO =1;
        GBuffer.IndirectIrradiance = DecodeIndirectIrradiance(InGBufferC.a);
#else
        GBuffer.GBufferAO = InGBufferC.a;
        GBuffer.IndirectIrradiance =1;
#endif
        GBuffer.CustomData = HasCustomGBufferData(GBuffer.ShadingModelID) ? InGBufferD :0; returnGBuffer;

}

//Direct lighting.
half3GetDirectLighting(
        FMobileLightData LightData,
        FMobileShadingModelContext          ShadingModelContext,
        FGBufferData      GBuffer,
        float3      WorldPosition,
        half3 CameraVector)
{
        half3 DirectLighting =0;

        float3 ToLight = LightData.Position - WorldPosition; floatDistanceSqr =
        dot(ToLight, ToLight); float3 L = ToLight * rsqrt(DistanceSqr);


        //Light source attenuation.
        floatAttenuation =0.0; if
        (LightData.bInverseSquared) {

                // Sphere falloff (technically just 1/d2 but this avoids inf) Attenuation =1.0f/
                (DistanceSqr +1.0f); Attenuation *= Square(saturate(1- Square(DistanceSqr *
                Square(LightData.InvRadius))));

        }
        else
        {
                Attenuation = RadialAttenuation(ToLight * LightData.InvRadius,
```

```
LightData.FalloffExponent);
    }

    //  Spotlight attenuation.
    if  (LightData.bSpotLight)
    {
        Attenuation *= SpotAttenuation(L, -LightData.Direction, LightData.SpotAngles);
    }

    //If the attenuation is not0,Direct lighting is
    then calculated. if(Attenuation >0.0) {

        half3 H = normalize(CameraVector + L);
        halfNoL = max(0.0, dot(GBuffer.WorldNormal, L)); half NoH = max(
        0.0, dot(GBuffer.WorldNormal, H));
        FMobileDirectLighting Lighting = MobileIntegrateBxDF(ShadingModelContext, GBuffer, NoL, CameraVector, H,
        NoH);
        DirectLighting = (Lighting.Diffuse + Lighting.Specular * LightData.SpecularScale)
* (LightData.Color * (1.0/ PI) * Attenuation); }

    returnDirectLighting;
}

//Lighting function.
halfComputeLightFunctionMultiplier(float3 WorldPosition); //Uses a light grid to add local
lighting. Dynamic shadows are not supported because per-light shadow maps are required.
half3GetLightGridLocalLighting(constFCulledLightsGridData InLightGridData, ...);

//Parallel lightPSMain entrance.
void MobileDirectLightPS(
    noperspective float4 UVAndScreenPos : TEXCOORD0, float4
    SvPosition : SV_POSITION,
    out half4 OutColor : SV_Target0)
{
    //Restore (read)GBufferdata. FGBufferData GBuffer =
    (FGBufferData)0; floatSceneDepth =0; {


        float4 GBufferA =0; float4 GBufferB =0; float4 GBufferC =0; float4 GBufferD =0;
        FetchGBuffer(UVAndScreenPos.xy, GBufferA, GBufferB, GBufferC, GBufferD, SceneDepth);




        GBuffer = DecodeGBufferMobile(GBufferA, GBufferB, GBufferC, GBufferD);
    }

    //Compute basis vectors.
    float2 ScreenPos = UVAndScreenPos.zw;
    float3 WorldPosition = mul(float4(ScreenPos * SceneDepth, SceneDepth,1),
    View.ScreenToWorld).xyz;
    half3 CameraVector = normalize(View.WorldCameraOrigin - WorldPosition); half NoV = max(0,
    dot(GBuffer.WorldNormal, CameraVector));
    half3 ReflectionVector = GBuffer.WorldNormal * (NoV *2.0) - CameraVector;

    half3 Color =0;
    // Check movable light param to determine if we should be using precomputed shadows half Shadow =
    LightFunctionParameters2.z >0.0f?1.0f:
```

