

Unreal Performance Guide

By Tianyi Zhang(Tanki)

Unreal version: 4.17.2

Content

Performance Profile tools available in Unreal 4	2
Frontend Tools	2
Session Frontend(ref)	2
Enable the Profiler:	2
What can I get from session frontend?	3
Figure out bottleneck through UFE	3
GPU Visualizer(ref)	4
Console Commands	12
stats xx commands	12
Debug Render Views	13
General Guides in Development	16Zoo Levels
	16
Be aware of your bottleneck	16
Performance in Daily Development Tips	17
When Should we concern about the Performance?	17
As a Programmer	17
As an Artist	17
As a (Level) Designer	18

Performance Profile tools available in Unreal 4

Unreal 4 as an advanced commercial engine has a lot of built-in profiler tools for developers. According to different scenarios, developers can choose various tools to help them to achieve the required performance. However, developers need to have enough knowledge about certain aspects of the hardware or software, so that they can understand the term and concepts occurring in the data.



Frontend Tools

Session Frontend([ref](#))

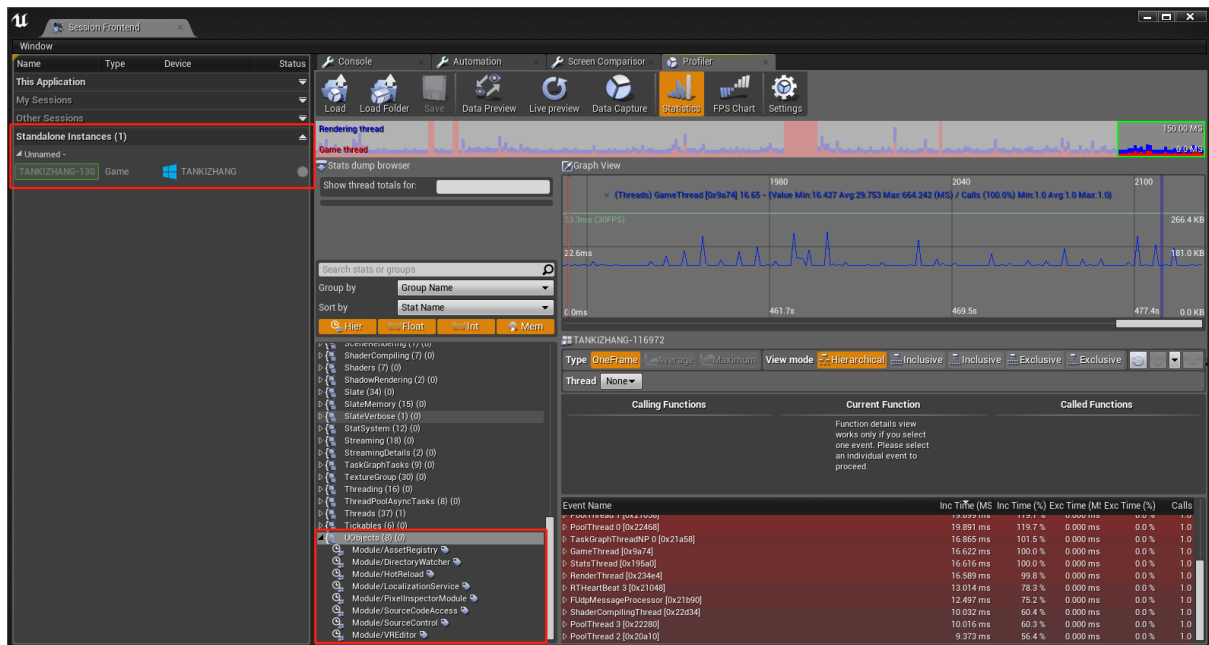
Enable the Profiler:

- Run the game with the parameter `-messaging` (for example **UE4Editor-Win64-Debug.exe -messaging**).
- Run the UFE with the parameter `-messaging` (for example **UnrealFrontend-Win64-Debug.exe -messaging**).
- Select **Session Frontend** from the **Developer Tools** section of the **Window** menu bar, then select the **Profiler** tab.

Here are the instructions from Unreal's official guide. There are several things requiring clarification:

- First, the session frontend tool(UFE) is enabled by default when you are developing the game, in the Unreal editor, no matter you play in editor  or run it with standalone lunch .
- Second, regarding 'Run the game,' that means the actual build executable, if you play the build with **-messaging**, we can profile the game with some limited information in session frontend. For more detail about the session frontend content, see the [next session](#).
- Third, Another helpful thing is if we use the command ``stat startfile`` and ``stat stopfile`` in the console. We can get a **.ue4stats** file. This **.ue4stats** file is the Unreal4 dump file which UFE can load in and visualize.

The file is primarily under the folder ``Profiling``. Depend on what kind of build you are using; it can be ``${BuildDir}/Profiling`` in packaged build, or ``${ProjectDir}/Saved/Profiling/UnrealStats`` in the editor mode.



What can I get from session frontend?

Once the data finishes dumping, it is available for analysis in UFE window. In short, it records how the time gets spent in different dimensions on CPU side. For example, Graph view is the time graph about frames. You can consider the x-Axis is the index of the frame and the y-Axis is how much time the specific category spent in that frame. [The Reference link](#) of the official document has well explained about the UFE layout.

The names of stats are confusing at first glance, here are some tips to identify the functionalities of the stats

- **CPU Stall - Wait for Event/Sleep ...**
CPU Stall means this thread is mainly idle for these time since the thread cannot step forward to the next frame until all the thread finishes this frame, which means if a thread finishes it is work too early, it has to wait, which is the event '**Wait for Event.**' In another case, when it shows '**CPU Stall - Sleep,**' that means that thread is paused and sleeping, doing nothing.
- **XXX Tick, Tick Time**
The keyword is **Tick**. The term tick in most cases indicates this event(and it is child events) is the event charged for the update of the current event. Most the problems happen here.
- **Self**
Self counts the time the function costs excluding all other function time.

