The Gap between Visualization Research and Visualization Software (VisGap) (2021)
C. Gillmann, M. Krone, G. Reina, T. Wischgoll (Editors)

Tools for Virtual Reality Visualization of Highly Detailed Meshes

M. B. Jensen, E. I. Jacobsen, J. R. Frisvad, and J. A. Bærentzen

Technical University of Denmark

Abstract

The number of polygons in meshes acquired using 3D scanning or by computational methods for shape generation is rapidly increasing. With this growing complexity of geometric models, new visualization modalities need to be explored for more effortless and intuitive inspection and analysis. Virtual reality (VR) is a step in this direction but comes at the cost of a tighter performance budget. In this paper, we explore different starting points for achieving high performance when visualizing large meshes in virtual reality. We explore two rendering pipelines and mesh optimization algorithms and find that a mesh shading pipeline shows great promise when compared to a normal vertex shading pipeline. We also test the VR performance of commonly used visualization tools (ParaView and Unity) and ray tracing running on the graphics processing unit (GPU). Finally, we find that mesh pre-processing is important to performance and that the specific type of pre-processing needed depends intricately on the choice of rendering pipeline.

CCS Concepts

• ***Human-centered computing*** → ***Visualization toolkits;***

1. Introduction

As of 2020, more than 2.5 quintillion (10^{18}) bytes of data are generated daily [Bul21]. Thus, we have truly entered the Age of Big Data, and we need good tools for analysis now more than ever. In the field of visual analytics, interactive user interfaces assist analytic reasoning [TC06] and Virtual Reality (VR) has been explored for better dealing with and analyzing big data [MGHK15]. The use of extended reality for visual analytics has led to the notion of immersive analytics [CCC*15], where a head-mounted display (HMD) offers many exploration modes that can improve task performance [WSN21]. However, this comes at the cost of significant rendering performance requirements (80+ frames per second) to avoid cybersickness issues [WSN21]. In many applications, a modern graphics processing unit (GPU) will likely provide adequate performance, but in areas like Earth science, where the main concern is exploration of details in very large geospatial datasets, rendering performance becomes highly important as it determines whether or not the user can immersively inspect the details of interest [ZWL*19].

Apart from use in visualization of geospatial data [KBB*06; ZWL*19], it seems that VR is rarely employed for visualization of large scale geometric data. We find this unfortunate since VR simplifies data exploration and thereby arguably aids inductive reasoning. For visualization purposes, a crucial benefit of VR is that the mapping from user movement to the virtual space is very intuitive. Head motion maps directly to camera movement, and both translation and rotation of an object can be achieved directly with completely analogous hand gestures. Simply put, the user controls both more degrees of freedom and does it in an more intuitive manner than if interacting with a mouse and keyboard while looking at a computer screen. Effectively, VR changes the role of the user from passively inspecting images to actively investigating data.

Using VR is not without its challenges, however. In particular, we are motivated by the concern that if frame rates drop or vary significantly, it will negatively impact the motion-to-photon latency (the time between a movement being registered by the HMD and the corresponding frame being rendered [ZAVJ17]) and this carries a real risk that users become cybersick [SNL20]. Clearly, this issue puts a limit on the size of the datasets that we can visualize in VR without a latency level that is too high.

In this regard, it is unfortunate that datasets grow rapidly in size in many scientific fields. Topology optimization (albeit on a supercomputer) now allows for discretization of models into more than 1 billion voxels [AALS17]. In 3D scanning, object surfaces can be scanned with a measurement sampling density (MSD) of 10,000 points per square millimeter [BSM11], and scanning a 39.3×28 cm^2 woodcut with a MSD at just 2500 points per square millimeter resulted in 277 GB of data [BS14]. Smooth surfaces can be simplified with little perceptual impact, but we often have unsmooth data and a need to inspect the details. The mentioned woodcut is an example of such data where lower MSD would make analysis hard [BSM11]. Some examples of meshes with details at varying scales are shown in Figure 1. The seal skull (1d–1f) is an example of a 3D scanned surface that includes per vertex colour information. A reduction in vertices therefore not only reduces the detail of the mesh but also means the loss of colour information. Moreover,

