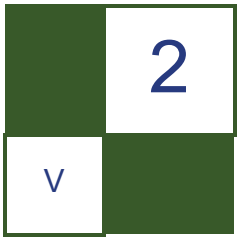

Understanding, Measuring, and Analyzing VR Graphics Performance

James Hughes, Reza Nourai, and Ed Hutchins

Given the meteoric growth of virtual reality (VR), augmented reality (AR), and mixed reality (MR) applications, it is important for us as graphics programmers to understand the performance characteristics of these apps. Specifically, we want to understand how output is generated and displayed and how frames are timed. With those goals in mind, we can then understand how to better optimize our applications. This chapter aims to do just that, providing not only an explanation of how things work, but also how to measure and analyze VR application performance.

Given our backgrounds, we focus on freely available Windows tools. We sometimes describe these in the context of the Oculus Rift consumer VR device. However, the concepts and approaches are broadly applicable.

2.1 VR Graphics Overview

Let's start by defining a few terms used throughout this discussion.

- An application is a software client on the system. These are typically what we as developers write.
- A system service is a component that is running externally to our application. Normally a service is part of the OS or software runtimes we have installed. For example, the audio mixer on Windows is a system service that is part of the OS.
- A compositor service is any specific system service that arbitrates the display between the applications running on the system, potentially mixing their outputs into a single output to send to the display. An example is the

Desktop Window Manager (DWM) on Windows. In all cases of interest for this discussion, either the operating system or the VR runtime environment we are targeting provides this compositor service. This is not a component we typically write, but it's important to understand its function at a basic level.

- An HMD is a head-mounted display, the hardware that presents a stereo-pair of images to the user's eyes.
- The position sensor is comprised of one or more hardware components responsible for establishing the position of the user's HMD in the real world.
- A pose is the position and orientation of the HMD measured from the position sensors and possibly extrapolated to some future point in time.
- A frame is a single iteration of the application in order to update its output. Typically, each time a frame is run, the output of the application is updated. This is often in the form of an image containing the rendered results.
- VSync refers to the interval of time during which the display can select a new frame to output. In the original electron-beam based displays (cathode ray tubes or CRTs), this interval was used to return the electron beam to the start point of the display to begin scanning out the next frame's scanlines.
- Frame start is the time that the application starts processing a frame based on the latest pose. The synchronization of frame start may be optimized for the application by the compositor service.
- The display interval is the time that a particular frame's pixels are illuminated for viewing.
- Latency, in the context of this discussion, is the delay between drawing a frame and displaying it to the user. Unless otherwise clarified, this will be specifically the time from application frame start to the time when the pixels become visible to the user.

2.1.1 Background

In VR applications, too much latency can lead to user discomfort and break the immersive experience. Minimizing latency, and therefore increasing user comfort, is a major goal of most VR systems. The way frames are timed and produced in these systems is optimized around minimizing latency, which is somewhat different than how traditional graphics applications are tuned.

In a non-VR application, frames are produced at some frequency and displayed on a two-dimensional screen. On most modern operating systems, the frame produced is not displayed directly, but buffered to a system-wide compositor service such as the Desktop Window Manager (DWM) on Windows or SurfaceFlinger on Android. The compositor service then prepares a final frame for sending to the output display device, combining the output of one or more running applications. Typically, the compositor service produces frames for output at a frequency that matches the display's refresh rate. It is often a requirement that the frequency of display in the compositor is coupled only to the display refresh, not to the frequency at which multiple applications may be producing frames.

Let's look at the typical breakdown of a frame for traditional applications targeting a two-dimensional display, such as a monitor. First, the application starts executing logic on the CPU, and then eventually starts calling a graphics API such as DirectX or OpenGL. Once the application has either completed its calls to the graphics API or exceeded the buffering limit for driver commands, the workload is submitted to the OS kernel or graphics driver. This work eventually gets submitted to the GPU hardware, where it begins execution. The GPU processes these commands asynchronously to the CPU. While this is happening, the CPU is often able to continue on and start doing the work for the next frame.

Once the GPU work for the frame is complete, the system-wide compositing service is notified. On the next composition iteration, the updated output of this app is composed into the final frame buffer and submitted to the display controller for scanout. Some display systems have additional latency while output is transmitted over a physical link before it can be displayed to the user. This all adds up to substantial latency between the frame start and the time the

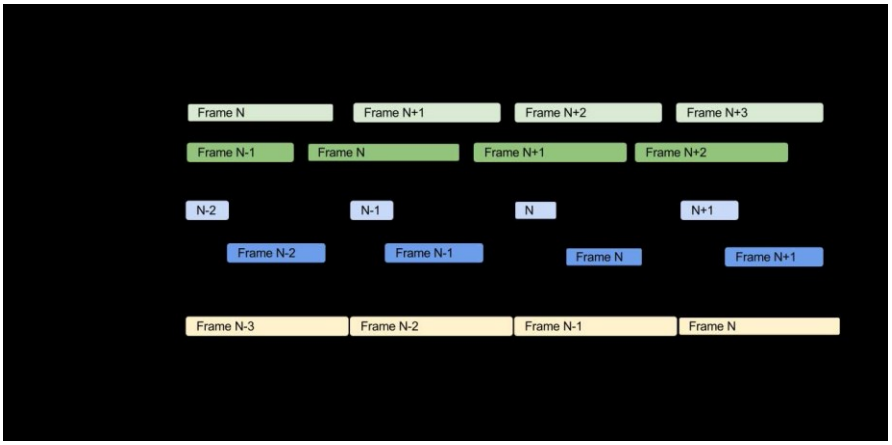

Figure 2.1. Example of traditional frame timing.

photons generated by that frame reach the user's eye (see Figure 2.1). Note the highlighted sequence that shows how Frame N gets from the CPU all the way to being scanned out.

In fact, applications often buffer multiple frames to the compositor service in order to smooth animation across variable workloads or OS interruptions. This adds additional latency. Some games incur additional latency even beyond that, if they decouple their simulation and rendering threads and let the simulation thread run ahead of the render thread. For example, both Unity and Unreal Engine do this.

What does all of this latency mean for VR? In a naive implementation (for example, a simple port of an existing 3D app), the position and orientation of the user's head is read at the frame start and an image is rendered based on that pose. That image is going to be sorely out of date by the time the user actually sees the pixels. Their actual head pose could have changed dramatically in that amount of time. This leads to swimmy, or laggy, visuals that can often make the user feel sick (https://en.wikipedia.org/wiki/Virtual reality sickness).

2.1.2 Pose Prediction

Virtually all VR libraries available today include the ability to predict the head pose at some specified point in the future. This increases comfort, because if you get an accurate prediction of where the user's head will be at the time you expect the pixel to illuminate, then you can render an image appropriate for that pose. However, since the head can change direction a lot in a relatively short period of time, prediction accuracy drops significantly as the time interval over which you are predicting increases. Predictions are generally pretty accurate for small intervals, on the order of five to ten milliseconds into the future. But the average error in the prediction, particularly in rotation, grows exponentially with interval length beyond that.

