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

Real-time rendering is not a new research topic. Naturally, developers and researchers have devised shortcuts and optimizations in hardware and software since the first applications in video games, visualization and simulation. There are methods and algorithms for almost any type of hardware, rendering technique and application scope. One more recently resurged trend is VR. The realization of affordable, comfortable and hassle-free VR 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 and engineering support and more.
However, VR poses challenges and requirements not commonly seen with traditional real-time applications on monoscopic screens. While goals such as stable motion tracking, low latency and high performance rendering are as old as early VR HMDs, this recent resurgence has prompted more active search for possible solutions. Among these requirements is the need for high framerates to give the user a feeling of increased visual fluidity and to avoid motion sickness. VR applications not only need to render at a high framerate, they also need to do so at higher resolutions than common flat screens since the shorter viewing distance and added lens distortion means pixels appear larger to the user. The only way to combat resulting visual aliasing and the so called SDE is to significantly increase pixel count and density.

Similarly to the opening example of [1], let us assume a traditional console video game running at Full HD resolution and 60 frames per second, which is a common 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 (Figure 1.1) at 90 frames per second for entry level headsets and up to or even exceeding 2000 pixels per axis per eye at 120 frames per second for high end devices like the Pimax 4K. Image warp will commonly require an even higher internal rendering resolution by a factor of around 20% or more. For the entry level headset, this 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 a 4.6x resolution and 4x framerate uptick for a combined 9.2x power requirement in comparison.

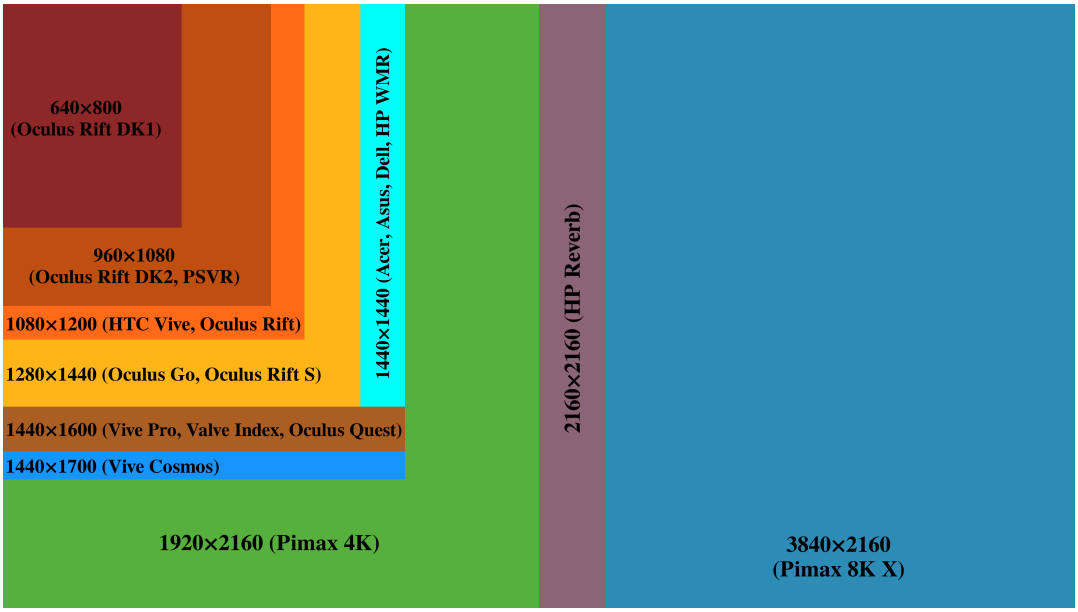


Figure 1.1: Comparison of common HMD resolutions (Oct 2019)[2]

1.1 The goal

While this is very rough napkin math and real applications scale differently due to a multitude of factors, it paints a clear picture: real-time VR rendering necessitates vastly faster rendering than traditional screens. Consequentially, VR rendering should take advantage of as much optimization as possible to attain that goal. There are many established options known to improve performance for various application types, with some of these options being de-facto standard features in popular rendering engines. However, there are a range of optimizations specific to VR or more generally stereoscopic rendering. There is less research and documentation available as of now, due in part to the aforementioned recent rise in popularity and prior lack of interest.
Thus, this paper aims to collect several VR optimizations, implement a subset of them in a real engine and subsequently benchmark and analyse the impact each has on performance. The goal is to have a better collection and understanding of the various possible methods and how much effort may be warranted for each. Firstly an explanation of the software and technical foundation used to implement practical samples is given, following this, optimizations aiming to reduce render *input* will be shown, succeeded by a chapter on optimizations aiming to reduce render *effort*. Finally, an analysis of the performance impact of each implemented approach. For the practical aspect of this thesis, the goal is to enable fast rendering of complex scenes with very large object counts, all while driving VR headsets at high resolution and framerate.

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 Echtzeitgraphik GmbH offers engineering services related to real-time visualization, VR applications and hardware prototyping to enterprise clients from a variety of industry branches such as automotive and industrial logistics[3].
The author of this thesis was employed at RTG Echtzeitgraphik GmbH at the time of writing. 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

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 Echtzeitgraphik GmbH's own Tachyon. The engine and its structure is further explained in chapter 2.
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. It promises increased flexibility and reduced overhead compared to prior graphics APIs as well as compatibility with a wide range of hardware architectures and operating systems[4]. 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.
For the practical goal of this thesis 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. This thesis hopes to add useful insights to this pool.
Lastly both Vulkan and the VR optimizations presented here are not just compatible with powerful x86 desktop machines but also 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[5][6].

2 The RTG Echtzeitgraphik GmbH Tachyon 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. Tachyon uses a fully Vulkan based forward renderer (internally rtvklib) with support for multiple viewports of various types, including an OpenVR-based VR 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. 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 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 and 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 VR 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. Resources include a framebuffer with one layer for each eye, created with the rendering resolution returned by OpenVR after querying `IVRSystem::GetRecommendedRenderTargetSize()` and using 4x multisampling as is often recommended for VR applications to minimize aliasing without obscene cost (see for example [7] p. 25f, [8], [9], [10], [11]).

All run-time relevant vector and matrix math in the renderer and the presented optimization implementations use the free MPL2 licenses Eigen 3.3 math library. Eigen was chosen for its extensive feature set and its special focus on high performance using SIMD vectorization including all current forms of SSE and AVX among others[12].

2.2 Render loop

After all startup and initialization is done, the engine's render loop executes `VKRenderer`'s `Update()` and `RenderFrame()` functions back to back. The flows of these are shown in Figure 2.1 and Figure 2.2, respectively.

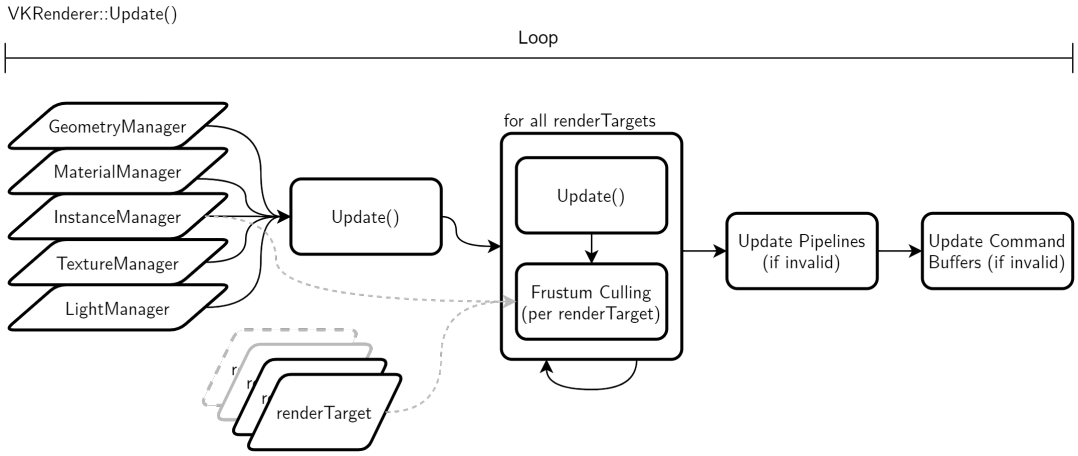


Figure 2.1: Renderer update function [TODO: autoref code snippet in appendix]

`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. 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()` prompts each render target to perform its per-frame rendering operations, be it regular monoscopic output for a traditional viewport or pose tracking and stereoscopic composition for a VR target.

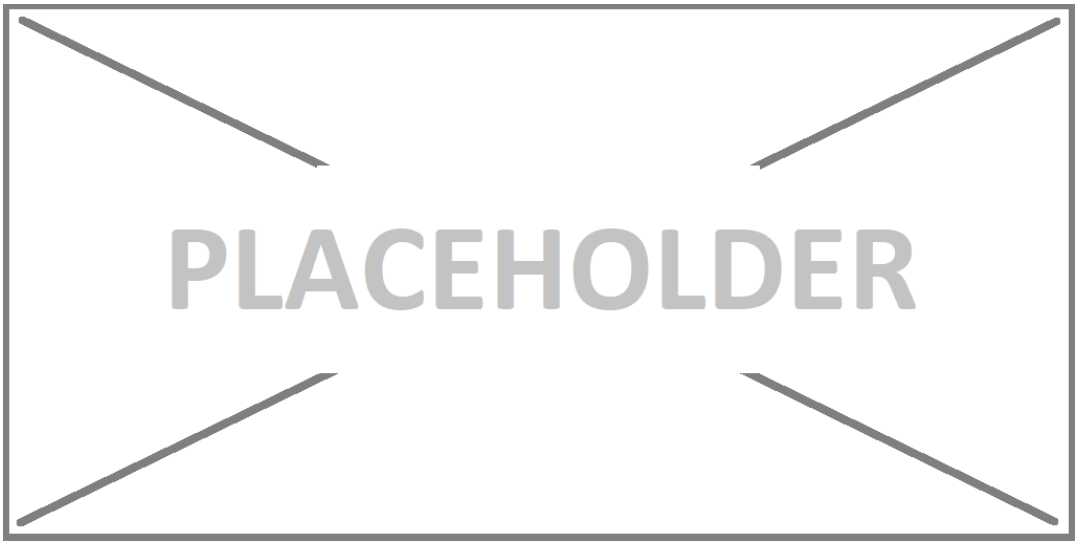


Figure 2.2: Renderer frame render function[TODO: autoref code snippet in appendix]

2.2.1 VR render loop

The VR 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. Note here, rtvklib's internal VR render target class is also called `OpenVR` unfortunately but is not equivalent to the external OpenVR library. Which of the two is meant in a given sentence is indicated by the formatting as showcased here.

As the VR view essentially moves constantly, its render target needs to update its Vulkan `VkCommandBuffer` every frame. This is done by setting the `mCommandBuffersInvalid` flag which has the renderer call `UpdateCommandBuffers()` which in turn prompts each render target to `RecordCommandBuffers()`. For each of the VR render target's command buffers this is a simple loop through the `RecordDrawCommand()`s of the pipelines assigned to this render target between the Begin and End commands of the command buffer and render pass. The draw command recording function has the respective pipeline query the `InstanceManager` for its post-culling draw command set and writes the contained entries as `vkCmdDrawIndexedIndirect()` draw calls.

Moreover, as seen in Figure 2.4, the VR render target's `OpenVR::RenderFrame()` render function first updates the API-supplied HMD pose matrices and consequently recalculates its virtual camera parameters. It then submits any given `VkCommandBuffer`s assigned to itself, followed by a submission of the multisample resolve command buffer. At the end of the loop iteration it gathers the resulting resolved image textures for each eye and submits them to the VR compositor for presentation inside the headset.

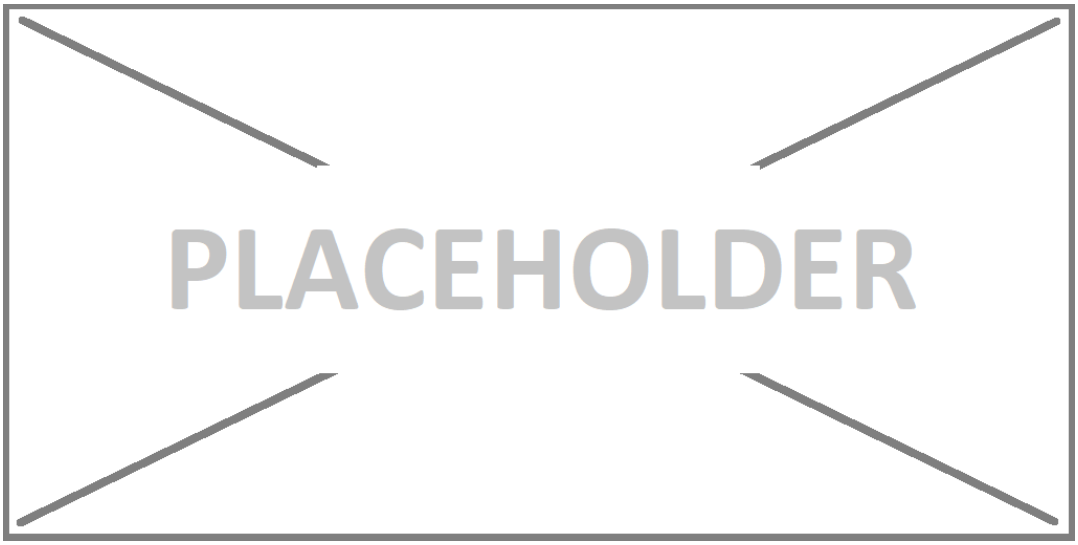


Figure 2.3: VR command buffer recording function[TODO: autoref code snippet in appendix]

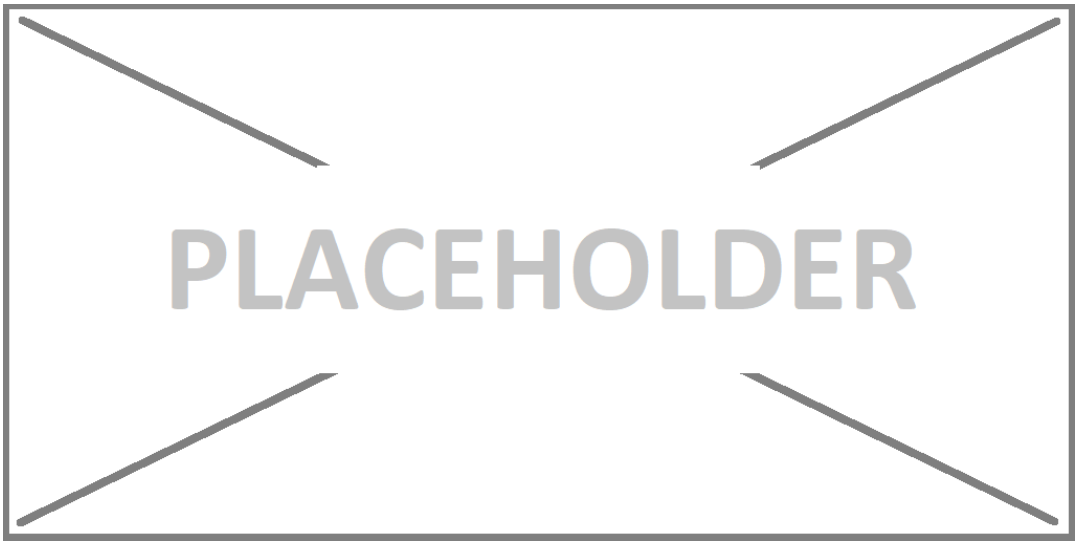


Figure 2.4: VR frame render function[TODO: autoref code snippet in appendix]

[TODO: overview illustration aka flowchart]

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 an 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 does the computational brunt to produce the required results. If one wants to speed up this process, there are two major ways. One is to reduce the amount of data 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. Note here that only a subset of the listed methods was implemented due to time constraints.

3.1 (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 or intersecting with the surface of the frustum. The checks themselves can be generally optimized in various ways, regardless of stereoscopy. Often, only an object's bounding geometry is checked. Collections of objects can be pre-computed 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*[13]. 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 obtain the list or buffer of objects visible by the given camera frustum in the scene.

3.2 Frustum (& distance) culling in Tachyon

The frustum culling approach in Tachyon was developed specifically for this thesis as a requirement for Superfrustum culling (section 3.3) and as a sensible general optimization, is fully CPU-based and utilizes pre-computed hierarchical draw buffers. More specifically, at

startup the scene is divided into a coarse grid of chunks where each such chunk possesses an octree. Also at startup, a thread pool with the double the number of detected 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 pre-computes 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 chunks 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:

```
pseudo SceneChunk::FrustumCull()
{
        out draw-calls[camfrusta.size];
        for all leftover octree cells
        {
                for all camera frusta
                {
                        if cell is valid
                        var checkResult = frustumCheck(frustum, cell);
                        if checkResult == INSIDE
                        {
                                add overall cell draw-call to draw-call list;
                        } else if checkResult == INTERSECT
                        {
                                add 8 child cells to leftover list;
                                add local cell draw-call to draw-call list;
                        }
                }
        }

        for all camera frusta
        {
                unique_lock(cullMutex);
                sort and aggregate draw-call list;
                add draw-call list to per-pipeline draw-call collection;
        }
}
```

Figure 3.1: per scene chunk frustum culling procedure (shortened pseudo code)

In essence, the octree is looped through in a hierarchical fashion until either all remaining

subcells of the current hierarchy level are outside of the frustum or no more subcells are left to check. Any resulting draw-calls are sorted to be memory layout friendly and aggregated into as few calls as possible to reduce invocation cost.

