DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Evaluation of rendering optimizations for virtual reality applications in Vulkan

Paul Preißner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

Evaluation of rendering optimizations for virtual reality applications in Vulkan

Evaluation von Renderoptimierungen für Virtual Reality Anwendungen in Vulkan

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

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

Garching bei München, February 15 2020 Paul Preißner

Acknowledgments

Abstract

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

Kurzfassung

Contents

1 Introduction

1.1 Briefly - what's the goal

Real-time rendering is by no means a new research topic. Naturally developers and researchers have devised shortcuts and optimizations in both hardware and software as far back as the first applications in video games, visualization - scientific or otherwise - and simulation. There are methods and algorithms for almost any type of hardware, rendering technique and application scope. One more recently resurged trend is virtual reality (VR). The realization of comparably affordable, comfortable and hassle-free virtual reality headsets like the Oculus Rift series, the HTC Vive, Windows Mixed Reality design or the recently launched Valve Index led to a newfound vigor for the technology among consumers. Similarly, businesses as well as scientists are increasingly looking to VR for various fields of research, marketing opportunities, engineering support and more.
However, virtual reality poses challenges and requirements not commonly seen with traditional real-time applications on monoscopic screens. And while goals such as stable motion tracking and low latency, high performance rendering are as old as early VR HMDs, this recent resurgence has prompted much more active search for possible solutions. Among these requirements is the need for high framerates to give the user a better feeling of visual fluidity and to avoid motion sickness. VR applications need not only render at high framerate, they also need to do so at higher resolutions than common flat screens. As the much lower viewing distance and added lens distortion means pixels appear much larger to the user, the only way to combat visual aliasing and the so called screen door effect is to significantly increase pixel count and density.

Similarly to the opening example in [SRC: GPUZEN1 article A], let's assume a traditional console video game running at Full HD resolution and 60 frames per second, a common sight for 8th generation consoles. Meanwhile the same game running in a VR headset may require resolutions and framerate ranging from 1200 pixels per axis per eye at 90 frames per second for entry level headsets up to or even exceeding 2000 pixels per axis per eye at 120 frames per second for high end devices like the Pimax 4K - not to mention image warp commonly requires an even higher internal rendering resolution by a factor of around 20% or more. In the entry level case that means a resolution increase of roughly 1.6x and a framerate increase of 1.5x for a combined 2.4x power requirement compared to the console baseline, while in the high end case it is roughly 4.6x resolution and 4x framerate uptick for a combined 9.2x power requirement in comparison.

[TODO: HMD res illustration]

While this is very rough napkin math and real applications scale differently to a degree due to a multitude of factors, it paints a clear picture: real-time virtual reality rendering necessitates vastly faster rendering than traditional screens. It obviously follows that VR rendering should take advantage of as much optimization as possible to attain that goal. There are many established options that are known to improve performance for various if not most application types, with some of these options being de-facto standard features in popular rendering engines. There is also, however, a range of optimizations specific to virtual reality or more generally stereoscopic rendering, and often there is less research and documentation available yet, due in part to aforementioned recent rise in popularity and prior lack of interest.

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

1.2 Industry collaboration

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

1.3 Technical foundation

From this industry collaboration follow influences when choosing the technical foundations of the thesis. In an effort to represent a realistic use case, an engine with actual production usage target was chosen as opposed to a purely synthetic "test vehicle", namely RTG's own Tachyon. The engine and its structure is further explained in chapter 2.
The graphics API and library of choice for this thesis then is the Khronos Group's Vulkan.

Vulkan is a low level graphics API officially created in 2014 and formally released in early 2016 with the promise of increased flexibility and reduced overhead compared to prior graphics APIs as well as compatibility with a wide range of hardware architectures and operating systems. On the flipside, the verbose and low level nature of the API brings increased development effort and means that these promised improvements can only be leveraged with sufficient care and caution by developers.
More specifically for Tachyon and this thesis, the goal is to enable fast rendering of complex scenes with very large object counts, all while driving virtual reality headsets at high resolution and framerate. Vulkan promises a good fit for this purpose. Furthermore Vulkan is seeing a steady growth in developer interest and industry adoption, but with documentation, resources and API-specific research still being lackluster in some ways. This thesis hopes to add useful insights to this pool.
Lastly - as a sort of tease to the future - both Vulkan and the VR optimizations presented here are 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.