DOI: 10.2312/visgap.20211088

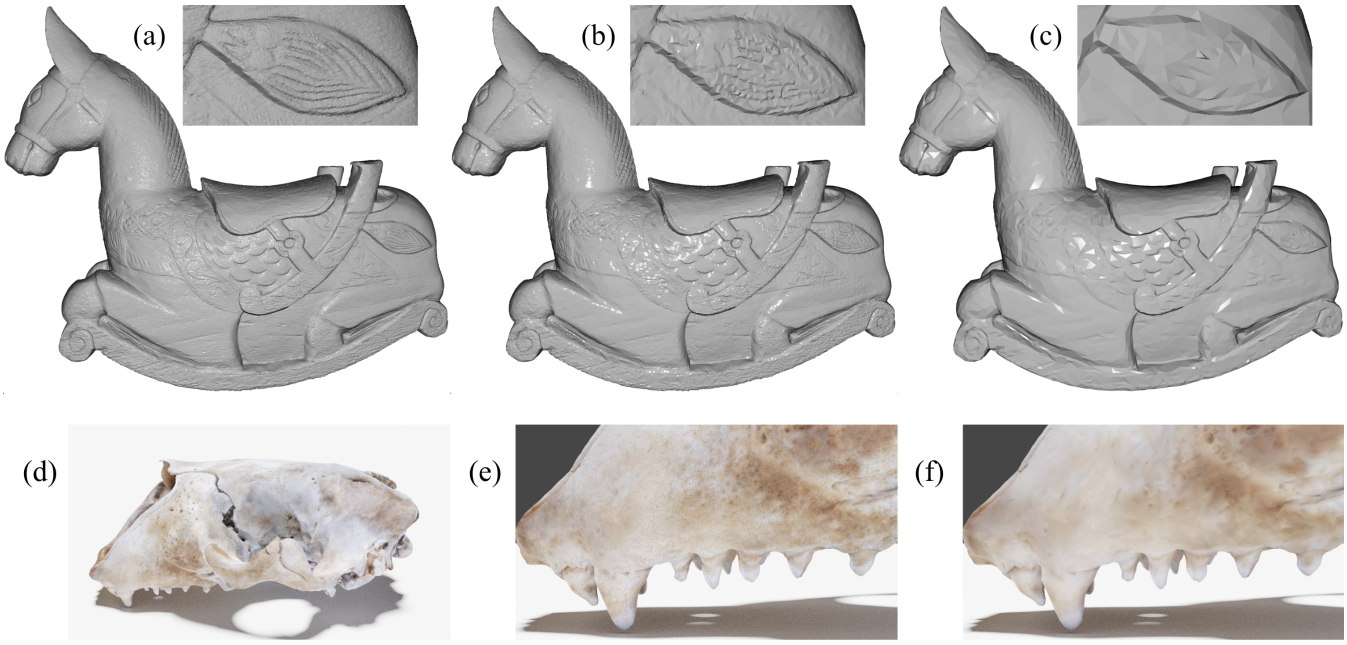


Figure 1: *The rocking horse (a) consists of 2.2 million triangles. We reduce it to 10% of the original number of triangles (b) and further to 1% (c). While this fairly large reduction has almost no effect on the silhouette, the fine scale geometric details are clearly impacted by the reduction to 10% and almost completely erased at 1%. Below, a 3D scan of a seal skull is shown with vertex colours (d). Looking at a close-up (e) and reducing to 1% (f), it is clear that the overall shape is completely unscathed, but the vertex colours are significantly blurred.*

many types of data might have a complexity that makes it infeasible to perform significant reductions to the level of detail in the first place (1a–1c). If we want the ability to interactively visualize the small details of large meshes in VR, we have to ensure that our visualization tools deliver high rendering performance, which means high and stable frame rates.