However, one problematic area remains and that is Z ordering. Modern graphics pipelines will perform early Z discard during the fragment stage. If a fragment fails the depth test for a given draw call, meaning its geometry would be occluded by triangles already written to the fragment, the shader will skip any further calculation for this geometry and fragment. For this to take hold in performance improvement, the draw calls need to be issued in a manner where geometry closer to the camera is drawn first. If draws are done in reverse order, the pipeline will draw distant geometry first and subsequent closer geometry at the same fragment will naturally not fail the depth test and overwrite the fragment. Effectively, all prior writes to any such fragment in that frame would be wasted and go unseen. This issue is called overdraw and can significantly deteriorate GPU render times in extreme cases. One such example would be the following, very dense synthetic test scene:

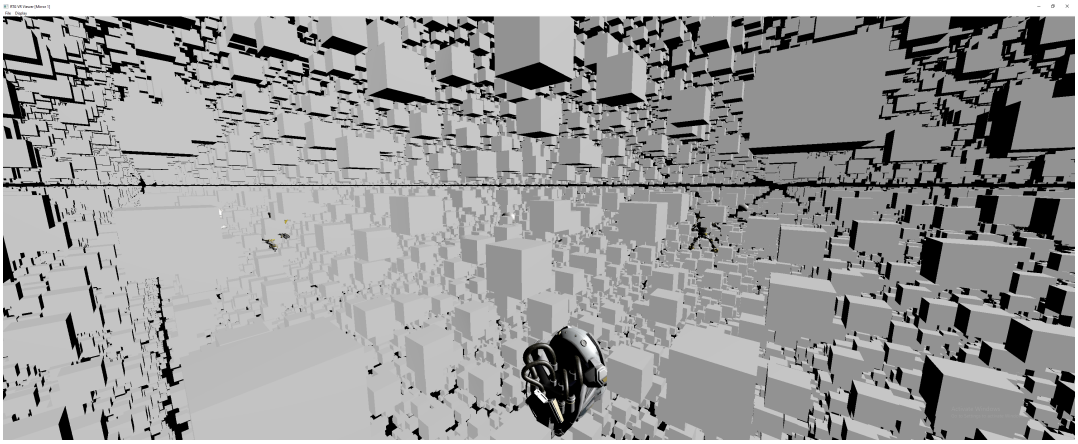


Figure 3.2: Sample scene with high object density, far draw distance and high degree of overdraw when rendered without per-frame Z ordering, screenshot taken of Tachyon's desktop viewport

If the draw calls for each populated octree cell are issued back to front, the overdraw of many dozens if not hundreds of layers can push frametimes upward to the point where the GPU is geometry-bottlenecked in the depth test, while issuing the calls front to back may result in sufficiently low frametimes. At this time, rtvklib does not employ such call reordering, meaning due to the nature of the octree cells' pre-recorded command buffers, overdraw cannot be avoided without a significant rewrite of these pre-recordings.

3.3 Superfrustum Culling

The basic idea behind so-called *Superfrustum Culling* is to do regular single 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 [TODO: illustration of stereo frusta], 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 the Superfrustum, assuming the frustum is the common six sided trapezoid. Cass Everitt of Facebook Technologies LLC, formerly Oculus LLC, has suggested this approach and provided computation sketches on his social media back in 2015 [14](Figure 3.3), and Nick Whiting at Oculus Connect 4 teased it as a future addition to Unreal Engine 4 [15]. 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. Vivien Oddou of Silicon Studios offered a generalized way to compute this recession for non-mirrored eye orientation [16](Figure 3.4), while Everitt has extended his sketches by an asymmetry normalization[17]. 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. While this superfrustum naturally eliminates all overlap of the naive variant, it in turn includes small false positive regions, notably the triangular void found close to the origin points between the two eye frusta and potential side edge regions in the case of asymmetrical lens orientations. In a typical application, the performance cost of these is negligible.

3.3.1 Estimated impact

The impact of using a Superfrustum will depend on the type of frustum culling math done and the combination with other techniques.
On its own, with a CPU based culling pass, only an appreciable benefit in CPU rendering time is to be expected, as the number of frustum checks will be reduced by up to 50% and only a single buffer needs to be transferred to the graphics unit. The GPU itself still needs to render each eye separately, including all vertex transformations, pixel shading and so forth. The specific impact in the case of Tachyon is elaborated on in chapter 6.

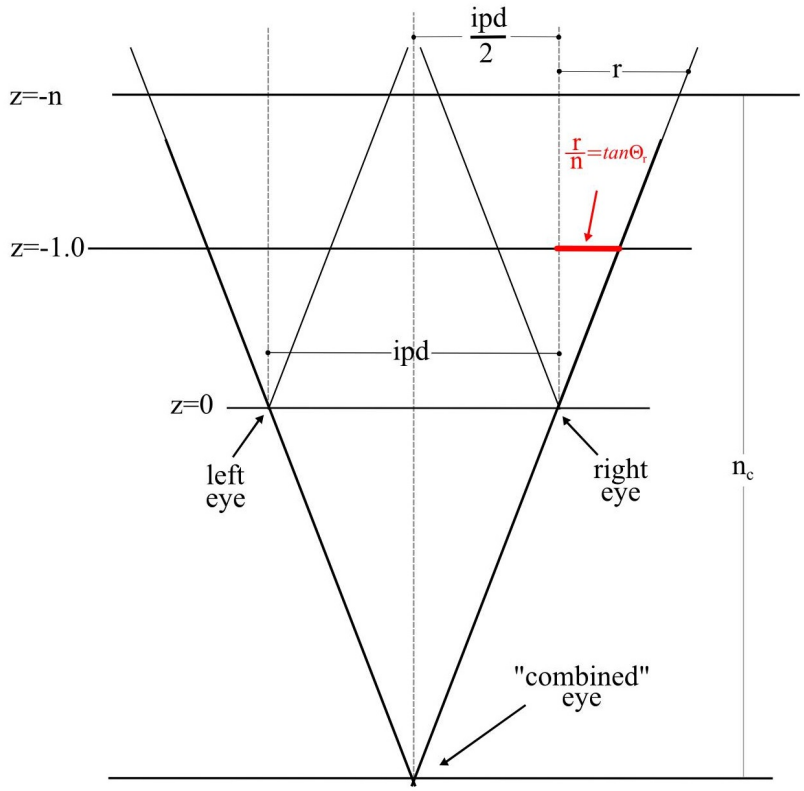


Figure 3.3: Symmetric Superfrustum (cropped to geometric construction)[14] [TODO: footnote with source/courtesy]

Superfrustum culling when performed directly on the GPU obviously has great potential to significantly cut down on related compute work, once again to the effect of up to 50% versus a baseline dual frustum culling. An interesting point to consider is whether any of the culling data needs to be synchronized back to the CPU, as, when not, the GPU compute workload will only depend on a single small buffer or pushconstant transfer containing the camera parameters. If, however, the resulting culling set is transferred back to the CPU, for example for preprocessing of the next frame, this transfer will present another speedup limiter.

3.3.2 Implementation specifics

Facilitating Superfrustum Culling in Tachyon required only straight-forward changes. At creation time of the virtual camera, the superfrustum is computed from the given OpenVR eye parameters following Everitt and Oddou's way. Assuming the two eye projections are asymmetric, they need to be symmetricized. For this, after grabbing each eye's projection matrix from OpenVR for the desired near and far clip distances, the M[0][0] and M[0][2] values of each matrix are of interest. These two values represent the OpenGL clipping space's X coordinate's scalars. OpenVR through SteamVR uses the same coordinate layout. At a

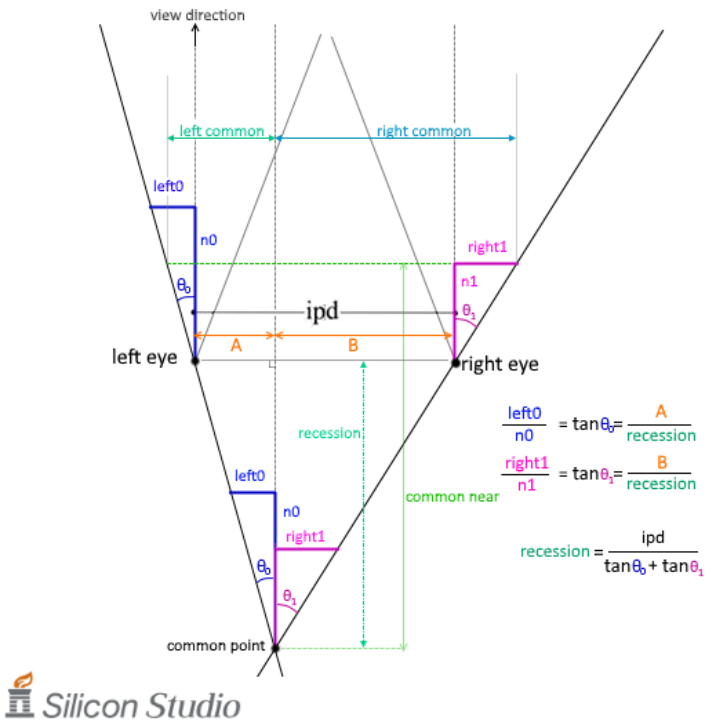


Figure 3.4: Non-mirrored superfrustum recession[16] [TODO: footnote with source/courtesy]

Z depth of -1, these two scalar terms are entirely dependent on center point and width of the projection cube. Solving them for center and width then gives $center = \frac{M[0][2]}{M[0][0]}$ and $width = \frac{2}{M[0][0]}$. A new symmetric projection then obviously could be constructed from this center and width by taking the center point and stepping "sideways" by half width. The new *right*, or *r* value, for example, will be $r_{sym} = abs(center) + \frac{width}{2}$, which if substituted by the matrix dependent fractions comes out to $abs(\frac{M[0][2]}{M[0][0]}) + \frac{1}{M[0][0]}$. This can be solved for a new $M_{sym}[0][0] = \frac{M[0][0]}{abs(M[0][2])+1}$ and $M_{sym}[0][2] = 0$ (see [17]). In the next step, these new $M_{sym}[0][0]$ values for both eyes are inserted into the recession math by Oddou[16]. For non-mirrored and asymmetric eyes, a superfrustum recession is simply calculated as $\frac{ipd}{tan\theta_0+tan\theta_1}$ with θ_i being the respective center-to-outside opening angle of each eye. Conveniently with the previously calculated M_{sym} values, these tangents of angles are equal to the reciprocal of the respective $M_{sym}[0][0]$. As such for the recession we get $\frac{ipd}{\frac{1}{M_{sym}[0][0]_l}+\frac{1}{M_{sym}[0][0]_r}}$. The calculated superfrustum recession and new combined field of view angles are saved and used every frame when re-transforming the superfrustum. The frustum transformation uses a simple geometric approach where the camera's world position, forward and up vectors in conjunction with the near and far distance, field of view and aspect ratio are extruded into the six planes of the frustum volume. The per-frame culling pass of Tachyon then naturally only checks against this single frustum and returns a single set of draw commands which are sent to both eyes.

3.4 Round Robin Culling

Another culling variant specific to stereoscopy uses the round robin principle. Again, the concept springs from the desire to avoid frusta overlap but instead of combining the frusta, it exploits a common property of current stereoscopy rendering techniques. As modern headsets use circular lens optics and flat displays distorting the displayed image, the framebuffers commonly need to be warped to compensate so the picture looks undistorted to the user. As a result, a lot of the edge data of the image is either considerably pushed together or outside of the visible area of the HMD displays. This conservative property means a virtual eye frustum can be smaller than the technical frustum of that respective eye and false negative discards in these edge regions would go unseen by the user. Assuming both eyes of the headset have similar opening angles and parallel or nearly parallel viewing direction, the overlap of the two frusta would encompass the entire stereo-visible volume. It follows that only culling for one of the eye frusta would already give a sufficient representation of the actually visible scene.

There is still a possibility of missing a few edge cases with this alone. So the extension of the idea to actual round robin assumes another common property of modern VR headsets, namely high refresh rates. Many of these aim for at least 80Hz (Rift S, WMR) ranging up to 144Hz (Index experimental mode) image refresh to give the user a smooth visual sensation. Exactly halving that refresh rate and reprojecting images for two refresh cycles with some pixel interpolation is an established way to still provide an acceptable experience with minor visual artifacting on slower devices as demonstrated by Oculus LLC's Asynchronous Space Warp[18] and SteamVR's reprojection[19] features. Subsequently, a conceivable compromise is to alternate which frustum is used for culling in a round robin fashion so that even if edge cases include visible false negatives, they only persist for one frame at a time. In the worst case this would manifest as shimmering or flickering at the outer edges of the visible screen area.

Overall this makes Round Robin Culling a viable candidate on systems with very limited culling performance but the tight constraints for sufficiently accurate results make it unfit as a general recommendation.

3.5 Conical Frustum Culling

This third alternative culling extension targets the circular shape of HMD lenses for leverage. Coming back to the conservative framebuffer size from the previous section, the lenses lead to a lot of invisible area in the corners of the display. Jonathan Hale attempts to demonstrate the method in his thesis [20] as both a contribution to the graphics middleware *Magnum*[21] and a UE4 extension at Vhite Rabbit[22] - albeit with limited success. However, his proof of concept shows the validity of the method. The traditional six sided trapezoid frustum is replaced by a cone encompassing a volumetric projection of the view through each respective lens as visualized in Figure 3.6.

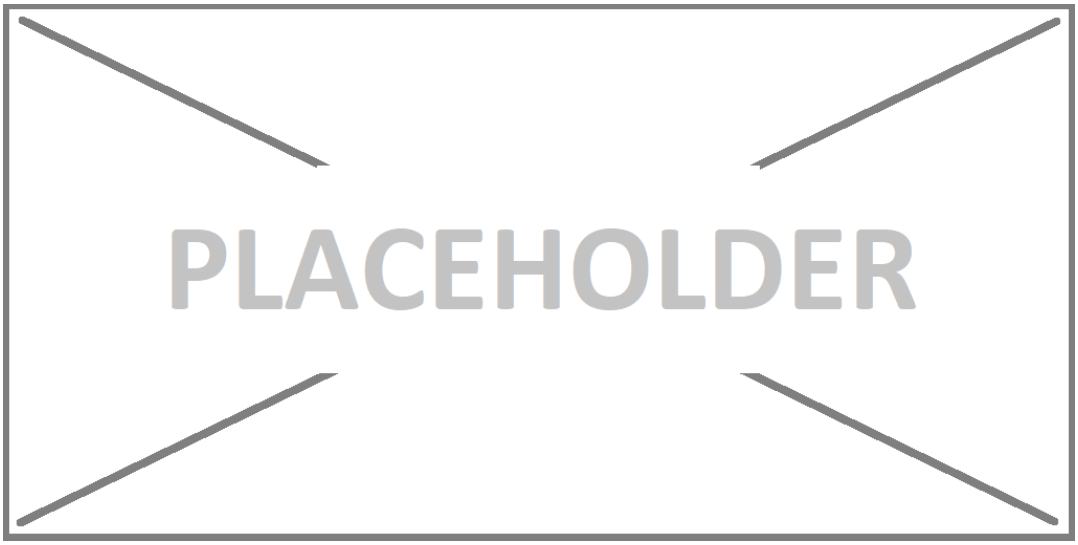


Figure 3.5: [TODO: small illustration?]

Hale examined various types of cone intersection math, including AABB and bounding sphere object checks against a spatially transformed frustum cone and the same checks for spatially inverse-transformed objects against an origin oriented frustum cone. While his results showed cone culling performing worse than traditional frustum culling, he notes optimization was not fully refined and all calculation was done on the CPU. Another note by Hale cautions that depending on the used HMD, a cone frustum may prove *less* accurate than a trapezoid, such as for the Oculus Rift CV1. As some headsets show not a fully circular image through their lenses but rather the entire distorted frame, it may not be possible to fit a cone frustum within the traditional frustum and instead that cone may actually exceed the traditional dimensions and thus cull fewer objects. For appropriate headsets and more geometry-bound GPUs, this method may provide a small relief if using even faster cone intersection math.

3.6 Merging approaches

A convenient side effect of these three presented optimizations is that they can in part be merged. For example, it is possible to do Conical Round Robin Frustum Culling in an effort to slice away as much of the conservative invisible area as possible and reduce the list of drawable objects to an optimistic minimum. It is also possible to construct a Conical Superfrustum aiming to avoid the mentioned edge false positives, albeit only feasible if the display per eye is square-like to avoid adding new false positive volume on other sides.