2 The RTG Tachyon Engine

2.1 The Engine

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

2.1.1 Render setup

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

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

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

but also holds data specific to individual objects in the virtual world, such as transforms or bounding geometry.

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

2.1.2 Render loop

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

```
void VKRenderer::Update()

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

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

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

Figure 2.1: Renderer update function

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

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

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

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

Figure 2.2: Renderer frame render function

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

2.1.3 VR render loop

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

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.

What's more, as seen in Figure 2.4, [TODO: briefly describe `OpenVR::RenderFrame`].

[TODO: overview illustration aka flowchart?]

```
void OpenVR::RecordCommandBuffers(uint32_t passId)

	for (uint32_t i = 0; i < mCommandBuffers.size(); ++i)
	{
		VkCommandBufferBeginInfo beginInfo = {};
		[...]
		vkBeginCommandBuffer(mCommandBuffers[i], &beginInfo);

		VkRenderPassBeginInfo renderPassInfo = {};
		[...]
		vkCmdBeginRenderPass(mCommandBuffers[i], &renderPassInfo, VK_SUBPASS_CONTENTS_I

		for (auto renderPipeline : mRenderPipelines)
		{
			renderPipeline->RecordDrawCommand(mCommandBuffers[i], passId);
		}

		vkCmdEndRenderPass(mCommandBuffers[i]);
	}
```

Figure 2.3: OpenVR command buffer recording function

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

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

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

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

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

Figure 2.4: OpenVR frame render function

3 Stereo Rendering Optimization - Input reduction

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

(Hierarchical) Frustum culling

The following options build on top of the regular frustum culling concept. In this the objects in the scene are checked against a camera frustum whether they are inside or outside of it or intersecting with the surface of the frustum. The checks themselves can be generally optimized in various ways, regardless of whether stereoscopy is desired or not. Often only an object's bounding geometry is checked, collections of objects can be precomputed so larger numbers may be discarded at once. An advanced option of culling is to delegate the calculations into a GPU compute shader so potentially less data needs to be transferred from the CPU per frame and much higher vector/matrix calculation is gained in exchange for slower branching. Some modern renderers also do very granular culling like bitmasked checks of precomputed triangle sets, as seen in Ubisoft's Anvil Next engine used in Assassin's Creed Origins [SRC: Anvil 64b triangle cull]. Another optional layer of the culling process is to maintain hierarchical container structures for the scene objects so larger numbers can be discarded or included early on. For all those options the goal is the same, to get as result the list or buffer of objects visible by the given camera frustum in the scene.

Frustum culling in Tachyon

The frustum culling approach in Tachyon is fully CPU-based and utilizes pre-computed hierarchical draw buffers. More specifically, at startup the scene is divided into a coarse grid of `chunk`s where each grid possesses an octree. Also at startup, a thread pool with the current number of hardware threads is created. At asset load time, these octrees are populated with the loaded objects through optimistic size-aware insertion. Each cell of a tree

then precomputes a draw buffer containing a combined draw call for all objects associated with that cell. These buffers can be recomputed at any time, but the operation should be avoided at runtime as it incurs costly CPU to GPU transfers. In a culling pass, first all `chunk`s within a certain draw distance radius of the camera are chosen, so there is an additional very primitive distance culling taking place. Then each `chunk` submits a culling call using its tree to the thread pool. Each such call works as follows:

[TODO: brief rundown of `SceneChunk::FrustumCull`]

One problematic area remains, however, 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:

[TODO: image of scene with high object density and distance]

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 to 300 milliseconds and beyond on the test machine, while issuing the calls front to back results in frametimes of 11 milliseconds or less. This stark contrast in performance highlights the importance of good Z ordering.

[TODO: Put the following in or not? *At this time, rtvklib does not successfully employ such call reordering. A conservative and thus only moderately beneficial front to back ordering is in the works but due to the nature of the octree cells' pre-recorded command buffers, overdraw cannot be avoided without a significant rewrite of these pre-recordings.*]

3.1 Superfrustum Culling

3.1.1 Theory

The basic idea behind Superfrustum culling is to essentially do regular frustum culling despite rendering into two cameras, one per eye. The naive way of extending the frustum concept to a stereoscopic camera is to add a second frustum so there is one per eye, then perform the culling check for both frusta and merge the results.
As is easily visible from [TODO: illustration of stereo frusta] though, the spatial proximity

of of these two frusta leads to a large overlap volume, especially as field of view increases with more advanced headsets. One possible strategy to leverage more performance when culling two eyes is a so-called Superfrustum, assuming the frustum is the common six sided trapezoid. Cass Everitt of Facebook LLC, formerly Oculus LLC [TODO: check correct corp names] has suggested this approach at Oculus Connect 5 [TODO: check conf name] and provided computation sketches at [SRC: everitt fb]. 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. [TODO: Name Surname] of Silicon Studios offered a generalized way to compute this recession for un-mirrored eye orientation, while Everitt has extended his sketches by an asymmetry normalization. Both of these are important to consider as virtual reality headsets can have slightly canted and asymmetrical lenses, either by design or by manufacturing tolerance. Ignoring these two corrections may still result in a sufficient superfrustum if computed conservatively, but should be included for fully correct setups. 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 will be negligible.