Our goal is to guide the choice of rendering technologies for interactive VR-based visualization of highly detailed meshes. We do this by comparing three visualization tools using a common benchmark. The compared tools are: Jinsoku, our own VR visualization engine based on C++/Vulkan; ParaView, which supports VR and is one of the most popular visualization tools; and Unity, which is a game engine and a popular tool for VR-based visualization [DDC*14; SLC*19; CCB*19]. Our aim is not simply to find out which of these three solutions is fastest but also to identify the choices of rendering pipeline and geometry-preserving mesh optimization that seem to have a big impact on performance. We discuss the underlying technologies in Section 2, the tested platforms in Section 3, and we present and analyze our results in Section 4.

We use three different large and detailed 3D models for our investigation. The three models are examples from natural heritage preservation (Seal Skull), topology optimization (Wing), and additive manufacturing (Nobby). Table 1 provides some mesh complexity info for the three models and example visualizations are in Figure 2 (rightmost column). The Seal Skull has been 3D scanned into a point cloud and digitally reconstructed as a triangle mesh. The topology optimized airplane wing [AALS17; ASLA20] is the largest model in our comparisons. The third mesh was created with PrusaSlicer (https://www.prusa3d.com/) using a model called Nobby (https://www.prusaprinters.org/prints/35338-nobby-octopus-sculpt). The three models are interesting case studies as they all have several orders of magnitude between the extent of the model and the size of the details that would be of interest in a VR-based inspection of the model.

Table 1: *Test Meshes*

	Seal Skull	Wing	Nobby
no. triangles	14,504,882	38,629,758	32,905,214
no. vertices	21,757,335	92,010,363	16,970,666
model size	1.154GB	3.819 GB	1.723GB

In addition to the main study, we also investigated the use of hardware accelerated ray-tracing for the purpose of visualization of large scale geometry. This study and its results are presented in Section 5. While all the results are discussed in Section 6.

2. The Graphics Pipeline

Traditionally, the graphics pipeline was easy to describe as a machine for processing and rasterizing triangles. Much of the performance of the graphics pipeline was derived from the fact that it was both data and task parallel, allowing processing of multiple vertices in parallel with multiple fragments [Hai06]. During this period, it was important to optimize meshes for the so-called post transform and lighting (post-T&L) cache which is a global cache that stores the transformed vertices, i.e. the output from the vertex shader

[SNB07]. On average a vertex is shared by six triangles. Thus, if a triangle needs a vertex that has already been transformed, it can simply be picked from the post-T&L cache, assuming the mesh is rendered with *indexed* primitives. Since the size of the cache might not be known - for instance if the mesh is to be used on a variety of graphics processors - meshes were often simply optimized to promote locality [For06]. If a vertex that is used by a given triangle is also used soon after, it is likely to be in the cache, and the result of vertex shading can be reused.

Modern graphics hardware has a different not-so-pipelined design: vertices and pixels are processed by the same streaming multiprocessors (SMs) imbued with local storage. If a modern GPU were to have a shared post-T&L cache, it would have to be outside the local storages of the SMs. In fact, it seems that modern GPUs do not have a post-T&L cache [KKI*18]. Instead each SM processes a small patch of the mesh at a time. Importantly, this means that mesh optimization which promotes locality is still highly beneficial but now for a different reason. If the triangles that share the same vertex are close in the stream of triangles, they are also likely to be in a patch processed at the same time on a given SM.

With the Turing architecture, NVIDIA also introduced a mechanism which directly exposes the way that meshes are processed by the GPU, namely *mesh shaders* [Kub17; Kub20]. Mesh shaders bring a programming model similar to that of compute shaders to the graphics pipeline: a workgroup of individual threads on the GPU are tasked with collaboratively producing both transformed vertices and triangle connectivity. To exploit this feature, one needs to break the mesh into smaller patches called *meshlets*. Essentially, this is automatic if the traditional vertex shader pipeline is used, but taking charge of meshlet generation affords additional freedom as described below.

The mesh shader based pipeline is highly flexible. While a meshlet is usually associated with a group of triangles, it can be seen simply as a descriptor that can carry any kind of information. Furthermore, the inputs and outputs between the shader stages can be decided by the programmer. A so-called *task shader* orchestrates the work and can generate workgroups that process meshlets, or decide that a meshlet is not visible and that resources should not be spent on its processing. This is very important since it allows the mesh shader to cull meshlets which are either outside the view frustum or backfacing. A meshlet is considered backfacing if all its faces are backfacing. This is easy to test if we store a cone that contains all face normals for each meshlet.