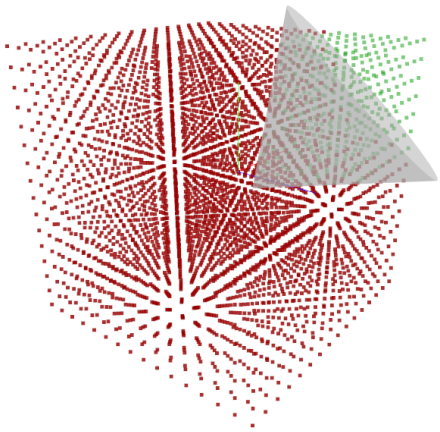


Figure 3.6: Point-cone intersection illustration by Hale ([20], p. 21)

4 Stereo Rendering Optimization - Effort reduction

The following chapter of optimization approaches targets the efficiency of rendering processes on the graphics chip itself. These approaches have little to no impact on CPU performance and tend to exploit and scale mostly with GPU power. Note again that only a subset of the listed methods was implemented due to time constraints.

4.1 Multiview stereo rendering

When rendering a stereo image using the naive method of simply going through the entire rendering pipeline once for each viewport, potentially a lot of computation is done twice with little or no change in data or parameters. With the general graphics pipeline (Figure 4.1) in mind, it is clear that for example the vertex stage will see very little change in output as geometry and index buffers are largely the same between multiple stereo viewport passes with only minor shifts in the view matrices. Similarly, the geometry stage is commonly not dependent on specific eye data and as such it would be a waste to process with the same data twice. Once the rasterizer stage of the pipeline is reached, the situation changes as stereo separation means the two images have notably different content and work from one can not realistically be recycled in the other.

An optimization exploiting this is called multiview stereo rendering. It very quickly surfaced as an idea after the introduction of the Nvidia Geforce 8 and ATi Radeon HD 2000 series in 2006 and 2007 brought unified shader architectures to the mass market [24][25]. Prior architectures relied on separate vertex and pixel shader units with relatively fixed capabilities and few ways to share data. Fully programmable shader units then allowed more customizable and efficient pipeline usage necessary for multiview to show any benefit. The lack of mainstream stereoscopic systems prohibited the feature from becoming more important until the official introduction of multiview extensions to graphics APIs like OpenGL (`GL_OVR_multiview`) and Vulkan (`VK_KHR_Multiview`, previously `_KHX` and `_NV`). The same resurgence saw the idea expanded and further optimized. In more recent vendor specific terms, Nvidia calls it Single Pass Stereo[26], Simultaneous Multi-Projection[27] and Multi-View[28] rendering and AMD calls it LiquidVR multiview[29][30]. The idea behind all these terms is the same, albeit with detail differences between the different flavors.

The core concept of multiview rendering is to submit all draw commands for a stereoscopic frame in one call instead of two separate passes, which can cut down CPU render and transfer time depending on the type and amount of data pushed to the GPU. Expensive

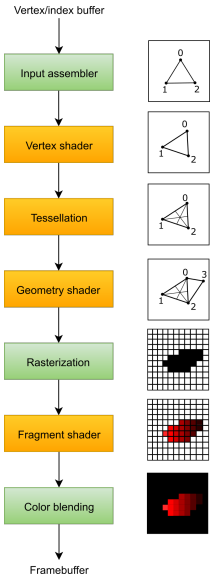
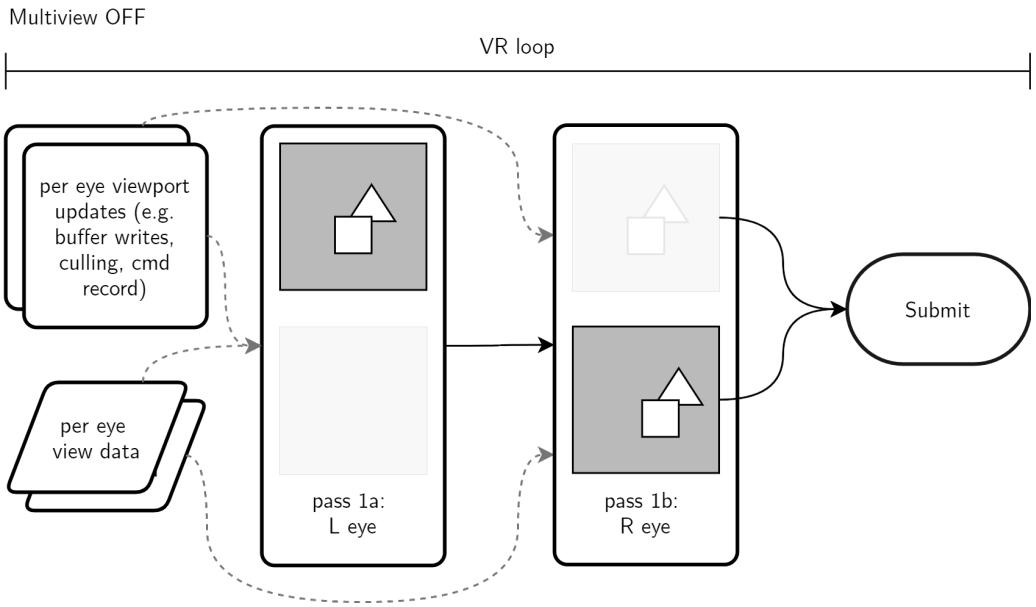


Figure 4.1: Simplified graphics pipeline of a modern GPU[23]

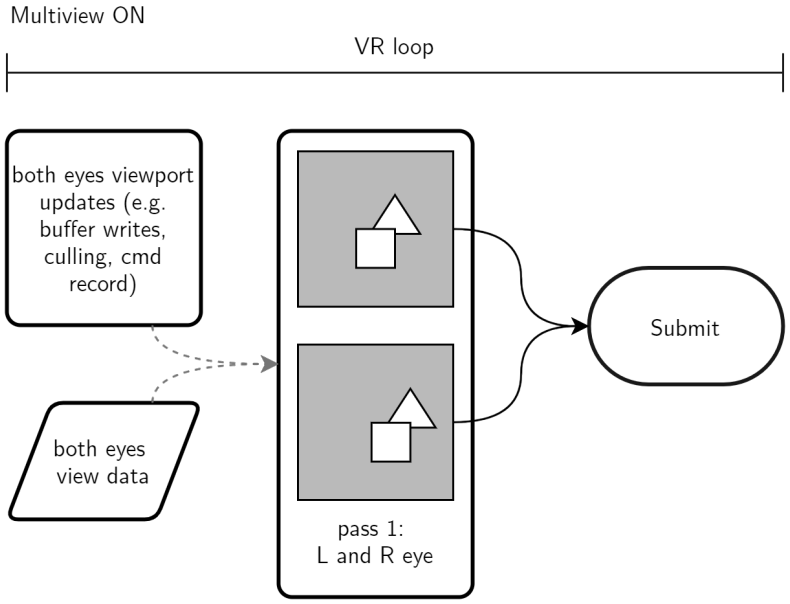
synchronization barriers are essentially halved and all necessary writes are performed in a single go. This is illustrated - albeit strongly simplified - in Figure 4.2, which makes it easily visible where multiview cuts out unnecessary work. As an addition, *hardware* multiview rendering is to only perform those pipeline stages multiple times which actually produce notably different data for each eye, such as the rasterizer and pixel shader stages, while only running the earlier stages with little changes once. The data from stages run only once can then be reused by the multiply run stages with very little extra cost. This expanded technique improves pipeline efficiency and will scale heavily depending on workload. For fragment-heavy applications the benefit will be limited while high vertex or geometry loads tend to scale more optimally. Hardware acceleration requires additional registers and pipeline shortcuts in the chip itself, which constrains it to more modern GPU architectures built with it already in mind. Nvidia can be considered the main drivers behind this, having pushed the technology from parallel geometry projection in Maxwell's Multi-Projection Acceleration to Pascal's SMP adding lens-matched shading to better approximate the lens shape. Finally, to Turing's Multi-View with a doubling of available views and positional independence to support state of the art HMDs with canted displays [28].
The small tradeoff is all relevant view data for each viewport having to be handed to the pipeline at once, creating a little higher memory overhead. Additionally, certain buffers such as geometry and indices for the vertex stage need to be uniform across all viewports and can not be altered for each eye as they are processed in a single pass.

Going by the user-maintained Vulkan Hardware Database [31], the major GPU vendors offer multiview support in their architectures as follows:

- Nvidia: hardware multiview from the Pascal generation and newer, software support from Kepler onward



(a) Simple stereo rendering loop flow



(b) Multiview stereo rendering loop flow

Figure 4.2: Simplified flow of standard stereo versus multiview render loops

- AMD: software support from Graphics Core Next 1.0 onward [TODO: hw support?]
- Intel: software support begins with Generation 9.0 (Skylake/Apollo Lake GT) onward under Windows, Generation 7.0 (Ivy Bridge GT) onward under Linux [TODO: hw support?]

- Qualcomm: software support from Adreno 500 onward
- ARM: software support from Bifrost onward, limited support on Midgard
- Imagination: software support from Rogue onward

Note here, while support of the desktop parts is solid and stable, the ARM-based mobile chipsets often have incomplete or unstable drivers [32][33].
For all submitted devices (892 at the time of writing) the database shows support coverage of 54% on Windows systems, 69% on Linux systems and only 23% on Android. This statistic is not very reliable, since an undetermined portion of the submissions contains incomplete or flawed information such as drivers versioned as 0.0.0 or API versions reported as 0.0.1.

4.1.1 Estimated impact

Impact of this extension is highly dependent on the specific workload, the used graphics hardware and the renderer structure.
Independent benchmark numbers are rare to find. In benchmarking the online racing simulation iRacing, Wolfgang Andermahr of ComputerBase GmbH attests a performance increase of between 14-21% when testing on a Nvidia GTX 1060 and GTX 1080 at 5760 by 1080 pixels. While this is not a VR application, it gives a first impression of potential gains when rendering multiple viewports using hardware-accelerated multiview as is the case on these Pascal-based GPUs [34]. Similarly, Croteam's Karlo Jez shows significant CPU performance gains for software multiview using AMD's LiquidVR suite, reducing CPU frame time in Serious Sam VR from 9ms down to 7ms [30]. He specifically notes that this includes all stages of CPU rendering, even ones that are not connected to multiview, and adds that render command processing itself is halved from 4ms to 2ms by using software multiview. Going further from traditional desktop rendering, Mozilla Corp. reports performance improvement of up to 40% when using the multiview extension for WebGL in a CPU-bound test application [35].

Overall this gives inconsistent performance expectations, but it does provide a trend. It stands to reason that with rising CPU load and increasing number of virtual viewports, multiview rendering offers significant performance improvement in the double digit range.

4.1.2 Implementation specifics

In Vulkan specifically, multiview is enabled through the `VK_KHR_multiview` extension. This extension's availability on the target hardware can be queried and if available, the individual hardware-dependent implementation is abstracted by Vulkan.
In the Tachyon implementation of multiview, the following changes to the render loop are introduced:

- The VR render target adds `VK_KHR_GET_PHYSICAL_DEVICE_PROPERTIES_2` to the required instance extensions and `VK_KHR_MULTIVIEW` to the required device extensions during Vulkan instance and device creation at startup

- the previously separate per-eye VR render passes are merged into a single render pass
- this VR render pass incorporates the multiview pNext extension using a view mask and correlation mask of 0x11, with each of those bits representing one of the eyes
- the Frustum Culling pass combines the two frustum checks with an early accept and outputs a merged set of draw call information
- the second command buffer recording - previously intended for the second eye - in `OpenVR::RecordCommandBuffers()` is cut out as multiview render passes can only take a single unified set of command buffers
- the underlying GLSL shader is modified to pick camera parameters based not only on a given camera index, but also the implicit `gl_viewIndex` as Vulkan multiview uses this integer to index the current viewport

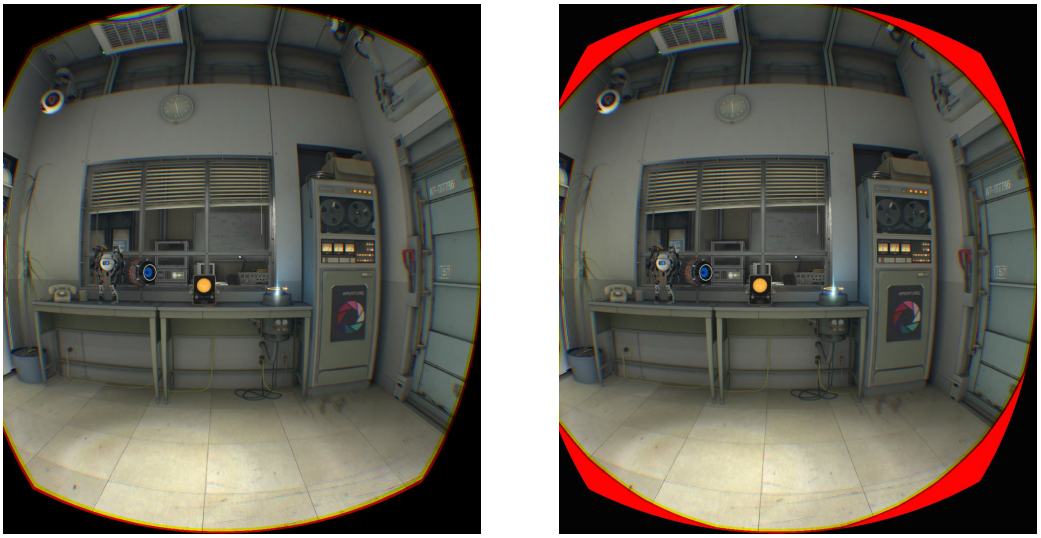
The VR render target's framebuffer, color attachment and depth attachment are already set up as dual-layered buffers which makes them readily compatible with multiview render passes. With these changes, the Tachyon renderer is fully switched over to single pass stereo rendering.

4.2 HMD Stencil Mask

Introduced to the mass market with 3Dlabs' Permedia II in 1997 and widely adopted since then, all modern graphics chips support a useful feature called the stencil buffer. This buffer uses low bit integer values - commonly 8 bits per pixel for a depth of 256 [36] - which can be read from and written to during the fragment stage, with stencil testing happening after alpha and before depth testing. Sometimes used for certain shadowing operations, the stencil buffer is primarily used for cheap masking efforts.
One such effort was presented by Alex Vlachos at GDC 2015 [7] as a possibility to improve performance in VR applications. Once again pointing at the significant areas of invisible screen space wasted outside the HMD lenses' warping reach (such as in Figure 4.3), the idea here is to write into a per-eye pixel matched stencil buffer a mask corresponding to the shape of the visible screen area. During the fragment stage of a frame render, the stencil test can early discard all masked fragments and thus avoid pixel shader work for all these areas. The operation works exactly as the classic idea of a stencil mask when painting surfaces. Paint will only hit the surface within the cutouts of the stencil. Similarly the graphics chip will only write color and depth values to unmasked fragments.

4.2.1 Estimated impact

The performance gain naturally scales well with both increased fragment shader bias of the per-frame workload as well as with the HMD's blind areas. In his GDC talk, Valve's Alex Vlachos, showcased gains of 17% lower GPU fill rate for the company's Aperture Labs



(a) HTC Vive full render after warp

(b) HTC Vive pixels wasted by lens cutout marked in red

Figure 4.3: Comparison of rendered rectangular right eye frame after warp versus pixels wasted by lens distortion ([7], pp. 52-54)

VR showcase scene using an HTC Vive headset ([7], p. 59). Assuming a roughly uniform distribution of objects in the scene and, accordingly, a roughly constant shader workload during use, the relative improvement in fragment render time is directly proportional to the masked percentage of the total framebuffer.

4.2.2 Implementation specifics

The rtvklib stencil masking feature is designed to query the require fitted meshes from OpenVR and render them to the respective eye at startup so the render loop can keep reusing it every frame without additional render load for the mask (see Figure 4.5). An example of the rendered mask is shown in Figure 4.4, displaying the Nsight VS stencil buffer capture when using a Valve Index headset.
Outfitting Tachyon for stencil masking required the addition of the entire stencil stack as the engine does not use the feature in any other capacity. The changes include extending the depth attachment of the OpenVR render pass with the `VK_IMAGE_ASPECT_STENCIL_BIT` so the additional buffer layer is created at startup. The involved Vulkan pipelines need stencil operation states defined in their `VkPipelineDepthStencilStateCreateInfo`, with the operations set as in Figure 4.6 so the bits are checked against but not modified. Next, the VR render pass needs its color attachment's `stencilLoadOp` set to `VK_ATTACHMENT_LOAD_OP_LOAD` so the renderer knows to load the stencil buffer at the start of the color pass, and the `stencilStoreOp` to `VK_ATTACHMENT_STORE_OP_DONT_CARE` so it can leave the buffer behind after use without saving anything more to it. This is important to make sure the stencil buffer

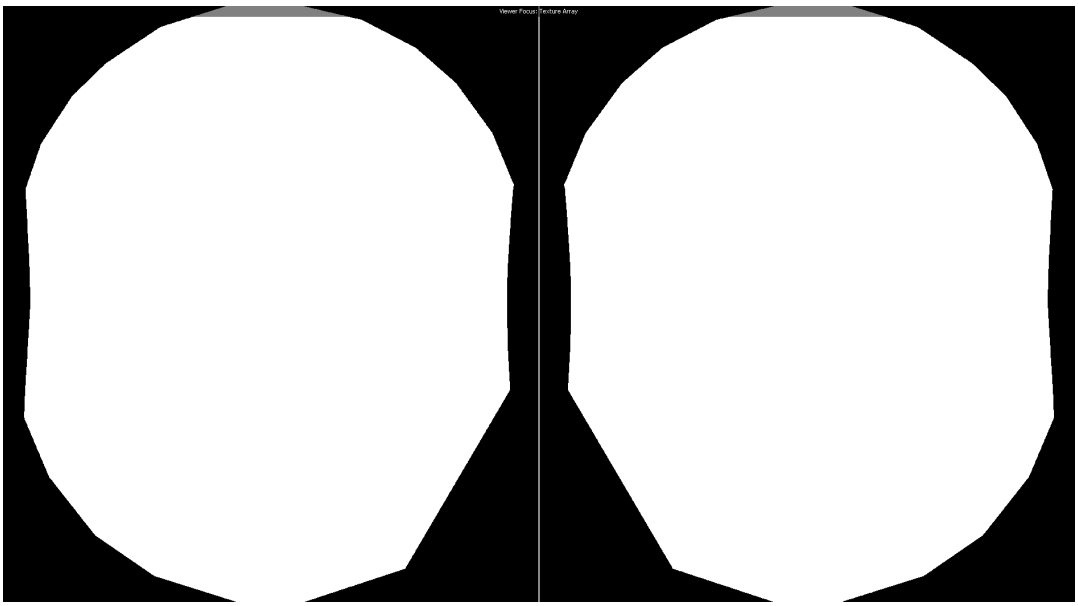


Figure 4.4: Nsight VS capture of stencil buffer as rendered for Valve Index HMD

remains unmodified and expensive writes are avoided.

