DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Evaluation of rendering optimizations for virtual reality applications in Vulkan

Paul Preißner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Evaluation of rendering optimizations for virtual reality applications in Vulkan

Evaluation von Renderoptimierungen für Virtual Reality Anwendungen in Vulkan

Author:	Paul Preißner
Supervisor:	Sven Liedtke
Advisor:	Advisor
Submission Date:	February 15 2020

I confirm that this master's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Garching bei München, February 15 2020 Paul Preißner

Acknowledgments

Abstract

Virtual reality (VR) and modern low-level graphics APIs, such as Vulkan, are hot topics in the field of high performance real-time graphics. Especially enterprise VR applications show the need for fast and highly optimized rendering of complex industrial scenes with very high object counts. However, solutions often need to be custom-tailored and the use of middleware is not always an option. Optimizing a Vulkan graphics renderer for high performance virtual reality applications is a significant task. This thesis will research and present a number of suitable optimization approaches. The goals are to integrate them into an existing renderer intended for enterprise usage, benchmark the respective performance impact in detail and evaluate those results. This thesis will include all research and development documentation of the project, an explanation of successes and failures during the project and finally an outlook on how the findings may be used further.

Kurzfassung

Contents

1 Introduction

1.1 Briefly - what's the goal

Real-time rendering is by no means a new research topic. Naturally developers and researchers have devised shortcuts and optimizations in both hardware and software as far back as the first applications in video games, visualization - scientific or otherwise - and simulation. There are methods and algorithms for almost any type of hardware, rendering technique and application scope. One more recently resurged trend is virtual reality (VR). The realization of comparably affordable, comfortable and hassle-free virtual reality headsets like the Oculus Rift series, the HTC Vive, Windows Mixed Reality design or the recently launched Valve Index led to a newfound vigor for the technology among consumers. Similarly, businesses as well as scientists are increasingly looking to VR for various fields of research, marketing opportunities, engineering support and more.
However, virtual reality poses challenges and requirements not commonly seen with traditional real-time applications on monoscopic screens. And while goals such as stable motion tracking and low latency, high performance rendering are as old as early VR HMDs, this recent resurgence has prompted much more active search for possible solutions. Among these requirements is the need for high framerates to give the user a better feeling of visual fluidity and to avoid motion sickness. VR applications need not only render at high framerate, they also need to do so at higher resolutions than common flat screens. As the much lower viewing distance and added lens distortion means pixels appear much larger to the user, the only way to combat visual aliasing and the so called screen door effect is to significantly increase pixel count and density. Similarly to the illustration in [GPUZEN1 article A], let's assume a traditional console video game running at Full HD resolution and 60 frames per second, a common sight for 8th generation consoles. Meanwhile the same game running in a VR headset may require resolutions and framerate ranging from 1200 pixels per axis per eye at 90 frames per second for entry level headsets up or exceeding 2000 pixels per axis per eye at 120 frames per second for high end devices like the Pimax 4K - not to mention image warp commonly requires and even higher internal rendering resolution by a factor of around 20% or more. In the entry level case that means a resolution increase of roughly 1.6x and a framerate increase of 1.5x for a combined 2.4x power requirement compared to the console baseline, while in the high end case it is roughly 4.6x resolution and 4x framerate uptick for a combined 9.2x power requirement in comparison.
While this is very rough napkin math and real applications scale differently to a degree due to a multitude of factors, it paints a clear picture: real-time virtual reality rendering necessitates vastly faster rendering than traditional screens. It obviously follows that VR rendering should take advantage of as much optimization as possible to attain that goal.

There are many established options that are known to improve performance for various if not most application types, with some of these options being de-facto standard features in popular rendering engines. There is also, however, a range of optimizations specific to virtual reality or more generally stereoscopic rendering, and often there is less research and documentation available yet, due in part to aforementioned recent rise in popularity and prior lack of interest.

Thus, this paper aims to collect several of these VR optimizations, implement a subset of them in a real engine and then benchmark and analyse the impact each has on performance. The goal is to have a better collection and understanding of the various methods to use and how much effort may be warranted for each. To this end, what follows is first an explanation of the software and technical foundation used to implement practical samples, then a chapter on optimizations aiming to reduce render *input*, succeeded by a chapter on optimizations aiming to reduce render *effort* and finally an analysis of the performance impact of each implemented approach. Note here that only a subset of the listed methods was implemented due to time constraints.

