Performance Analysis and Optimization for PC-Based VR Applications: From the CPU's Perspective

Virtual Reality (VR) is becoming more and more popular these days as technology advancement following Moore's Law continues to make this brand new experience technically possible. While VR brings a fantastic immersive experience to users, it also puts significantly greater computing workloads on both the CPU and GPU compared to traditional applications due to dual-screen rendering, low latency, high resolution and high frame rate requirements. As a result, performance issues are especially critical in VR applications since a non-optimized VR experience with insufficient frame rate and high latency could cause nausea for users. In this article, we'll introduce a general methodology to profile, analyze, and tackle bottlenecks and hotspots in a PC-based VR application regardless of the underlying engine or VR runtime used. We use a PC VR game from Tencent* called Pangu* as an example to showcase the analysis flow.

The rendering pipeline in VR games and conventional games

Before digging into the details of the analysis, we want to explain why the CPU plays an important role in VR and how it affects VR performance. Figure 1 shows the rendering pipeline in conventional games where CPU and GPU are processed in parallel in order to maximize the hardware utilization. However, the scheme cannot be applied to VR since VR requires a low and stable rendering latency, the rendering pipeline in conventional games doesn't meet this requirement.

Let's take Figure 1 as an example, if we look at the rendering latency of Frame N+2, we find that the latency is much longer than normal because GPU has to finish the workload of Frame N+1 before starts working on the workload of Frame N+2, thus introducing a significant latency to Frame N+2. Besides, the rendering latency is varying for Frame N, Frame N+1 and Frame N+2 due to different execution circumstances, which is also unfavorable in VR since it will introduce simulation sickness to users.

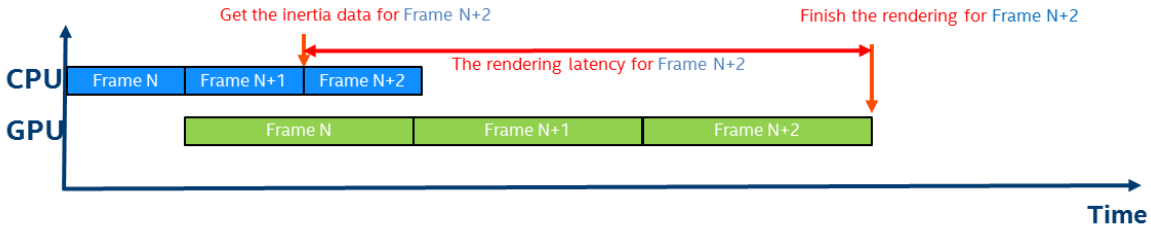


Figure 1: *The rendering pipeline in conventional games.*

As a result, the rendering pipeline in VR is changed to Figure 2 in order to achieve a shortest latency for each frame. In Figure 2, the CPU/GPU parallelism is intentionally broken in order to exchange efficiency for a low and stable rendering latency for each frame. In this case, CPU could be a bottleneck in VR since GPU has to wait for the CPU to finish pre-rendering jobs (drawcall preparation, initialization

of dynamic shadowing, occlusion culling, etc.), optimization on CPU can help reduce the GPU bubbles and improve the performance.

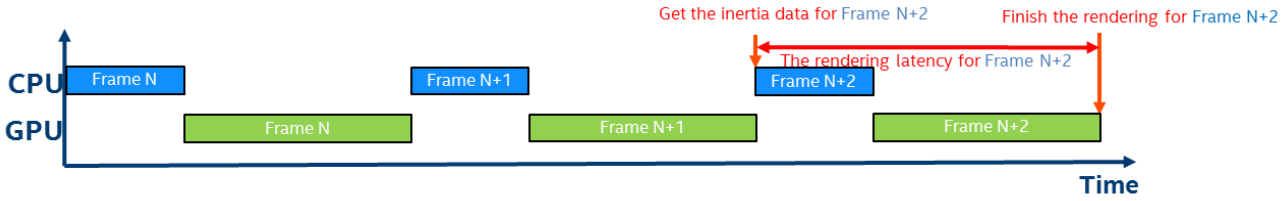


Figure 2: *The rendering pipeline in VR games.*

Background of the Pangu* VR workload

Pangu* is a PC-based VR title from Tencent*, it's a DirectX* 11 FPS VR game developed with Unreal Engine* 4 and supports both Oculus Rift* and HTC Vive*. We worked with Tencent* to improve the performance and user experience of the game in order to achieve a best- in-class gaming experience on Intel® Core™ i7 processors. Our result shows that during the development work outlined in this article the frame rate was significantly improved from 36.4 frames per second (fps) on Oculus Rift* DK2 (1920x1080) during early testing to 71.4 fps on HTC Vive* (2160x1200) at the time of this article. Here are the engines and VR runtimes used at the start and end of the development work:

- Initial development platform: Oculus v0.8 x64 runtime and Unreal 4.10.2
- Final development platform: SteamVR* v1463169981 and Unreal 4.11.2

The reason why different VR runtimes were used during development is that Pangu was initially developed on Oculus Rift DK2 since both Oculus Rift CV1 and HTC Vive have not been released yet at that time. Pangu was then migrated to HTC Vive once the device had been officially released. The adoption of different VR runtimes was evaluated and didn't make a significant difference in the performance since both Oculus and SteamVR runtimes adopted the same VR rendering pipeline as shown in Figure 2, and the rendering performance is mainly determined by the game engine in this situation. It can also be verified in Figure 5 and Figure 14 that both Oculus and SteamVR runtimes inserted GPU work(for distortion pass) after the GPU rendering of each frame, which consumed only a small proportion of time with respect to the rendering.

