DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

An evaluation of culling methods in VR? Rendering optimizations for Vulkan-based VR applications?

Paul Preißner

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics: Games Engineering

An evaluation of culling methods in VR? Rendering optimizations for Vulkan-based VR applications?

...

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

Kurzfassung

Contents

1 Outline

1.1 Introduction

1.1.1 Briefly - what's the goal

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

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

Thus, this paper aims to collect several of these VR optimizations, implement a subset of them in a real engine and then benchmark and analyse the impact each has on performance. The goal is to have a better collection and understanding of the various methods to use and how much effort may be warranted for each.

1.1.2 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.

1.1.3 Technical foundation

From this industry collaboration follow influences when choosing the technical foundations of the thesis. In an effort to represent a realistic use case, an engine with actual production usage target was chosen as opposed to a purely synthetic "test vehicle", namely RTG's own Tachyon. The engine and its structure is further explained in [chapter 2 - Tachyon Engine]. The graphics API and library of choice for this thesis then is the Khronos Group's Vulkan. Vulkan is a low level graphics API officially created in 2014 and formally released in early 2016 with the promise of increased flexibility and reduced overhead compared to prior graphics APIs as well as compatibility with a wide range of hardware architectures and operating systems. On the flipside, the verbose and low level nature of the API brings increased development effort and means that these promised improvements can only be leveraged with sufficient care and caution by developers. More specifically for Tachyon and this thesis, the goal is to enable fast rendering of complex scenes with very large object counts, all while driving virtual reality headsets at high resolution and framerate. To this end, Vulkan promises a good fit. Furthermore Vulkan is seeing a steady growth in developer interest and industry

adoption, but with documentation, resources and API-specific research still being lackluster in some ways. This thesis hopes to add useful insights to this pool. Lastly - as a sort of tease to the future - both Vulkan and the VR optimizations presented here are compatible with many ARM-based SoCs, 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 Prior research [TEMP]

2.1 Culling methods research

2.1.1 Assorted sources

1. Experiments in GPU-based occlusion culling (Anagnostou, Interplay of Light) [1]:
tldr CPU occlusion culling is very coarse as the CPU isn't great at rasterising. GPU can rasterise efficiently and supports hardware occlusion queries. Problem: "drawcall level" granularity, needs query and fence around every drawcall, thus can't handle instancing very well. Other problem: readback to CPU necessary, to avoid stalling use previous frame occlusion info for current frame. Thus popping likely for fast moving objects or camera. Other solution: render low res occlusion buffer on GPU, produce mip chain, determine screen-space size of each prop and compare vs appropriate mip-level of buffer. Still needs CPU readback though, unless... compute shader in modern API: "In this case I will be using a compute shader to perform the occlusion tests, producing a list of visible props that I will then be consuming on the GPU, avoiding the CPU roundtrip."

2. Why Frustum Culling Matters, and Why It's Not Important [2]:
tldr disregardm, oversimplified explanation of frustum culling.

3. Overview on popular occlusion culling techniques [3]:
tldr temporal culling methods are prone to artifacting with fast object or camera movement, CPU culling "is the most efficient, forward-looking approach", then again the article is written by someone from Umbra3D, a CPU occlusion culling middleware. Still, precomputed visibility sets for weak devices are brought up, low res HI-Z depth buffer rasterised on the CPU with SIMD units for dynamic culling, see Intel's 2015/2016 occlusion culling article.

4. Frustum Culling [4]:
tldr code samples and brief performance numbers for bounding sphere, AABB, OBB based culling, SSE support, MT support, somewhat simple GPU culling. Sphere or AABB with SSE + MT support quite fast, 0.1-0.2ms for 100k objects on unspecified i5 quadcore.

5. Dual-Cone View Culling for Virtual Reality Applications [5]:
tldr not discernibly faster than frustum pyramid culling plus stencil mesh. Needs careful tuning for each HMD to avoid performing worse when a larger image is rendered and warped to view. With SIMD, more accurate cones etc may be more efficient.

6. Culling Techniques [6]:
tldr general explanation. Of interest maybe hierarchical bounding volumes for frustum culling. As the goal is a fast rendering of extreme number of objects, hierarchical ordering seems obviously necessary.

7. A Survey of Visibility for Walkthrough Applications [7]:
tldr .

8. Occlusion Culling Methods [8]:
tldr overview of methods anno 2001 (duh). Hierarchical frustum culling, occluder fusion, etc.

9. Math for Game Developers - Frustum Culling[9]:
tldr as expected, brief mathematical explanation on how to compute whether a prop is inside the view frustum (based on position and prop radius).

10. Wie Sie einen Einstieg in das Frustum Culling finden [10]:
tldr D3D sample code for view frustum culling. Very brief, not very useful. But raises the questions how to compute (and store) bounding sphere radius for each prop, and how to compute and update frustum planes each frame (aka reconstruct from view + proj matrices or set up at start and then move all planes according to camera movement).