The Turing architecture also saw the introduction of the so-called RT cores which allow for much faster hardware accelerated ray tracing on the GPU than previously [Bur20]. It has also recently become possible to mix ray tracing and rasterization using the Vulkan API [KHBW20]. While ray tracing makes it far easier to implement shadows, non-planar reflections, ambient occlusion and other global effects, it is not likely to lead to faster rendering if only local illumination (e.g. Phong shading) is required.

3. VR Visualization Tools

ParaView is a tool designed for visualization and analysis of extremely large datasets [AGL05]. Paraview is built on the Visualization Toolkit (VTK), and it includes easy-to-use VR-based visualization [MDJA18], making it a good choice for our purposes.

Unity is a game engine that includes VR support. In previous work, it has been referred to as "a standard platform for developing immersive environments" [CCB*19]. However, in our initial testing, we experienced surprisingly poor performance with Unity when rendering our large meshes: average render times per frame ranging from 20 to 140 milliseconds. To remedy this, we optimized the application by switching to Unity's *Universal Render Pipeline* and by allowing Unity to optimize the mesh without decimating it. This means that Unity is free to reorder the index buffer to increase performance, but it is not allowed to change the number of vertices. These optimizations led to significantly better render times. However, Unity does not implement the new mesh shading pipeline described above [Uni20].

We compare these two solutions to our own (bespoke) VR visualization application implemented in C++ using the Vulkan API [SK17]. We refer to our own application as *Jinsoku*. Since Jinsoku is white box, it is easy to analyze and well-suited as a benchmark when comparing the different tools. Jinsoku incorporates two pipelines: one based on vertex shading and one based on mesh shading. This enables us to better analyze the practical importance of mesh shaders.

As an additional experiment, we implemented a VR ray tracer. While we found that GPU ray tracing scales well with an increasing polygon count, the ray tracer was a factor of two slower than Jinsoku and Unity. We therefore focus on rasterization techniques. Ray Tracing is however becoming more viable and will continue to do so as the recently introduced hardware acceleration matures.

3.1. Auxiliary Tools

We use SteamVR to interface with the headset for all the applications. SteamVR is a runtime API that interfaces with the backend of OpenVR. As such, SteamVR enables developers to interface with a broad range of different HMDs. SteamVR has several options for analyzing the performance of an application and is capable of recording frame data and saving it to a file. We use these data for our comparisons (except in the case of ray tracing, see Section 5). This means that applications are subject to the same asynchronous time warping implementation.

The three test meshes have an increasing number of triangles and vertices. The Seal Skull mesh and the Nobby mesh were optimized using Tootle (https://github.com/GPUOpen-Archive/amd_tootle). This program greedily reorganizes the mesh so that triangles using a given vertex are as close as possible in the list of triangles. Tootle was created for the vertex shader pipeline where locality is useful for vertex caching [NBS06], but it also makes the meshlets more compact. Unfortunately, this software could not handle the topology optimized Wing mesh, presumably because of its size. For the skull, the optimized version has not only increased locality but also reduced the overall number of meshlets needed to represent the mesh. For Nobby, the optimization has not changed the number of cullable meshlets nor has it changed the total number of meshlets. The optimized version is however still used since it might

Table 2: *Meshlets*

meshlets	Skull	Skull opt	Wing	Nobby	Nobby opt
cullable	170,400	156,930	274,589	16	16
total	229,043	163,264	1,699,388	307,774	307,774

have changed the vertex order. Table 2 shows the total and cullable number of meshlets for each mesh.

The ability to process only the parts of the mesh that can be seen by the camera is often very powerful when dealing with large amounts of data. We used the meshlet builder from the official NVIDIA github (https://github.com/nvpro-samples/-gl_vk_meshlet_cadscene) when implementing Jinsoku. Because the mesh optimization in Unity is a black box, we also implement a vertex shading pipeline in Jinsoku to directly compare the traditional vertex shading pipeline with the mesh shading pipeline.