Figure out bottleneck through UFE

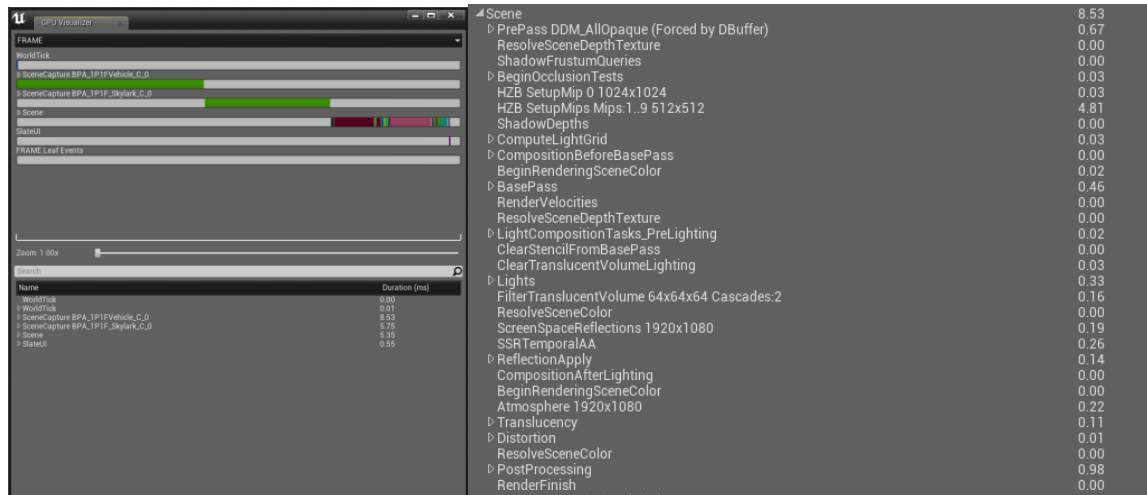
If the frame is CPU bound, probably you can finish the profiling only with the UFE. A typical process is that the team find there is a frame rate issue. So we open the UFE and dump some data. At that time we need to figure out its game tick too slow, or the render path(CPU side) is too slow, or it is just the GPU bottleneck, in which case it needs to draw too many things.

If the game tick is too slow, that means the game scripts are not efficient enough. In this case, please refer to general blueprint tips in the [As a Programmer](#) section and [As a \(Level\)](#)

[Designer](#) section. However, generally, most of the useful information would go into the **Game Thread**.

GPU Visualizer([ref](#))

In Editor, if you type **`ProfileGPU`** in the console, you can dump a current frame and get a visual version of the data in the GPU Visualizer. The time here is all GPU time.



As shown in the picture, the rendering path in ue4 has a lot of different stages.

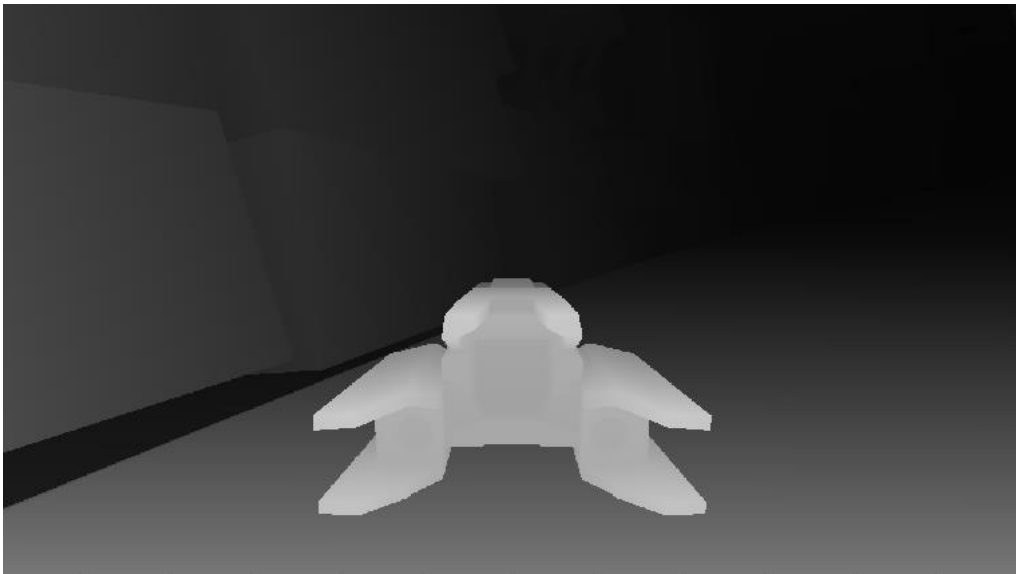
Take one frame from the game and analyze it in render doc as an example. Side note, you can enable RenderDoc plugin in ue4, **`Edit>Plugin>Rendering`**. The stats shows in the Render Doc matches the ones in the GPU Visualizer.

[This](#) is a link to the render doc frame capture in the example

5-4381	▼ Scene
66-828	➤ PrePass DDM_AllOpaque (Forced by DBuffer)
830-835	➤ ResolveSceneDepthTexture
836-837	➤ ShadowFrustumQueries
838-1265	➤ BeginOcclusionTests
1266-1286	➤ HZB SetupMip 0 1024x512
1288-1395	➤ HZB SetupMips Mips:1..9 512x256
1396-1397	➤ ShadowDepths
1398-1441	➤ ComputeLightGrid
1443-1456	➤ CompositionBeforeBasePass
1457-1468	➤ BeginRenderingSceneColor
1469-3372	➤ BasePass
3382-3390	➤ RenderVelocities
3391-3398	➤ ResolveSceneDepthTexture
3399-3403	➤ LightCompositionTasks_PreLighting
3405-3411	➤ ClearStencilFromBasePass
3412-3435	➤ ClearTranslucentVolumeLighting
3436-3530	➤ Lights
3531-3563	➤ FilterTranslucentVolume 64x64x64 Cascades:2
3564-3565	➤ ResolveSceneColor
3566-3600	➤ ScreenSpaceReflections 1280x720
3601-3632	➤ ReflectionApply
3634-3635	➤ CompositionAfterLighting
3636-3640	➤ BeginRenderingSceneColor
3641-3667	➤ Atmosphere 1280x720
3668-3680	➤ Translucency
3681-3884	➤ Translucency
3886-3958	➤ Distortion
3963-3964	➤ ResolveSceneColor
3965-4378	➤ PostProcessing



- **SlateUI** Render Pass is all Unreal editor related things so that we can ignore them.
- **ParticleSimulation** pass calculates particles related properties for all particle emitter to through GPU and encoding the information on two render targets. RGBA32_Float for positions and RGBA16_Float for velocities. In our frame sample, this pass is not involved.
- **PrePass DDM_AllOpaque**. This is technically a Z-Prepass where the engine renders all the opaque meshes to an R24G8 depth buffer. One thing we need to be careful is that ue4 use [reverse-Z](#) when rendering Depth, which can bring more precision.