[TODO: illustrations by Everitt and Silicon]

3.1.2 Estimated impact

The impact of using a Superfrustum will depend very specifically on the type of frustum culling done and 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 though, including all vertex transformations, pixel shading and so forth. The specific impact in the case of Tachyon is elaborated on in section 5.2.
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 of course present another speedup

limiter.

3.1.3 Implementation specifics

To facilitate 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 Silicon's way, outlined in [TODO: pseudo code]. 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 that are then sent to both eyes.

3.2 Round Robin Culling

Another culling variant specific to stereoscopy uses the round robin principle. The concept again 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 again. As a result a lot of the edge data of the image is either considerably squeezed 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 in most cases encompass the entire stereo-visible volume. So it naturally follows that only culling for one of the eye frusta would already give a sufficient representation of the actually visible scene.

However, 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 [TODO: and/or Time Warp?] and SteamVR's reprojection features. It follows that 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.

[TODO: small illustration?]

3.3 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. [TODO: student's name] attempts to demonstrate the method in [SRC: conical cull BA paper] albeit with limited success. That proof of concept does show the validity of the method though. The traditional six sided trapezoid frustum is replaced by a cone encompasses a volumetric projection of the view through each respective lens. For a more geometry-bound GPU, this method will provide a small relief.

[TODO: more specific description of approach from paper] [TODO: small illustration]

Merging approaches

A convenient side effect of these three presented optimizations is that they can in part be merged. It is conceivable, for example, 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.

4 Stereo Rendering Optimization - Effort reduction

This second 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.

4.1 Multiview stereo rendering

4.1.1 Theory

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 [TODO: illustration of general graphics pipeline] in mind, it's 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 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, and very quickly surfaced as an idea after the introduction of the Nvidia Geforce 8 and ATi Radeon HD 2000 series in 2006 brought unified shader architectures to the market [SRC: arch whitepaper]. 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 and - in more recent vendor specific terms - Nvidia calling it Single Pass Stereo, Simultaneous Multi-Projection and Multi-View rendering and AMD calling it LiquidVR multiview [SRC: dev page for each]. 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 synchronization barriers are essentially halved and all necessary writes are performed in a single go. As an addition, *hardware* multiview rendering is to only perform those pipeline stages multiple times that 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 though, which constraints it to more modern GPU architectures built with it in mind [SRC: Nv GDC 2015, etc]. Nvidia could be considered the main progress drivers for this, having pushed the technology from parallel geometry projection in Maxwell's Multi-Projection Acceleration to Pascal's SMP which adds lens-matched shading to better approximate the lens shape and 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 [SRC: T MV dev entry].

[TODO: illustration]

The small tradeoff then is that all relevant view data for each viewport has 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't be altered for each eye as they are processed in a single pass. [TODO: is there any warping happening? -> very slight artifacts possible]

Going by the user-maintained Vulkan Hardware Database [SRC: vulkan.gpuinfo.org/listreports.php?extensi
The major GPU vendors specify multiview support in their architectures as follows:

- Nvidia: hardware multiview from the Maxwell generation and newer, software support from Kepler onward
- 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 [SRC: reports?].

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 may not be very reliable, however, as 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.

4.1.2 Estimated impact

Impact of this extension is highly dependent on the specific workload, the used graphics hardware and renderer structure.

[TODO: AMD LiquidVR numbers, Anand SMP tests, Nv MV dev blog]

4.1.3 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

4.2.1 Theory

Introduced to the mass market with 3Dlabs' Permedia II in 1997 and widely adopted since then, all modern graphics chips support a nifty feature called the stencil buffer. This buffer uses low bit integer values - commonly 8 bits per pixel for a depth of 256 [SRC: learnopengl.com/Advanced-OpenGL/Stencil-testing] that 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 [SRC: Valve talk] 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, 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. Then 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 can essentially be imagined exactly as the classic idea of stencil mask when painting surfaces, where 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.

[TODO: illustration]

4.2.2 Estimated impact

The performance gain then naturally scales well with both increased fragment shader bias of the per-frame workload as well as with the HMD's blind areas. Valve's Vlachos in his GDC talk showcased gains of 17% lower GPU frametimes for the company's Aperture Labs VR showcase scene using an HTC Vive headset. [SRC: Valve talk] 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.
[TODO]

4.2.3 Implementation specifics

Outfitting Tachyon for stencil masking required the addition of the entire stencil stack as the engine did not use the feature in any capacity yet. An inherent advantage of this is that it allows to render the stencil mask only once at startup and keep reusing it every frame without additional render load for the mask.
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. What's more, the involved Vulkan pipelines need stencil operation states defined in their `VkPipelineDepthStencilSta` with the operations set as in Figure 4.1 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 our stencil buffer remains unmodified and expensive writes are avoided.