```
GBuffer.PrecomputedShadowFactors.r;

    // CSMshadow.
#if  APPLY_CSM
    floatShadowPositionZ          = 0;
    float4 ScreenPosition = SvPositionToScreenPosition(float4(SvPosition.xyz,SceneDepth)); floatShadowMap =
    MobileDirectionalLightCSM(ScreenPosition.xy, SceneDepth, ShadowPositionZ);

    Shadow = min(ShadowMap, Shadow);
#endif

    //Coloring model context.
    FMobileShadingModelContext ShadingModelContext = (FMobileShadingModelContext)0; {

        half DielectricSpecular =0.08* GBuffer.Specular; ShadingModelContext.DiffuseColor =
        GBuffer.BaseColor - GBuffer.BaseColor * GBuffer.Metallic;
                        // 1 mad
        ShadingModelContext.SpecularColor = (DielectricSpecular - DielectricSpecular * GBuffer.Metallic) +
GBuffer.BaseColor * GBuffer.Metallic;                                    // 2 mad
        //Computing EnvironmentBRDF.
        ShadingModelContext.SpecularColor = GetEnvBRDF(ShadingModelContext.SpecularColor,
GBuffer.Roughness, NoV);
    }

    //Local light source.
    float2 LocalPosition = SvPosition.xy - View.ViewRectMin.xy;
    uint GridIndex = ComputeLightGridCellIndex(uint2(LocalPosition.x, LocalPosition.y), SceneDepth);

    //Clustered light sources
#if  USE_CLUSTERED
    {
        constuint EyeIndex =0;
        constFCulledLightsGridData CulledLightGridData = GetCulledLightsGrid(GridIndex, EyeIndex);

        Color += GetLightGridLocalLighting(CulledLightGridData, ShadingModelContext, GBuffer,
WorldPosition, CameraVector, EyeIndex,0);
    }
#endif

    //Computes parallel light.
    halfNoL = max(0, dot(GBuffer.WorldNormal,
    MobileDirectionalLight.DirectionalLightDirectionAndShadowTransition.xyz));
    half3 H = normalize(CameraVector +
    MobileDirectionalLight.DirectionalLightDirectionAndShadowTransition.xyz);
    half NoH = max(0, dot(GBuffer.WorldNormal, H));
    FMobileDirectLighting Lighting;
    Lighting.Specular = ShadingModelContext.SpecularColor *
CalcSpecular(GBuffer.Roughness, NoH);
    Lighting.Diffuse = ShadingModelContext.DiffuseColor;
    Color += (Shadow * NoL) * MobileDirectionalLight.DirectionalLightColor.rgb * (Lighting.Diffuse +
Lighting.Specular *
MobileDirectionalLight.DirectionalLightDistanceFadeMADAndSpecularScale.z);

    //Handling reflectionsIBL,Reflection
#if  Catcher). APPLY_REFLECTION
    uint NumCulledEntryIndex = (ForwardLightData.NumGridCells + GridIndex) *
    NUM_CULLED_LIGHTS_GRID_STRIDE;
    uint NumLocalReflectionCaptures =
```

```
min(ForwardLightData.NumCulledLightsGrid[NumCulledEntryIndex +0],
ForwardLightData.NumReflectionCaptures);
        uint DataStartIndex = ForwardLightData.NumCulledLightsGrid[NumCulledEntryIndex +1];

        float3 SpecularIBL = CompositeReflectionCapturesAndSkylight(
            1.0f,                   WorldPosition,
            ReflectionVector,//RayDirection,
            GBuffer.Roughness,
            GBuffer.IndirectIrradiance,   1.0f,
            0.0f,
            NumLocalReflectionCaptures,
            DataStartIndex,
            0,



            true);