Rendering the stencil mask itself at startup is done as follows: OpenVR by now has some helper functions for masking built in, such as the `GetHiddenAreaMesh(EVREye eEye)` function that returns a screenspace-normalized list of vertices representing the ideal mask mesh for the current HMD if available to OpenVR. Exceptions apply, as the API does not contain mask definitions for all headsets. As an example, the mesh query returns an empty list for the Oculus Rift CV1 as seen in Figure 4.7. With this in mind, the main benefit is the simplicity of getting a fitted mask for most HMDs instead of either approximating with circular masks or going through the trouble of manually creating fitted meshes for existing and upcoming OpenVR-enabled headsets. For exception cases a fast circular approximation mask may be an adequate compromise.
Tachyon queries OpenVR for the mesh of each eye, converts the vertex lists into a renderer-compatible vertex format and writes out a vertex and index buffer each. At the end of the VR render pass, an ad-hoc command buffer is recorded and submitted to render the two masks into the VR framebuffer's depth attachment's stencil layer. The `VK_IMAGE_ASPECT_STENCIL_BIT` is set to ensure only that layer is written to.

Due to Vulkan's verbose nature, rendering these two masks also requires its own pipeline. The default material and PBR pipelines only do stencil test compares, no writes and they perform color writes and rasterizer face culling, which are all things the stencil mask pipeline should do differently. A separate stencil pipeline is introduced with the stencil operation state set like in Figure 4.8, the color writes masked off and rasterizer culling disabled. Should the need to perform per-frame stencil writes arise, merging the material pipelines and the stencil pipeline may prove beneficial to avoid expensive pipeline re-binds but for the sake of

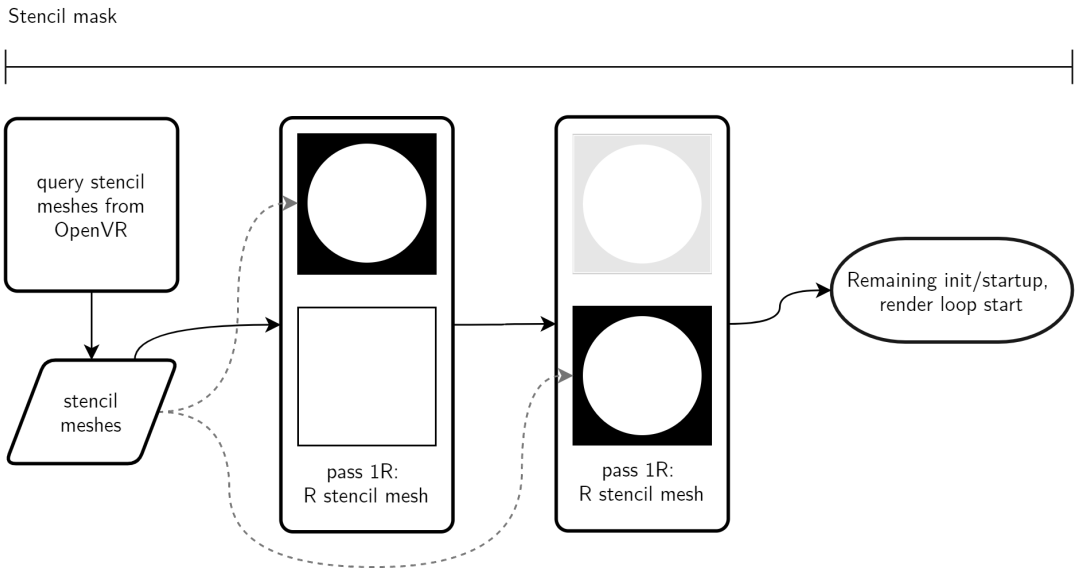


Figure 4.5: HMD stencil mask query and render flow in rtvklib

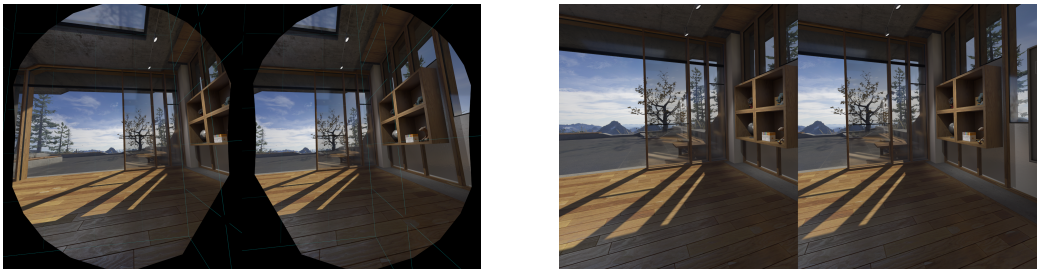
cleaner separation the described setup was used in Tachyon's current implementation.

4.3 Monoscopic Far-Field Rendering

Monoscopic Far-Field Rendering (MFFR) is an approach strongly leaning toward an inherent property of many optimizations in the field of rendering and real-time computing in general, which is the property of trading accuracy for speed. MFFR is a topic brought up again soon after Oculus Rift CV1's retail launch by Oculus developers Rémi Palandri and Simon Green at developer keynotes like the ARM GDC 2017 talk[37] and the Oculus Developer Blog as Hybrid Mono Rendering[38].

```
//stencil op settings for pipelines using
//the stencil mask for comparison
VkStencilOpState stencilOpState = {};
stencilOpState.compareOp = VK_COMPARE_OP_NOT_EQUAL;
stencilOpState.failOp = VK_STENCIL_OP_KEEP;
stencilOpState.depthFailOp = VK_STENCIL_OP_KEEP;
stencilOpState.passOp = VK_STENCIL_OP_KEEP;
stencilOpState.compareMask = 0xff;
stencilOpState.writeMask = 0xff;
stencilOpState.reference = 1;
```

Figure 4.6: Stencil operation flags set for default material Pipeline



(a) Stencil mask returned by SteamVR for the Index Vive

(b) (lack of) Stencil mask returned by SteamVR for the Rift CV1

Figure 4.7: Comparison of stencil mask availability (while running SteamVR home)[TODO: Odyssey, Vive]

```
//stencil op settings for the stencil pipeline
VkStencilOpState stencilOpState = {};
stencilOpState.compareOp = VK_COMPARE_OP_ALWAYS;
stencilOpState.failOp = VK_STENCIL_OP_REPLACE;
stencilOpState.depthFailOp = VK_STENCIL_OP_REPLACE;
stencilOpState.passOp = VK_STENCIL_OP_REPLACE;
stencilOpState.compareMask = 0xff;
stencilOpState.writeMask = 0xff;
stencilOpState.reference = 1;
```

Figure 4.8: Stencil operation flags set for stencil mask Pipeline

Understanding the concept requires some explanation of the technical and visual background. Depth perception of the human eye relies on the slight spatial distance between both eyes as each eye sees a slightly different angle of a given object. This difference in perceived angle is called stereo separation. Without it, the brain has difficulties determining the depth at which a certain object or surface lies. Regular stereo rendering recreates this separation correctly when rendering the two virtual eyes at their respective spatial offset from the HMD center - given correct projection and view matrices and accurate world scale at least.

However, as distance grows, stereo separation shrinks - the aspect MFFR exploits. In infinity, separation would be infinitely small. Even at more reasonable distances separation is small enough so even with good vision it is hard to properly judge depth unless the object is large. This of course also holds true for rendered stereoscopy, but an additional limit is the pixel density of the output displays. This means that at a certain distance from the virtual camera, stereo separation will shrink to less than a full screen space pixel once projected. If the difference can not physically be displayed by the HMD, it is a waste of resources to still render both eyes.
Mono Far-Field Rendering opts to skip the second view during rendering of the name-giving

far field of objects. The hope here is to only render a single view past a certain distance, reducing rendering load without the user noticing the theoretical loss in accuracy. This approach has caveats however. The value at which a field split - the distance at which the stereo rendering is cut off and followed by only mono rendering - will depend on the individual user, their quality of vision and spatial awareness. It will also potentially depend on the resolution of the used headset given the user's vision is good enough to not deteriorate before that point. Note here, this thesis will not explore these constraints of MFFR further than approximate values used for testing as time does not allow more.
MFFR has been implemented by Oculus LLC and Epic Games Inc in Unreal Engine 4 and was recommended for certain types of pixel-bound mobile VR experiences with very limited GPU power but has been removed from the engine in update 4.20 without further explanation. An odd decision, as UE documentation posts prior to the removal indicated continued optimization efforts such as added compatibility with UE4's multiview path[39].

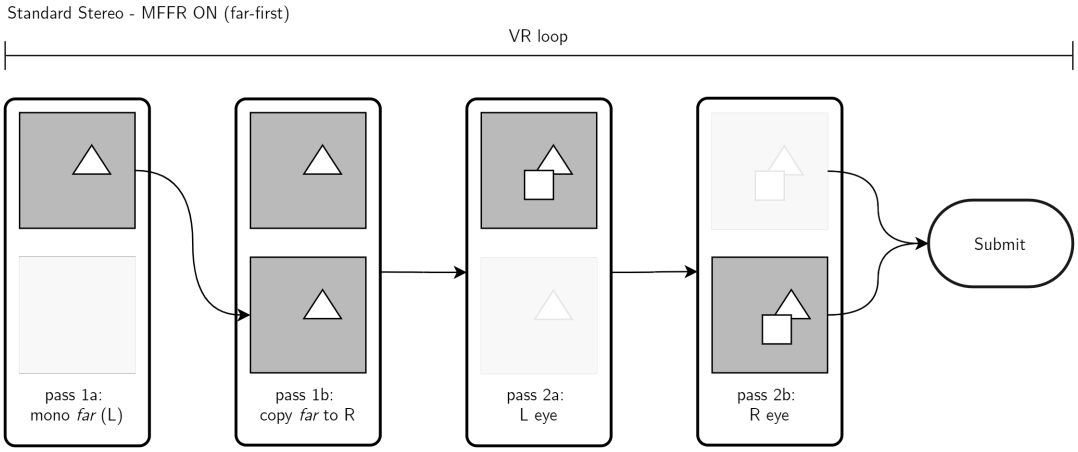


Figure 4.9: Per-frame render pass flow of far-first MFFR;
Each row represents one image buffer, each column represents the process steps the respective buffer is subjected to

4.3.1 Estimated impact

The makeup of the scene itself will also affect the effectiveness of the solution. As cautioned by Oculus LLC in their developer reference on Mono Far-Field Rendering [38], there is a certain baseline overhead simply for enabling the additional render pass necessary for the monoscopic image and the associated context switches. Furthermore, only scenes with a significant amount of distant geometry beyond the field split distance will benefit from the optimization, as obviously the second view workload can only be saved for objects originally contained within that second pass.
Going purely by Palandri and Green's blog entry, their forward-rendered UE4 implementation supposedly running on a GearVR device saw frametimes in the Epic SunTemple sample scene drop by 25% and their best-case Unity test scene demonstrated a 49% drop[38]. This

would indicate that if circumstances allow - meaning pixel-bound forward-rendered large scale scenes - impressive performance gains well upwards of 20% can be expected, while less suitable cases in the worst case will see no change if not minor regression.

4.3.2 Far-first approach

Implementing Mono Far-Field Rendering into Tachyon - or any renderer for that matter - requires care and a number of changes. There are at least two possible ways of doing it, both by way of multiple render passes and with their own respective drawbacks and advantages. One way of implementing MFFR is doing the far pass first in each frame, followed by a near pass, both possibly using the same framebuffer as illustrated in Figure 4.9. At the start of the frame, the framebuffer is cleared - color and depth clear are necessary - after which the far-field command buffers are submitted and executed. This pass can render directly into the buffer of one eye and copy color and depth buffer to the other eye. Finally, the near-field command buffers are submitted and render their values into the buffers already containing the far field information. For this case, one extra step needs to be performed. As the projection matrix normally projects the world as seen by the camera into a uni-cube with each axis being length 1 going from 0 to 1 and this same uni-cube is used for both depth value calculation and projection clipping (triangle discard via clip space evaluation), the two passes can not share the same uni-cube. Ways around this would be to either use regular full-depth projection matrices and clear the depth buffer before the near pass or to scale both passes' projection matrices by 0.5 and translate the far projection by 0.5 so the uni-cube is shared between the two fields without overlap.
If neither is done, the effective depth values of far-pass objects will be closer than they should be and possibly closer than those of near-pass objects, leading to some near-field objects being drawn behind far ones. The depth buffer clear option has the benefit of utilizing full projection precision and correct triangle clipping. The matrix squashing option avoids an extra clear but means vertices otherwise clipped outside of the uni-cube are pulled into the uni-cube and not skipped.
The overall advantage of this far-first approach is it yields a full precision far-field depth buffer which may be useful for stereo interpolation as briefly described in the subsection 4.3.6. The disadvantage is early Z discard cannot be fully effective as all far geometry is rendered first even if opaque geometry close to the camera would later obstruct it. Whether the benefits of lower split distance coupled with interpolation can outweigh the additional cost of overdraw heavily depends on the scene and how much geometry is contained within the far volume. The goal of this approach is to reduce memory operations and locality to a minimum and avoid more costly compositing methods.

4.3.3 Near-first approach

The alternative to rendering the far pass first is to render the stereo near volume at the start of each frame as illustrated in Figure 4.10. A key difference to the former approach is the first pass needing to flag the stencil buffer pixel when an opaque fragment is within the uni-cube

and a color value is written. After the stereo pass, each eye contains a binary stencil mask of the near-field occluded screen areas. In the next step, the far pass can then execute regularly except with stencil testing enabled. This sequence has no direct need for depth buffer clearing or uni-cube sharing. However, the constraint is the far pass also needing to sample from left to right eye separately as a straight buffer copy cannot work anymore. The advantage of this approach over far-first is early Z discard takes full effect and stencil masking reducing the leftover render area even more. The downside is the need to perform additional stencil writes every frame and potentially costly sampling operations.