4. Experiment Setup and Results

For our experiments, we set up the three visualization tools as follows.

- In Jinsoku, we used Phong shading with a fixed light position. Texture mapping was not employed. Hence, each vertex carries only one attribute in addition to its position, namely the surface normal.
- In Unity (UnityURP in Figure 1), we also used Phong shading with a fixed light position. The Phong shading is implemented with a so-called unlit shader, meaning that no shadows are cast from the light source. Texture mapping was not employed. The out-of-the-box version of Unity (UnityNoop in Figure 1) uses a deferred rendering pipeline and includes shadows.
- ParaView uses flat shading and has no options for changing this in VR.

When measuring the render time with SteamVR we get the time between each update to the HMD. Each update requires that two frames are rendered and presented to the HMD. By using these SteamVR render times, we obtain times that are comparable to those that you would get during an actual inspection of the meshes.

Performance plots are in Figure 2. The bar charts are all plots of average render times for each application. The whiskers show the variance of the render time for each frame. For all plots, the vertical axis is time in milliseconds. Each mesh has been visualized under two different conditions, on two different hardware setups. In the first condition, the entire mesh is visible, and in the second, the mesh is inspected up close (this is exemplified in Figure 3).

The same transformations are applied to the meshes in both Unity and Jinsoku. Since ParaView does not allow for the same precision in placing meshes the objects are inspects in approximately the same positions. The first hardware platform uses an Oculus Quest which has a pixel resolution of 1440×1600 for each eye and runs with a refresh rate of 72 Hz. The Quest is tethered to a 2019 Razer Blade 15 with an NVIDIA GeForce RTX 2080 with Max-Q Design and 8GB GDDR6 VRAM, a 9th Gen Intel Core i7-9750H 6-Core, 16GB of RAM and a 512GB SSD (NVMe). The second hardware platform uses a Valve Index which has a pixel resolution of 1440×1600 for each eye and can run with a refresh rate of up to 144 Hz. The Index is connected to a desktop that has an Intel Core i9-9900k, 64GB of DDR4-2666 RAM, and one NVIDIA GeForce RTX 2080 Ti Turbo OC with 11GB of GDDR6 RAM.