Here shows the screenshots of the game before and after the optimization work, note that the number of drawcalls was reduced by 5X after optimization, and the GPU execution period for each frame was also reduced from 15.1ms to 9.6ms in average in order to fit the 90fps requirement on HTC Vive*, as seen in Figure 12 and 13:

Figure 3: *Screenshots of the game before(left) and after(right) optimization.*

The specifications of the test platform:

- Intel® Core™ i7-6820HK processor (4 cores, 8 threads) @ 2.7GHz
- NVIDIA GeForce* GTX980 16GB GDDR5
- Graphics Driver Version: 364.72
- 16 GB DDR4 RAM
- Windows* 10 RTM Build 10586.164

Spotting the performance issues

In order to better understand the potential performance issues of Pangu*, we first collected the basic performance metrics of the game, shown in Table 1. All the data in this table were collected using various tools including GPU-Z, TypePerf, and Unreal Frontend. If we compare the data to system idle, several observation can be made:

- Relatively low GPU utilization (49.64 percent on GTX980) with respect to the low frame rate (36.4 fps). If the GPU utilization were improved, a higher frame rate could be achieved.
- High numbers of draw calls. The rendering in DirectX 11 is single threaded and has relatively high draw call overhead in the render thread as compared to DirectX 12. Since the game was developed on DirectX 11 and VR rendering pipeline breaks the CPU/GPU concurrency in order to achieve a shorter Motion-to-Photon(MTP) latency, the performance will be significantly decreased if the game is render thread bound. Less draw calls can help relief the render thread bound in this case.
- CPU utilization doesn't seem to be an issue in this table since it is only 13.6 percent on average. In the following session we show that this statement is not true, that the workload is actually bounded by some CPU threads.

	System Idle	Pangu* on Oculus Rift* DK2 (before optimization)
GPU Core Clock (MHz)	135	1337.6
GPU Memory Clock (MHz)	162	1749.6
GPU Memory Used (MB)	184	1727.71
GPU Load (%)	0	49.64
Average Frame Rate (fps)	N/A	36.4
Draw Calls (/frame)	0	4437
Processor(_Total)\Processor Time (%)	1.04 (5.73/0.93/0.49/0.29/0.7/0.37/0.24/0.2)	13.58 (30.20/10.54/26.72/3.76/12.72/8.16/12.27/4.29)
Processor Information(_Total)\Processor Frequency (MHz)	800	2700

Table 1: *Basic performance metrics of the game before optimization.*

In the following section, we use GPUView and Windows Performance Analyzer (WPA) from the Windows Assessment Development Kit (ADK) [1] to profile and analyze the bottlenecks in the VR workload.

A deeper look into the performance issues

GPUView [2] is a tool that can be used to investigate the performance interaction between graphics applications, CPU threads, graphics driver, Windows graphics kernel, and related interactions. This tool can also show whether an application is CPU bound or GPU bound in the timeline view. On the other hand, WPA [3] is an analysis tool that creates graphs and data tables of Event Tracing for Windows (ETW) events. It has a flexible UI that can be pivoted to view call stacks, CPU hotspots, context switches, and so on. It can also be used to explore the root cause of performance issues. Both GPUView and WPA can be used to analyze the event trace log (ETL) file captured by Windows Performance Recorder (WPR), which can be run from the user interface (UI) or from the command line, and have built-in profiles that can be used to select the events to be recorded.

For a VR application, it's better to determine whether the application is bounded by the CPU, GPU, or both. We can focus our optimization efforts on the most critical part of the performance bottlenecks, thus achieving as much performance gain as possible with minimum effort.

Figure 4 shows the timeline view of Pangu* in GPUView before optimization, where the GPU work queue, CPU context queues, and CPU threads are all shown in Figure 4. Several facts can be concluded from the chart:

- The frame rate is about 37 fps.
- GPU utilization is about 50 percent.
- The user experience of this VR workload is bad since the frame rate is far less than 90 fps, which is easy to induce motion sickness and nausea to end users.
- As seen in the GPU work queue, only two processes submitted tasks to the GPU: Oculus VR runtime and VR workload. Oculus VR runtime performed works including distortion, chroma aberration, and time warp at the last stage of frame rendering.
- The VR workload was bounded by both the CPU and GPU:
    - For CPU bound, the GPU was idle for 50 percent of the time (GPU bubbles) and was bounded by the execution of some CPU threads (T1864, T8292, T8288, T4672, T8308), which means that GPU works could not be submitted and executed as long as the CPU tasks in these threads had not been finished. If CPU tasks were optimized, GPU utilization could be greatly improved to allow more works to be accomplished in the GPU, thus achieving a higher frame rate.
    - For GPU bound, we can see that even if we could eliminate all the GPU bubbles, the GPU execution period of a single frame was still larger than 11.1ms (about 14.7ms in this workload), which means that without further optimization on the GPU side, the VR workload is not able to run at 90 fps, which is the required frame rate for premier VR head-mounted displays (HMDs) including Oculus Rift* CV1 and HTC Vive*.

Figure 4: *A timeline view of Pangu* in GPUView.*

Preliminary recommendations for improving the frame rate and GPU utilization:

- Some non-urgent CPU work such as physics and AI could be deferred to let graphics rendering jobs get submitted earlier, in order to reduce GPU bubbles during CPU bottlenecks
- Apply multithreading techniques efficiently to increase the amount of parallel execution and reduce the CPU bottleneck in the game
- Reduce tasks that lead to CPU bottleneck such as draw calls, dynamic shadowing, cloth simulation, physics and AI navigation, etc..
- Submit the CPU task of the next frame earlier to reduce GPU gaps. Although motion-to-photon latency might be slightly increased, performance and efficiency could be greatly improved.
- DirectX 11 has a high drawcall and driver overheads, having too much drawcalls will lead to serious CPU bound caused by the render thread, consider migrating to DirectX 12 if possible.

- Have to optimize GPU workloads as well(e.g. overdraw, bandwidth, texture fillrate, etc.) since GPU active period for a single frame is longer than a vsync period, leading to frames dropping.

In order to take a deeper look into the bottleneck, we can use WPA to explore the same ETL file analyzed with GPUView. WPA can also be used to identify CPU hotspots in terms of CPU utilization or context switches; readers who are interested in this topic can refer to [4] for more details. Here we introduce the main methodology for CPU bottleneck analysis and optimization.

Look at a single frame of the VR workload that has performance issues. Since the present packet is submitted to the GPU once per frame after rendering, the timing between two succeeding present packets is the period of a single frame, as shown in Figure 5 (26.78 ms, which is equivalent to 37.34 fps).