```
//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;
                [TODO: check what both masks do]
        stencilOpState.reference = 1;
```

Figure 4.1: Stencil operation bits set for default material Pipeline

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 as screenspace normalized list of vertices representing the ideal mask mesh for the current HMD as known to OpenVR [TODO: check "ideal"ity for various HMDs]. The obvious benefit of this 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 coming OpenVR-enabled headsets. Unfortunately, however, OpenVR in a few instances returns empty masks, such as for the Oculus Rift CV1. For those 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 pipeline, PBR pipeline et cetera only do stencil test compares, no writes, they perform color writes and rasterizer face culling, all things the stencil mask pipeline should do differently. So a separate stencil pipeline is introduced with stencil operation state set like in Figure 4.2, 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 cleaner separation the described setup was used in Tachyon's current implementation.

```
//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;
                        [TODO: check what both masks do]
        stencilOpState.reference = 1;
```

Figure 4.2: Stencil operation bits set for stencil mask Pipeline

4.3 Monoscopic Far-Field Rendering

4.3.1 Theory

Monoscopic Far-Field Rendering (MFFR) is a curious approach strongly aligning with 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 in the early days of Rift CV1's retail launch by [TODO: Oculus? Epic? check timeframe] at developer keynotes.
Understanding the concept requires some explanation of the technical and visual background. Proper 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 [TODO: word this more scientifically?] and without it, the brain has a hard to impossible time determining the depth of and at which a certain object or surface lies. Regular stereo rendering obviously 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, and that's one aspect MFFR exploits, as distance grows, stereo separation shrinks. In infinity, separation would be infinitely small, but even at more reasonable distances separation becomes so small that even with good vision it becomes 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. What this means is that at a certain distance from the virtual camera, stereo separation will shrink to less than a full screen space pixel once projected. Obviously if the difference can physically not be displayed by the HMD, it seems a waste of resources to still render both eyes.
Mono Far-Field Rendering thus opts to skip the second view during rendering of the name-giving far field of objects. The hope here is that only rendering a single view past a certain distance reduces rendering load without the user noticing the theoretical loss in accuracy. This approach has caveats unfortunately. For one, 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 recognition. 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 at some point and was recommended for example for certain types of mobile VR experiences with very limited GPU power, but curiously has been removed from the engine in update 4.20 without further explanation. An odd decision surely, as Oculus LLC developer blog posts prior to the removal indicated continued optimization efforts such as added compatibility with UE4's multiview path.

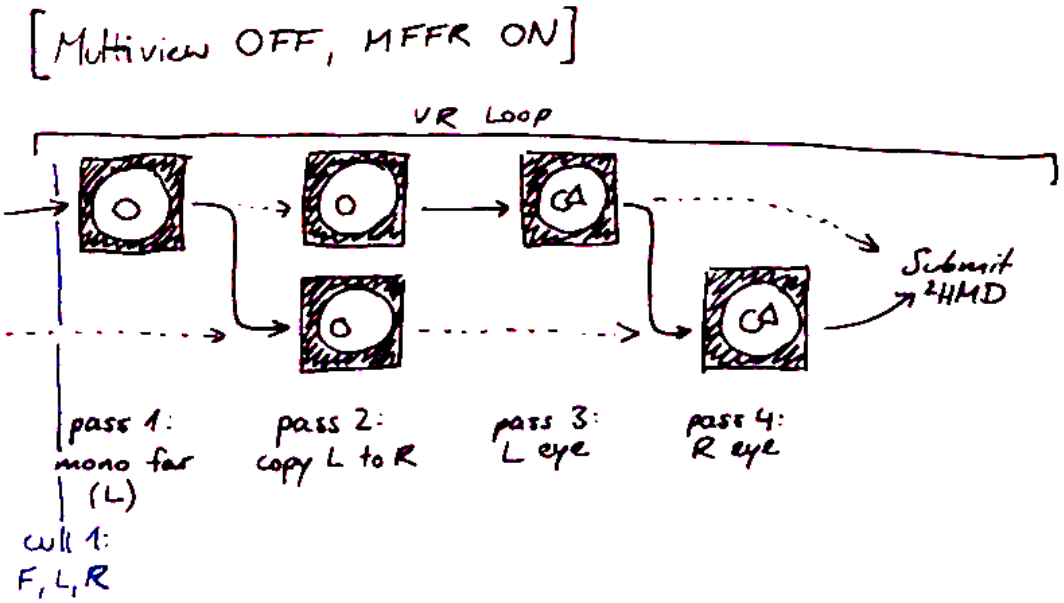

Figure 4.3: Far-First MFFR flowchart [TODO: redo properly]

4.3.2 Estimated impact

The makeup of the scene itself will also affect effectiveness of the solution. As cautioned by Oculus LLC in their developer reference on Mono Far-Field Rendering [SRC: MFFR dev entry], 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 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 that would be contained within that second pass.

[TODO]

4.3.3 Implementation specifics

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 two render passes and with their own respective drawbacks and advantages.

Far-first approach