1.2 Industry collaboration

This thesis is created in collaboration with RTG Echtzeitgraphik GmbH, an engineering and consulting office based in Garching bei München. RTG offers engineering services related to real-time visualization, virtual reality applications and hardware prototyping to enterprise clients from a variety of industry branches such as automotive and industrial logistics.
At this point it needs to be disclaimed that the author of this thesis was employed at RTG Echtzeitgraphik GmbH at the time of making. This allowed the use of company assets like workstations, a range of VR headsets and relevant literature as well as expertise with regard to related technologies and enterprise requirements for a rendering engine. The thesis, accompanied research efforts and all material created for the purpose of this thesis, however, are the author's own work.

1.3 Technical foundation

From this industry collaboration follow influences when choosing the technical foundations of the thesis. In an effort to represent a realistic use case, an engine with actual production usage target was chosen as opposed to a purely synthetic "test vehicle", namely RTG's own Tachyon. The engine and its structure is further explained in [chapter 2 - Tachyon Engine]. The graphics API and library of choice for this thesis then is the Khronos Group's Vulkan. Vulkan is a low level graphics API officially created in 2014 and formally released in early 2016 with the promise of increased flexibility and reduced overhead compared to prior graphics APIs as well as compatibility with a wide range of hardware architectures and operating systems. On the flipside, the verbose and low level nature of the API brings increased development effort and means that these promised improvements can only be leveraged with sufficient care and caution by developers. More specifically for Tachyon and this thesis, the goal is to enable fast rendering of complex scenes with very large object counts, all while driving virtual reality headsets at high resolution and framerate. To this end, Vulkan promises a good fit. Furthermore Vulkan is seeing a steady growth in developer interest and industry adoption, but with documentation, resources and API-specific research still being lackluster in some ways. This thesis hopes to add useful insights to this pool. Lastly - as a sort of tease to the future - both Vulkan and the VR optimizations presented here are compatible with many ARM-based SoCs running Google's Android as well as Apple's iOS, promising prospects for the category of standalone low power VR headsets which are seeing increasing popularity.

2 The RTG Tachyon Engine

2.1 The Engine

Rendering optimizations can only be implemented if there actually is a renderer at hand. In order to see how the chosen approaches would perform in an engine intended for productive use and industrial applications, rather than an ad hoc renderer only built for some specific tests, Tachyon was chosen, an engine currently in development at RTG Echtzeitgraphik GmbH. Tachyon uses a fully Vulkan based forward renderer (internally called rtvklib) with support for multiple viewports of various types, including an OpenVR based virtual reality path, an optional physically-based shader pipeline, a user interface module, a network module and a physics module with more extensions on the development schedule for the future. The renderer integrates Vulkan version 1.1.85 and up and OpenVR version 1.4.18 and up with support for all major SteamVR headsets at the time of writing, including roomscale tracking of the Valve Index, HTC Vive and Vive Pro, the Windows Mixed Reality series and Oculus Rift series.

2.1.1 Render setup

As typical for the verbose nature of Vulkan, render initialization starts with the creation of all necessary basic Vulkan resources such as descriptors, descriptor sets, activation of a minimal set of Vulkan extensions and layers and device enumeration. More specific to rtvklib, multiple Vulkan pipelines are active by default:

- a material pipeline offering support for a Phong and a PBR shader as well as geometry and index buffers of arbitrary size
- a skybox pipeline with a simplified skybox shader
- a point cloud pipeline, primarily to allow rendering of LiDAR scan data

To facilitate rendering into multiple viewports, Tachyon uses the concept of render targets. Each render target can reference an arbitrary subset of pipelines, comes with its own set of Vulkan framebuffers, command buffers and render pass and its own virtual camera. Whenever any 3D object is to be loaded, rtvklib uses several manager classes to keep track of the various resource types needed for an object. There are managers for geometry, materials, textures and instances among other types. When an object is loaded, the former three hold the respective buffers. When an object is to be rendered, it first needs to be instanced, handled by the latter. An instance references the various geometry and materials of the original object again, but also holds data specific to individual objects in the virtual world, such as transforms or bounding geometry.

The renderer initialization also encompasses virtual reality through OpenVR. Given a valid OpenVR environment and HMD is detected, a special render target is created with a Vulkan renderpass for multiple views and the respective resources.