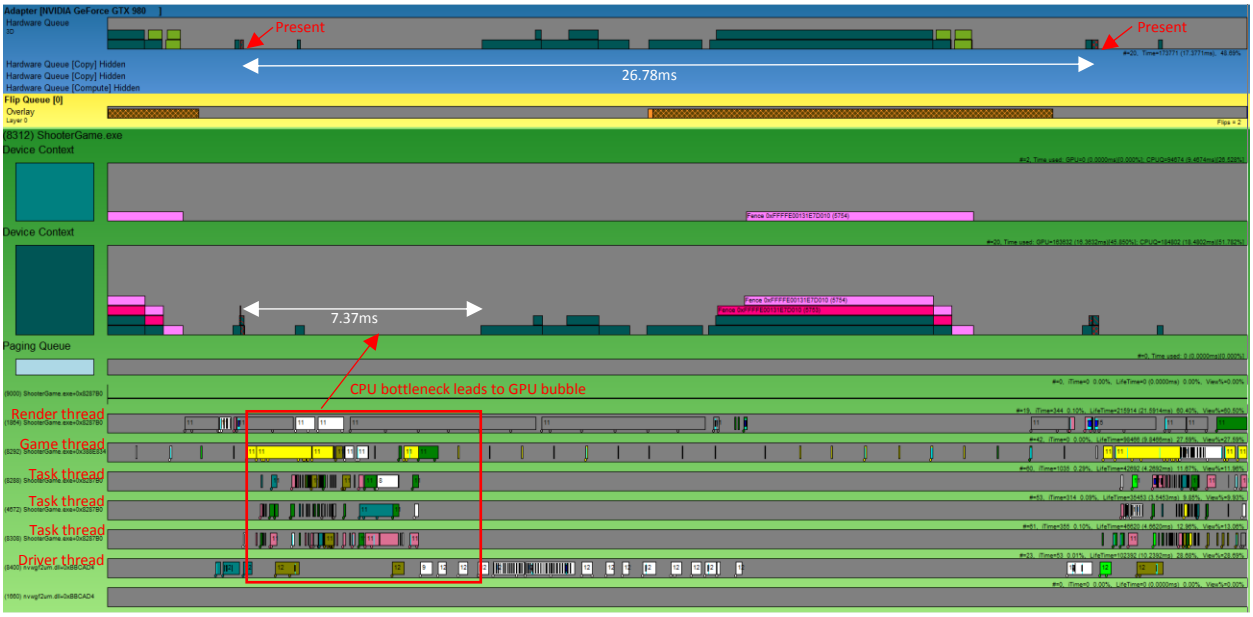


Figure 5: *A timeline view of Pangu* in GPUView for a single frame. Note the CPU threads that lead to GPU bubble.*

Note that there are GPU bubbles in the GPU work queue (for example, 7.37 ms at the beginning of a frame) which were actually caused by the CPU thread bound in the VR workload, as marked in the red rectangle. It is because CPU tasks such as draw call preparation, culling, and the like must finish before GPU commands are submitted for rendering.

If we use WPA to look at the CPU bound periods shown in GPUView, we are able to find out the key CPU hotspots that prevent the GPU from execution. Figures 6–11 show the utilization and the call stacks of CPU threads in WPA, within the same time period in GPUView.

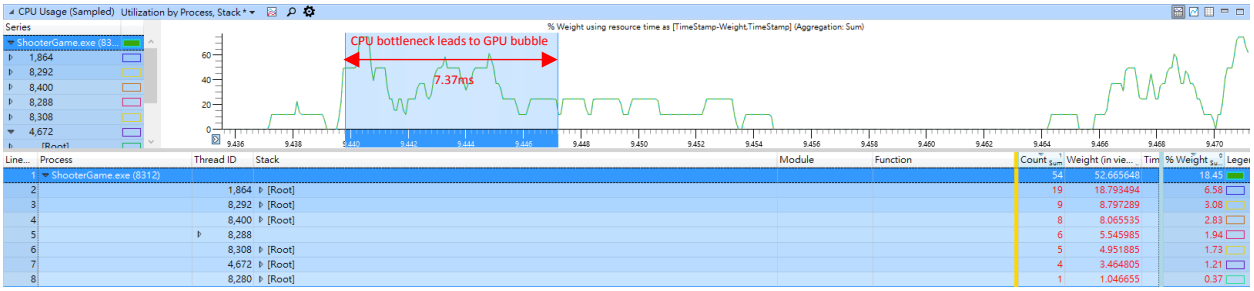


Figure 6: *A timeline view of Pangu* in WPA with the same period as Figure 5.*

Let's look at the bottleneck of each CPU thread.

Process	Thread ID	Stack	Module	Function	% Weight
ShooterGame.exe (8312)					37.93
	1,864	[Root]			9.92
		ntdll.dll!RtlUserThreadStart			9.92
		kernel32.dll!BaseThreadInitThunk			9.92
		ShooterGame.exe!FRunnableThreadWin::GuardedRun			9.92
		ShooterGame.exe!FRunnableThreadWin::Run			9.92
		ShooterGame.exe!FRenderingThread::Run			9.92
		ShooterGame.exe!RenderingThreadMain			9.92
		ShooterGame.exe!FTaskThread::ProcessTasksUntilQuit			9.92
		ShooterGame.exe!FTaskThread::ProcessTasks			9.92
		ShooterGame.exe!TGraphTask<`FRendererModule::BeginRenderingViewFamily'::`20'::EURCMacro_FDrawSceneCommand>::ExecuteTask			9.92
		|- ShooterGame.exe!FRelevancePacket::RenderThreadFinalize			8.35
		| ShooterGame.exe!FDeferredShadingSceneRenderer::Render			8.35
		| |- ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePass			4.96
		| | ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassView			4.96
		| | ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassStaticData			4.96
		| | ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassStaticDataDefault			4.96
		| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<TDistanceFieldShadowsAndLightMapPolicy<1> > >::DrawVisibleInner			4.96
		| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<TDistanceFieldShadowsAndLightMapPolicy<1> > >::DrawElement			4.96
		| |- ShooterGame.exe!FMeshDrawingPolicy::DrawMesh			3.20
		| |- ShooterGame.exe!TBasePassDrawingPolicy<TDistanceFieldShadowsAndLightMapPolicy<1> >::SetMeshRenderState			1.77
		|- ShooterGame.exe!FDeferredShadingSceneRenderer::InitViews			3.39
		| |- ShooterGame.exe!FDeferredShadingSceneRenderer::InitDynamicShadows			1.69
		| |- ShooterGame.exe!FSceneRenderer::ComputeViewVisibility			1.69
		|- ShooterGame.exe!TGraphTask<`FRendererModule::BeginRenderingViewFamily'::`20'::EURCMacro_FDrawSceneCommand>::ExecuteTask<itself>	ShooterGame.exe	TGraphTask<`FRendererM...	1.57