- **ResolveSceneDepthTexture**. This is nothing in this frame. According to the source code:

```
void FSceneRenderTargets::ResolveSceneDepthTexture(FRHICmdList& RHICmdList, const FResolveRect& ResolveRect)
{
    SCOPED_DRAW_EVENT(RHICmdList, ResolveSceneDepthTexture);

    if (ResolveRect.IsValid())
    {

```

```

        RHICmdList.SetScissorRect(true, ResolveRect.X1, ResolveRect.Y1, ResolveRect.X2,
ResolveRect.Y2);
    }

    FSceneRenderTargets& SceneContext = FSceneRenderTargets::Get(RHICmdList);
    uint32 CurrentNumSamples = SceneDepthZ->GetDesc().NumSamples;

    const EShaderPlatform CurrentShaderPlatform =
GShaderPlatformForFeatureLevel[SceneContext.GetCurrentFeatureLevel()];
    if ((CurrentNumSamples <= 1
|| !RHISupportsSeparateMSAAAndResolveTextures(CurrentShaderPlatform)) || !GAllowCustomMSAAResolves)
    {
        RHICmdList.CopyToResolveTarget(GetSceneDepthSurface(), GetSceneDepthTexture(), true,
FResolveParams());
    }
    else
    {
        ResolveDepthTexture(RHICmdList, GetSceneDepthSurface(), GetSceneDepthTexture(),
FResolveParams());
    }

    if (ResolveRect.IsValid())
    {
        RHICmdList.SetScissorRect(false, 0, 0, 0, 0);
    }
}

```

This step is platform related. My current running platform(Win10 x64) did not hit this part.

→ ShadowFrustumQueries.

There are two places this event gets triggered.

Source\Runtime\Renderer\Private\SceneOcclusion.cpp(1287)

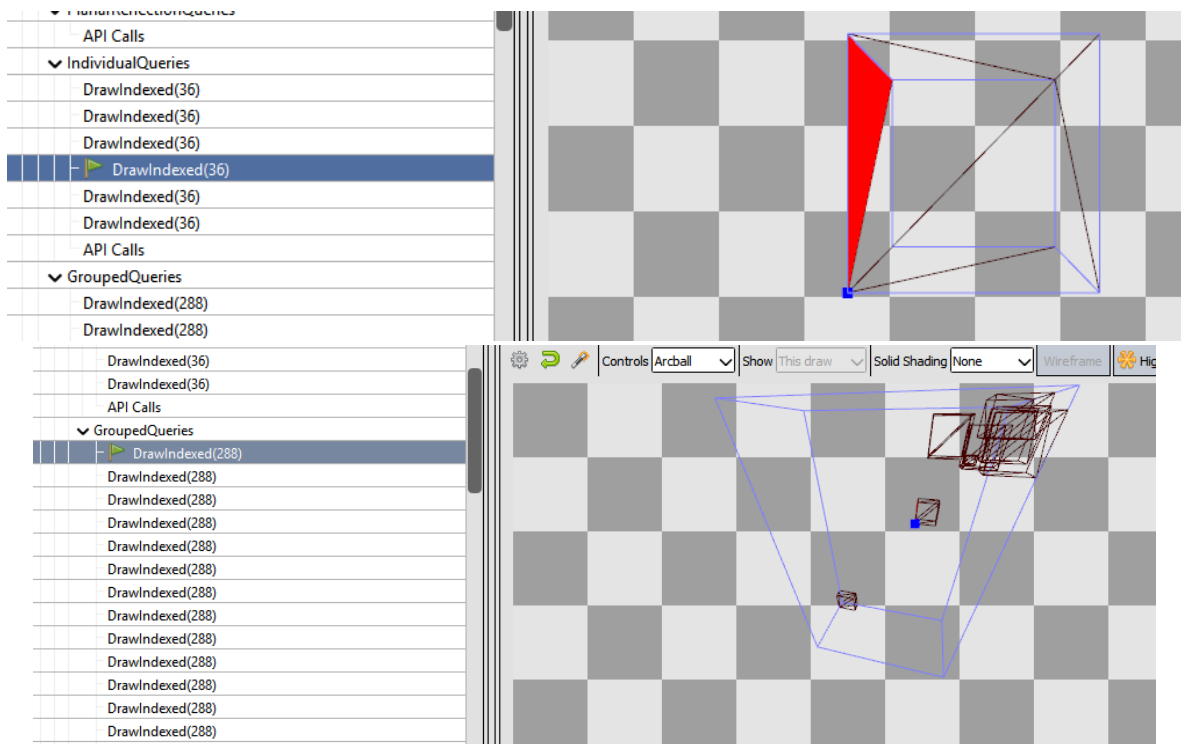
Source\Runtime\Renderer\Private\SceneOcclusion.cpp(1447)

Since it is empty for now, there is no actual drawing step get included.