11. Superfrustum culling [11]:
tldr instead of one frustum per eye, construct one superfrustum covering both. On paper means somewhat fewer objects are culled, but the reduced overhead still comes out on top. In their testing, shaved off 1ms on lower end CPUs. Also monoscopic far-field for lower end devices and far draw distances: past a certain distance, stereo separation becomes very small, nigh indistinguishable, so it may be faster to render the far field monoscopic and then composit with the near-field stereo image.

12. OpenGL sample for shader-based occlusion culling [12]:
tldr shader-based batched occlusion culling system: leverages multi_draw_indirect and works well with setups where all geo is stored in one big buffer.

13. GPU occlusion culling using compute shader with Vulkan [13]:
tldr follows [1] to showcase GPU occlusion culling.

14. Understanding Culling Methods | Live Training | Unreal Engine [14]:
tldr basic explanation of methods employed in UE4.

More:
15. Occlusion Culling Algorithms [15].
16. View Frustum Culling [16].

17. How Occlusion Culling Works [17].
18. Cull and LOD: Compute shader culling and LOD using indirect rendering [18].
19. Occlusion queries: Using occlusion query for visibility testing [19].
20. Visibility and Occlusion Culling [20].

2.1.2 Other conference talks

21. Advanced VR Rendering with Valve's Alex Vlachos (GDC'15) [21].
22. Advanced VR Rendering (GDC'16) [22].
23. Practical Development for Vulkan (GDC'16) [23].
24. Fast and Flexible: Technical Art and Rendering for The Unknown (VRDC'16) [24].
25. Bringing Vulkan to VR (DevU'17) [25].

2.1.3 Culling aspects worth looking at

Thus, things up in the air:
CPU-based: (hierarchical/chunk-based) frustum culling, occlusion culling (e.g. mipped depth raster) and front-to-back ordering, detail culling (e.g. size-distance function or center-distance culling of peripheral objects), simple distance culling
GPU-based: same, really, but shader/compute based (However, I have a lot of respect for these approaches as GPU-based culling is a whole other beast. I would prefer to stick with CPU-based approaches for starters.)
Specifically for frustum culling: dual-view frustum culling vs superfrustum culling evaluation
General: stencil meshing, SIMD and MT optimization

2.2 Another angle: Monoscopic far-field rendering

Slides 14-31 in [26], [27] and [28]: Basic idea is to render near-field (e.g. first 30ft) in stereo for good separation effect, anything beyond that in mono, then mask the background and composite the two images. This leverages the fact that stereo separation becomes very minimal beyond a certain distance, even so small that it's less than 1px shift. When tuned correctly and used in scenes with many objects beyond the threshold distance, this can yield good performance improvements of up to 25%. The savings from only drawing far objects once must outweigh the cost of adding another camera and more render passes. Another question is how savings scale when multiview rendering is used, as it already avoids a lot of redundant vertex shader work. While Epic Games seem to have dropped the feature from Unreal Engine in 4.22 for yet unknown reasons, I couldn't find much further research about it. One curious extension would be to use the far-field depth buffer to shift the image slightly in post-processing to create a fake stereo separation effect. Or perhaps to render not a centered far-field image, but to use the far-field image of each eye in alternating fashion. This may allow a further reduction in near-field far plane while still being a convincing approximation.

2.3 Research during implementation and testing

GPU Zen 1, 2
Intel Software, Masked Occlusion Culling

2.4 Renderer current status

- using Vulkan 1.1.97+ and OpenVR 1.4.18
- using RTG's rtvklib rendering library and wx-vk UI library, both still in development
- available rendering features:
  - VK_multiview based stereo rendering
  - pipelines for blinn-phong shading, PBR shading, skybox
  - OpenVR roomscale tracking
  - assets supported in in-house .hvr format
- so far tested successfully with Valve Index, HTC Vive and Vive Pro, Samsung Odyssey
- rendering features to be ported from discontinued Valkun project:
  - VR stencil mesh mask
  - OpenVR rendermodel readout

3 The RTG Tachyon Engine

3.1 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.

3.1.1 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.