        Color += SpecularIBL * ShadingModelContext.SpecularColor;
#elifAPPLY_SKY_REFLECTION
        floatSkyAverageBrightness =1.0f;
        float3 SpecularIBL = GetSkyLightReflection(ReflectionVector, GBuffer.Roughness, SkyAverageBrightness);

        SpecularIBL *= ComputeMixingWeight(GBuffer.IndirectIrradiance, SkyAverageBrightness, GBuffer.Roughness);

        Color += SpecularIBL * ShadingModelContext.SpecularColor;
#endif
        //   Diffuse sky light.
        half3 SkyDiffuseLighting = GetSkySHDiffuseSimple(GBuffer.WorldNormal); Color+=
        SkyDiffuseLighting * half3(View.SkyLightColor.rgb) * ShadingModelContext.DiffuseColor *
        GBuffer.GBufferAO;
        half LightAttenuation = ComputeLightFunctionMultiplier(WorldPosition);

#if   USE_PREEXPOSURE
        // MobileHDR applies PreExposure in tonemapper
        LightAttenuation *= View.PreExposure;
#endif

        OutColor.rgb = Color.rgb * LightAttenuation; OutColor.a =1;

}

//Local light sourcePSMain entrance.
void MobileRadialLightPS(
        float4 InScreenPosition : TEXCOORD0, float4
        SVPos                               : SV_POSITION,
        out half4 OutColor                     : SV_Target0
)
{
        FGBufferData GBuffer = (FGBufferData)0; float
        SceneDepth =0; {

            float2 ScreenUV = InScreenPosition.xy / InScreenPosition.w *
View.ScreenPositionScaleBias.xy + View.ScreenPositionScaleBias.wz;
            float4 GBufferA =0; float4 GBufferB =0; float4 GBufferC =0; float4 GBufferD =0;
            FetchGBuffer(ScreenUV, GBufferA, GBufferB, GBufferC, GBufferD, SceneDepth);
```

```
        GBuffer = DecodeGBufferMobile(GBufferA, GBufferB, GBufferC, GBufferD);
    }

    // With a perspective projection, the clip space position is NDC * Clip.w // With an orthographic
    projection, clip space is the same as NDC
    float2 ClipPosition = InScreenPosition.xy / InScreenPosition.w * (View.ViewToClip[3] [3] <1.0f? SceneDepth:1.0f);

    float3 WorldPosition = mul(float4(ClipPosition, SceneDepth,1),
    View.ScreenToWorld).xyz;
    half3 CameraVector = normalize(View.WorldCameraOrigin - WorldPosition); half NoV = max(0,
    dot(GBuffer.WorldNormal, CameraVector));


    //Assemble the light source data structure.
    FMobileLightData LightData = (FMobileLightData)0; {

        LightData.Position          =DeferredLightUniforms.Position;
        LightData.InvRadius          =DeferredLightUniforms.InvRadius;
        LightData.Color = DeferredLightUniforms.Color;
        LightData.FalloffExponent              = DeferredLightUniforms.FalloffExponent;
        LightData.Direction          =DeferredLightUniforms.Direction;
        LightData.SpotAngles          =DeferredLightUniforms.SpotAngles;
        LightData.SpecularScale              =1.0;
        LightData.bInverseSquared LightDa=t aIN.bVSEpRoStLEi_gShQtU ARED_FALLOFF;
        = IS_SPOT_LIGHT;
    }

    FMobileShadingModelContext ShadingModelContext = (FMobileShadingModelContext)0; {

        half DielectricSpecular =0.08* GBuffer.Specular; ShadingModelContext.DiffuseColor =
        GBuffer.BaseColor - GBuffer.BaseColor * GBuffer.Metallic;
                            // 1 mad
        ShadingModelContext.SpecularColor = (DielectricSpecular - DielectricSpecular * GBuffer.Metallic) +
GBuffer.BaseColor * GBuffer.Metallic;                                                    // 2 mad
        //Computing EnvironmentBRDF.
        ShadingModelContext.SpecularColor = GetEnvBRDF(ShadingModelContext.SpecularColor,
GBuffer.Roughness, NoV);
    }