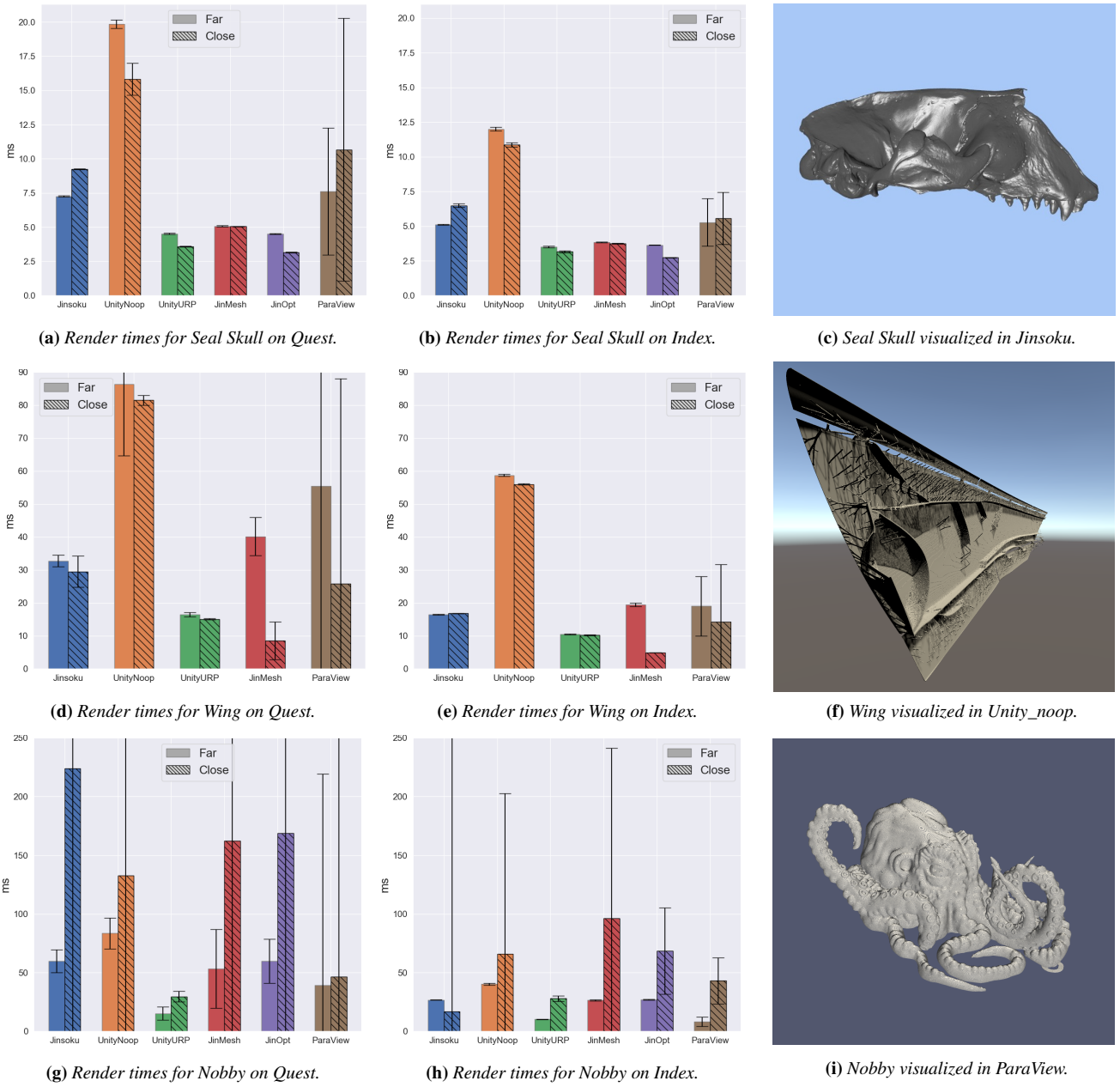
When converting the average render times to frames per second (FPS) and comparing to a target of 80+ FPS [WSN21], we observe that this is only achieved consistently for the Seal Skull. For the Seal Skull we get low variance and average render times of 3.7–5.0 ms (∼200–270 FPS) for UnityURP and 2.7–4.5 ms (∼222–370 FPS) for Jinsoku with mesh shading and the Tootle-optimized meshes. For the Wing, we see a different picture with UnityURP timings in the range of 10.2–16.4 ms (∼61–98 FPS) on both platforms. Here the mesh shading pipeline does really well when inspecting the wing up close getting between 4.9–8.5 ms (∼118–204 FPS). The variance on the Quest platform is however quite high. For Nobby, we get good results for UnityURP and ParaView. However, this is only on the Index platform with average rendering times around 8.1–10.2 ms (∼98–123 FPS) while inspecting the mesh from afar. All other tests show average rendering times from 16–223.9 ms (∼4.5–62.5 FPS) while exhibiting large variance across the board. Rendering performance is thus still a major concern when it comes to visualization of some types of large meshes. We suggest future development of better optimization of meshes for the mesh shading pipeline to avoid discomfort in VR visualization of such meshes.

4.1. Vertex Shading vs Mesh Shading

We can compare the vertex and mesh shading pipeline by inspecting the blue and red bars in Figures 2a, 2b, 2d, 2e, 2g, 2h. When we are inspecting the mesh up close the mesh shading pipeline performs better in 5 out of 6 test cases. When inspecting the mesh from afar the mesh shading pipeline performs better in 3 out 6 cases. We see that the mesh shading pipeline exhibits larger variance in render time for the wing and Nobby but not the skull. For Nobby the normal vertex shading pipeline performs better on the Index but worse on the Quest. This can be seen in Figure 2g and 2h. Figure 4 shows the Nobby mesh up close with a visualization of the meshlets. This gives some insight into why the mesh shading pipeline exhibit these high render times. The mesh is comprised of elongated cylinders, and since the meshlets are not generated so as to combine faces with similar normals, it is likely that no meshlets can ever be culled because they all contain faces that are visible from almost any direction. On the other hand, the mesh shading pipeline is extremely efficient on the largest data set. Figure 2e and 2d show that the quest and index mesh shader pipeline produces the smallest average render times across headsets when inspecting the mesh up close.