If the predicted pose matches the actual head pose when those pixels are displayed to the user, then the user does not perceive any latency. It does not mean the latency isn't there—it is. It still took some number of milliseconds between when the rendering occurred and the pixels were illuminated for the user. For a great VR experience, it is therefore critical to reduce the perceived latency as much as possible. This necessitates having great prediction algorithms and reducing the prediction intervals to improve accuracy.

For more information about prediction, please see the following blog post: http://ocul.us/2awPpcy.

2.1.3 The Modern VR Frame

One of the simplest improvements over typical frame timing for VR is to bypass the default operating system compositor service, and this is exactly what many of the current VR rendering systems on the market do. Technically, they still go

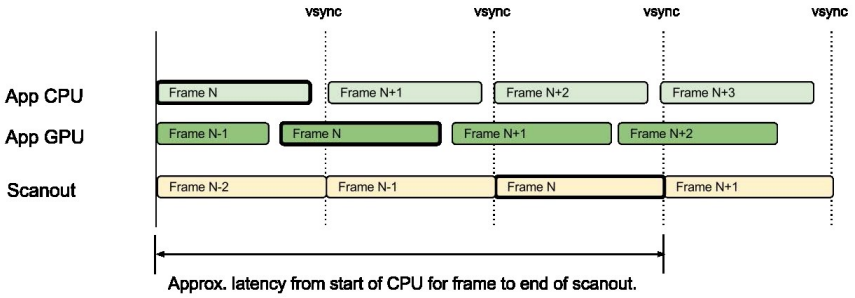

Figure 2.2. Removing the OS compositor.

through a separate composition pass, but it's their own highly tuned replacement for the OS default one. See Figure 2.2, which is a large improvement!

Looking closer at the diagram, we see that the total amount of time for both the app's CPU and GPU usage is greater than a single frame. This is quite common, and it is what allows applications to fully utilize the hardware available. However, for some trivial titles, it may be possible to fit both the CPU and GPU work into a single frame. This would allow you to complete the GPU work in the same frame as you started the CPU work, and scanout immediately on the next frame, reducing latency by another full frame. This is certainly possible, but does limit the utilization of the CPU and GPU resources, which is quite restricting to the application. Instead of making that scenario a special case, a couple of other techniques have been developed to help minimize the perceived latency.

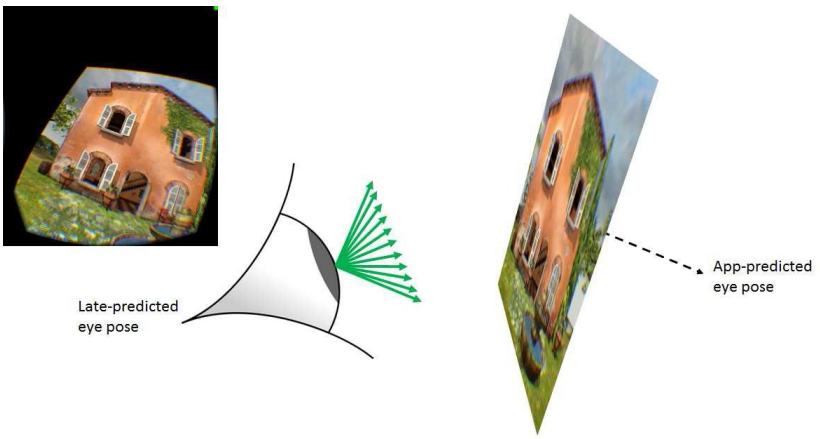

Figure 2.3. Actual vs predicted pose.

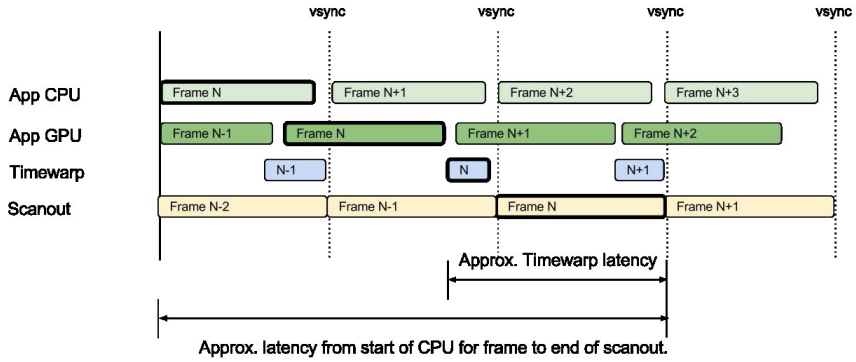

Figure 2.4. Timewarp.

The first of these is a reprojection technique called timewarp. Timewarp is a post-processing step that runs after the app's GPU work has completed. A new head-pose prediction is made for the same point in time for which the original was intended. Since this is happening much later, the accuracy of that prediction is much better than the original one. The difference of the two head-pose predictions is calculated and used to reproject or "warp" the original output from the application to a more accurate location so that it matches the user's head orientation more closely.

Figure 2.3 shows a particularly large change in actual versus predicted pose; in the typical case the change is small and there is a sufficient excess border around the rendered frame to hide the effect of the warping. Timewarp reduces perceived latency dramatically (see Figure 2.4).

For more information about Timewarp and how it can operate asynchronously, please see the following post on the Oculus developer blog: http://ocul.us/1TGXXtY.

Another technique for reducing latency is to delay the start of the CPU frame if the full time is not needed. Even if the full time is needed, the application can delay requesting the predicted pose until as late as possible. This can be done by the application or by the VR library being used. For example, the Oculus Rift PC SDK already manages and optimizes frame start times for the application, so no additional work is required. In the illustration in Figure 2.5, frame start is delayed by the SDK to allow a more accurate pose estimation based on the performance of prior frames:

This optimization of frame start times can work both ways. For example, it is not uncommon for the application to complete its CPU work while the GPU is still busy rendering the results. In this case the VR runtime can increase inter-frame parallelism by starting the CPU work for the next frame early, before the current frame has been displayed. Whether it is starting the work earlier, or

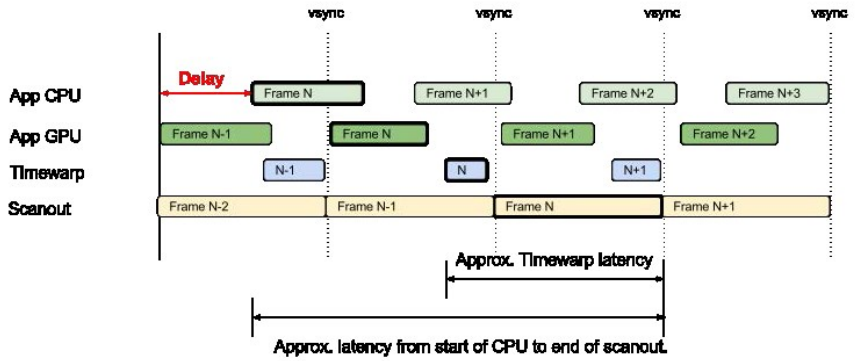

Figure 2.5. Delayed frame start.

later, carefully optimized frame starts are a key to reducing overall latency and improving prediction accuracy.