3.1.2 Render loop

After all startup and initialization is done, the engine's render loop has the following structure.

```
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 3.1: Renderer update function

3.1.3 VR loop(?)

blabla

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

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

Figure 3.2: Renderer frame render function

4 Stereo Rendering Optimization

4.1 Explanation - Optimizations

4.1.1 Multiview stereo rendering

- How does it work? - Expected estimated impact - Implementation details blabla

List of Figures

List of Tables

Bibliography

[1] K. Anagnostou. *EXPERIMENTS IN GPU-BASED OCCLUSION CULLING*. 2017. URL: https://interplayoflight.wordpress.com/2017/11/15/experiments-in-gpu-based-occlusion-culling/.

[2] S. Barrett. *Why Frustum Culling Matters, and Why It's Not Important*. 2017. URL: https://gist.github.com/nothings/913056601b56e5719cc987684a16544e.

[3] *Overview on popular occlusion culling techniques: The experts at Umbra share their knowledge on solving the 3D visibility problem*. 2016. URL: https://www.gamesindustry.biz/articles/2016-12-07-overview-on-popular-occlusion-culling-techniques.

[4] A. Gerlits. *Frustum Culling*. 2017. URL: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/frustum-culling-r4613/.

[5] J. Hale. *Dual-Cone View Culling for Virtual Reality Applications*. 2018. URL: https://blog.squareys.de/dual-cone-view-culling-for-vr/.

[6] Dr. K.R. Subramanian. *Culling Techniques*. Charlotte, NC. URL: https://webpages.uncc.edu/krs/courses/5010/ged/lectures/cull_lod2.pdf.

[7] D. Cohen-Or, Y. L. Chrysanthou, C. T. Silva, and F. Durand. "A Survey of Visibility for Walkthrough Applications". In: *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS* 9.3 (2003), pp. 412–431. URL: https://www.researchgate.net/publication/2440562_A_Survey_of_Visibility_for_Walkthrough_Applications.

[8] H. Hey and W. Purgathofer. "Occlusion Culling Methods". In: *EUROGRAPHICS STAR* (2001). URL: https://pdfs.semanticscholar.org/c1a0/aa9000f62bb9b1a60f27182e23872c375e1e.pdf.

[9] J. Rodriguez. *Math for Game Developers - Frustum Culling*. 2013. URL: https://www.youtube.com/watch?v=4p-E_31XOPM.

[10] *Wie Sie einen Einstieg in das Frustum Culling finden*. URL: https://viscircle.de/wie-sie-einen-einstieg-in-das-frustum-culling-finden/.

[11] 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.

[12] C. Kubisch. *OpenGL sample for shader-based occlusion culling*. 2014. URL: https://github.com/nvpro-samples/gl_occlusion_culling.

[13] sydneyzh. *GPU occlusion culling using compute shader with Vulkan*. 2018. URL: https://github.com/sydneyzh/gpu_occlusion_culling_vk.

[14] T. Hobson. *Understanding Culling Methods | Live Training | Unreal Engine*. 2019. URL: https://www.youtube.com/watch?v=6WtE3CoFMXU.

[15] E. Haines and T. Müller. *Occlusion Culling Algorithms*. 1999. URL: https://www.gamasutra.com/view/feature/131801/occlusion_culling_algorithms.php.

[16] *View Frustum Culling*. 2011. URL: http://www.lighthouse3d.com/tutorials/view-frustum-culling/.

[17] thebennybox. *How Occlusion Culling Works*. 2018. URL: https://www.youtube.com/playlist?list=PLEETnX-uPtBVszVeAID1jp5j9YT9EFae.

[18] S. Willems. *Cull and LOD: Compute shader culling and LOD using indirect rendering*. 2016. URL: https://github.com/SaschaWillems/Vulkan/blob/master/examples/computecullandlod/computecullandlod.cpp.

[19] S. Willems. *Occlusion queries: Using occlusion query for visbility testing*. 2016. URL: https://github.com/SaschaWillems/Vulkan/blob/master/examples/occlusionquery/occlusionquery.cpp.

[20] *Visibility and Occlusion Culling*. n.d. URL: https://docs.unrealengine.com/en-US/Engine/Rendering/VisibilityCulling/index.html.

[21] A. Vlachos. *Advanced VR Rendering with Valve's Alex Vlachos*. 2016. URL: https://www.youtube.com/watch?v=ya8vKZRBXdw.

[22] A. Vlachos. *Advanced VR Rendering Performance*. 2016. URL: https://www.youtube.com/watch?v=eIlb688pUu4.

[23] D. Ginsburg, B. Karlsson, and D. Sekulic. *Practical Development for Vulkan (presented by Valve Software)*. 2016. URL: https://www.gdcvault.com/play/1023577/Practical-Development-for-Vulkan-(presented.

[24] J. Answer. *Fast and Flexible: Technical Art and Rendering For The Unknown*. 2016. URL: https://www.youtube.com/watch?v=ClRtgeUsdOg.

[25] C. Everitt. *2017 DevU - 06 Bringing Vulkan to VR - Cass Everitt*. 2017. URL: https://www.youtube.com/watch?v=jbCQI3DZ9w8&list=PLYO7XTAX41FOURVQnOMFI-uHyuyIGh6eX&index=8&t=0s.

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

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

[28] *Hybrid Monoscopic Rendering*. 2016. URL: https://developer.oculus.com/documentation/unreal/1.14/concepts/unreal-hybrid-monoscopic/.