Figure 7: *The call stack of the render thread T1864.*

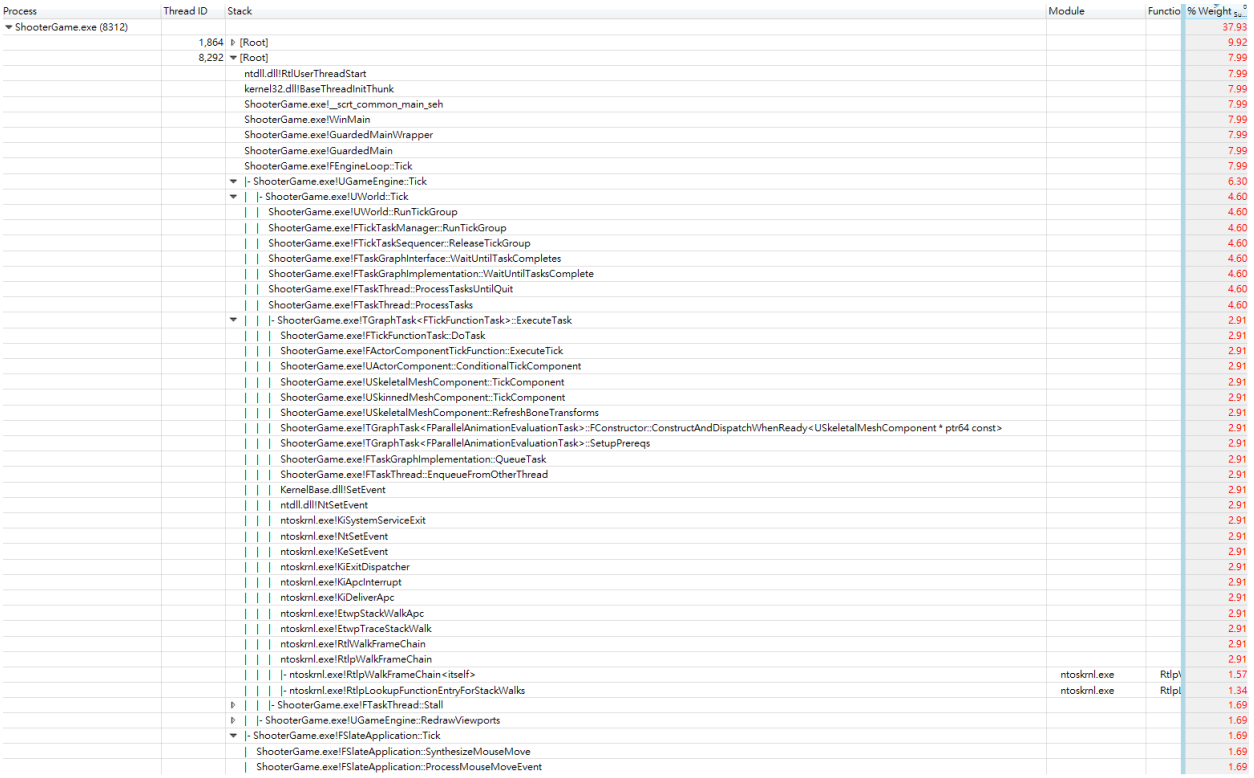
As seen in the call stack, the top three bottlenecks in the render thread are

1. Base pass rendering for static meshes (50 percent)
2. Initialization of dynamic shadows (17 percent)
3. Compute view visibility (17 percent)

These bottlenecks are caused by too many draw calls, state changes, and shadow map rendering in the render thread. Some suggestions to optimize the render thread performance:

- Apply batching in Unity* or actor merging in Unreal to reduce static mesh drawing. Combine close objects together and use Level of Details (LOD). Using fewer materials and putting separate textures into a larger texture atlas can also help.
- Use Double Wide Rendering in Unity or Instanced Stereo Rendering in Unreal to reduce draw call submission overhead for stereo rendering.
- Reduce or turn off real-time shadows. Objects that receive dynamic shadowing will not be batched, thus incurring a severe draw call penalty.

- Avoid using effects that cause objects to be rendered multiple times (reflections, per-pixel lights, transparent, and multi-material objects).



Process	Thread ID	Stack	Module	Functio	% Weight
ShooterGame.exe (8312)					37.93
	1,864	[Root]			9.92
	8,292	[Root]			7.99
		ntdll.dll!RtlUserThreadStart			7.99
		kernel32.dll!BaseThreadInitThunk			7.99
		ShooterGame.exe!__scrt_common_main_seh			7.99
		ShooterGame.exe!WinMain			7.99
		ShooterGame.exe!GuardedMainWrapper			7.99
		ShooterGame.exe!GuardedMain			7.99
		ShooterGame.exe!FEngineLoop::Tick			7.99
		|- ShooterGame.exe!UGameEngine::Tick			6.30
		| |- ShooterGame.exe!UWorld::Tick			4.60
		| | ShooterGame.exe!UWorld::RunTickGroup			4.60
		| | ShooterGame.exe!FTickTaskManager::RunTickGroup			4.60
		| | ShooterGame.exe!FTickTaskSequencer::ReleaseTickGroup			4.60
		| | ShooterGame.exe!FTaskGraphInterface::WaitUntilTaskCompletes			4.60
		| | ShooterGame.exe!FTaskGraphImplementation::WaitUntilTasksComplete			4.60
		| | ShooterGame.exe!FTaskThread::ProcessTasksUntilQuit			4.60
		| | ShooterGame.exe!FTaskThread::ProcessTasks			4.60
		| | |- ShooterGame.exe!TGraphTask<FTickFunctionTask>::ExecuteTask			2.91
		| | | ShooterGame.exe!FTickFunctionTask::DoTask			2.91
		| | | ShooterGame.exe!FActorComponentTickFunction::ExecuteTick			2.91
		| | | ShooterGame.exe!UActorComponent::ConditionalTickComponent			2.91
		| | | ShooterGame.exe!USkeletalMeshComponent::TickComponent			2.91
		| | | ShooterGame.exe!USkinnedMeshComponent::TickComponent			2.91
		| | | ShooterGame.exe!USkeletalMeshComponent::RefreshBoneTransforms			2.91
		| | | ShooterGame.exe!TGraphTask<FParallelAnimationEvaluationTask>::FConstructor::ConstructAndDispatchWhenReady<USkeletalMeshComponent * ptr64 const>			2.91
		| | | ShooterGame.exe!TGraphTask<FParallelAnimationEvaluationTask>::SetupPrereqs			2.91
		| | | ShooterGame.exe!FTaskGraphImplementation::QueueTask			2.91
		| | | ShooterGame.exe!FTaskThread::EnqueueFromOtherThread			2.91
		| | | KernelBase.dll!SetEvent			2.91
		| | | ntdll.dll!NtSetEvent			2.91
		| | | ntoskrnl.exe!KiSystemServiceExit			2.91
		| | | ntoskrnl.exe!NtSetEvent			2.91
		| | | ntoskrnl.exe!KeSetEvent			2.91
		| | | ntoskrnl.exe!KiExitDispatcher			2.91
		| | | ntoskrnl.exe!KiApcInterrupt			2.91
		| | | ntoskrnl.exe!KiDeliverApc			2.91
		| | | ntoskrnl.exe!EtwpStackWalkApc			2.91
		| | | ntoskrnl.exe!EtwpTraceStackWalk			2.91
		| | | ntoskrnl.exe!RtlWalkFrameChain			2.91
		| | | ntoskrnl.exe!RtlpWalkFrameChain			2.91
		| | | |- ntoskrnl.exe!RtlpWalkFrameChain<itself>	ntoskrnl.exe	Rtlp\	1.57
		| | | |- ntoskrnl.exe!RtlpLookupFunctionEntryForStackWalks	ntoskrnl.exe	Rtlpl	1.34
		| | |- ShooterGame.exe!FTaskThread::Stall			1.69
		| |- ShooterGame.exe!UGameEngine::RedrawViewports			1.69
		|- ShooterGame.exe!FSlateApplication::Tick			1.69
		| ShooterGame.exe!FSlateApplication::SynthesizeMouseMove			1.69
		| ShooterGame.exe!FSlateApplication::ProcessMouseMoveEvent			1.69