2.1.4 Optimizing Stereo Rendering

Current VR libraries expect the application to render two views, one for each eye. A naive implementation would just run the rendering code in the game or application twice, but this is the theoretical worst case for performance due to bandwidth duplication and poor cache utilization. These two views have many similarities that can be exploited, and many techniques and GPU features have been developed to help reduce the duplication of work. See Íñigo Quílez's article (Chapter 1 in this section) for a discussion of major methods. There are many other good resources available online for further reading.

2.2 Trace Collection

Understanding where an application's performance can potentially be improved requires the collection and analysis of data from typical use cases. In the past, rolling your own instrumentation or using any of the vendor-specific analysis tools worked well enough for 3D applications where the metric to be tuned was frame throughput. For VR applications, a whole-system approach is required due to the multi-process nature of frame composition and the increased emphasis on reducing latency.

High-end desktop VR is currently primarily a Microsoft Windows ecosystem phenomenon. Therefore, we will focus on Event Tracing for Windows (ETW), the primary system-level tracing mechanism for Windows. If you come from a Linux or Apple background, ETW is the rough equivalent of ptrace or dtrace. In addition, we will focus on the practical use of ETW as it relates to VR application

analysis. We refer you to the literature (http://tinyurl.com/j9fohk3) for in-depth discussion of ETW itself. Finally, due to the authors' familiarity with the Oculus SDK, we will also focus on that particular vendor's support for ETW.

2.2.1 A Brief Description of ETW

ETW is a high-performance thread-safe structured logging mechanism that permits both kernel-level and per-process recording of events in an efficient binary (as opposed to textual) format. Events have a common header that identifies the event type, process and thread Id, timestamp, optional call stacks, and other common information. The optional payload can contain fixed or variable-sized binary-format primitives described using an XML-format manifest file that is also used to decode the collected trace data.

When tracing is disabled, the tracing code is skipped with an inline conditional, making the inclusion of instrumentation in released code practical. Traces can be written to a file or processed in real time, but be aware that the current implementation uses a fixed-size circular buffer and can drop events when the system is under heavy load. The trace will indicate dropped data if this occurs. In practice, ETW has proven to be quite robust when analyzing VR applications.

Instrumented code is identified as one or more ETW "providers" and has an associated globally unique name and GUID. Since ETW is a system-level tracing mechanism, external tooling typically is used to enable trace collection. In Microsoft's nomenclature, these apps are called "controllers," examples of which include LogMan, wevtutil, TraceLog and XPerf. These tools manage tracing sessions, which can remain active across multiple process invocations.

Finally, ETW analysis applications are referred to as "consumers," some examples of which include PerfView, Windows Performance Analyzer, and GPUView. We will focus on GPUView in this chapter since it provides the best view of the internal workings of GPUs and the graphics software stack.

2.2.2 Collecting ETW Traces

The Windows tool XPerf has a wrapper script called log.cmd that can be used to collect ETW traces suited for use with GPUView (https://msdn.microsoft.com/en-us/library/windows/desktop/jj585574.aspx). The Oculus SDK provides an enhanced version of this script named ovrlog.cmd under the OculusSDK\Tools\ETW directory of the Oculus SDK. In addition, there is an ovrlog win10.cmd for Win10 users, and an install.cmd for installing Oculus-specific ETW manifests. Installation of the manifests is required in order to enable ETW consumers to parse the binary-format event payloads. Installed manifests will persist in the system once installed until explicitly removed or updated with newer versions.

Assuming you've installed the Oculus SDK, open an Administrator shell and change directory to the SDK's Tools\ETW directory. In this directory you should see the following:

```
EventsForStackTrace.txt
LibOVREvents.man
OVRUSBVidEvents.man
install.cmd
ovrlog.cmd
ovrlog_win10.cmd
```

The EventsForStackTrace.txt file is used by XPerf to annotate certain ETW events with full stack traces. The .man files are the ETW manifests that describe the providers and custom payloads for the Oculus SDK. The install.cmd script will install these manifests into the system so that Oculus providers can be properly enumerated and so that recorded SDK events will be parsed correctly by tools such as GPUView. Run the install.cmd script from an Administrator console and verify that it has installed the manifests correctly. Note that one failure mode that is difficult to diagnose is having a mismatch between runtime and installed manifests; when in doubt re-run install.cmd.

You should now be able to run the appropriate ovrlog.cmd or ovrlog_win10.cmd to start collecting an ETW trace. After running the application or applications to be traced and performing whatever actions you are interested in analyzing, run the same command with the argument "stop" to stop tracing. Note that running the command again with no arguments from the same command prompt will also stop tracing, much like the original Microsoft log.cmd. However, this method is more failure-prone due to its reliance on environment variable settings.

2.2.3 GPUView

While there are a number of tools that consume ETW traces and provide graphing and event-search features, the most useful of these tools for VR purposes is Microsoft's GPUView.

GPUView provides an interactive graphical view of an ETW trace, with an emphasis on GPU and CPU utilization. It displays per-packet and per-thread details for these key resources along with per-process breakdowns. These views allow you to visually trace the impact of a process' actions on the system as a whole. Arbitrary events can be filtered for and displayed as time markers on the utilization graphs, allowing accurate measurement of time deltas between events.

An in-depth introduction to using GPUView is beyond the scope of this article (see https://graphics.stanford.edu/~mdfisher/GPUView.html). However, we will show how to identify an HMD's VSync, which can be somewhat tricky. This example will assume you've captured a trace of an active VR application and loaded it into GPUView.