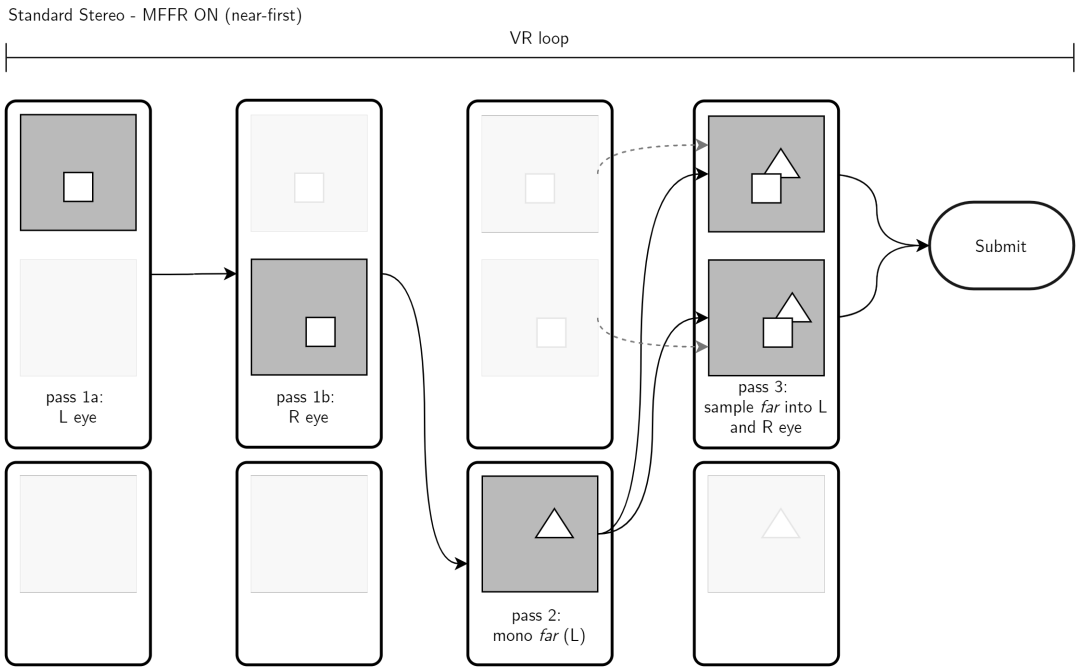


Figure 4.10: Per-frame render pass flow of near-first MFFR;
Each row represents one image buffer, each column represents the process steps the respective buffer is subjected to

4.3.4 Implementation specifics

In this thesis, the plan is to do far-first MFFR, albeit the measured results are not favorable. In the first step of each VR frame, a monoscopic render pass renders the far clip volume's color and depth values into the index 0 layer of the framebuffer. The next step is to copy that layer to index 1 as well. In the final step the regular stereoscopic render commands are executed with reduced near field clip volume, including clearing the depth buffer at the start of the stereo pass to avoid the uni-cube issue. While the general approach is mostly straight-forward and both described ways are possible, implementing either in a Vulkan renderer is no trivial task and requires the following changes to rtvklib:

- for the virtual camera, a field split distance parameter is introduced and an additional

frustum is added; the stereo frusta cover the volume from near plane to split plane while the far field frustum covers split plane to far plane

- the frustum culling procedure is altered to write the far frustum's resulting draw commands into a separate set instead of merging them into one (as would happen for regular multifrustum culling in Tachyon)
- at initialization time of the VR render target an additional render pass is created for monoscopy with the main difference - compared to the regular VR render pass - being the removal of the multiview extension
- an additional set of Vulkan `VkCommandBuffer`s is added, into which the draw commands of the far frustum cull set are to be recorded
- an additional set of `VkSemaphore`s is added to synchronize the two render passes and create them in `CreateSyncObjects()` during initialization
- an additional `VkCommandBuffer` for the layer copy operation is added
- at initialization time of the VR render target this copy command buffer is pre-recorded so it can be reused every frame; this recording includes transitioning the layout of both the color and depth image from `VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL` to `VK_IMAGE_LAYOUT_GENERAL`, `vkCmdCopyImage(...)` both images' layer 0 to layer 1 and then transitioning the layouts in reverse again
- in the per-frame recording procedure of the VR render target, `RecordCommandBuffers(...)`, the entire structure of begin and end of command buffers and render pass and per-pipeline `RecordDrawCommand(...)` calls is duplicated with the monoscopy render pass and far field command buffers set; afterward the prior regular stereo recording still takes place
- in the render target's `RenderFrame(...)` function the far field command buffer and layer copy command buffer submission is inserted before the regular stereo submits; the mono submit is set to wait on `mRenderCompleteSemaphores` and signal `mFarfieldCompleteSemaphores`, while the stereo submit is only set to wait on the latter
- in VR render target's camera setup, the far field frustum projection matrix is constructed as outlined in [40] pp. 515-519 - albeit transposed as Tachyon still retains the OpenGL matrix format for a few matrix types
- an additional set of camera data struct and camera index is added
- in the render target's `UpdateCameras()` call, the far field volume's view projection matrix is updated by transforming aforementioned projection matrix by the current HMD pose and the updated matrix is written to the camera data buffer on the GPU

4.3.5 rtvklib MFFR failure

Some issues with this implementation prevail and make it unfit for productive deployment in Tachyon. For one, while the projection matrix itself uses the correct parameters, view matrix transformation used for the far field remains incorrect, so head movement leads to incorrect projection distortion effects on the horizontal axis. While the lack of pure separation at little to no sideways tilt may be acceptable and not easily noticed with a correct view matrix, tilting the head means more severe separation mismatch as the spatial disconnect is expanded from being mostly horizontal to being horizontal and vertical. Fixing this disparity would require translating the far field image of each eye to conform more closely to the expected overall stereo shift. Vitally, this visual distortion does not affect performance measurements presented in chapter 6 as the used internal culling frustum is computed from the virtual camera's world position and direction in any given frame and not subject to incorrect projection transformation. Subsequently, submitted draws and render passes are correct and executed in the described order, as exemplified by the Nvidia Nsight VS debug timeline in Figure 4.11.

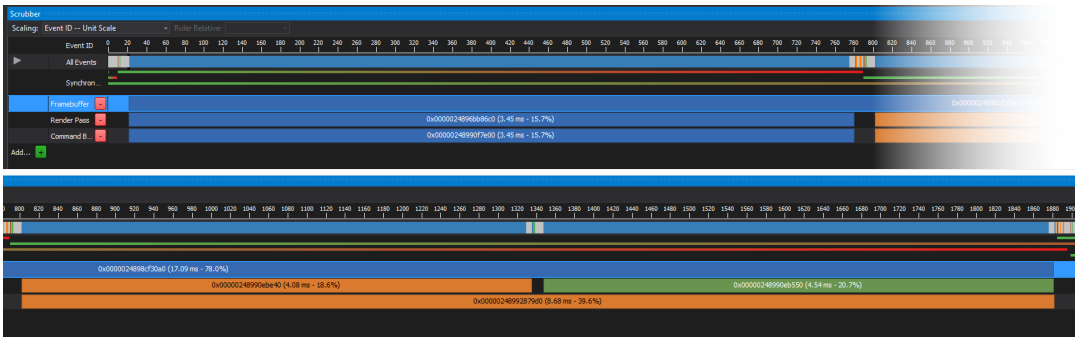


Figure 4.11: Draw submission and render pass binding timeline of a typical rtvklib MFFR frame, captured in Nsight VS. (Split after far-field render pass and stacked vertically for better page fit)
On the "Render Pass" row, the blue pass denotes the far-field pass, orange is the left eye and green is the right eye. Far and near are each synchronized by a VkSemaphore.

Secondly, Vulkan render passes discard fragments at their end by default if depth testing is enabled and the depth value of said fragment equals 1.0 (and is thus on the far plane). This means any color values written by the far pass and not overwritten by the near pass are discarded right at the end of the stereo pass and before presentation. A workaround for this is to clear the depth buffer at the start of the stereo pass not to 1.0 but to 0.999999, which is the closest value a IEEE-754 float can reach below 1.0 (22 of 23 mantissa bits set to 1). With this tweak, color values were composited together without any visible loss at the near field's split plane, even if it technically entails a depth clamp in that last mantissa bit.
Lastly, and most importantly, performance simply is not up to par. On the i7 and RTX2080 test system (**WS-Big**, see chapter 5), enabling MFFR with a sufficiently far split distance

yielded a 34% performance loss. The culprit for this is most likely two-fold. One factor is the detrimental Z ordering of the submitted draw commands since much of the far field color buffer is overdrawn by the near pass and thus counts as wasted effort. The other factor is the lack of parallelism due to the use of a shared framebuffer. The two render passes need to be synchronized to run in order and cannot be scheduled on parallel thread warps to increase GPU utilization and thus performance. What's puzzling in this scenario is that Palandri et al. in their Oculus blog entry[38] saw a clear performance uplift in a Unity sample scene with seemingly very similar MFFR construction.
As such, the rtvklib MFFR implementation as presented here remains beneficial only in theory and will require further work to result in real performance gain.

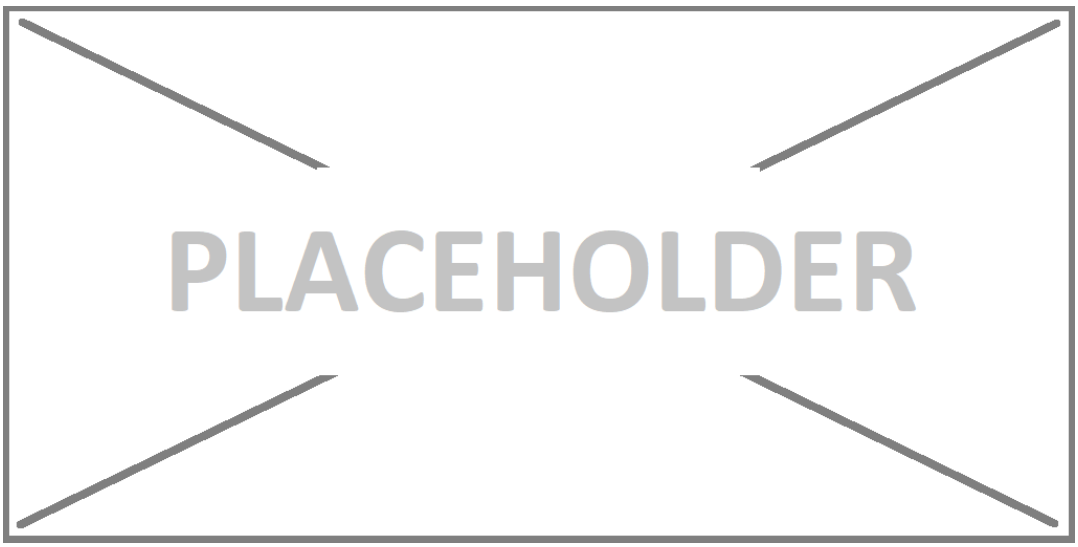


Figure 4.12: [TODO: distorted picture example?]

4.3.6 MFFR Variant: Depth Shift

In its basic version MFFR as described in section 4.3 completely foregoes separation beyond the split distance and as such the split distance has to be set relatively far back to minimize the visual inaccuracy. Naturally, an attempt to try and reduce that split distance closer to the camera is by artificially and cheaply increasing that accuracy again. One such way is to use the data already contained in the framebuffer's depth layer during rendering. As stereo separation is mostly dependent on depth, given the object itself and its properties are known, an improvement is to approximate small amounts of separation based on that depth buffer. Instead of simply copying or sampling the far image to both eyes after the far field pass, one can do an additional sampling or post-processing pass which includes slightly shifting pixels according to the depth value of the respective given fragment.
This interpolation should allow pulling the field split distance closer to the camera and save some stereo render time, but it is unclear whether the savings outweigh the additional

processing cost and this thesis does not explore MFFR beyond the base variant.

[TODO: shift math or geometric illustration]

4.3.7 MFFR Variant: Alternate eye

Another possible way of pulling in the split distance value while retaining approximated separation comes back to the VR property of high framerate as explained in earlier chapters like the Round Robin Culling (section 3.4). Assuming high framerate and refresh rate, the mono render pass could be called with the camera parameters not custom calculated as a middle point between the two eyes but alternating between left and right eye parameters each frame. This way, each alternating eye would be correctly projected every other frame and incorrect data would likewise only persist for one frame at a time in each eye. If this eye-alternating MFFR were to be combined with interpolation of the respective incorrect eye, the perceived inaccuracy (seen as flickering as the visual information is only incorrect every other frame) may be further reduced. Candidates for this are a simple frame interpolation or a single frame temporal reconstruction, although a single previous frame may not be enough for stable results.
Once again this thesis does not encompass this additional option and thus it is unknown how far the split distance could be pulled in and whether related savings would outweigh interpolation cost. Just as with Round Robin Culling, the same potential artifacts and issues are present here.

4.4 Foveated Rendering

This technique lends its name from the fovea centralis, a spot in the center of the primate retina, responsible for sharp central vision of the eye. The regions around it gradually lose visual sharpness as they contain fewer and fewer cone cells. Foveated rendering in its various forms builds on this limitation of the human photoreceptive system. For example, as described in the Oculus Go optimization guide by Palandri et al.[41] the image produced by a VR renderer is warped by the VR compositor to closely match the spherical shape of the lenses and the distorted field of view they create. This warping means that toward the edge of the frame, more pixels are compressed into a given angle of vision than in the center, effectively leading to much higher pixel density for peripheral regions. Conversely, the peripheral vision of the human eye is usually significantly less sharp than the aforementioned foveal vision. Therefore the outer rim of the rendered image can be rendered at much lower pixel density without sacrificing a humanly noticeable amount of detail if the HMD can even physically display this density. Exactly that is the concept of foveated rendering, it decreases resolution of the edge regions of the image while rendering the central area at full resolution. However, there is one big constraint. The foveal vision of the user and the artificial foveal center of the image need to match up, otherwise a user may easily notice the lowered pixel density. To accurately match these two focus points, the user's eye movements need to be

tracked and the supposed focus center of the image adjusted accordingly.

4.4.1 Fixed versus True Foveated Rendering

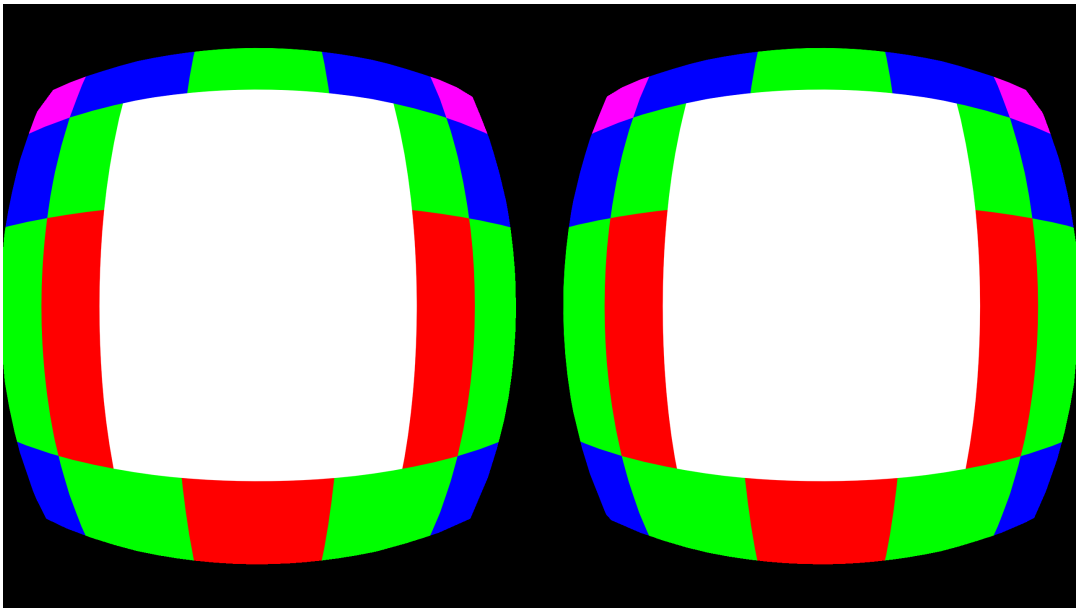
Eye tracking can be foregone in theory, assuming the headset's field of view is low enough to physically discourage a wide range of eye movement and instead rely on head rotation. Additionally, the opening angle of the foveated region should be wider so eye movement on a small scale is still without consequence for the perceived image quality. The resolution of peripheral areas may not be reduced as much as with so-called *true* foveated rendering so even when the user briefly looks at such an area, the perceived loss is not as distracting. This compromised form of the technique is commonly called Fixed Foveated Rendering, for example available on Oculus Go (4.13a).
Given eye tracking of some adequate sort is available, true foveated rendering can be performed in which in each frame the current focus position of each eye is queried and the virtual view matrix adjusted accordingly. This makes the rendering setup more complex as the high pixel density area can shift to any location within the image but it allows for a tighter foveal angle and lower peripheral resolution as it is much less likely - if not entirely impossible due to limits or occasional mistakes by the tracker - for the user to focus on any such low density frame data. See ZeroLight Limited's Nvidia VRS based approach in 4.13b.

4.4.2 Radial Density Mask

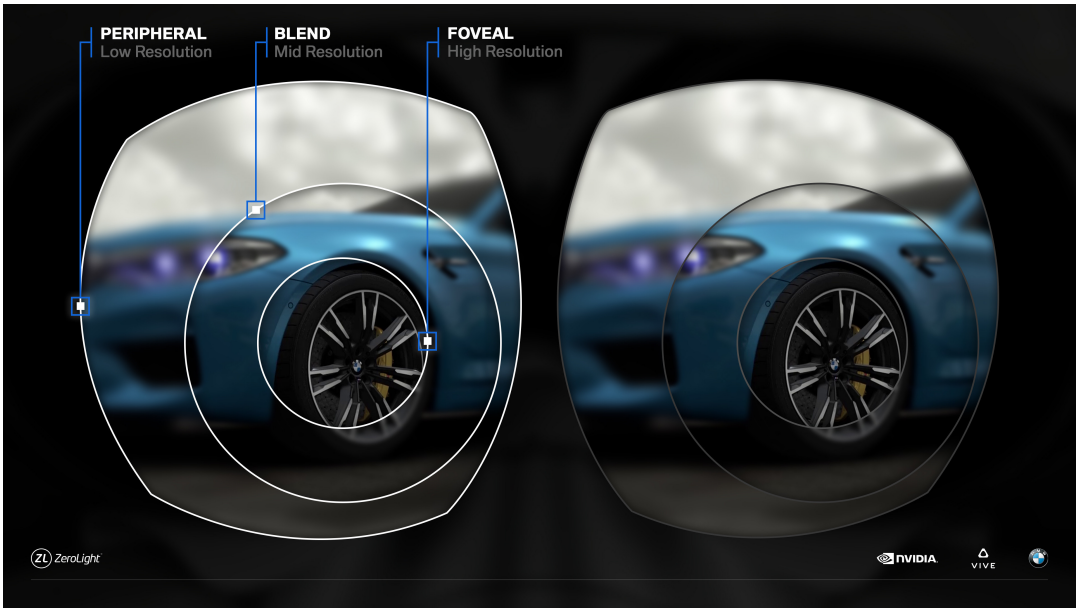
A somewhat related technique is called radial density masking as shown by Valve's Alex Vlachos in his talk at GDC 2016 [43]. The goal is the same as with foveated rendering but the approach is different to better exploit traditional GPU architectures. Instead of reducing the theoretical rendering resolution of the peripheral ring, a mask is overlaid. The mask has the render pipeline skip a certain pattern of pixels in that ring. This can either be done by marking a checkerboard pattern of pixels in the stencil buffer so the pixel shader fails the stencil test on them - coming back to section 4.2 - to get a pixel perfect mask, or by overlaying a masking mesh right at the near plane of the render volume so the respective fragments are discarded during early Z tests. The latter would allow to approximately match a relative area reduction target but may not be pixel perfect depending on internal frame resolution. The resulting checkerboard area can then be interpolated and filtered to reconstruct the missing information. Vlachos claims gains of up to 15% for the Valve Aperture Robot Repair demo scene, but warns that reconstruction cost needs to be kept lower than mask savings.