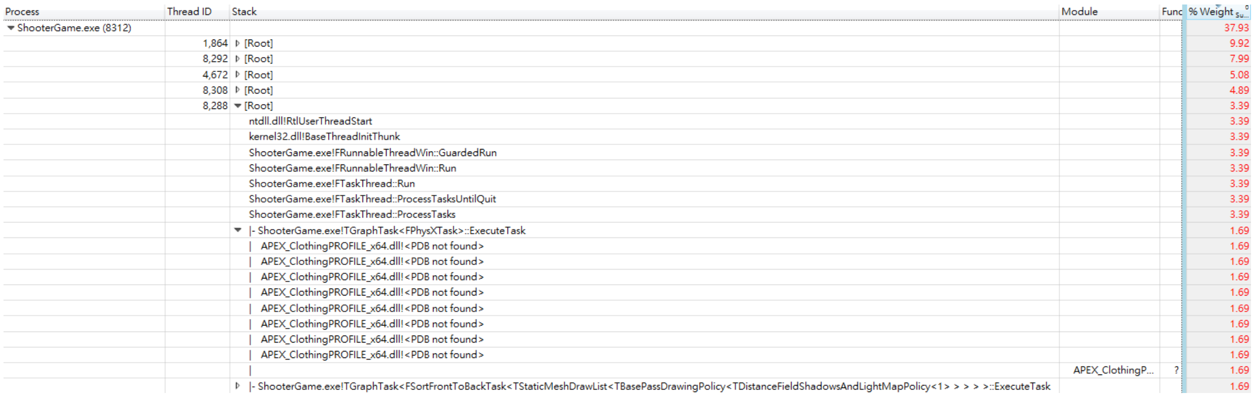
Figure 8: *The call stack of the game thread T8292.*

For the game thread, the top three bottlenecks are

1. Set up pre-requirements for parallel processing of animation evaluation (36.4 percent)
2. Redraw view ports (21.2 percent)
3. Process Mouse Move Event (21.2 percent)

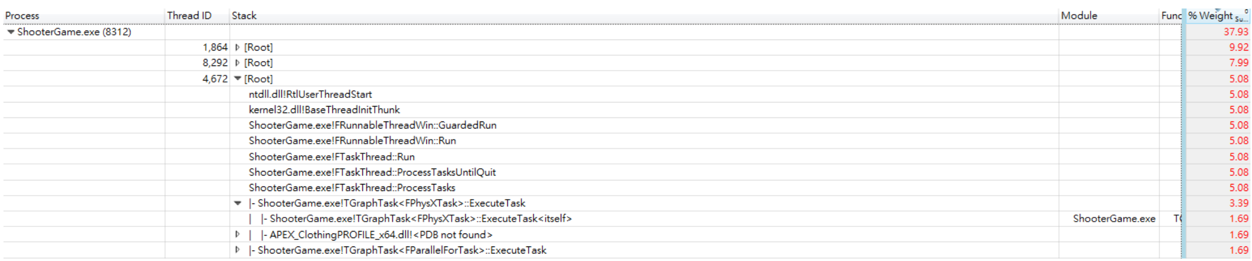
These bottlenecks can be optimized by reducing the number of view ports and the overhead of parallel animation evaluation at the CPU side. Use single-thread processing instead if only a few number of animation nodes are used, and examine the usage of mouse control at the CPU side.

Task threads (T8288, T4672, T8308):