    //Calculates direct lighting.
    half3 Color = GetDirectLighting(LightData, ShadingModelContext, GBuffer, WorldPosition,
    CameraVector);

    // IES,Lighting function.
    half LightAttenuation = ComputeLightProfileMultiplier(WorldPosition,
    DeferredLightUniforms.Position,        - DeferredLightUniforms.Direction,
DeferredLightUniforms.Tangent);
    LightAttenuation*= ComputeLightFunctionMultiplier(WorldPosition);

#if  USE_PREEXPOSURE
    // MobileHDR applies PreExposure in tonemapper
    LightAttenuation*= View.PreExposure;
#endif

    OutColor.rgb = Color * LightAttenuation; OutColor.a =
    1;
}
```

From the above, we can see that the PS of parallel light and local light source are different entrances, mainly because the difference between the two is quite large. Parallel light directly calculates the lighting at the main entrance, and also calculates the reflection (IBL , catcher) and diffuse reflection of sky light; while the local light source will construct a light source structure, enter the direct light calculation function, and finally process the IES and lighting functions unique to the local light source.

In addition, when obtaining GBuffer, a SubPass-specific reading mode is used, which varies for different shading platforms:

```
// Vulkan
[[vk::input_attachment_index(1)]] SubpassInput<float4>
GENERATED_SubpassFetchAttachment0;
#defineVulkanSubpassFetch0() GENERATED_SubpassFetchAttachment0.SubpassLoad()

//Metal
Texture2D<float4> gl_LastFragDataRGBA_1;
#defineSubpassFetchRGBA_1() gl_LastFragDataRGBA_1.Load(uint3(0, 0, 0), 0)

//DX / OpenGL
Texture2DSampleLevel(GBufferATexture, GBufferATextureSampler, UV,0);
```

-
-
-
-

-
  -
  -
-
-
-
-
-

References

- Unreal Engine Source
- Rendering and Graphics
- Materials
- Graphics Programming
- Mobile Rendering
- Qualcomm® Adreno™ GPU
- PowerVR Developer Documentation
- Arm Mali GPU Best Practices Developer Guide Arm
- Mali GPU Graphics and Gaming Development Moving
- Mobile Graphics
- GDC Vault
- Siggraph Conference Content
- GameDev Best Practices
- Accelerating Mobile XR
- Frequently Asked Questions
- Google Developer Contributes Universal Bandwidth Compression To Freedreno Driver
- Using pipeline barriers efficiently
- Optimized pixel-projected reflections for planar reflectors
- The difference between UE4's mobile and PC graphics and how to minimize the difference
- Deferred Shading in Unity URP
- General techniques for optimizing mobile game performance

- In-depth understanding of GPU hardware architecture and operation mechanism
- Adaptive Performance in Call of Duty Mobile
- Jet Set Vulkan : Reflecting on the move to Vulkan
- Vulkan Best Practices - Memory limits with Vulkan on Mali GPUs A
- Year in a Fortnite
- The Challenges of Porting Traha to Vulkan
- L2M - Binding and Format Optimization
- Adreno Best Practices
- Summary of knowledge on GPU architecture of mobile devices Mali
- GPU Architectures
- Cyclic Redundancy Check Arm
- Guide for Unreal Engine Arm
- Virtual Reality
- Best Practices for VR on Unreal Engine
- Optimizing Assets for Mobile VR
- Arm® Guide for Unreal Engine 4 Optimizing Mobile Gaming Graphics
- Adaptive Scalable Texture Compression
- Tile-Based Rendering
- Understanding Render Passes
- Intro to Moving Mobile Graphics
- Mobile Graphics 101
- Intro to Moving Mobile Graphics
- Mobile Graphics 101
- Vulkan API
- Best Practices for Shaders
-

https://github.com/pe7yu