Adaptive resolution

The described methods of fixed/true foveated rendering and radial density masking can be combined further with another - by now almost universally used - optimization compromise: dynamic resolution scaling. While monitoring GPU load or frametimes and framepacing, the resolution of not just the central fovea but the peripheral regions too can be reduced or increased within given bounds. This can stabilize performance at the cost of some visual



(a) Oculus Go Fixed Foveated Rendering map (colored tiles decrease in resolution outward)[41]



(b) Dynamic Foveated Rendering using Nvidia Variable Rate Shading on HTC Vive Pro Eye[42]

Figure 4.13: Foveated rendering examples

quality in the outer regions or - at the other end of the spectrum - alleviate some of the quality reduction if the available power overhead allows.

Figure 4.14: Radial Density Masking (2x2 px checkerboard)[43] [TODO: footnote of page in slides]

4.4.3 Relevancy of GPU architecture

The previous sections describe various methods but the choice which of them is ideal for a given engine depends a lot on the target hardware. Foveated rendering relies on splitting the frame into several rectangular sub-frames and rendering them at differing internal resolution. Thus, it is only suited for GPU architectures supporting or better yet being built as so-called tile based renderers. Tile based GPUs such as Qualcomm's Adreno line and most other low power mobile SoCs compute the frame parallely in a number of set tiles. On the other hand traditional GPUs execute render commands only on a frame as one entire unit, albeit with possibly many more compute units at once. These tile based architectures naturally make it very simple to render tiles outside the foveal center at lower resolution without additional overhead, with the fovea circle being better approximated the higher the tile count is.
More traditional architectures like those found in Nvidia and AMD desktop graphics chipsets are rarely built as tile renderers and may only support this tiling in software. Such software solutions exist, even low-level optimized and accelerated like Nvidia's Multi-Res Shading[44] and AMD's LiquidVR MultiRes[29]. On those, rendering frame regions at different resolutions requires internally splitting a frame into multiple viewports with a subsequent composition pass. This sequential process incurs additional overhead as it requires broadcasting geometry information to all these viewports, which is not possible in an efficient way on all GPU architectures as it relies on similar functionality as multiview (section 4.1) except with significantly more target viewports. The radial density mask approach may be more suitable for such traditional GPUs as it manages to render multiple resolutions within a single render pass by leveraging fragment discard features. This masking will necessitate an interpolation

pass to blend away the masked pixels, while foveated rendering can technically skip further interpolation. Filtering the low resolution perifoveal pixels is recommended to reduce aliasing.

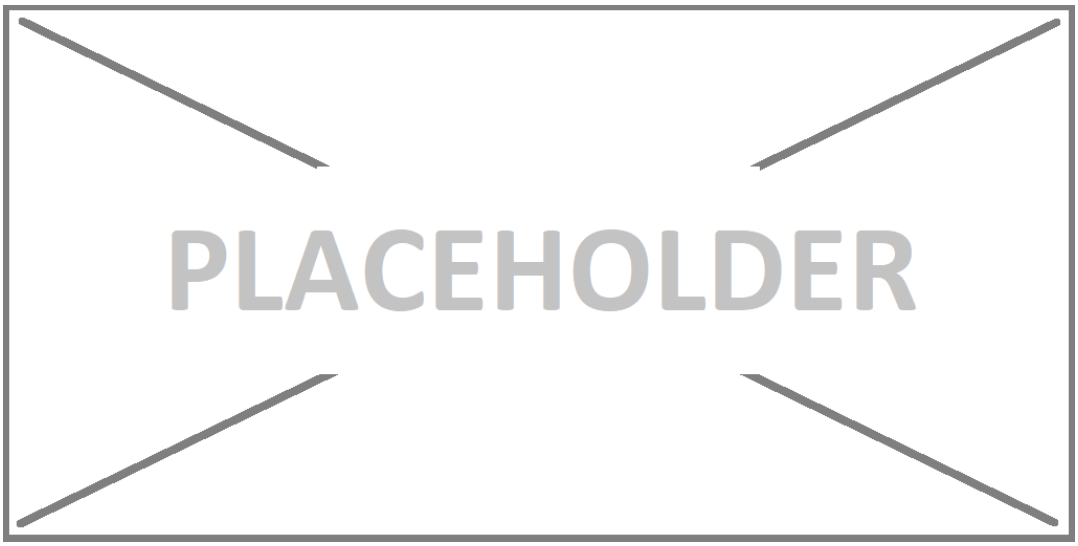


Figure 4.15: [TODO: illustration Tile vs Whole]

5 Performance testing setup

As can be gathered from the previous chapters, the optimizations implemented in Tachyon for this thesis are

- Superfrustum Culling
- (vendor agnostic) Multiview Stereo Rendering
- HMD-matched Stencil Mask
- Monoscopic Far-Field Rendering

While expectations for the first three items were optimistic, it shall be noted again that MFFR unfortunately turned out unsuccessful, which will reflect in this following chapter.

5.1 Benchmark scene

In order to properly assess how each of these implementations fares at run-time and how it impacts performance of the engine, a series of benchmarks were conducted. To ensure repeatability of the benchmark, a synthetic test scene was constructed, aimed to stress the tested systems to a degree not likely found in many real scenarios. While this may seem counterintuitive, it paints a worst-case picture of performance to be expected and how the tested methods hold up. [TODO: conduct a few short tests with a lower workload just to get an idea of scaling?]
This test scene is built as follows: the scene dimensions are set up as 32 by 32 `chunk`s, each `chunk` sized 80 units on each axis. This gives an overall scene volume of 2560 by 80 by 2560 units. This seems strongly skewed towards lateral expansion rather than vertical, chosen primarily due to the expected productive use being industrial scenes covering large factory floors but not necessarily very vertical setups. Another reason is that the Tachyon scene `chunk` system currently does not allow stacking of `chunk`s and as such an adequate compromise between scene scale and octree scale had to be chosen. Filling this test scene is a selection of objects, the geometries namely being a primitive cube and three high polycount objects, a robot called "Robi", a material showcase sphere and a PBR showcase helmet. These objects are placed in the scene by iterating through a counter for each axis and placing an object instance at each new count. To determine the instance position, the three axis counts at that moment are multiplied by a spacing of 3.6 and a entropy value is added to each axis. This entropy value is combined as `x = sinf(sinf(x) + cosf(y) + tanf(z))`, `y = cosf(sinf(y) + cosf(z) + tanf(x))`, `z = sinf(cosf(sinf(z) + cosf(x) + tanf(y)))`. Adding this artificial entropy

makes the scene look more chaotic but is still deterministic and repeatable. By default the placed object is a primitive cube, but at every intersection of x+z, y-z and z counts valued 11, one high polycount instance is placed, with the chosen type being modulo-index-incremented over the available high polycount types. This placement setup is done with target counts of 711 for the X and Z and 22 for the Y axis respectively. This utilizes the scene height as much as possible and results in a total instance count of 11.121.462, a respectable number even for detailed industrial applications.
Still in the interest of repeatability, the head-tracked headset pose had to be disabled for these tests in favor of a scripted on-rails camera pose that follows a simple circular pattern for its virtual position and another one for rotation, both based on the sine and cosine of the number of frames completed since the first rendered frame. This gives a simple and arbitrarily repeatable pattern resulting in the same camera position and angle at the same frame count for each run, obviously making it much easier to match measurements.

5.2 Timing code & metrics

To get an accurate idea of how the computational effort within a frame changes and is split up over its several steps, `STL::chrono high_resolution_clock` timings calls were used in strategic places. For each frame, the measured metrics are:

- total frame-time (microseconds)
- CPU-only time (microseconds)
- Culling-only time (microseconds)
- GPU-only time (microseconds)
- number of draw calls submitted
- number of triangles submitted through draw calls

Times are measured by calculating elapsed counts between start and end time points of the respective function call. GPU-only time is measured by artificially placing a `VkQueueWaitIdle(graphicsQueue)` at the end of the `VKRenderer::RenderFrame` procedure. At first glance this seems counterproductive as it prevents frames from overlapping resource usage, but this is where synthetic repeatability becomes relevant.
To guarantee runs with different optimizations enabled can be compared to each other, the render loop is modified to follow aforementioned camera pattern and and save the current frame number with each data point. This obviously flies in the face of desired real-world decoupling of motion from framerate and overlapping execution, but it ensures identical workload per frame for each tested configuration. Additionally, the camera pattern and timings loops are tuned to run exactly twice in exactly 5400 frames each for 10 loops. The resulting 54000 samples of frame data for each configuration are then filtered to exclude the worst outliers and the 10 loops are averaged into one representative set of 5400 data points.

The very first frame of the averaged data sets is removed as the very first frame of the first loop after program start includes all initialization commands and major buffer transfers and thus results in an outlier frame time an order of magnitude higher than the remaining 53999 frames of each run.
However in the interest of real-world scaling, a later section will also briefly explore average frametimes for a selection of configurations without these limits in place.
Lastly, and this part is crucial, the `Submit()` calls to SteamVR are disabled for the testing runs as these calls otherwise force vertical refresh synchronization (Figure 5.1), completely distorting actual performance numbers. While this means during benchmarking there is no output to the connected HMD, the internal rendering loop still uses all of the HMD's rendering parameters and more importantly renders as many frames as it can without a framerate cap or refresh synchronization, exactly the behavior we need for accurate measurements.

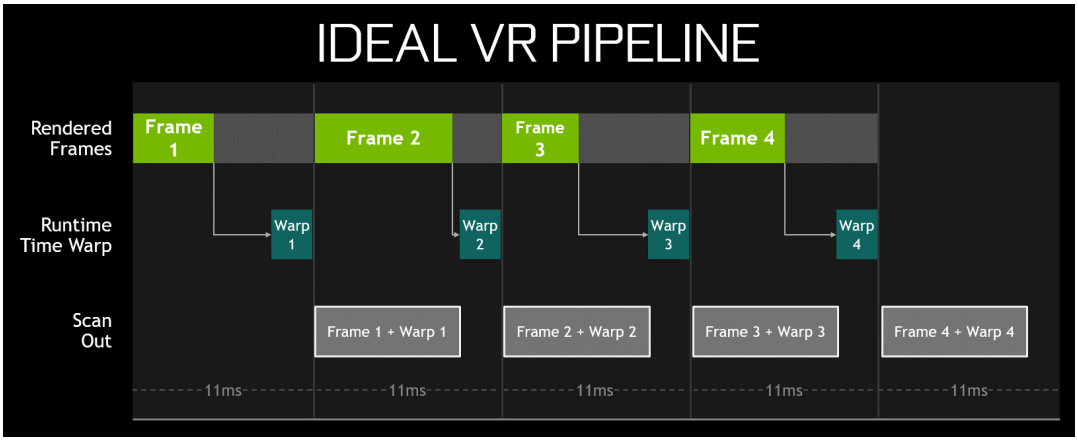


Figure 5.1: Virtual Reality Pipeline as imagined by Nvidia[45] [TODO: footnote of page in slides]

[TODO: resource usage aka memory per frame for relevant configurations, grabbed e.g. from representative NSight Frame]

5.3 Compilation parameters

Naturally productive deployment would go for fastest possible optimization and as such the tested configurations were all compiled with `-O2` in Release mode using Microsoft Visual Studio 2015 and its v140 MSVC toolset. To further eliminate potential slowdown from branching or data tracking, each of the 16 permutations of enabled optimizations is not done simply through `if` statements or `switch` cases, but through preprocessor defines. These defines are `SUPERFRUSTUM_ON`, `MULTIVIEW_ON`, `STENCILMASK_ON`, `MFFR_ON`. Similarly, benchmark timing code is enabled via the `BENCHMARK_MEASURE`, `BENCHMARK_CHUNKINFO` and `BENCHMARK_CAMRAILS` flags.
As a sidenote here, *all* configurations were tested with distance culling (`DISTANCECULLING_ON`, draw distance 320 units) and frustum culling (`FRUSTUMCULLING_ON`) enabled as running the

massive 11.1 million instance scene without these in place would realistically bring any test machine near a grinding halt. It also seems unrealistic to run any scene with advanced VR optimizations enabled but leave such simple measures disabled. Demonstrably, these culling methods are also not posing as dangerous bottlenecks to the tested configurations as the baseline measurement details will show.

5.4 System specifications

Two test machines were used to perform measurements on. Each graph and table will denote which system it is based on, respectively.
The first machine, further titled **WS-Big**, is specified as:

- CPU: Intel Core i7 6700 (4c8t Skylake, 4x3.4GHz base, 4.0GHz boost)
- RAM: 2x16GB DDR4-2400/15
- GPU: Nvidia GeForce RTX 2080 Founders Edition (2944 Turing cores at 1800MHz core, 8GB GDDR6 at 1750MHz) - driver version 432.00
- Storage: Samsung SSD 840 Evo 500GB
- OS: Microsoft Windows 10 Pro x64 1809

The second machine, further titled **WS-Small**, is specified as:

- CPU: AMD Ryzen 5 1600 *12nm* (6c12t Zen+, 6x3.2GHz base, 3.7GHz boost)
- RAM: 2x8GB DDR4-3066/14
- GPU: Hewlett-Packard Radeon RX 580 (2304 Polaris cores at 1200MHz, 4GB GDDR5 at 1750MHz) - driver version 19.12.3
- Storage: ADATA SSD SX6000 Pro 500GB
- OS: Microsoft Windows 10 Pro x64 1909

Furthermore, the following virtual reality headsets were available with the respective resolution setting and in the respective capacity:

- Valve Index (0x0 pixels[TODO]) - **WS-Big** performance measurements, functionality verification
- HTC Vive (0x0 pixels[TODO]) - **WS-Big** functionality verification
- HTC Vive Pro (0x0 pixels[TODO]) - **WS-Big** functionality verification
- Oculus Rift CV1 (1344x1600 pixels) - **WS-Big** functionality verification

- Samsung Odyssey (0x0 pixels[TODO]) - **WS-Small** performance measurements, functionality verification

To ensure repeatability and level the field as much as possible, the CPU of each system is locked to a fixed clock multiplier and thus operating frequency. For the i7 6700 and its lack of unlocked multiplier this is achieved by setting minimum and maximum processor state in Windows to 100 and 99% respectively, which effectively disables singlecore turbo boost and speedshift and fixates the frequency at the allcore boost multiplier of 37x for 3.7GHz allcore. For the Ryzen 5 1600 a more reliable method is available, that being manually overriding the CPU multiplier within the mainboard UEFI, in this case to a fixed value of 38x for 3.8GHz across all 12 processor threads.
Keeping GPU clock speeds at the same level for all test runs is harder to achieve as all modern graphics cards employ some version of dynamic boosting algorithms based on temperature, power target, voltage target and usage. The Radeon RX 580 is set to target a maximum of 1200MHz boost frequency via a VBIOS modification that disables higher automatic boost, and during core load it maintains this target stable unless the power target of 100 Watts is exceeded. For Nvidia graphics cards later than 2014's Maxwell 2 architecture, such a firmware mod is unfortunately not possible. There is, however, one way of reaching a mostly stable core frequency. To achieve this, the card needs to be pre-heated using any application generating GPU load. To prime **WS-Big**'s RTX 2080, SteamVR Home is idled for 30 minutes before any test runs are performed, resulting in an equilibrium of 70°C at 1905MHz core frequency. This temperature and frequency is maintained for the entirety of each test run as the 210 Watts power target of the card is never exceeded.