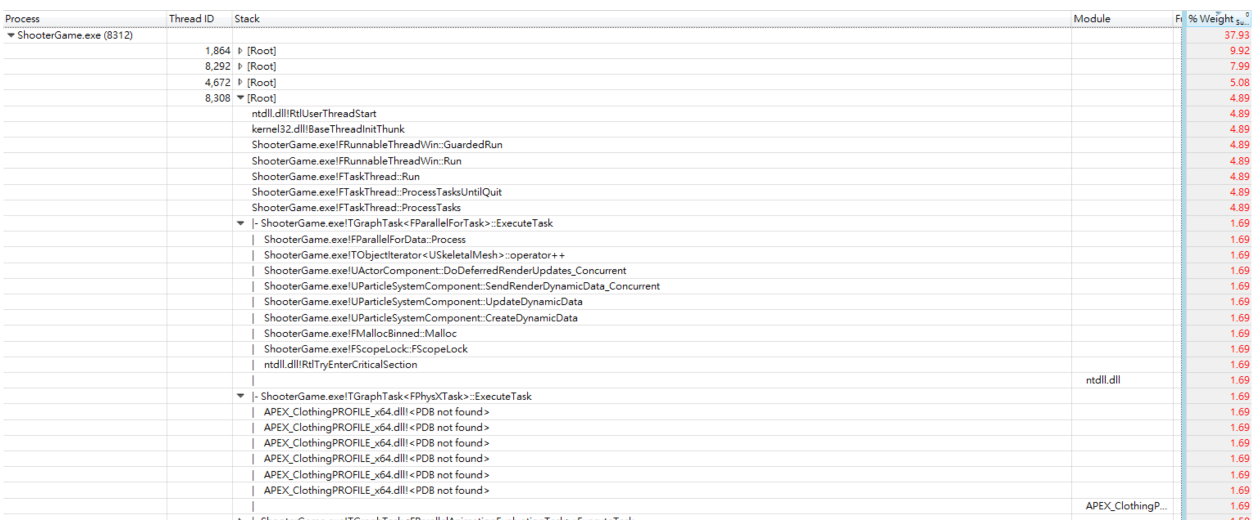
Process	Thread ID	Stack	Module	Func	% Weight
ShooterGame.exe (8312)					37.93
	1,864	[Root]			9.92
	8,292	[Root]			7.99
	4,672	[Root]			5.08
	8,308	[Root]			4.89
	8,288	[Root]			3.39
		ntdll.dll!RtlUserThreadStart			3.39
		kernel32.dll!BaseThreadInitThunk			3.39
		ShooterGame.exe!FRunnableThreadWin::GuardedRun			3.39
		ShooterGame.exe!FRunnableThreadWin::Run			3.39
		ShooterGame.exe!FTaskThread::Run			3.39
		ShooterGame.exe!FTaskThread::ProcessTasksUntilQuit			3.39
		ShooterGame.exe!FTaskThread::ProcessTasks			3.39
		|- ShooterGame.exe!TGraphTask<FPhysXTask>::ExecuteTask			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		|	APEX_ClothingP...	?	1.69
		|- ShooterGame.exe!TGraphTask<FSortFrontToBackTask<TStaticMeshDrawList<TBasePassDrawingPolicy<TDistanceFieldShadowsAndLightMapPolicy<1> > > >::ExecuteTask			1.69

Figure 9: *The call stack of the task thread T8288.*



Process	Thread ID	Stack	Module	Func	% Weight
ShooterGame.exe (8312)					37.93
	1,864	[Root]			9.92
	8,292	[Root]			7.99
	4,672	[Root]			5.08
		ntdll.dll!RtlUserThreadStart			5.08
		kernel32.dll!BaseThreadInitThunk			5.08
		ShooterGame.exe!FRunnableThreadWin::GuardedRun			5.08
		ShooterGame.exe!FRunnableThreadWin::Run			5.08
		ShooterGame.exe!FTaskThread::Run			5.08
		ShooterGame.exe!FTaskThread::ProcessTasksUntilQuit			5.08
		ShooterGame.exe!FTaskThread::ProcessTasks			5.08
		|- ShooterGame.exe!TGraphTask<FPhysXTask>::ExecuteTask			3.39
		| |- ShooterGame.exe!TGraphTask<FPhysXTask>::ExecuteTask<itself>	ShooterGame.exe	T(	1.69
		| |- APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		|- ShooterGame.exe!TGraphTask<FParallelForTask>::ExecuteTask			1.69

Figure 10: *The call stack of the task thread T4672.*



Process	Thread ID	Stack	Module	F	% Weight
ShooterGame.exe (8312)					37.93
	1,864	[Root]			9.92
	8,292	[Root]			7.99
	4,672	[Root]			5.08
	8,308	[Root]			4.89
		ntdll.dll!RtlUserThreadStart			4.89
		kernel32.dll!BaseThreadInitThunk			4.89
		ShooterGame.exe!FRunnableThreadWin::GuardedRun			4.89
		ShooterGame.exe!FRunnableThreadWin::Run			4.89
		ShooterGame.exe!FTaskThread::Run			4.89
		ShooterGame.exe!FTaskThread::ProcessTasksUntilQuit			4.89
		ShooterGame.exe!FTaskThread::ProcessTasks			4.89
		|- ShooterGame.exe!TGraphTask<FParallelForTask>::ExecuteTask			1.69
		| ShooterGame.exe!FParallelForData::Process			1.69
		| ShooterGame.exe!TObjectIterator<USkeletalMesh>::operator++			1.69
		| ShooterGame.exe!UActorComponent::DoDeferredRenderUpdates_Concurrent			1.69
		| ShooterGame.exe!UParticleSystemComponent::SendRenderDynamicData_Concurrent			1.69
		| ShooterGame.exe!UParticleSystemComponent::UpdateDynamicData			1.69
		| ShooterGame.exe!UParticleSystemComponent::CreateDynamicData			1.69
		| ShooterGame.exe!FMallocBinned::Malloc			1.69
		| ShooterGame.exe!FScopeLock::FScopeLock			1.69
		| ntdll.dll!RtlTryEnterCriticalSection			1.69
		|	ntdll.dll		1.69
		|- ShooterGame.exe!TGraphTask<FPhysXTask>::ExecuteTask			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		| APEX_ClothingPROFILE_x64.dll!<PDB not found>			1.69
		|	APEX_ClothingP...		1.69
		|- ShooterGame.exe!TGraphTask<FParallelAnimationEvaluationTask>::ExecuteTask			1.50

Figure 11: *The call stack of the task thread T8308.*

For the task threads, bottlenecks are mostly located in physics-related simulations such as cloth simulation, animation evaluation, and particle system update.

Table 2 shows a summary of the CPU hotspots (percent of clockticks) during GPU bubble periods.

THREAD	FUNCTION	CLOCKTICK %	
Render thread	Base pass rendering for static meshes	13.1%	22.1%
	Initialization of dynamic shadows	4.5%	
	Compute view visibility	4.5%	
Game thread	Set up pre-requirements for parallel processing of animation evaluation	7.7%	16.7%
	Redraw view ports	4.5%	
	Process Mouse Move Event	4.5%	
Physics	Cloth simulation	13.5%	22%
	Animation evaluation	4%	
	particle system update	4.5%	
Driver			4.4%

Table 2: *CPU hotspots during GPU bubble periods before optimization.*

Optimization

After implementation of some of the optimization including Level of Detail (LOD), instanced stereo rendering, dynamic shadow removal, deferred CPU tasks and optimized physics, the frame rate was increased from 36.4 fps on Oculus Rift* DK2 (1920x1080) to 71.4 fps on HTC Vive* (2160x1200); the GPU utilization was also increased from 54.7 percent to 74.3 percent due to fewer CPU bottlenecks.