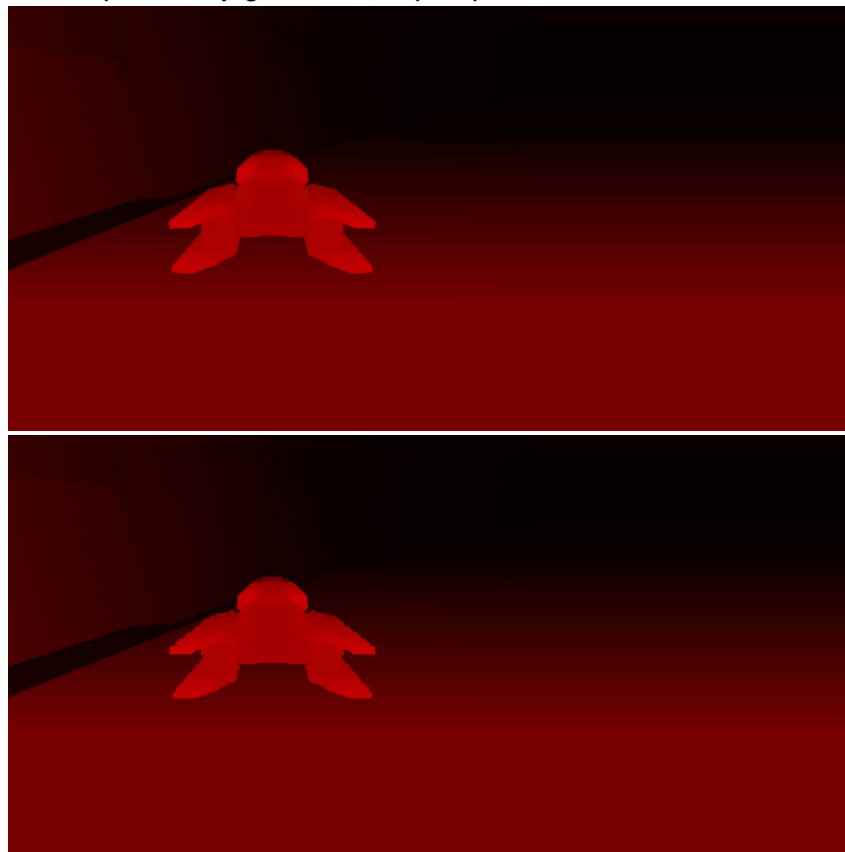
→ BeginOcclusionTests. All Unreal's occlusion test happens here. Unreal uses [hardware occlusion queries](#) for occlusion testing by default.

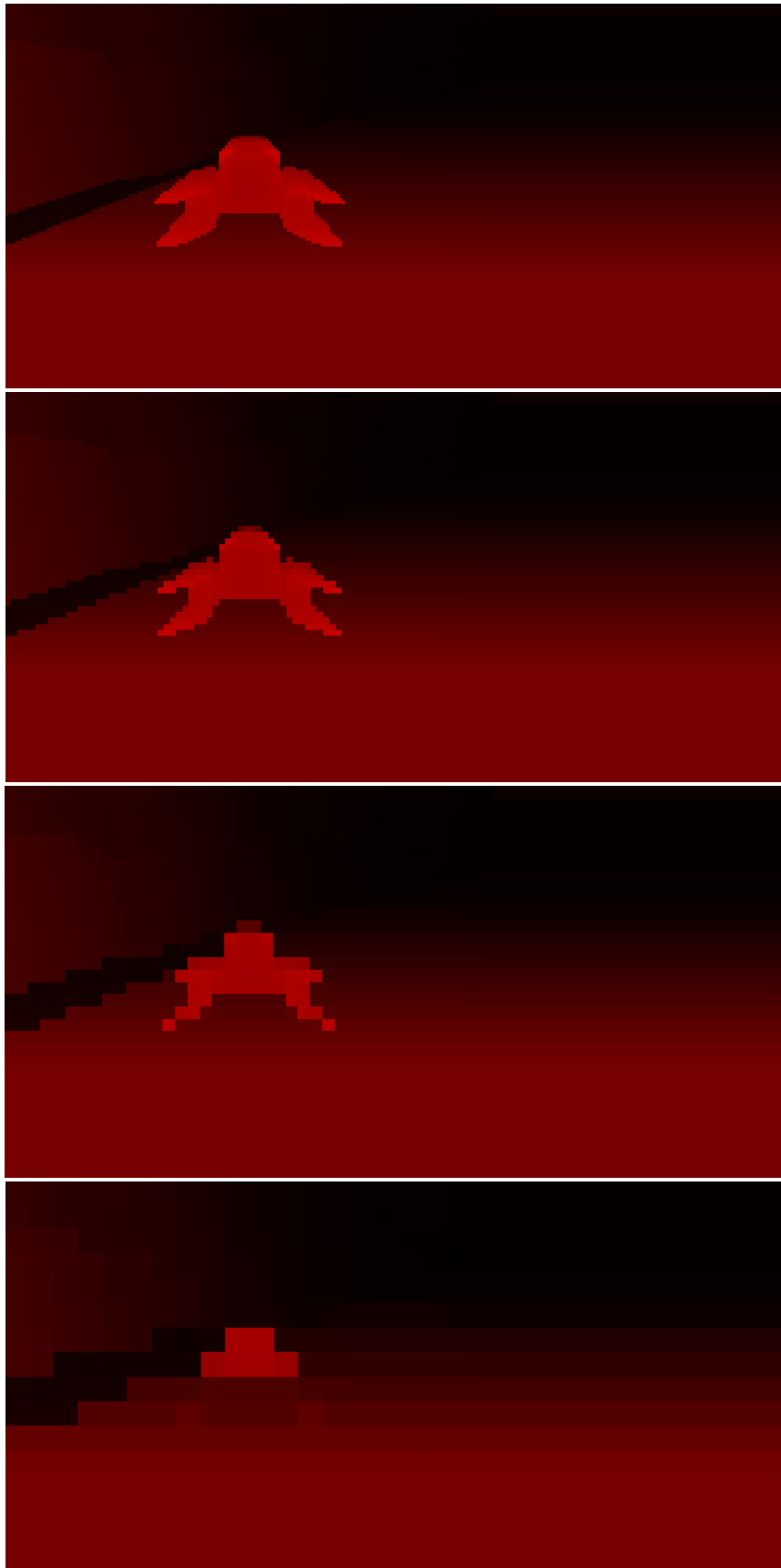
▼	BeginOcclusionTests
▼	ViewOcclusionTests 0
▼	ShadowFrustumQueries
	API Calls
▼	PlanarReflectionQueries
	API Calls
▼	IndividualQueries
	DrawIndexed(36)
	DrawIndexed(36)
	DrawIndexed(36)
	DrawIndexed(36)
	DrawIndexed(36)
	DrawIndexed(36)
	API Calls
▼	GroupedQueries
	DrawIndexed(288)
	DrawIndexed(288)
	DrawIndexed(288)
	DrawIndexed(288)

Different types of occlusion queries are applied based on the context. According to the geometry, Unreal use different bonding box to do the test. For example, for dynamic point lights, a sphere is submitted.(We do not have this in the sample frame). For a general shape, it submits a cube for the test. In the GroupedQueries, Several geometries are grouped as a single draw call.