6 Performance benchmark results

Testing each combination of *ON* and *OFF* states of the four optimizations logically yields 16 permutations. This chapter aims to show the metrics declared in section 5.2 for each permutation and explore causes and implications for those permutations that would reasonably be employed in an application. To get a frame of reference for any potential performance improvement - or degradation - a baseline measurement is necessary, meaning a set of data points with all four optimizations set to *OFF*. This baseline as measured on **WS-Big** looks as follows (Figure 6.1):

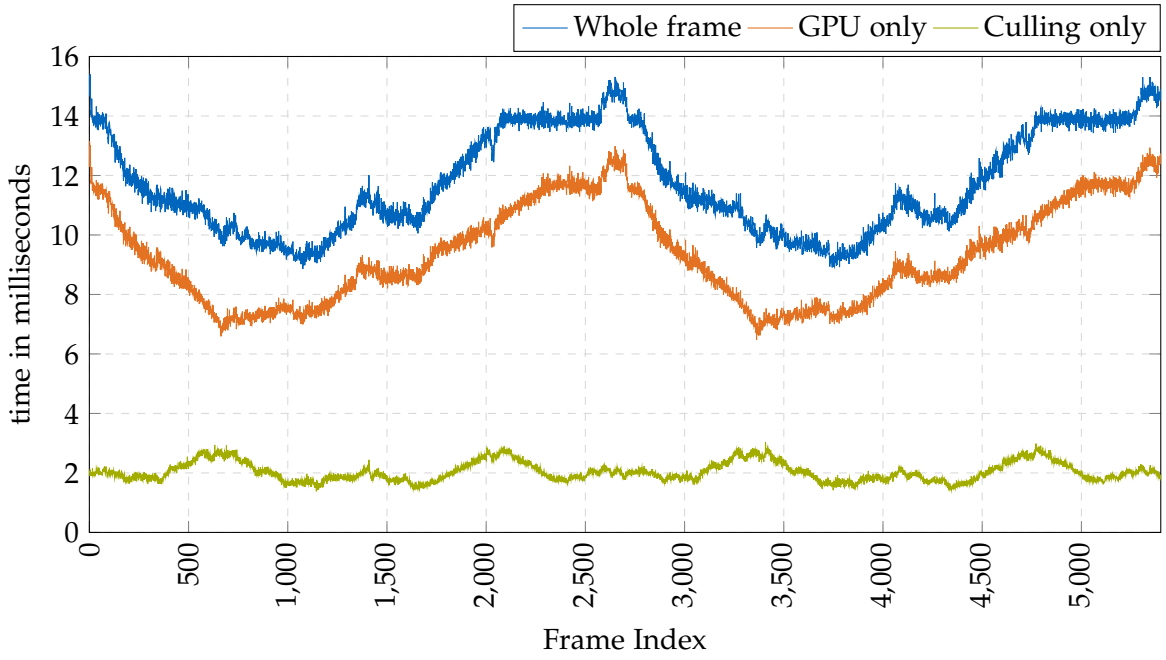


Figure 6.1: baseline/all OFF performance (10 run average)

While these performance numbers were recorded, the sensors of the RTX 2080 and i7 6700 were monitored using the tool *HWiNFO64*. For the duration of the test, this tool attests utilization averages of 80% for the GPU core, 37% memory capacity, 32% for the GPU memory controller and 20% for the PCIe bus link. For the CPU an overall average utilization of 49% with each of the 8 threads averaging between 30-40% was seen. RAM usage from asset load at program start up until termination reported 5.4GB for the application by itself.

6.1 Individual impact

[TODO: discussion of each, also why I omit MFFR for remaining permutations]

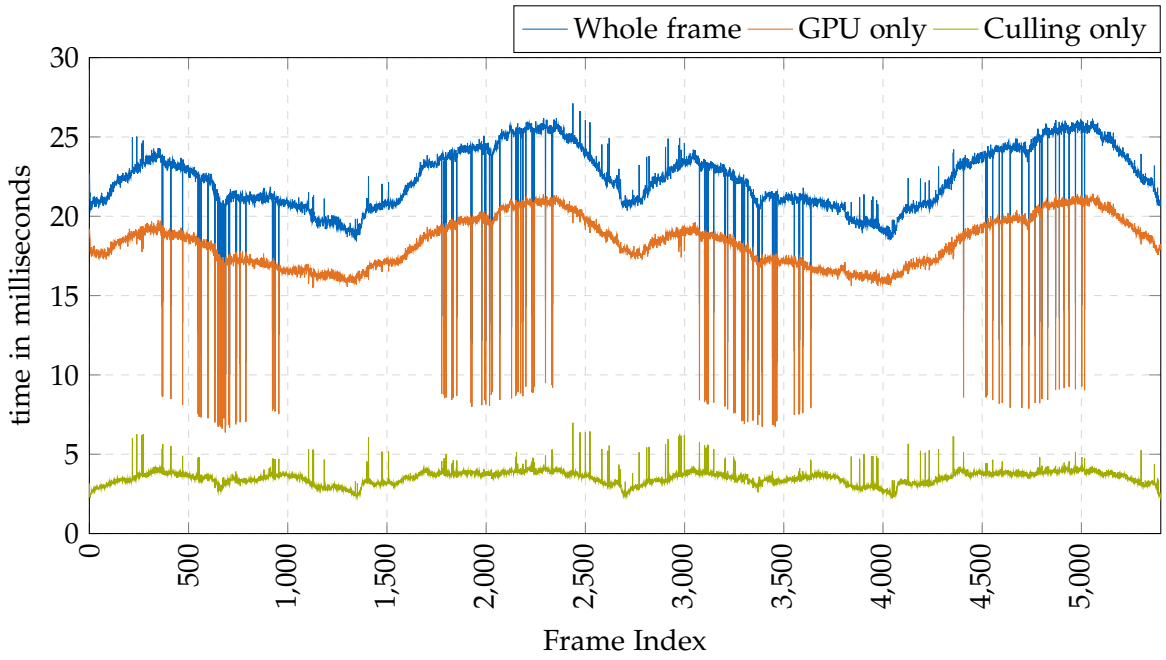


Figure 6.2: MFFR ON performance (10 run average)

Comparison of optimizations (individual use)

6.2 Combined and partially combined impact

[TODO]

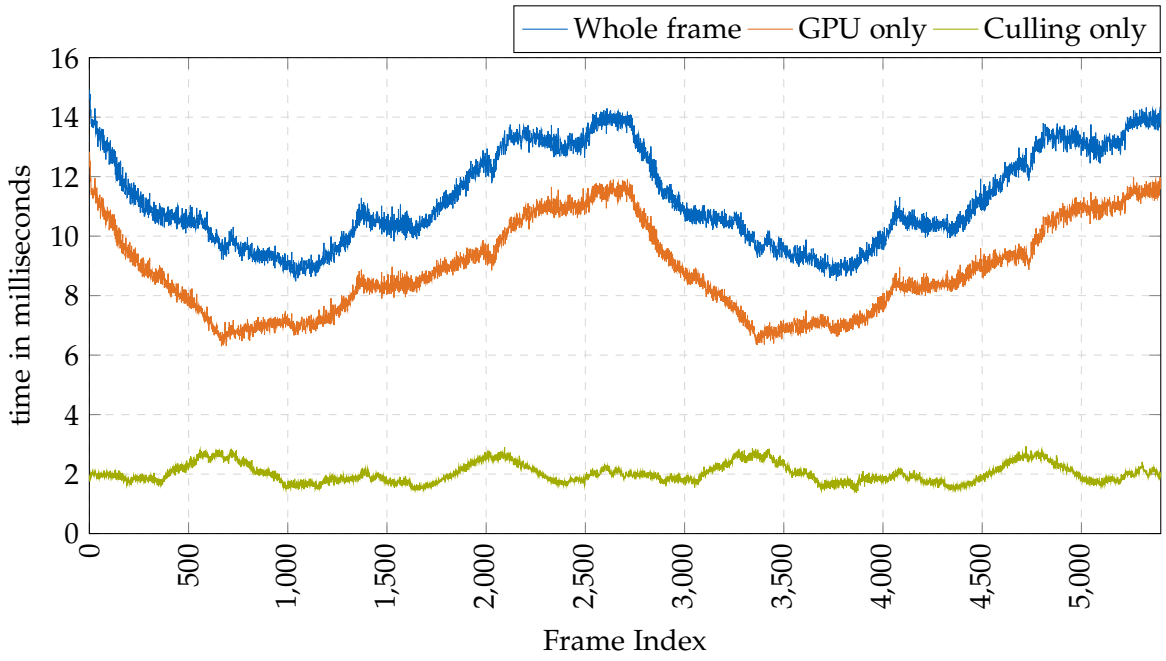


Figure 6.3: SMASK ON performance (10 run average)

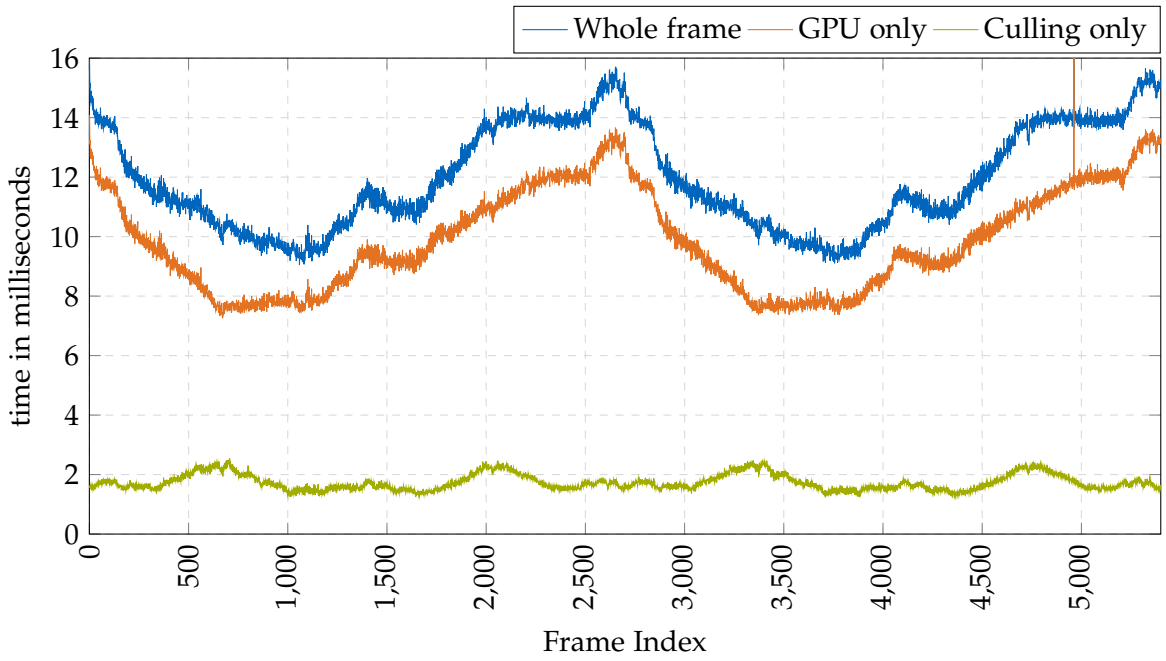


Figure 6.4: SFRUST ON performance (10 run average)

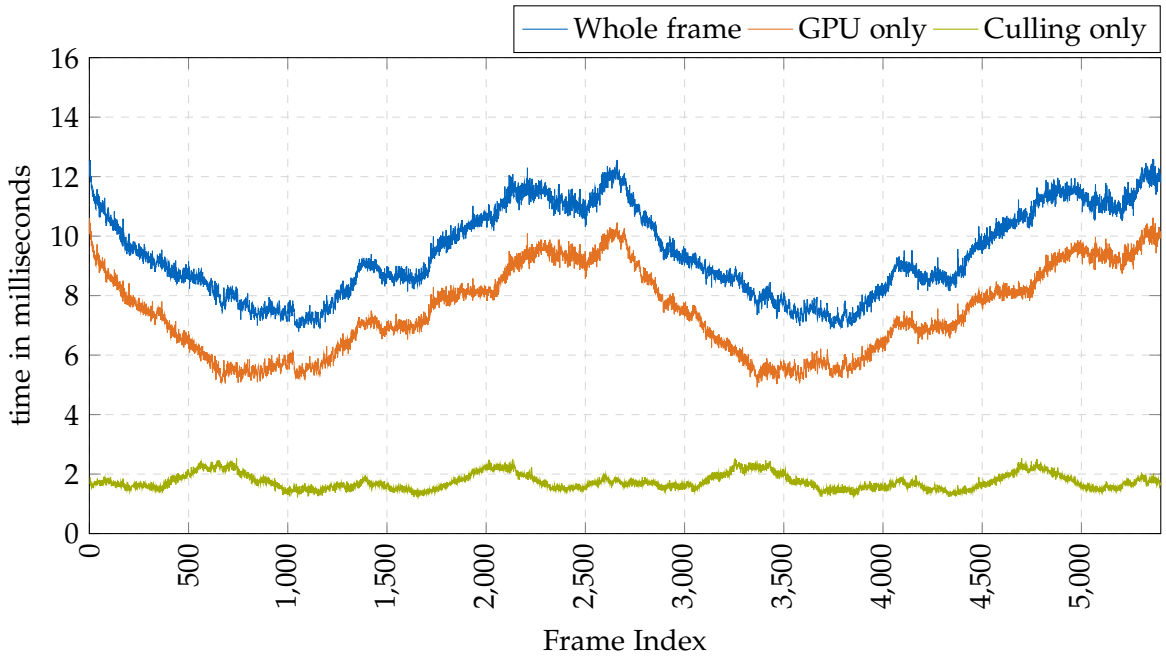


Figure 6.5: MVIEW ON performance (10 run average)

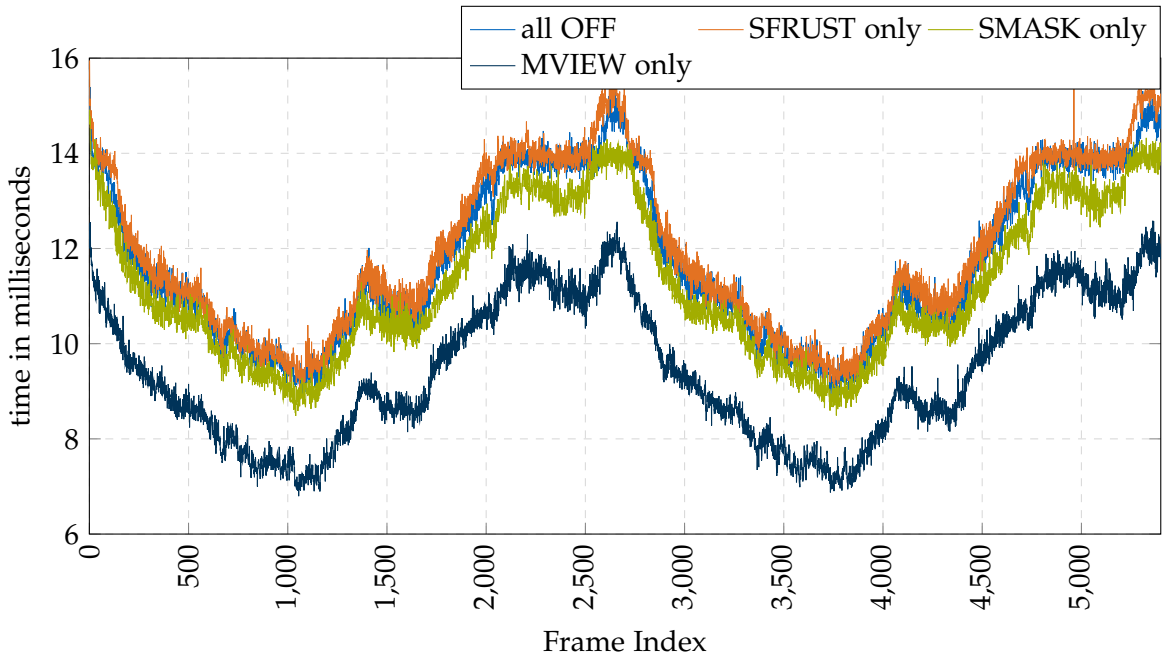


Figure 6.6: comparison (Whole frame time)

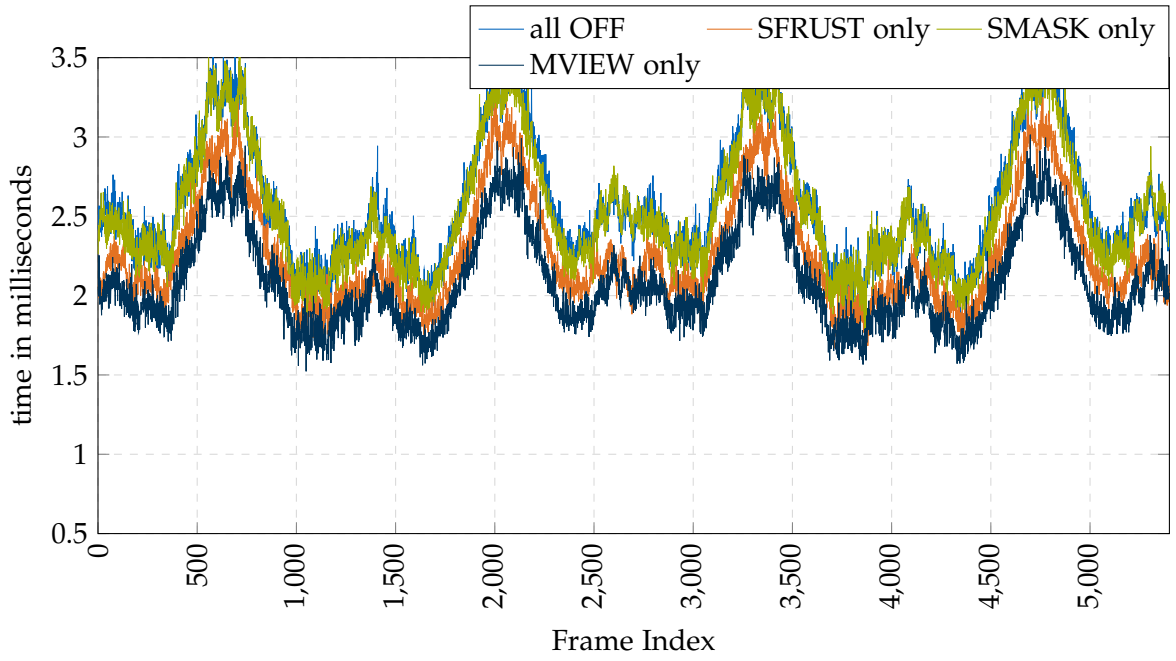


Figure 6.7: comparison (CPU time)