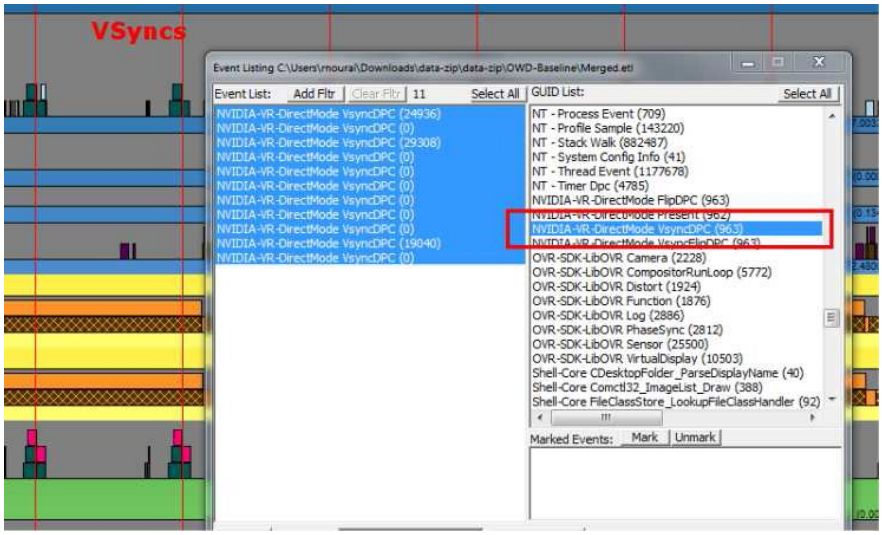

Figure 2.6. VSync event list.

Identifying VSync. GPUView has built-in support for displaying vertical sync information when the Windows OS is aware of the display. As of the time this chapter was written, HMD support is implemented by the GPU vendors in a way that hides their display from the OS in order to prevent the OS from using the HMD as part of the desktop. This means that traces gathered on these systems won't have VSync information available. As Windows improves native support for HMDs, this limitation will disappear and the standard GPUView VSync display information should work.

The good news is that, in the meantime, GPU vendors have instrumented the HMD's vertical sync to enable performance analysis. To view the information, you'll need an ETW manifest from the GPU vendor. AMD deploys the manifest as part of their typical driver install. NVIDIA provides theirs only with their GameWorks VR SDK.

Once installed, both GPU vendors provide VSync events in the event list. Figure 2.6 shows an example using an NVIDIA GPU.

2.2.4 Adding Custom Tracing Events

In some cases, additional visibility into the internal workings of your application would be very helpful when combined with the rest of the ETW data being collected. The Oculus SDK provides ovr_TraceMessage(), which can be used to trace string messages as ETW events. In cases where structured events are desired it is also possible to add your own custom provider to your application.

While ETW is powerful, it unfortunately suffers from a cumbersome API that makes adding ETW tracing support to your application fairly difficult (https://mollyrocket.com/casey/stream_0029.html). We'll briefly sketch what needs to be done in order to accomplish this.

The first step is to create a manifest. Look for ecmangen.exe under Windows\Kits and launch it. Try loading the LibOVREvents.man file and examine how the different categories are utilized. Hitting F1 to open the help page is a good idea at this point. Create a new manifest and try adding the application tasks you wish to trace (e.g., loading, rendering, AI, etc.) and templates for any custom data structures. Then try adding events corresponding to specific points in your code you wish to trace.

Once you save the manifest, you'll need to compile it using the message compiler mc.exe found under the top-level Windows\Kits directory. For a detailed description and example C++ code see https://blogs.msdn.microsoft.com/seealso/2011/06/08/use-this-not-this-logging-event-tracing/. Once successfully compiled, you will have a generated C++ header file that you can include in your application's source, as well as resources to add to your project.

Once your source is instrumented and compiled, you will need to install the manifest you've created. See the Oculus SDK's install.cmd as an example of how to do this. After installation, the command "wevtutil ep" should show your provider as available in your system. Add your provider's GUID and appropriate start and stop commands to a local copy of ovrlog.cmd, then start tracing, run your application, and stop tracing. If all has gone well, at this point you should be able to see your custom events in GPUView's event viewer.

2.3 Analyzing Traces

Now that we've seen how to capture and look at traces, it's time for deeper investigation. We will cover examples demonstrating the lifecycle of a VR frame to better understand the effects of CPU and GPU constraints. Analyzing the behavior of a number of different applications will allow us to decode the interaction between the application and runtime and identify potential problems. We'll conclude by discussing common pitfalls and possible tradeoffs.

2.3.1 Let's Baseline

Let's work towards understanding a frame in VR. Ideally, an application exploits the asynchronous nature of the GPU by beginning the next frame on the CPU before the prior frame finishes. Nowadays this staggering of the CPU work relative to the GPU work is optimized by the SDK's frame start API. We'll assume this is happening for this example. Let's begin by taking a look at a well-behaved VR frame's lifetime.

We start by selecting a VSync and working backwards through the lifecycle of a frame. Once the VSync is selected, we determine what application GPU

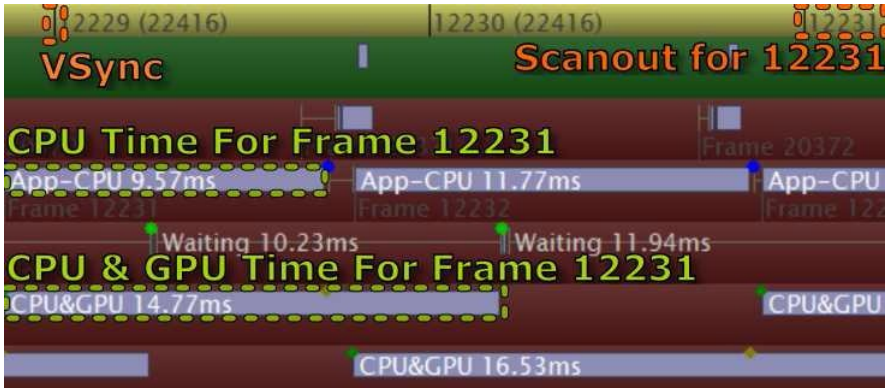

Figure 2.7. Conceptual view.

work was scanned out at the VSync and when that GPU work was scheduled. In Figure 2.7, the topmost diagram depicts the high-level concept of a real-world frame—the conceptual view. You'll see an actual GPUView trace of the application in Figure 2.8. In both diagrams, application-specific annotations are green while VR runtime or system specific annotations are orange.

In these two figure, we've chosen the VSync at which frame 12231 started scanning out. Both the conceptual view and GPUView traces in this section follow the lifecycle of this frame, but you could just as easily follow along in GPUView using some of your own data.

We'll begin with the high level conceptual view of the frame we intend to cover. In the conceptual view, finding the CPU work for our frame is straightforward: "CPU Time For Frame 12231" (App-CPU) and "CPU & GPU Time For Frame 12231" (CPU & GPU). "CPU Time" represents only the time spent by the application on the CPU, while "CPU & GPU Time" represents the time from the beginning of CPU work to when the application's GPU work finished. These conceptual regions align directly with the GPUView trace below.

In the GPUView trace in Figure 2.8, you'll notice the beginning of application CPU work denoted by "App CPU Begin" and the end of GPU work specified by "App GPU End." The VSyncs are clearly marked in both figures.