Figures 12 and 13 show the GPU utilization of Pangu* before and after optimization, respectively, as seen from the GPU work queue.

Figure 12: *The GPU utilization of Pangu* before optimization.*

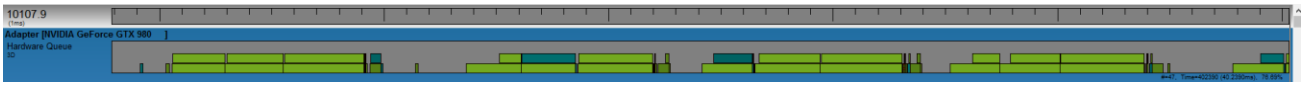


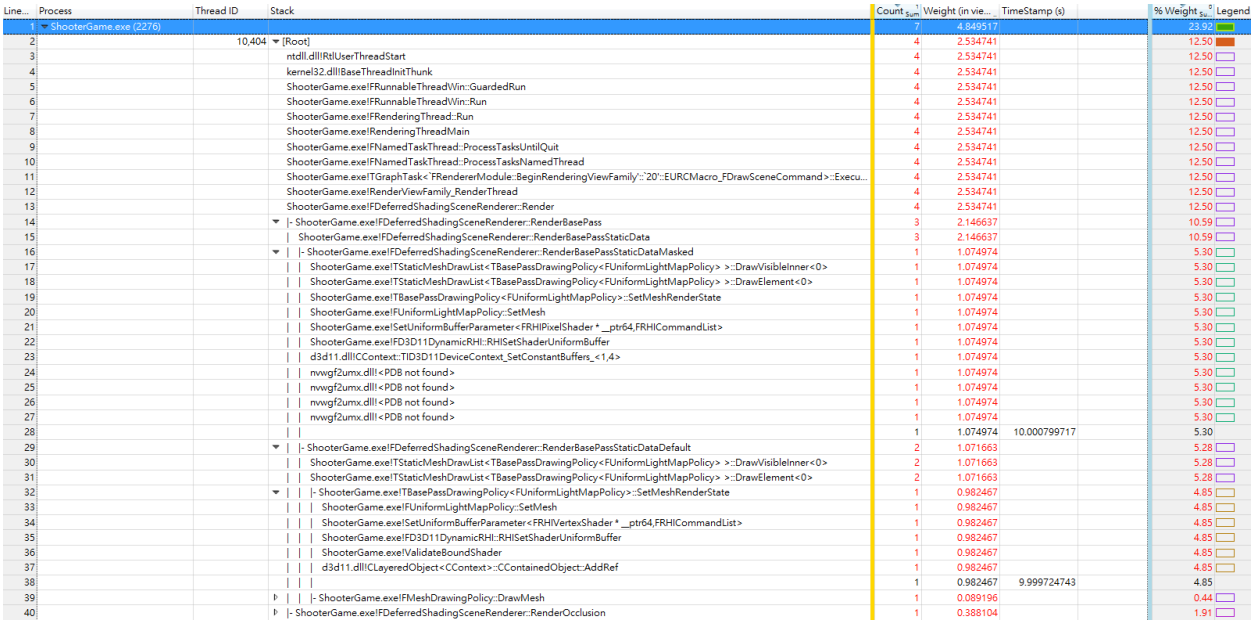
Figure 13: *The GPU utilization of Pangu* after optimization.*

Figure 14: *A timeline view of Pangu* in GPUView after optimization.*

Figure 14 shows the Pangu* VR workload viewed from the GPUView after optimization. The CPU bottleneck period was decreased from 7.37 ms to 2.62 ms after optimization, which is achieved by the following optimizations:

- Running start of the render thread(a method that reduces CPU bottleneck by introducing an extra MTP latency) [5]
- Reduction on the number of draw call and overheads, including the adoption of LOD, Instanced Stereo Rendering, and the removal of dynamic shadowing
- Works in game thread and task threads are deferred to process

Figures 15 shows the call stack of the CPU render thread in the CPU bottleneck period, as marked in the red rectangle shown in Figure 14.