→ **HZB SetupMip xxx.** Unreal setup Hi-Z buffer stored as R16_Float textures, where it use the Depth buffer rendered in the PrePass as input, output a MipMap chain. Each time it downsamples the previously generated mipmap.





→ **ShadowDepths**. Compute the Shadow Map. It creates quads for directional lights and static point lights and cube map for movable point lights.

- **ComputeLightGrid**. The light grid gets generated in this stage using compute shaders. In this sample, the size of the light grid is 20x12x32. The light grid split the space into small boxes, which is used for clustered shading so that Unreal can support more lights in the scene. The shader file is at **Engine\Shaders\Private\LightGridInjection.usf**. Unreal uses light grids during the Volumetric Fog Pass to add light scattering to the fog, the environment reflection pass and the translucency rendering pass/

▼	ComputeLightGrid
▼	CullLights 20x12x32 NumLights 0 NumCaptures 0
	Dispatch(240, 1, 1)
	ClearUnorderedAccessViewUint(0, 0, 0, 0)
	ClearUnorderedAccessViewUint(0, 0, 0, 0)
	Dispatch(5, 3, 8)
	API Calls
▼	Compact
	Dispatch(5, 3, 8)
	API Calls

- **CompositionBeforeBasePass**. For here, this calls to clear the GBuffers to get ready for the Main rendering part.

▼	CompositionBeforeBasePass
▼	DeferredDecals DRS_BeforeBasePass
▼	DBufferClear
	ClearRenderTargetView(0.000000, 0.000000, 0.000000,
	ClearRenderTargetView(0.501961, 0.501961, 0.501961,
	ClearRenderTargetView(0.000000, 1.000000, 0.000000,
	API Calls
	API Calls

- **BasePass**. First, Let's clarify EBasePassDrawListType enum value in ue4. This draw event get logged in

**Engine\Source\Runtime\Renderer\Private\BasePassRendering.cpp(970),
 SCOPED_DRAW_EVENTF(RHICmdList, StaticType, TEXT("Static
 EBasePassDrawListType=%d"), DrawType);**

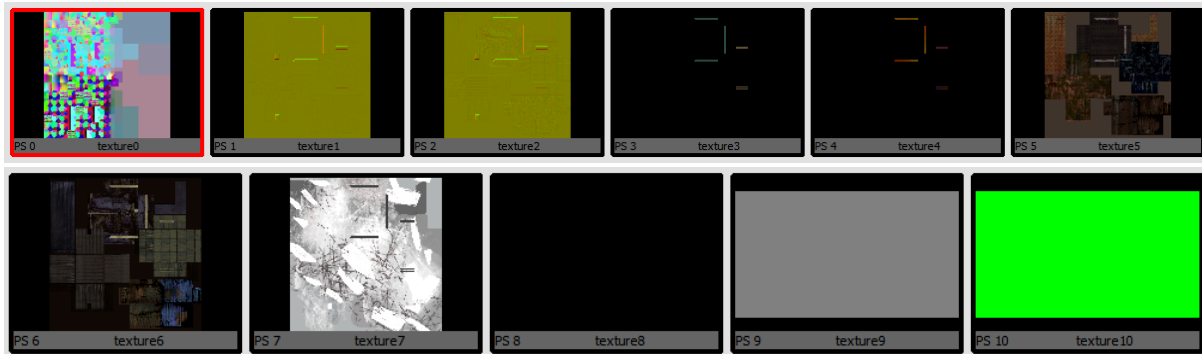
The Definition of the type:

```
enum EBasePassDrawListType
{
    EBasePass_Default=0,
    EBasePass_Masked,
    EBasePass_MAX
};
```

That means Unreal renders Masked item first, then other items.

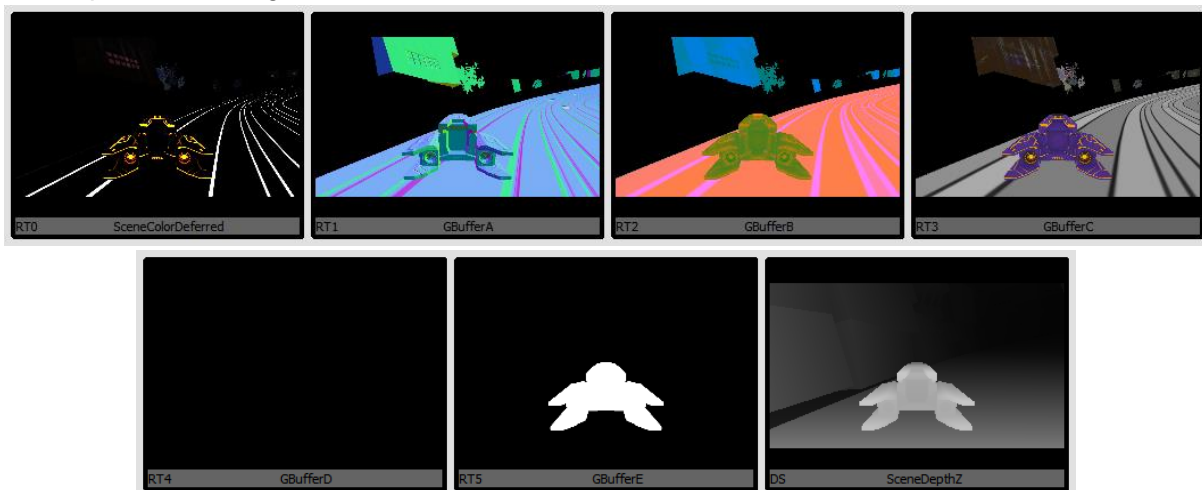
In each base pass, each draw call takes 11 textures as inputs, and output to seven separate render targets(GBuffer A~E, SceneColor, SceneDepth).