Using GPUView we want to deduce what CPU work interval led to the frame that was scanned out. Since we have already chosen a VSync in GPUView, look at the "LibOVR-Log - App EndFrame" event directly before this VSync. This event is denoted by "End Frame Timing." Figure 2.9 is a screenshot of the events dialog in GPUView showing the "End frame app timing" event selected. Find the "FrameIndex" data within this event since that contains which application frame index was scanned out at our target VSync. In this case "FrameIndex" contains 12231. This tells us the frame index we want to look for in the events that tell us the bounds of our CPU work.

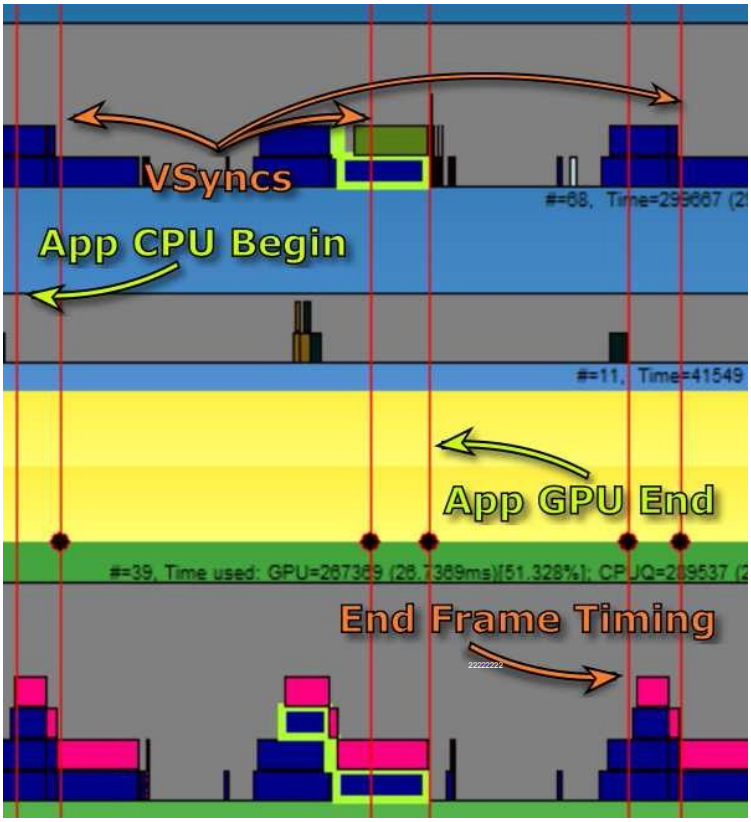

Figure 2.8. GPUView trace.

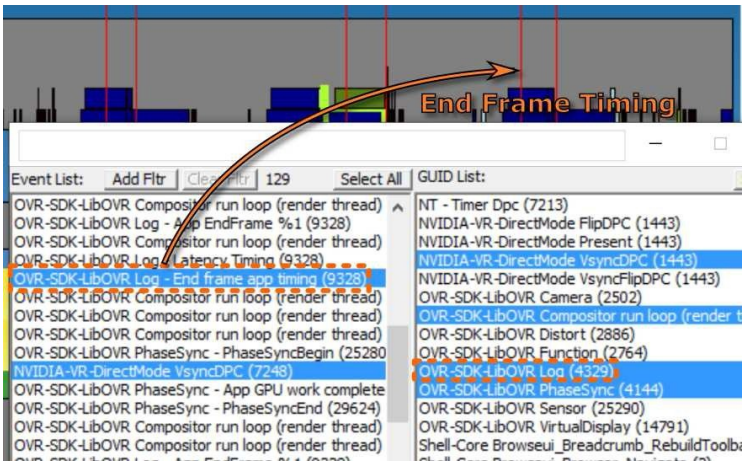

Figure 2.9. Events dialog in GPUView.

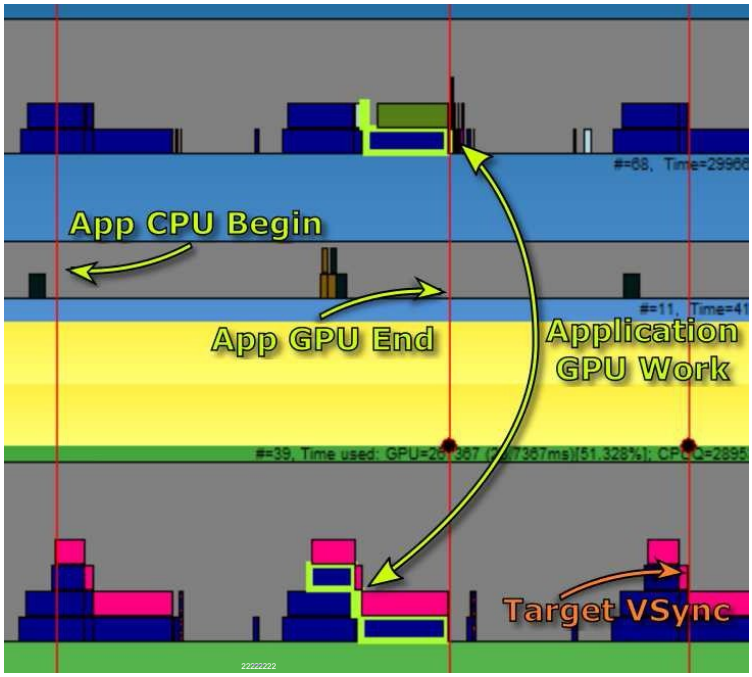

Figure 2.10. Example for frame 12231.

Find the "LibOVR-Phasesync – PhaseSyncEnd" event and match our frame index with the "FrameIndex" event data as before. It should occur within one VSync interval before our target VSync; an example for frame 12231 is given in Figure 2.10.

Now that we know the appropriate PhaseSyncEnd event we can easily identify when the CPU work began for the application: look for the corresponding

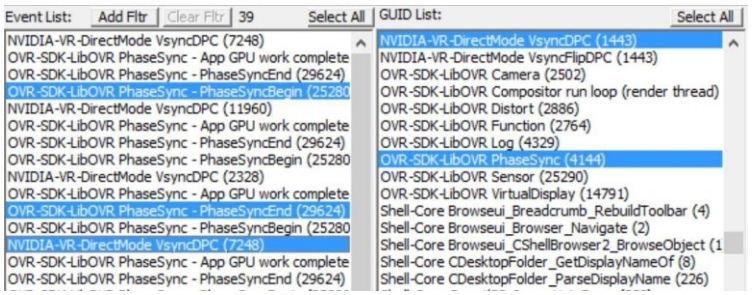

Figure 2.11. The events bounding the application's CPU work.

PhaseSyncBegin event. These two events bound the application's CPU work as depicted in Figure 2.11

Though these two events bound the application's CPU work, the PhaseSyncEnd event represents when the application GPU work ended, not when its CPU work finished. The PhaseSyncBegin event, on the other hand, truly demarcates when the application began CPU work.