2.1.2 Render loop

After all startup and initialization is done, the engine's render loop executes `VKRenderer`'s `Update` and `RenderFrame` functions back to back. These functions are shown in [Listing 3.1] and [Listing 3.2].

```
void VKRenderer::Update()

    mRenderMutex.lock();
    mGeometryManager->Update();
    mMaterialManager->Update();
    mInstanceManager->Update();
    mTextureManager->Update();
    mLightManager->Update();
    mPointCloud->Update();
    UpdateGlobalParamsBuffer();

    for (auto renderTarget : mRenderTargets)
    {
            renderTarget->Update();
            mInstanceManager->FrustumCulling(renderTarget);
    }

    if (mPipelinesInvalid) UpdateRenderPipelines();
    if (mCommandBuffersInvalid) UpdateCommandBuffers();
    mRenderMutex.unlock();
```

Figure 2.1: Renderer update function

`VKRenderer::Update` first prompts all aforementioned managers to update their databases, buffers and anything else they hold in case they are dirty. Then it prompts each render target to update, which may involve camera transformation updates and buffer synchronization for example. Also for each render target, the loop will then have the instance manager perform a frustum culling pass, which will for this target save a conservative list of draw call information for objects visible by this target's camera viewpoints. If any of these updates and culling passes set a pipeline or command buffer state invalid, these will be rebuilt accordingly.

`VKRenderer::RenderFrame` simply prompts each render target to perform its per-frame

```
void VKRenderer::RenderFrame()
{
        if (!mDevice)
                return;

        mRenderMutex.lock();
        for (auto renderTarget : mRenderTargets)
        {
                renderTarget->RenderFrame();
        }
        mRenderMutex.unlock();
}
```

Figure 2.2: Renderer frame render function

rendering operations, be it regular monoscopic output for a traditional viewport or pose tracking and stereoscopic composition for a virtual reality target.

2.1.3 VR render loop

As seen in [Listing 3.3], the virtual reality render target's `RenderFrame` function is rather straight-forward. As long as the target and compositor are active, it updates the OpenVR device poses and virtual camera transforms. It then renders the stereoscopic views, resolves the multisampling layers into single sample and finally submits both eyes' images to SteamVR, which serves as the chosen OpenVR compatible compositor on Windows systems.

```
void OpenVR::RenderFrame()
{
        if (!mIsActive || !vr::VRCompositor())
        {
                return;
        }

        // update ovr poses and camera transforms
        UpdateHMDMatrixPose();
        UpdateCameras();

        // render stereo
        [...]
        submitInfo.pCommandBuffers = &mCommandBuffers[mCurrentFrame];
        vkQueueSubmit(queue, 1, &submitInfo, mInFlightFences[mCurrentFrame]);

        // blit/resolve array layers
        [...]
        submitInfo.pCommandBuffers = &mResolveCommandBuffer;
        vkQueueSubmit(queue, 1, &submitInfo, VK_NULL_HANDLE);

        // Submit to SteamVR
        [...]
        for (int i = 0; i < 2; i++)
        {
                vulkanData.m_nImage = (uint64_t)mResolveImage[i];
                vr::VRCompositor()->Submit(static_cast<vr::EVREye>(i),
                &texture, &bounds);
        }
}
```

Figure 2.3: Renderer frame render function

3 Stereo Rendering Optimization - Input reduction

When looking at real-time rendering as it is done today - albeit from a strongly simplified perspective - the CPU could be described as a employer and the GPU as an employee. For each frame, the CPU produces certain render tasks and supplies the necessary information such as draw calls, shader parameters, buffers and so forth. The GPU then consumes these tasks and associated items and dos the heavy lifting to produce the required results. Now if one wants to speed up that overall process, there are two major ways. One way is to reduce the amount of data that is put into the pipeline so less data needs to be processed overall, the other way is to increase the efficiency of the processing itself. This first chapter of optimization approaches presents ways of reducing the amount of data or work input.

(Hierarchical) Frustum culling

The following options build on top of the regular frustum culling concept. In this the objects in the scene are checked against a camera frustum whether they are inside or outside of it or intersecting with the surface of the frustum. The checks themselves can be generally optimized in various ways, regardless of whether stereoscopy is desired or not. Often only an object's bounding geometry is checked, collections of objects can be precomputed so larger numbers may be discarded at once. An advanced option of culling is to delegate the calculations into a GPU compute shader so potentially less data needs to be transferred from the CPU per frame and much higher vector/matrix calculation is gained in exchange for slower branching. Some modern renderers also do very granular culling like bitmasked checks of precomputed triangle sets, as seen in Ubisoft's Anvil Next engine used in Assassin's Creed Origins [Source]. Another optional layer of the culling process is to maintain hierarchical container structures for the scene objects so larger numbers can be discarded or included early on. For all those options the goal is the same, to get as result the list or buffer of objects visible by the given camera frustum in the scene.