Input textures are as followed:



I only want to talk about the texture 0; another paragraph said this is the sample lighting information from 3 mipmapped atlases that appear to cache shadows and surface normals, but I am not sure about this. Please absolutely tell me if you have more information :D

Output render targets are as followed:



GBufferA: RGB10A2_UNORM, world normal

GBufferB: PBR material properties(metalness, roughness, specular intensity...)

GBufferC: Albedo in RGB, AO in Alpha

GBufferD: Custom Data based on the shading model

GBufferE: Pre-baked shadowing factors

The following steps are pretty straightforward from there name, for more information about the Unreal rendering frame, you can check the reference link.

Console Commands

stats xx commands



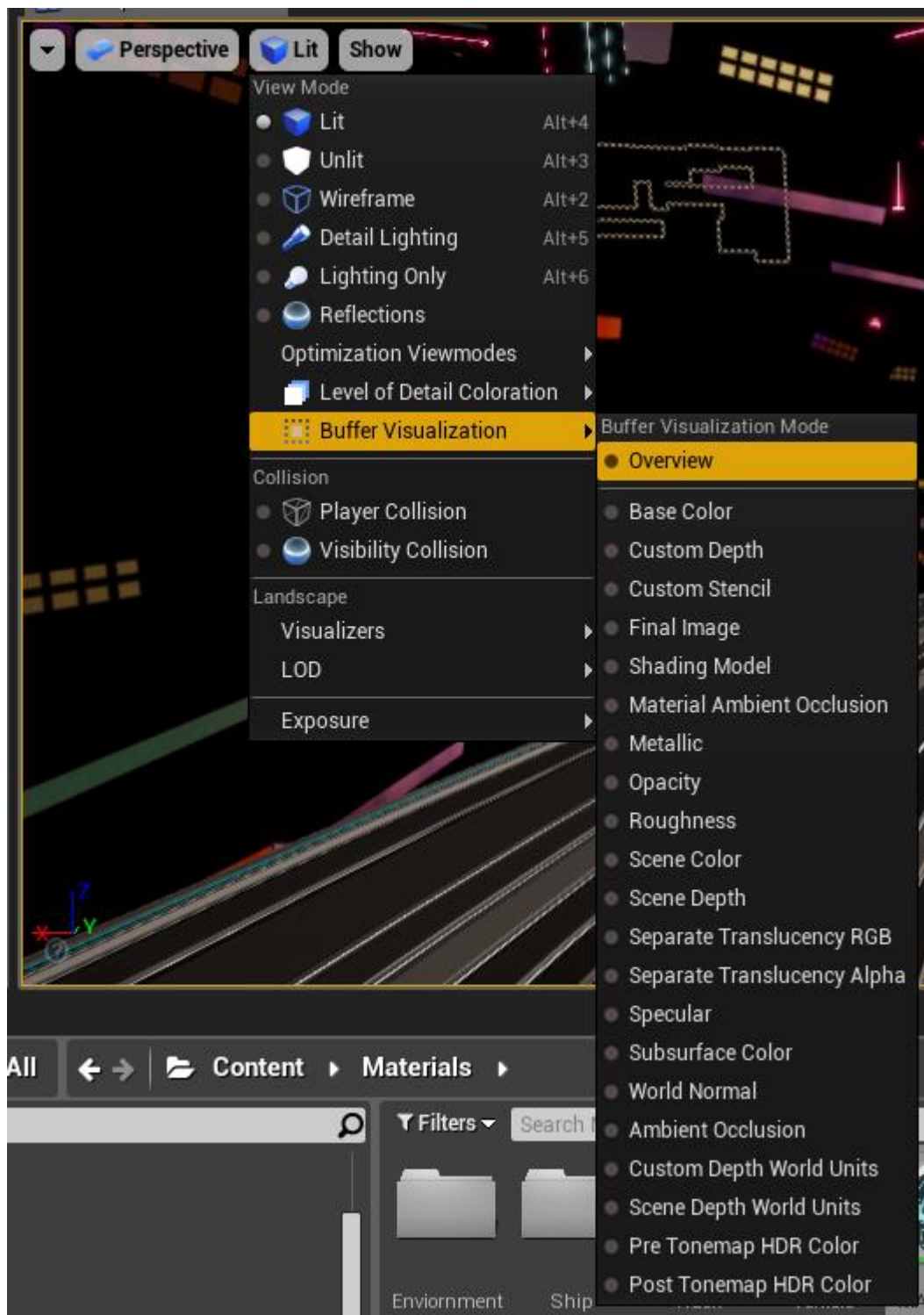
There are a bunch of commands to see the statistics of the game. The most commonly used ones are:

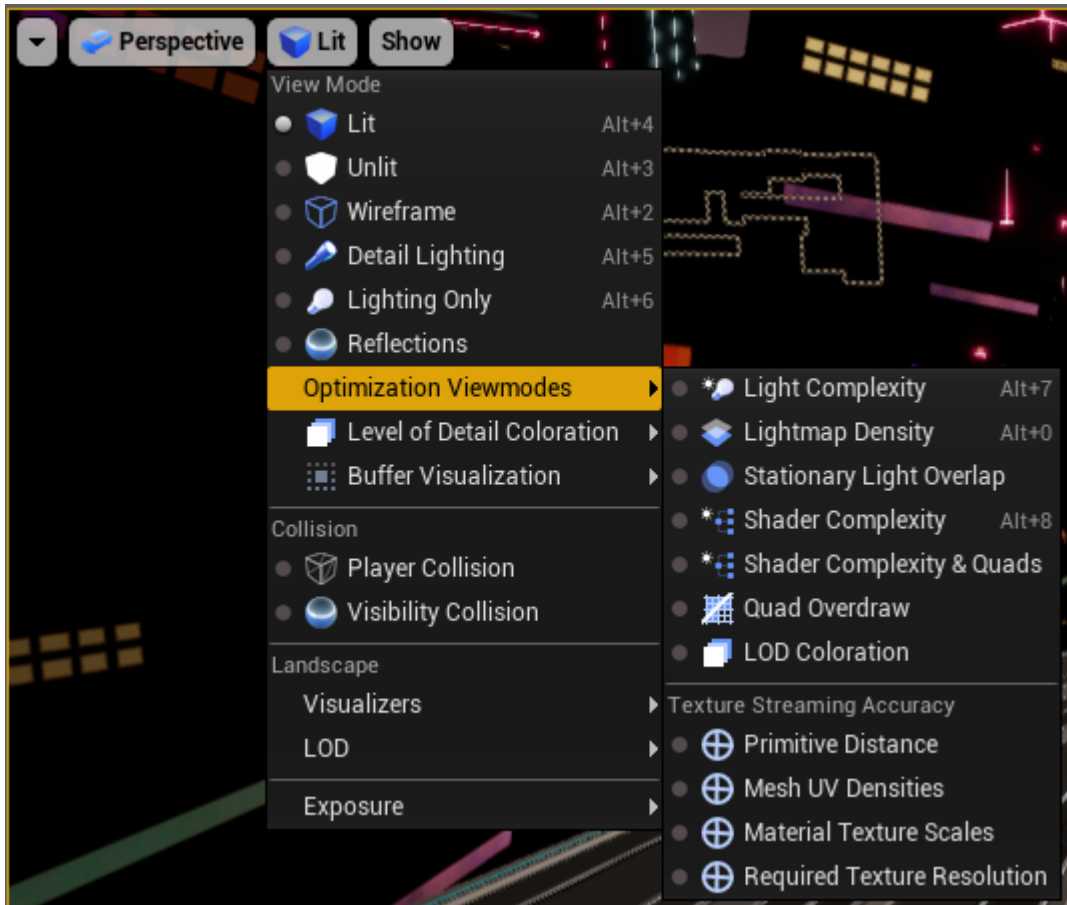
Stat FPS
Stat UNIT
Stat UnitGraph
Stat Game
Stat RHI
Stat GPU

You can play around with them to see what they do.

Debug Render Views

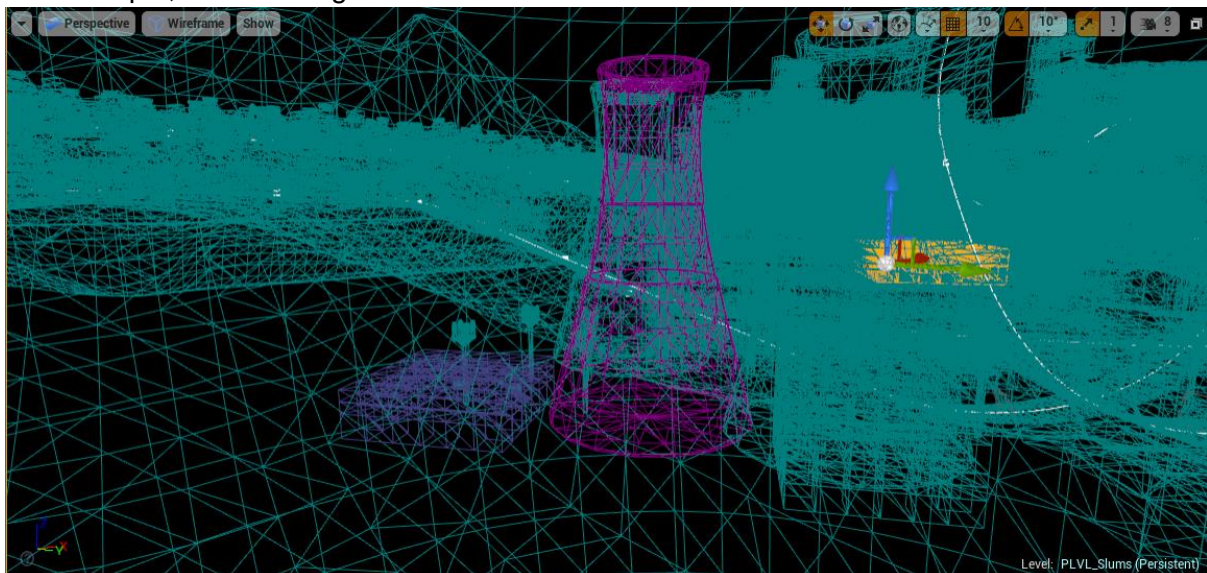
Here is the location of the debug view in Unreal Editor:





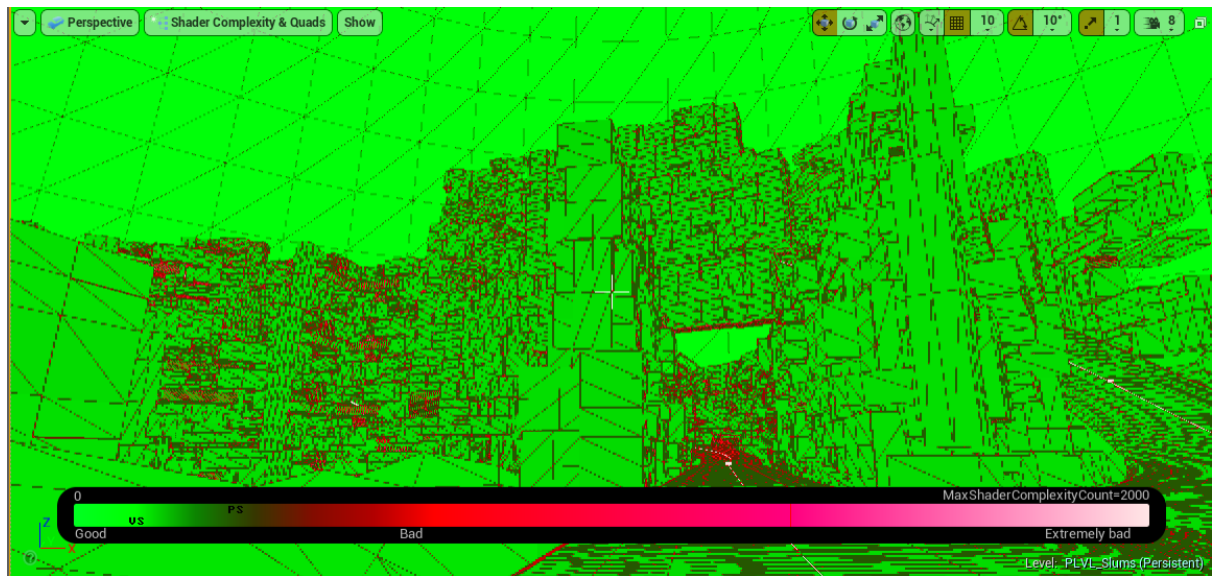
Be aware of these debug render views, which help to optimize the level.