Note the "Application GPU Work" in Figure 2.10. The selected regions depict the execution of the application's GPU work on hardware and the device context queue. Application GPU work will always finish directly before the PhaseSyncEnd event. This figure closely matches the intervals in the conceptual view of frame 12231. The time between the "App CPU Begin" and "App GPU End" markers are representative of the "CPU & GPU Time For Frame 12231" interval in the conceptual view.

Knowing the PhaseSync events, the end-frame timing event, and a target VSync, you will be able to follow the lifecycle of any VR frame as outlined above. If you run into performance issues, this is a good place to start diagnosing the problem.

You can find the compositor GPU distortion work for the frame presented at the VSync. The conceptual view (Figure 2.12) shows this work in the green "Compositor GPU" section directly below the VSync markers. In GPUView (Figure 2.13), look for GPU packets directly before VSync that correspond to the process of the compositor (OVRServer.exe in the case of Oculus' runtime). Click on one of the packets in the hardware queue, and you'll be able to locate the corresponding process and CPU thread responsible for generating the GPU work. It is helpful to collapse all other processes' graphs and explicitly open only the CPU graphs for your application and OVRServer.exe to make this more obvious.

Once you've identified these major parts of a frame's anatomy you're well on the way to diagnosing the different possible failure modes of a misbehaving VR application!

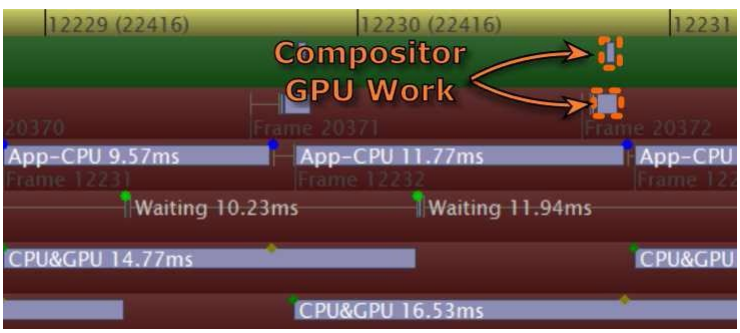

Figure 2.12. Conceptual view.

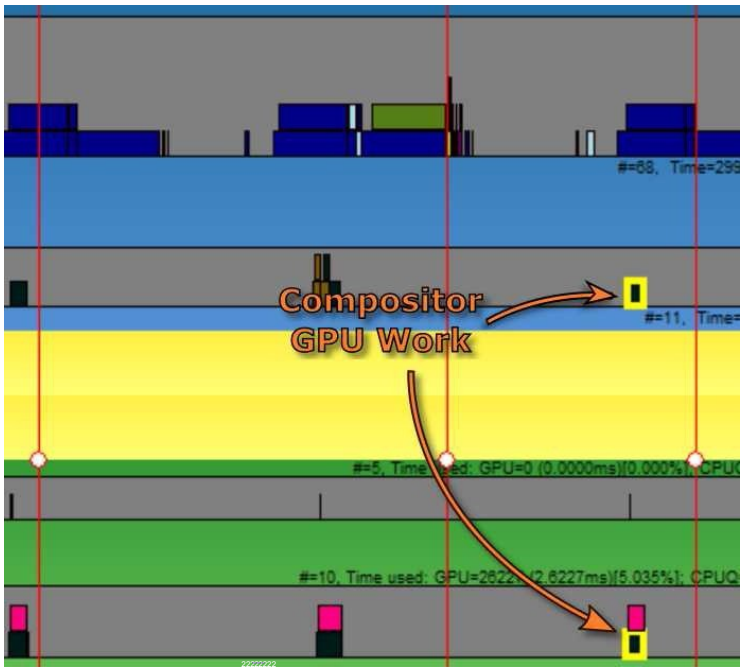

Figure 2.13. GPUView.

2.3.2 Lost App Frames

Now that we understand the lifecycle of a VR frame, let's look at problematic applications for additional perspective. In Figure 2.14, you'll see the lifecycle of frame 3744 where the application is under high GPU (and CPU) load.

Each pair of yellow boxes on the conceptual diagram (top) means the compositor flagged the application as missing a frame. In this case, frame 3743 was scanned out twice before the application was able to submit the next frame to the VR runtime. The runtime compensates for the missed frame by re-timewarping frame 3743 to correctly display it with the most recently available HMD pose.

In both the conceptual and GPUView diagrams (bottom), the "App CPU Begin" to "App GPU End" exceeds two VSync intervals, indicating the application has missed a frame. The application's CPU alone takes 16 milliseconds to complete, which means we can't meet the 11 millisecond (90Hz) refresh rate of the HMD. If we inspect the GPU work for the application, we also note that the total GPU work is roughly 20 milliseconds. So, while we take a combined time of 27 milliseconds to generate two frames in VR, more than twice the refresh rate of the device, we still only miss one frame every other VSync. This is due to starting the next frame before the GPU work for the prior frame completes.

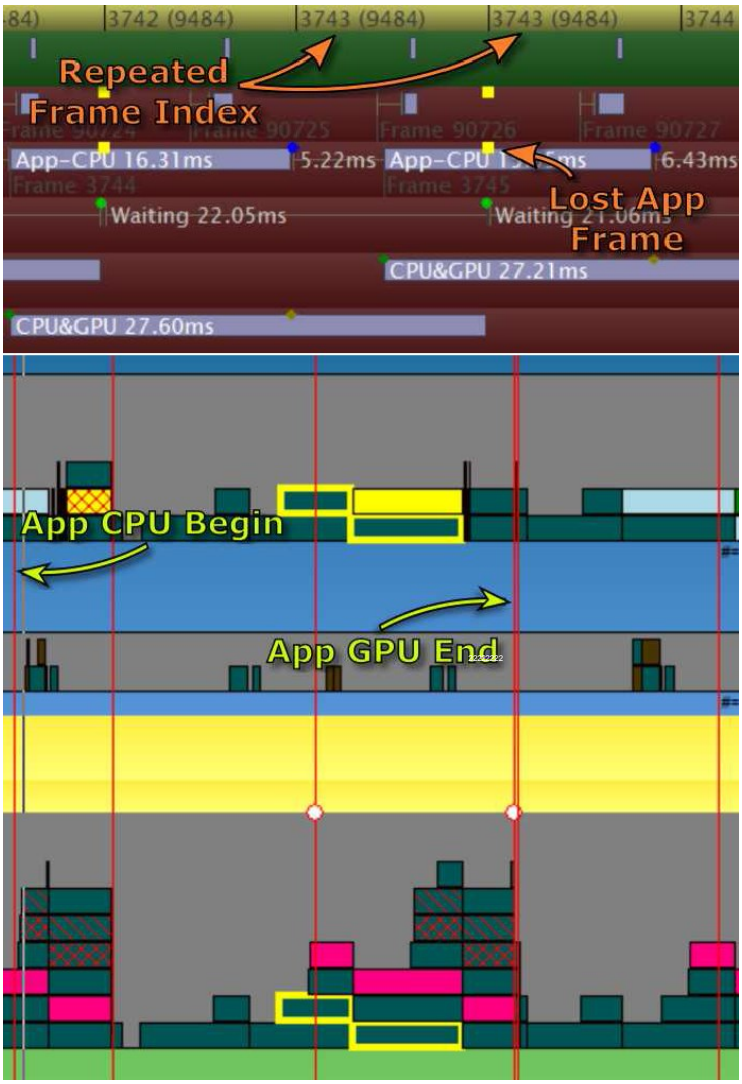

Figure 2.14. Lifecycle of frame 3744.