Line...	Process	Thread ID	Stack	Count Sum	Weight (in vie...	TimeStamp (s)	% Weight Su...	Legend
1	ShooterGame.exe (2276)			7	4.849517		23.92	
2		10,404	[Root]	4	2.534741		12.50	
3			ntdll.dll!RtlUserThreadStart	4	2.534741		12.50	
4			kernel32.dll!BaseThreadInitThunk	4	2.534741		12.50	
5			ShooterGame.exe!FRunnableThreadWin::GuardedRun	4	2.534741		12.50	
6			ShooterGame.exe!FRunnableThreadWin::Run	4	2.534741		12.50	
7			ShooterGame.exe!FRenderingThread::Run	4	2.534741		12.50	
8			ShooterGame.exe!RenderingThreadMain	4	2.534741		12.50	
9			ShooterGame.exe!FNamedTaskThread::ProcessTasksUntilQuit	4	2.534741		12.50	
10			ShooterGame.exe!FNamedTaskThread::ProcessTasksNamedThread	4	2.534741		12.50	
11			ShooterGame.exe!TGraphTask<`FRendererModule::BeginRenderingViewFamily'::`20'::EURCMacro_FDrawSceneCommand>::Execu...	4	2.534741		12.50	
12			ShooterGame.exe!RenderViewFamily_RenderThread	4	2.534741		12.50	
13			ShooterGame.exe!FDeferredShadingSceneRenderer::Render	4	2.534741		12.50	
14			|- ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePass	3	2.146637		10.59	
15			| ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassStaticData	3	2.146637		10.59	
16			| |- ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassStaticDataMasked	1	1.074974		5.30	
17			| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<FUniformLightMapPolicy> >::DrawVisibleInner<0>	1	1.074974		5.30	
18			| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<FUniformLightMapPolicy> >::DrawElement<0>	1	1.074974		5.30	
19			| | ShooterGame.exe!TBasePassDrawingPolicy<FUniformLightMapPolicy>::SetMeshRenderState	1	1.074974		5.30	
20			| | ShooterGame.exe!FUniformLightMapPolicy::SetMesh	1	1.074974		5.30	
21			| | ShooterGame.exe!SetUniformBufferParameter<FRHIPixelShader * __ptr64,FRHICommandList>	1	1.074974		5.30	
22			| | ShooterGame.exe!FD3D11DynamicRHI::RHISetShaderUniformBuffer	1	1.074974		5.30	
23			| | d3d11.dll!CContext::TID3D11DeviceContext_SetConstantBuffers_<1,4>	1	1.074974		5.30	
24			| | nvwgf2umx.dll!<PDB not found>	1	1.074974		5.30	
25			| | nvwgf2umx.dll!<PDB not found>	1	1.074974		5.30	
26			| | nvwgf2umx.dll!<PDB not found>	1	1.074974		5.30	
27			| | nvwgf2umx.dll!<PDB not found>	1	1.074974		5.30	
28			| |	1	1.074974	10.000799717	5.30	
29			| |- ShooterGame.exe!FDeferredShadingSceneRenderer::RenderBasePassStaticDataDefault	2	1.071663		5.28	
30			| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<FUniformLightMapPolicy> >::DrawVisibleInner<0>	2	1.071663		5.28	
31			| | ShooterGame.exe!TStaticMeshDrawList<TBasePassDrawingPolicy<FUniformLightMapPolicy> >::DrawElement<0>	2	1.071663		5.28	
32			| | |- ShooterGame.exe!TBasePassDrawingPolicy<FUniformLightMapPolicy>::SetMeshRenderState	1	0.982467		4.85	
33			| | | ShooterGame.exe!FUniformLightMapPolicy::SetMesh	1	0.982467		4.85	
34			| | | ShooterGame.exe!SetUniformBufferParameter<FRHIVertexShader * __ptr64,FRHICommandList>	1	0.982467		4.85	
35			| | | ShooterGame.exe!FD3D11DynamicRHI::RHISetShaderUniformBuffer	1	0.982467		4.85	
36			| | | ShooterGame.exe!ValidateBoundShader	1	0.982467		4.85	
37			| | | d3d11.dll!CLayeredObject<CContext>::CContainedObject::AddRef	1	0.982467		4.85	
38			| | |	1	0.982467	9.999724743	4.85	
39			| | |- ShooterGame.exe!FMeshDrawingPolicy::DrawMesh	1	0.089196		0.44	
40			|- ShooterGame.exe!FDeferredShadingSceneRenderer::RenderOcclusion	1	0.388104		1.91	

Figure 15: *The call stack of the render thread T10404.*

Table 3 shows a summary of the CPU hotspots (percent of clockticks) during GPU bubble periods after optimization. Note that many of the hotspots and threads were removed from the CPU bottleneck as compared to Table 2.

THREAD	FUNCTION	CLOCKTICK %	
Render thread	Base pass rendering for static meshes	44.3%	52.2%
	Render occlusion	7.9%	
Driver		38.5%	

Table 3: *CPU hotspots during GPU bubble periods after optimization.*

More optimizations, such as actor merging or using fewer materials, can be done to optimize the static mesh rendering in the render thread and further improve the frame rate. If CPU tasks were fully optimized, the processing time of a single frame could be further reduced by 2.62 ms (the period of CPU bottleneck in a single frame) to 11.38 ms, which is equivalent to 87.8 fps on average.

Table 4 shows the performance metrics before and after the optimization.

	System Idle	Pangu* on Oculus Rift* DK2 (before optimization)	Pangu* on HTC Vive* (after optimization)
GPU Core Clock (MHz)	135	1337.6	1316.8
GPU Memory Clock (MHz)	162	1749.6	1749.6
GPU Memory Used (MB)	184	1727.71	2253.03
GPU Load (%)	0	49.64	78.29
Average Frame Rate (fps)	N/A	36.4	71.4
Draw Calls (/frame)	0	4437	845
Processor(_Total)\Processor Time (%)	1.04 (5.73/0.93/0.49/0.29/0.7/0.37/0.24/0.2)	13.58 (30.20/10.54/26.72/3.76/12.72/8.16/12.27/4.29)	31.37 (46.63/27.72/33.34/18.42/39.77/19.04/46.29/19.76)
Processor Information(_Total)\Processor Frequency (MHz)	800	2700	2700

Table 4: *Basic performance metrics of the game before and after optimization.*

Conclusion

In this article, we worked closely with Tencent* to profile and optimize the Pangu* VR workload on premier HMDs in order to achieve 90 fps on Intel® Core™ i7 processors. After implementing some of our recommendations, the frame rate was increased from 36.4 fps on Oculus Rift* DK2 (1920x1080) to 71.4 fps on HTC Vive* (2160x1200), the GPU utilization was also increased from 54.7 percent to 74.3 percent on average due to fewer CPU bottlenecks. The CPU bound period in a single frame was also reduced

from 7.37 ms to 2.62 ms. Additional optimizations such as actor merging and texture atlasing could be done to further optimize the performance.

Profiling and analyzing a VR application with various tools gives insights on the behaviors and bottlenecks of the application, and it is essential to VR performance optimization since performance metrics alone might not reflect the real bottlenecks. The methodology and tools discussed in this article can be used to analyze VR applications developed with different game engines and VR runtimes, and determine whether the workload is bounded by CPU, GPU, or both. Sometimes the CPU has a larger impact to VR performance than the GPU due to drawcall preparation, physics simulation, lighting, or shadowing. After analyzing various VR workloads with performance issues, we found that many of them were CPU bounded, implying that CPU optimization can help improve the GPU utilization, performance, and the user experience of the applications.

Reference

[1] https://developer.microsoft.com/en-us/windows/hardware/windows-assessment-deployment-kit

[2] http://graphics.stanford.edu/~mdfisher/GPUView.html

[3] https://msdn.microsoft.com/en-us/library/windows/hardware/hh162981.aspx

[4] https://randomascii.wordpress.com/2015/09/24/etw-central/

[5] http://www.gdcvault.com/play/1021771/Advanced-VR

About the author

Finn Wong is a senior application engineer in the Intel Software and Solutions Group (SSG), Developer Relations Division (DRD), Advanced Graphics Enabling Team (AGE Team). He joined Intel in 2012 and has been actively enabling third-party media, graphics and perceptual computing applications for the company's PC products since then. Before joining Intel, Finn has seven years of experience and expertise in the fields of video coding, digital image processing, computer vision, algorithms and performance optimization, with several academic papers published in the literature as well. Finn holds a bachelor's degree in electrical engineering and a master's degree in communication engineering, all from National Taiwan University.

Notices