For example, the following one is the view rendered in the wireframe mode:



The selected mesh is rendered in yellow. Static Meshes are cyan; Stationary Meshes are Magenta and Movable Meshes are purple. Though this view, you can have an overview of how crowded your meshes are in specific view and whether all the meshes have reasonable mobility property.

We can use the Shader Complexity & Quads view to find whether the LOD is set up correctly and the shader for different geometry is reasonably complicating.



General Guides in Development

Zoo Levels

Zoo levels are isolations of a particular feature, which are also living documentation for other developers (commonly are also referenced in the TDD/GDD).

Performance wise, Zoos are set up to do pressure test about a particular aspect of the engine. The Zoo levels should be optimized so that it is precisely 60 fps to give a precise idea how much we can have for this aspect in this engine with this device. There are several categories we can build around when starting playing with a new engine or new device. Noticing that all these are optional, we do not necessarily do them if those categories are very trivial for the game.

Mesh: static/stationary/movable

Physics: collision...

lighting: static/stationary/movable, point/directional/spot

...

A critical thing I need to address again is that always make zoos in isolation. Additional costs can be very easily created and affect the frame rate. For example, when testing how many movable meshes we can have in a specific view, we might not put some small script for the mesh so that it can rotate a little bit every frame. Enough numbers of small scripts can affect performance if we spawn them without constraints. So be careful about the zoo level and try to double check the result with build-in frontend tools.

Be aware of your bottleneck

The bottleneck is the most significant issue that drags down the frame rate. For example, in a frame, if your logic (CPU) takes 20ms, the GPU side spends 16ms/frame to render the scene. Then you see the GPU thread would stall for another 4ms. As a result, this frame takes 20ms. In this case, we call it **CPU bounded**. Similarly, if CPU takes 16ms while GPU costs 20ms, the frame still takes 20ms, which is called **GPU bounded**.

Furthermore, if the frame is **GPU bounded**, it can be **vertex bounded** or **pixel/fragment bounded**.

If it is vertex bounded, the most common reason is that it takes too much time for the GPU to finish the computation for all the vertices, which means either there are too many triangles or the shader logic for every single vertex is too complicating.

If it is fragment bounded, it is commonly related to the screen solution. Maybe there are too many post-processing steps, or some fragment shader is too heavy.

Performance in Daily Development Tips

When Should we concern about the Performance?

When developing the game, things change rapidly every day. So we do not want to concern about the performance too early since the contents maybe are just for prototype or temporary contents which might be replaced later.

When we are in **Vertical Slice**, the whole team need to be concerned about the performance problem because we commonly consider the vertical slice as the final game quality level. So at that time, Artists can try to optimize the assets according to the level and designer can layout the level with final quality meshes. The programmer can profile the level and do some specific optimization.

When we are in the **Alpha or later milestones**, performance gets most prioritized. The frame rate should get monitored through daily build so that we can immediately be aware of the potential problems and allocate team resource to handle it if necessary.

As a Programmer

- Most of the CPU bottlenecks are not hard to figure out and optimize, so make it works first!
- When there is a bottleneck from your blueprint or your code, you can use Session Frontend to profile your logic.
- Some general things to avoid:
 - Don't do typecast per frame.
 - Don't do actor searching by class per frame
 - Be aware of what you are costing, though possibly it is just a simple node in Blueprint, it still can be expensive(the cast is a good example).
 - Expensive update probably can be done with a lower frequency. For example, if you have 8 units, each one has an expensive operation, they get update 60 times/sec. While if you update one unit per frame, each unit still can get updated 7.5 times/sec.

As an Artist

- For all models, there should be a budget. As an artist, if you are not sure about the budget, ask your programmers or designers! If they can not tell you the budget, then push them and consider that as a blocker. This process helps you to mostly reduce the chance to re-make/optimize the models later.
- Save your budget, makes your models as simple as possible. If it is a cube, then there is no excuse to have more than 12 triangles for that.
- Being aware of how expensive is your materials, if you are not sure about that, consult your programmer teammates.
- When making the models, make sure you communicate with your designers to check how they use their models so that you can do some specific optimization(like removing hidden triangles, making LOD)

As a (Level) Designer

- Same to Artist, be aware of your budget so that you can more confidently layout your level. That means pushing your programmers instead of waiting for notifications from them.
- When white boxing your level, always keep in mind that we need to hit 60 FPS. Check all the player views to make sure there is no view with too many unnecessary details/contents/overdraw.
- When playing with Blueprint, be aware that even small script can be non-trivial when there are thousands of them running in a frame. Your programmer teammates will be glad to provide some advice.

Thanks for the help and review from the following friends:

Xia Hua(Bryan)

Joe Holan

reference:

- <https://tw.wxwenku.com/d/105049640>
- <https://interplayoflight.wordpress.com/2017/10/25/how-unreal-renders-a-frame/>
- <https://docs.unrealengine.com/en-us/Engine/Performance/GPU>
-