One way of implementing MFFR is to do the far pass first in each frame, then followed by a near pass, both using the same framebuffer as illustrated in Figure 4.3. At the start of the frame, the framebuffer is cleared - only color and depth necessary - after which the far-field command buffers are submitted and executed. This pass can render directly into the buffer of one eye, then 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 though. 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't 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 also translate the far projection by 0.5 so the uni-cube is split between the two fields.
If neither is done, the depth values of far-pass objects may appear to be closer than those of near-pass objects, effectively leading to some near-field objects being drawn *behind* far ones. The depth buffer clear option has to benefit of utilizing full projection precision and correct triangle clipping. The matrix squashing option avoids an extra clear but will mean vertices otherwise clipped outside of the uni-cube will be pulled into the uni-cube and not skipped.
The overall advantage of this far-first approach then is that it yields a full precision far-field depth buffer which may be useful for stereo interpolation as briefly described in the next section 4.3.4. The disadvantage then is that 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 outpace the additional cost of overdraw will heavily depend 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 also avoid more costly compositing methods.

Near-first approach

The obvious alternative to rendering the far pass first is of course to render the stereo near volume at the start of each frame [TODO: mffr near first flowchart]. A key difference to the former approach is that for near-first to work correctly, this first pass also needs to flag then stencil buffer pixel when a opaque fragment is within the uni-cube and a color value is written. Then after the stereo pass, each eye will contain 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. The constraint, however, is that the far pass then also needs to sample from left to right eye separately as a straight buffer copy cannot work anymore. The obvious advantage of this approach over far-first is that early Z discard takes full effect, not to mention stencil masking may reduce the leftover areas even more. The downside is that it needs to perform additional stencil writes every frame and potentially costly sampling operations.

rtvklib MFFR

In this case, the plan was to do far-first MFFR. It eventually became evident that for the given test scenario, this approach proved unsuccessful for various reasons. Before that, the theory. In the first step of each VR frame, a monoscopic render pass would render the far clip volume's color and depth values into the index 0 layer of the framebuffer. The next step would copy that layer to index 1 as well. In the final step then the regular stereoscopic render commands would be 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 sounds mostly straight-forward and both described ways are possible, implementing either in a Vulkan renderer is no trivial task and required 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 then cover the volume from near plan 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 VR render pass being that it foregoes the multiview extension
- add an additional set of Vulkan `VkCommandBuffer`s into which the draw commands of the far frustum cull set are to be recorded
- add an additional set of `VkSemaphore`s to synchronize the two render passes and create them in `CreateSyncObjects()` during initialization
- add an additional `VkCommandBuffer` for the layer copy operation
- at initialization time of the VR render target, pre-record this copy command buffer 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 transition 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; after that block 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`, the latter of which the stereo submit is set to wait on
- in VR render target's camera setup, construct the far field frustum projection matrix as outlined in [SRC: Vulkan Cookbook] - transposed as Tachyon still retains OpenGL matrix format for a few matrix types