This staggering of frames 3744 and 3745 is easier to see in the conceptual view. In this case, the second frame begins work six milliseconds before the GPU work of the last frame completes, saving us from dropping two application frames in a row. In GPUView you can see this overlap between disjoint pairs of "PhaseSync Begin" and "PhaseSync End" events.

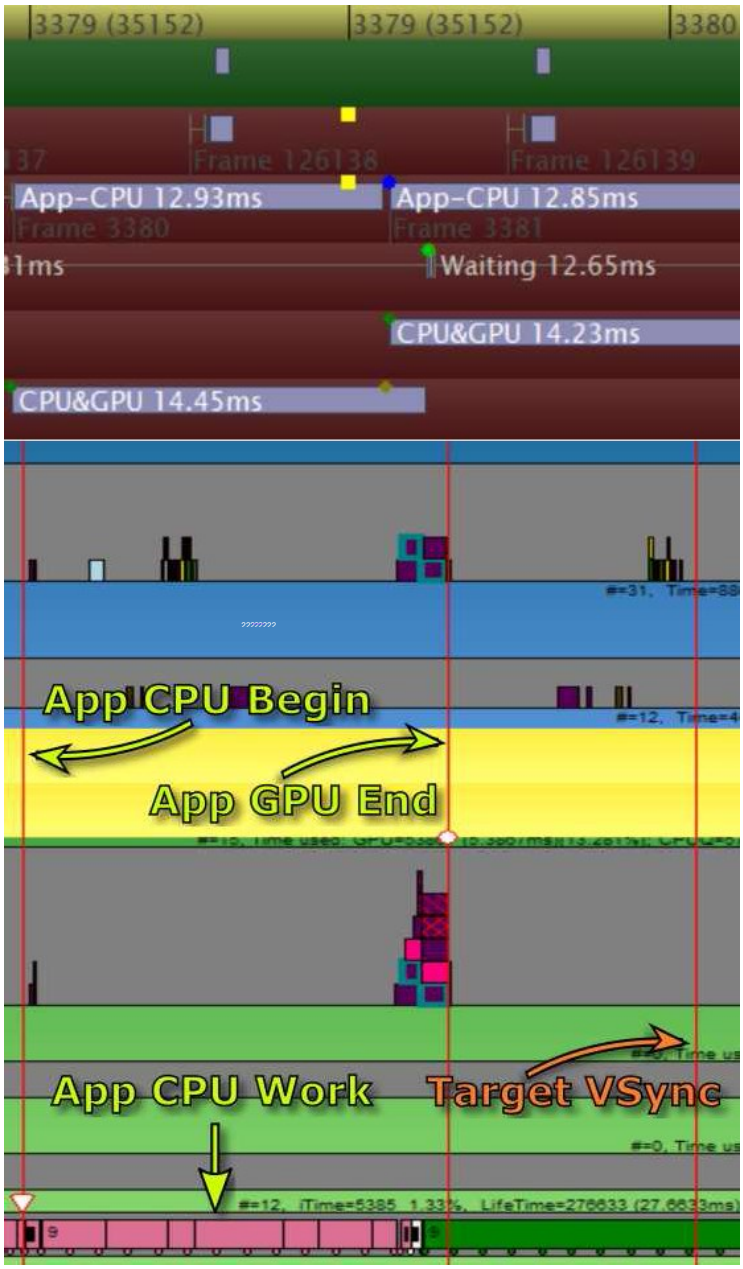

Figure 2.15. High CPU load.

2.3.3 High CPU Load

Moving on to a purely CPU-bound application, we'll find that there's not much the runtime can do to help with parallelism.

This application's CPU load (see Figure 2.15) is high relative to the amount of GPU work it performs. This is easy to see in GPUView (bottom image), as the hardware queue remains empty for nearly the entire frame. Worse yet, the frame CPU time of 12.85 milliseconds is longer than a frame interval. As such, the runtime is unable to compensate, and we see an application lost frame in the form of a re-timewarp of frame 3379. These sorts of failures are a strong indication that attempting to exploit further CPU parallelism and advancing the start of GPU work will benefit your application.

One Sundance Film Festival virtual reality experience utilized high-priority multi-threaded CPU video decoding. The total CPU load was sufficient to starve the VR compositor, causing lost frames. Switching to a different GPU-based decoding library and reducing the CPU thread priorities resulted in a flawless VR experience.

2.3.4 Lost Compositor Frames

In some cases the VR compositor may be the source of a missed frame. These cases are particularly bad since they result in judder perceived by the user. There's not much an application developer can do to prevent compositor lost frames, but it is useful to understand what these traces look like (see Figure 2.16).

In this case the distortion GPU-work for frame 21729 took too long to complete. As a result, the compositor missed VSync and was unable to present a frame on the device. In the conceptual view (top of figure), you see this manifest as no compositor GPU work following the missed VSync. In GPUView (bottom of figure), the GPU work that was unable to finish in time is highlighted and its work ends right against VSync, yielding no time to present the results. If this failure mode occurs frequently with your application or certain hardware configurations, consider reaching out to your VR SDK vendor.

In our experience, these sorts of reports have sometimes uncovered edge cases in the VR compositor. For example, one constructive solid geometry ray-tracing application was generating primitives that proved difficult to preempt, requiring collaboration with GPU hardware vendors to improve their preemption performance.