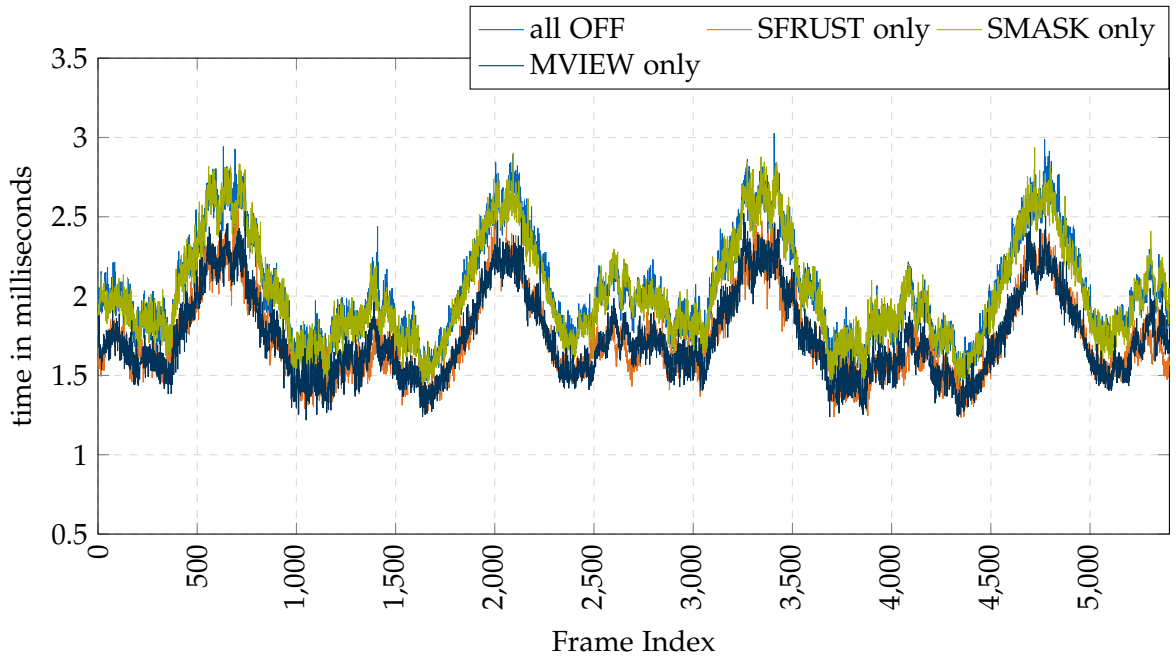


Figure 6.8: comparison (Cull time)

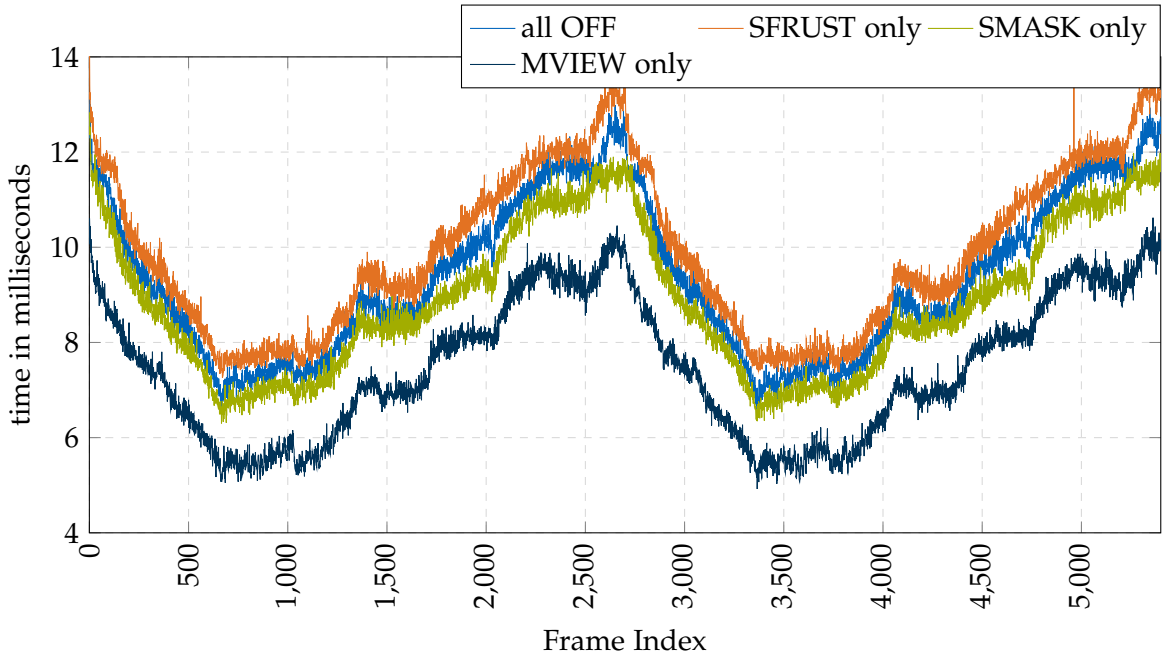


Figure 6.9: comparison (GPU time)

7 Outlook

[TODO]
[TODO: use glossary feature for abbreviations, rtvklib etc?]

List of Figures

List of Tables

Glossary

HMD head mounted display. 1, 5, 6, 23, 25, 29, 32, 37, 46

MFFR Monoscopic Far-Field Rendering (see section 4.3). vii, 24–28, 30–32, 43, 46, 47

Nsight VS NVIDIA Nsight Visual Studio Edition, a graphics debugging plugin for Microsoft Visual Studio[46]. 22, 23, 30, 46

Nvidia VRS Variable Rate Shadin, a technology by Nvidia Corporation to decouple raster resolution and shading resolution for geometry[47]. 33

OpenVR VR abstraction API developed by Valve Software. 4–6

RTG Yeet dab. vi, 3–7, 49

rtvklib RTG Echtzeitgraphik GmbH Vulkan library. vii, 4, 6, 22, 24, 30, 31, 46

SDE screen door effect. 1

Tachyon Real-time visualization engine currently in development at RTG Echtzeitgraphik GmbH. vi, 3–8, 10–13, 20–24, 27, 29, 30, 46

VR Yeet VR dab. 1–6, 14, 20–23, 26, 28, 29, 32, 49

WMR Windows Mixed Reality, a moniker for VR headset specifications introduced by Microsoft Corporation. 14

Bibliography

[1] Íñigo Quílez. "Efficient Stereo and VR Rendering". In: *GPU Zen*. Ed. by W. F. Engel. Vol. 1. Encinitas, CA: Black Cat Publishing Inc, 2017, pp. 241–251. ISBN: 9780998822891.

[2] Veikko Mäkelä. *VR headset resolution per eye comparison (CC BY-SA 4.0)*. 2019. URL: https://commons.wikimedia.org/w/index.php?curid=82765161.

[3] H. Vollmer. *RTG Echtzeitgraphik GmbH Homepage*. 2020. URL: http://www.echtzeitgraphik.de/.

[4] The Khronos® Group Inc. *Khronos Releases Vulkan 1.0 Specification*. 2016. URL: https://www.khronos.org/news/press/khronos-releases-vulkan-1-0-specification.

[5] Antony Vitillo. *What is Standalone Virtual Reality, and Why Are Enterprises Betting On It?* 2018. URL: https://www.isaca.org/resources/news-and-trends/isaca-now-blog/2018/what-is-standalone-virtual-reality-and-why-are-enterprises-betting-on-it.

[6] *IDC - AR & VR Headsets Market Share*. 2020. URL: https://www.idc.com/promo/arvr.

[7] A. Vlachos. "GDC2015: Advanced VR Rendering". In: (2015), pp. 51–65. URL: http://media.steampowered.com/apps/valve/2015/Alex_Vlachos_Advanced_VR_Rendering_GDC2015.pdf.

[8] T. Porter. *VR Optimization Tips from Underminer Studios*. 2017. URL: https://software.intel.com/en-us/articles/vr-optimization-tips-from-underminer-studios.

[9] J. Carmack. *Avoiding aliasing in VR*. 2016. URL: https://www.facebook.com/permalink.php?story_fbid=1818885715012604&id=100006735798590.

[10] *Was Sie über VR Performance-Guides für Unity-Projekte wissen sollten. | 3D Konfigurator 3D Animationen 3D Renderings*. 2018. URL: https://viscircle.de/was-sie-ueber-vr-performance-guides-fuer-unity-projekte-wissen-sollten/.

[11] N. Pettit. *VR Performance Guidelines for New Unity Projects*. 2017. URL: https://blog.teamtreehouse.com/vr-performance-guidelines-new-unity-projects.

[12] G. Guennebaud, B. Jacob, et al. *Eigen v3: a C++ template library for linear algebra*. 2010. URL: http://eigen.tuxfamily.org.

[13] U. Haar and S. Aaltonen. "SIGGRAPH 2015: Advances in Real-Time Rendering in Games: GPU-Driven Rendering Pipelines". In: (2015), pp. 10–25. URL: http://advances.realtimerendering.com/s2015/aaltonenhaar_siggraph2015_combined_final_footer_220dpi.pdf.

[14] C. Everitt. *single combined camera matrix*. 2015. URL: https://scontent-dus1-1.xx.fbcdn.net/v/t31.0-8/11334168_10154006919426632_2185539868454578065_o.jpg?_nc_cat=107&_nc_ohc=QEbGuUs7xiIAX_q3hjC&_nc_ht=scontent-dus1-1.xx&oh=4c8cb329ef851d5ab51161943910fed9&oe=5ECFE2F1.

[15] N. Whiting. *Oculus Connect 4 | The Road to Shipping: Technical Postmortem for Robo Recall: Superfrustum culling*. 2017. URL: https://www.youtube.com/watch?v=BZhOUGG45_o&feature=youtu.be&t=46m12s.

[16] V. Oddou. *VR and frustum culling - Computer Graphics Stack Exchange*. 2017. URL: https://computergraphics.stackexchange.com/a/4765.

[17] C. Everitt. *asymmetric eye matrix normalization*. 2015. URL: https://scontent-dus1-1.xx.fbcdn.net/v/t31.0-8/10460839_10154007978676632_3794989256420316318_o.jpg?_nc_cat=110&_nc_ohc=JuTLkrDwfTOAX_uUssk&_nc_ht=scontent-dus1-1.xx&oh=a3837266309d26af90584470482a1fea&oe=5EC2AD91.

[18] D. Beeler, E. Hutchins, and P. Pedriana. *Asynchronous Spacewarp | Oculus Developer Blog*. 2016. URL: https://developer.oculus.com/blog/asynchronous-spacewarp/.

[19] *Steam :: SteamVR :: Introducing SteamVR Motion Smoothing*. 2018. URL: https://steamcommunity.com/games/250820/announcements/detail/1705071932992003492.

[20] J. Hale. "Dual-Cone View Culling for Virtual Reality Applications". Bachelor's Thesis. Squareys, 10.04.2018. URL: https://squareys.de/downloads/bachelors-thesis-dual-cone-view-culling-for-vr.pdf.

[21] V. Vondruš. *Magnum Engine*. 30.10.2019. URL: https://magnum.graphics/.

[22] Vhite Rabbit. *Vhite Rabbit Website*. 8.01.2020. URL: https://vhiterabbit.org/#about.

[23] A. Overvoorde. *Vulkan Tutorial: Introduction*. URL: https://vulkan-tutorial.com/Drawing_a_triangle/Graphics_pipeline_basics/Introduction.

[24] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. "NVIDIA Tesla: A Unified Graphics and Computing Architecture". In: *IEEE Micro* 28.2 (2008), pp. 39–55. ISSN: 0272-1732. DOI: 10.1109/MM.2008.31.

[25] R. Sommefeldt. *AMD R600 Architecture and GPU Analysis*. 2007. URL: https://www.beyond3d.com/content/reviews/16/5.

[26] I. Cantlay. *Pascal VR Tech*. 2016. URL: https://developer.nvidia.com/pascal-vr-tech.

[27] R. Smith. *The NVIDIA GeForce GTX 1080 & GTX 1070 Founders Editions Review: Kicking Off the FinFET Generation*. 2016. URL: https://www.anandtech.com/show/10325/the-nvidia-geforce-gtx-1080-and-1070-founders-edition-review/11.

[28] S. Bhonde and M. Shanmugam. *Turing Multi-View Rendering in VRWorks | NVIDIA Developer Blog*. 2018. URL: https://devblogs.nvidia.com/turing-multi-view-rendering-vrworks/.

[29] L. Gallagher. "Radeon Software Crimson ReLive". In: (2016), p. 19. URL: https://awesome.nwgat.ninja/crimson/Radeon_Software_Crimson_ReLive_%5BNDA_Only_-_Confidential%5D_v4.pdf.

[30] K. Jez. *GPUOpen: AMD LiquidVR MultiView Rendering in Serious Sam VR*. 2017. URL: https://gpuopen.com/amd-liquidvr-multiview-rendering-in-serious-sam-vr/.

[31] S. Willems. *Vulkan Hardware Database by Sascha Willems: Reports (extension feature multiview)*. 2020. URL: http://vulkan.gpuinfo.org/listreports.php?extensionfeature=multiview.

[32] S. Willems. *Getting a Vulkan application up and running on a low-spec device with buggy drivers*. 2019. URL: https://www.saschawillems.de/blog/2019/03/08/getting-a-vulkan-application-up-and-running-on-a-low-spec-device-with-buggy-drivers/.

[33] JMC47. *The Current State of Dolphin on Android: Running Dolphin on Android in 2018*. 2018. URL: https://dolphin-emu.org/blog/2018/08/14/state-of-android/.

[34] W. Andermahr. *Nvidia Pascal: 21 Prozent mehr Leistung in iRacing durch SMP*. 2016. URL: https://www.computerbase.de/2016-09/pascal-smp-iracing-benchmark/.

[35] F. Serrano. *Multiview on WebXR*. 2019. URL: https://blog.mozvr.com/multiview-on-webxr/.

[36] J. de Vries. *Learn OpenGL: Stencil testing*. 2014. URL: https://learnopengl.com/index.php?p=Advanced-OpenGL/Stencil-testing.

[37] D. Di Donato, R. Palandri, and R. Vance. *High quality mobile VR with Unreal Engine and Oculus*. 1.03.2017.

[38] R. Palandri and S. Green. *Oculus Developer Blog: Hybrid Mono Rendering in UE4 and Unity*. 2016. URL: https://developer.oculus.com/blog/hybrid-mono-rendering-in-ue4-and-unity/.

[39] *Monoscopic Far Field Rendering*. 2016. URL: https://docs.unrealengine.com/en-US/Platforms/VR/DevelopVR/MonoFarFieldRendering/index.html.

[40] P. Lapinski. *Vulkan cookbook: Work through recipes to unlock the full potential of the next generation graphics API–Vulkan / Pawel Lapinski*. Birmingham, UK: Packt Publishing, 2017. ISBN: 9781786468154.

[41] R. Palandri, S. Gosselin, and C. Pruett. *Oculus Developer Blog: Optimizing Oculus Go for Performance*. 2018. URL: https://developer.oculus.com/blog/optimizing-oculus-go-for-performance/.

[42] *Foveated Rendering on the VIVE PRO Eye*. 2019. URL: https://zerolight.com/news/tech/foveated-rendering-on-the-vive-pro-eye.

[43] A. Vlachos. "GDC2016: Advanced VR Rendering Performance". In: (2016), pp. 18–24. URL: https://alex.vlachos.com/graphics/Alex_Vlachos_Advanced_VR_Rendering_Performance_GDC2016.pdf.

[44] *VRWorks - Multi-Res Shading*. 2016. URL: https://developer.nvidia.com/vrworks/graphics/multiresshading.

[45] S. Cleveland. "NVIDIA FCAT VR Reviewer's Guide". In: (2017), p. 35. URL: https://international.download.nvidia.com/geforce-com/international/pdfs/NVIDIA_FCAT_VR_Reviewer's_Guide_Public.pdf.

[46] *NVIDIA Nsight Visual Studio Edition*. 2013. URL: https://developer.nvidia.com/nsight-visual-studio-edition.

[47] *VRWorks - Variable Rate Shading (VRS)*. 2018. URL: https://developer.nvidia.com/vrworks/graphics/variablerateshading.