- add an additional set of camera data struct and camera index
- in the render target's `UpdateCameras()` call, update the far field volume's view projection matrix by transforming aforementioned projection matrix by the current HMD pose and write that updated matrix to the camera data buffer on the GPU

[TODO: sort this list a bit better? normal text some of it?]

Some issues with this implementation do prevail and make it unfit for productive deployment in Tachyon yet, unfortunately. For one, the projection and view matrices used for the far field remain incorrect and sufficiently accurate projection math could not be determined in time, so head rotation leads to incorrect separation effects on the horizontal axis. And while the lack of pure separation at little to no sideways tilt may be acceptable and not easily noticed, tilting the head means more severe separation mismatch as the spatial disconnect is expanded from being mostly horizontal to being horizontal and vertical.
Another issue is that Vulkan render passes by default discard fragments at the their end if depth testing is enabled and the depth value of said fragment equals 1.0 and is thus on the far plane. This meant any color values written by the far pass and not overwritten by the near pass would be discarded right at the end of the stereo pass and before presentation. A workaround for this was to clear the depth buffer at the start of the stereo pass not to 1.0 but to 0.999999 (22 of 23 mantissa bits set to 1), the closest value a IEEE-754 float can reach below 1.0. With this tweak, color values were correctly composited together without any visible loss at the near field's split plane.
Lastly, and most importantly, performance simply was not up to par. On the i7 and RTX2080 test system, *enabling* MFFR with a sufficiently far split distance yielded a 34% performance *loss*. The culprit for this most likely two-fold. One factor is the detrimental Z ordering of the submitted draw commands, as much of the far field color buffer is overwritten 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 warps to increase GPU utilization and thus performance. As such, the rtvklib MFFR implementation as presented here remains beneficial only in theory and will require further work - if not a complete overhaul to near-first approach - to result in real performance gain.

[TODO: example picture comparison]

4.3.4 MFFR Variant: Depth Shift

MFFR in its basic version as described above obviously completely foregoes separation beyond the split distance and as such that split distance has to be set relatively far back to minimize the visual inaccuracy. So naturally it seems profitable to attempt to try and reduce that split distance closer to the camera by artificially 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, it is logical to approximate small amounts of separation based on that depth buffer. Instead of simply copying layer 0 to layer 1 after the far field pass, one can do a sampling pass over layer 1 from layer 0 or a post-processing pass after the copy operation and slightly shift 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, however it is unclear whether the savings outweigh the additional processing cost and this thesis did not explore MFFR beyond the base variant.

[TODO: shift math or geometric illustration]

4.3.5 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 underlying for earlier chapters like the Round Robin Culling. 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 also 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 single frame temporal reconstruction, although a single previous frame may not be enough for good results, or a simple frame interpolation.
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. And just as with Round Robin Culling, the same potential artifacts and issues are present here.

4.4 Foveated Rendering

4.4.1 Theory