2.3.5 GPU Resource Contention

Since the GPU normally works on tasks asynchronously after the CPU is done submitting, there arise scenarios where the CPU would like to access the resource again but the GPU is not done with it. In these cases, contention often leads

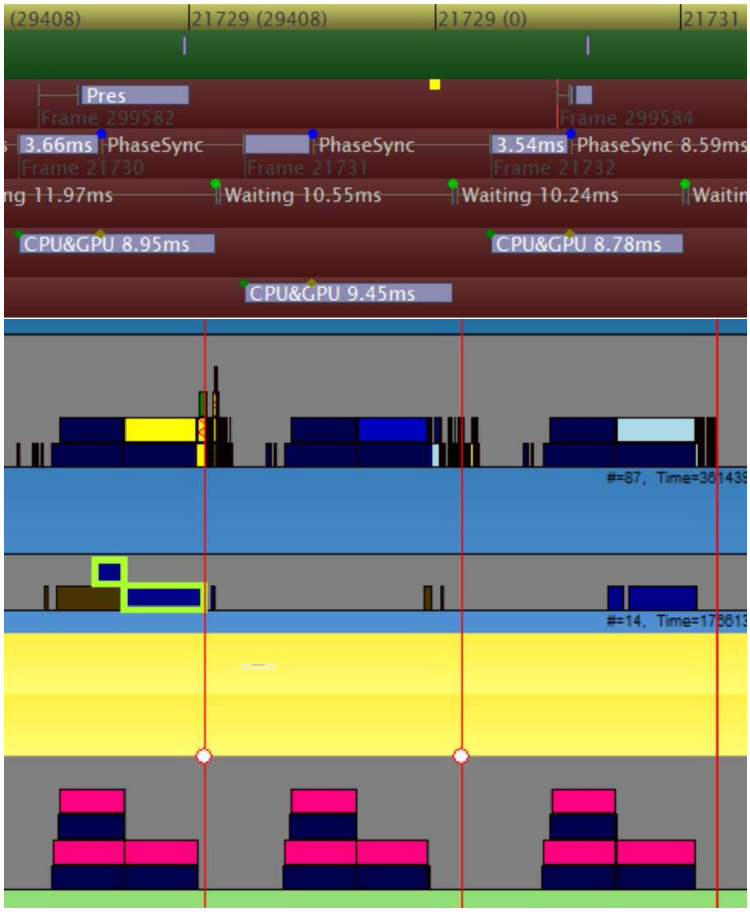

Figure 2.16. Lost compositor frames.

to the CPU stalling until the GPU work completes and no longer requires the resource. Some common examples of this are explored in the following sections.

Driver resource renaming. Prior to D3D12 and Vulkan, older graphics APIs normally did a bunch of work in the driver to assist applications with common use cases. One of the most common use cases is something called driver resource renaming. When an application wants to update a resource with new data from the CPU, it requires either gaining access to a pointer to the resource, or providing an application pointer to be copied to that resource. However, the GPU may still have work pending that depends on the existing data in the resource. In that case, the driver often creates a new copy of the resource (or, more practically, keeps a circular buffer of copies of that resource) and provides

that new pointer to the application. In this way, the application updates this new instance, while the GPU is still referencing the old reference. Using the same constant buffer to feed world transforms to all draw calls might trigger this case, for example.

Once the GPU has completed using the previous copy, it is released back to the driver and reclaimed, to be used again. This trick is called "resource renaming." Drivers have limits on how much memory they set aside for renaming. When this limit is reached, the driver blocks in the call until the GPU has released and reclaimed some of the existing instances. This stall can cause dropped frames and jumps in the frame rate.

In VR applications, it can be tempting to reuse the same buffers between left and right eyes when doing stereo rendering. Be aware that this will generate twice as many updates to the resource in a single frame and can push you into the resource-renaming failure mode twice as soon as a typical application. In general, reuse of non-trivial buffers multiple times in a frame should be avoided to help prevent running into resource-renaming limits. For VR, in particular, it is good practice to use unique buffers for the left and right eye if sufficient memory is available.

As a concrete example, let's walk through a case where the application has a single constant buffer used to transfer constants (or uniforms) to the GPU for several different draw calls. Since the CPU work for the render thread is often ahead of the GPU work, when the app maps the constant buffer a second time to put in new values, it will trigger resource renaming.

With each additional mapping of that buffer to write new values, new copies are being created in the driver. Finally, when the GPU catches up a frame or two later (depending on how deep the rendering queue is) and those values are read from the older copies of the resource, the memory is again made available. Since rendering left and right eyes can double the draw calls in a naive implementation, you now have twice as many renames of the same buffer and, hence, are more likely to stall the CPU.

We ran into a particularly nasty example of this in a popular game engine right before a major trade show. A small dynamically-updated vertex buffer was causing many demos to drop a frame approximately once per second. Upon debugging with GPUView, we found that resource renaming was the culprit. Some heroic last-minute hex-editing of the binaries to enlarge the buffer swept the issue under the rug and saved our bacon at that show.

The new low-level graphics APIs such as Vulkan or D3D12 have eliminated driver renaming, making the tradeoff between buffer duplication and explicit synchronization visible to your application. One simple trick to avoid this particular pitfall on low level APIs is to avoid reusing the same constant buffer multiple times in a frame. Instead, create a unique buffer for each use. Leverage this to create your own circular buffer of resources to avoid stomping over in-use resources or causing stalls.

Mapping an in-use resource. Mapping a resource for CPU access can cause stalls in other scenarios as well. Consider the case where there are pending GPU writes to a buffer and the CPU would like to read from that resource by mapping a pointer to it. In this case, the CPU would not be able to read the latest data unless the GPU has completed the writes, which requires waiting for the GPU work to complete. Similarly, a stall can occur if the CPU attempts to map a surface for writing but the GPU is still reading from it. In these cases, advanced techniques such mapping the region no overwrite (which is an implicit promise from the app to the driver that it will take care not to write to portions being access by the GPU) may allow the driver to skip synchronization.

GPU preemption. In cases where the VR compositor is the cause of missed frames, one thing to consider is the possibility that the final GPU work is difficult to preempt. You can spot this in GPUView as a GPU packet spanning the period where the VR compositor should be running. This can arise when large full-screen quads are being rendered at the end of the scene for deferred rendering or post-processing steps. If you suspect this might be the case, experiment with dicing these large primitives into tiles to see if that improves compositor performance. Note that larger triangles may improve performance of your app itself, but if that is producing a workload that is difficult for your GPU to preempt, it can have detrimental effects on the overall VR experience.

2.4 The Big Picture

If there are only four things you remember after reading this chapter, they should be the following:

- Keep your CPU and GPU work per frame each under a single frame interval. Neither the CPU nor GPU work can individually exceed the frame interval;
- Keep the total time from the start of the CPU to the end of the GPU work under two frame intervals in total;
- Look for and avoid common resource contention cases that can cause stalls in the pipeline;
- Ensure no single draw call from the application is too expensive. For example, consider tiling full-screen passes if they appear to be causing problems with the performance of the compositor. Be sure to test your assumptions on hardware from all of the vendors your customers are likely to be using.

We hope that this introduction to VR performance analysis has given you a solid foundation on which to build a deeper understanding of the VR pipeline. Taking the time to learn the tools and examine collected traces will yield more comfortable and compelling VR experiences.