4.2. Index Buffer order and Mesh shaders

Allowing Unity to optimize the mesh is in part what resulted in the performance that can be seen in Figure 2. This motivated us to try and see if the mesh shading pipeline would also benefit from similar treatment. It is clear that meshlets also benefit from locality optimizing the index buffer, not only does it produce more cullable meshlets but it also decrease the total number of meshlets and



(a) *Render times for Seal Skull on Quest.* **(b)** *Render times for Seal Skull on Index.* **(c)** *Seal Skull visualized in Jinsoku.*

(d) *Render times for Wing on Quest.* **(e)** *Render times for Wing on Index.* **(f)** *Wing visualized in Unity_noop.*

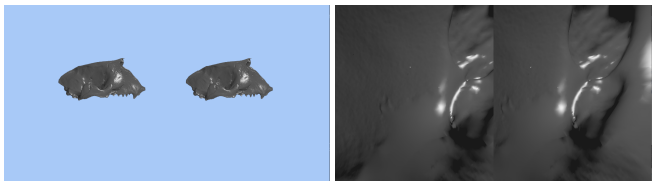
(g) *Render times for Nobby on Quest.* **(h)** *Render times for Nobby on Index.* **(i)** *Nobby visualized in ParaView.*

Figure 2: *Test results as bar plots (a,b,d,e,g,h). Each bar plot has render time in milliseconds on the vertical axis and shows two test cases for one mesh on one platform. The crosshatched bar is for close-up inspection while the flat-coloured bar is for far-away inspection. The whiskers show the variance of the render time. In the right column, we visualize the Seal Skull in Jinsoku (c), the Wing in Unity (f), and Nobby in ParaView (i). Explanation of abbreviations: Jinsoku - Vulkan-based vertex shading pipeline; UnityNoop - none-optimized Unity; UnityURP - Unity when using its Universal Render Pipeline and its mesh optimization; JinMesh - Jinsoku when using its mesh shading pipeline; JinOpt - Jinsoku with mesh shading and mesh optimized by Tootle; ParaView - the VR support of ParaView.*

the variance in the render time. This indicates that less vertices are shared across meshlets. Figures 2a and 2b also reflect this by showing improvements when comparing the mesh shading pipeline with (purple bars) and without (red bars) the optimized mesh. The mesh shading pipeline even edges out Unity when inspecting the skull up close. Nobby on the other hand exhibits a case where the optimization algorithm fails to optimize the mesh.

5. Ray tracing

Hardware rasterization of triangles is by far the more common approach when we are aiming at rendering of objects at the high frame rates required by VR. Rasterization is the process of drawing a triangle by first projecting it into the image plane and then shading the pixels covered by the triangle. Instead of projecting triangles to an image plane, we could trace a ray from a position in each pixel



Inspection from far away Inspection up close

Figure 3: *The two test conditions.*

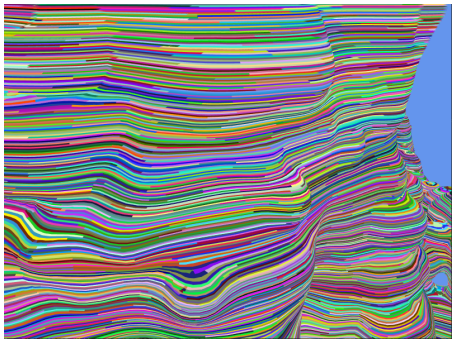


Figure 4: *Nobby meshlets.*

into the scene and figure out what triangle the ray hit (if any). This is the ray tracing paradigm.