This technique lends its name from the *fovea centralis*, a spot in the center of the human 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 described in the Oculus Go optimization guide [SRC: developer.oculus.com/blog/optimizing-oculus-go-for-performance/], the image produced by a VR renderer is warped by the VR compositor to more 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 commonly significantly less sharp than aforementioned foveal vision. So it seems logical that the outer rim of the rendered image could be rendered at much lower pixel density without sacrificing a *humanly noticeable* amount of detail. Exactly that is the concept of foveated rendering, to decrease resolution of the edge regions of the image while rendering the central area at full resolution. There is one big constraint, however. 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.

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 to the perceived image quality. And the resolution of peripheral areas may not be reduced as much as with "true" foveated rendering so even when the user briefly looks at such an area, the perceived loss is not as distracting. This compromise form of the technique is commonly called Fixed Foveated Rendering.
Given eye tracking of some adequate sort is available, "true" FR can be performed, where in each frame the current focus position of each eye is queried and the virtual view matrix updated accordingly. This makes the rendering setup more complex, of course, as the high pixel density area can shift to any location within the image, but it possibly 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.

[TODO: illustration]

4.4.2 Radial Density Mask

A somewhat related technique is called radial density masking as also shown by Valve's Alex Vlachos in his [SRC: Valve GDC talk]. The goal is the same as with FR, but the approach is different. Instead of reducing the theoretical rendering resolution of the peripheral ring, a mask is overlaid that has the render pipeline skip a certain pattern of pixels in that ring. This could, for example, be done by marking a checkerboard pattern of pixels in the stencil buffer so the pixel shaders 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 way would allow to approximately match a relative 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.

[TODO: illustration by Vlachos]

Adaptive resolution

The described methods of FFR, FR and RDM can of course be combined further with another, by now almost universal, optimization compromise: dynamic resolution. While monitoring GPU load or frametimes and framepacing, the resolution of not just the central fovea but obviously the peripheral regions could be reduced or increased within given bounds as these metrics allow. This can stabilize performance at the cost of some visual 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.

Relevancy of GPU architecture

The previous sections describe various methods, but the choice of which 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, FR 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 in parallel in a number of set tiles as opposed to traditional GPUs which execute render commands in sequence and can only work 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 fovea at lower resolution, 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. On those, rendering frame regions at different resolutions requires multiple render passes in sequence with a subsequent composition pass that writes the multiple layers into a single frame. This sequential process incurs additional overhead and may not yield ideal core utilization if the low resolution pass does not provide enough work for a large number of shader cores. 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 though to reduce aliasing.

[TODO: illustration Tile vs Whole?]

5 Performance impact

5.1 Explanation - Performance measurements

5.1.1 Compilation parameters, system specifications

[TODO: compiler settings, test machines, HMDs]

5.1.2 Measured metrics

[TODO: Framerate, frametimes, framepacing?, resource usage cpu threads, gpu, memory, cull cells count, polycount, overdraw estimate??, frame count, timestamp, virtual camera rails pos/rot, etc]

5.2 Benchmark results

5.2.1 Individual impact

[TODO]

5.2.2 Combined and partially combined impact

[TODO]

5.2.3 Interpretation

[TODO]

6 Outlook

[TODO]

List of Figures

List of Tables