3.1 Superfrustum Culling

3.1.1 Theory

The basic idea behind Superfrustum culling is to essentially do regular frustum culling despite rendering into two cameras, one per eye. The naive way of extending the frustum concept to a stereoscopic camera is to add a second frustum so there is one per eye, then perform the culling check for both frusta and merge the results. As is easily visible from [Figure X.x] though, the spatial proximity of of these two frusta leads to a large overlap volume, especially as field of view increases with more advanced headsets. One possible strategy to leverage more performance when culling two eyes is a so-called Superfrustum, assuming the frustum is the common six sided trapezoid. Cass Everitt of [Facebook LLC], formerly [Oculus LLC] has suggested this approach at [Oculus Connect 5?] and provided computation sketches at [FB source]. The idea is to combine the left and right eye frusta by taking the respective widest outer FOV tangent - usually the left eye's right side and the right eye's left side - and using these as the new side tangents of the superfrustum. Another way to express these is to take the widest half opening angles of each eye and adding them up to a combined opening angle. Similar is done for the top and bottom tangents, although these will usually be nearly identical for the two eyes. A pitfall of the superfrustum is its necessary depth recession. This is easy to visualize when combining the two frusta by extending aforementioned side tangents backwards until they cross. The meeting point of this step is the new origin of the superfrustum, slightly recessed behind the two separate eyes. [Name Surname?] of Silicon Studios offered a generalized way to compute this recession for un-mirrored eye orientation, while Everitt has extended his sketches by an asymmetry normalization. Both of these are important to consider as virtual reality headsets can have slightly canted and asymmetrical lenses, either by design or by manufacturing tolerance. Ignoring these two corrections may still result in a sufficient superfrustum if computed conservatively, but should be included for fully correct setups.

[Figures by Everitt and Silicon]

3.1.2 Estimated impact

blabla

3.1.3 Implementation specifics

The frustum culling approach in Tachyon is fully CPU-based and utilizes pre-computed hierarchical draw buffers. More specifically, at startup the scene is divided into a coarse grid of `chunk`s where each grid possesses an octree. Also at startup, a thread pool with the current number of hardware threads is created. At asset load time, these octrees are populated with the loaded objects through optimistic size-aware insertion. Each cell of a tree then precomputes a draw buffer containing a combined draw call for all objects associated with that cell. These buffers can be recomputed at any time, but the operation should be avoided at runtime as it incurs costly CPU to GPU transfers. In a culling pass, first all `chunk`s within a certain draw distance radius of the camera are chosen, so there is an additional very primitive distance culling taking place. Then each `chunk` submits a culling call using its tree to the thread pool. Each such call works as follows:

[blablabla, Figure etc]

3.2 Round Robin Culling

blabla

3.3 Conical Frustum Culling

blabla

4 Stereo Rendering Optimization - Effort reduction

4.1 Multiview stereo rendering

4.1.1 Theory

blabla

4.1.2 Estimated impact

blabla

4.1.3 Implementation specifics

blabla

4.2 HMD Stencil Mask

4.2.1 Theory

blabla

4.2.2 Estimated impact

blabla

4.2.3 Implementation specifics

blabla

4.3 Monoscopic Far-Field Rendering

4.3.1 Theory

blabla

4.3.2 Estimated impact

blabla

4.3.3 Implementation specifics

blabla

4.4 MFFR Variant: Depth Shift

blabla

4.5 MFFR Variant: Alternate eye

blabla

4.6 Foveated Rendering

4.6.1 Theory

blabla

4.6.2 Estimated impact

blabla

4.6.3 Implementation specifics

blabla

5 Performance impact

5.1 Explanation - Performance measurements

5.1.1 Framerate, frametimes, framepacing

5.1.2 Resource usage

5.2 Benchmark results

5.2.1 Individual impacts

5.2.2 Combined and partially combined impacts

5.2.3 Interpretation

6 Outlook

bla

List of Figures

List of Tables