Ray tracing eases rendering of shadows in general and rendering of multiple reflections and refractions in specular surfaces. Since we can place our triangle mesh in a spatial data structure, we can find the closest triangle that a ray might intersect in logarithmic time. If the number of triangles is very large, this is a great advantage. However, it becomes more expensive if the digital object is interactively modified, as the spatial data structure must then be updated. This can be done in parallel on the GPU, but still incurs some overhead. Conventional ray tracing also requires that we consider all pixels, which means that performance depends more directly on the screen resolution. In rasterization, we need only consider the pixels where fragments end up, but then in return we have to process each triangle.

Use of ray tracing for VR became tractable on consumer platforms only with recently introduced hardware support. To employ this hardware support, we used NVIDIA OptiX [PBD*10; WP19]: a CUDA-based API that requires CUDA to Vulkan/OpenGL interoperability to efficiently interact with OpenVR. The ray tracer renders directly to textures that OpenVR can access. This unfortunately has the effect that the HMD cannot directly measure render times (as it can always use the texture whether it was updated or not). For this reason, we did not include ray tracing in Figure 2. Instead, we discuss the prospects of ray tracing for VR.

We designed our ray tracer to provide a frame for each vertical synchronization (vsync) of the HMD. When measuring render times, everything was kept unchanged except that we did not connect an HMD to avoid this vsync lock. As in our results for rasterization, we tested our VR ray tracer using a GPU on a stationary computer (Figure 5) and on two GPUs on laptop computers (Figure 6) with models of different complexity (numbers of triangles).

△: 300,603, t: 9.43 ms △: 15,740,813, t: 9.80 ms

Figure 5: *VR ray tracing with one sample ambient occlusion (reason for the noise) rendered using an NVIDIA RTX 2080 graphics card. Here, △ is number of triangles and* t *is render time.*

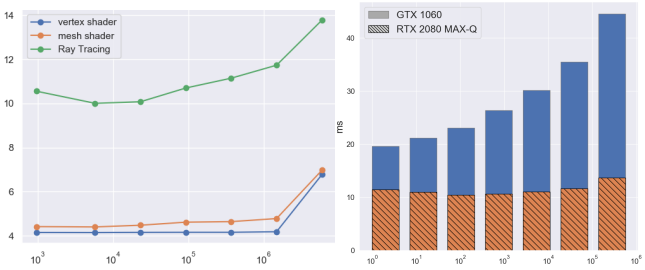


Figure 6: *Performance of our GPU VR ray tracer when rendering the Blender monkey [Wik21] with increasing number of subdivisions. We compare with the two shading pipelines in Jinsoku (left) and with a GPU architecture from before RTX (right). The horizontal axes are logarithmic, meaning that the development in performance should be a straight line for logarithmic time complexity. This is not quite obtained, but RTX is getting there.*

We tested performance for GPUs with different hardware architectures. The ones called RTX have special RT cores dedicated to hardware acceleration of ray tracing [Bur20].

The RTX graphics card almost achieves the logarithmic time complexity with increasing number of triangles (Figure 6). The difference in performance as a function of the number of triangles is very small across several orders of magnitude (Figures 5 and 6). Even so, GPU ray tracing is still significantly slower than Jinsoku when it comes to the visualization with local illumination that we are testing in this work (Figure 6). RTX cards for stationary computers are fast enough to support the frame rates needed for ray traced virtual reality (Figure 5). We could even afford a so-called ambient occlusion ray, which is a shadow ray traced in a random direction. Ambient occlusion is a visual effect that is expensive to compute in rasterization. In ray tracing, we can get a noisy version of it at low cost. Since the RTX architecture has special tensor cores dedicated to hardware acceleration of deep learning techniques [Bur20], the future will see very efficient denoising that can also exploit temporal correspondences between frames [HMS*20]. GPU accelerated denoising is however still too expensive for the time budget allowed by VR.

Interestingly, ray tracing was recently made available as a core extension in Vulkan [KHBW20] (released in December 2020). This provides the first open, cross-vendor, cross-platform standard for hardware accelerated ray tracing. In addition, Vulkan ray tracing enables use of a hybrid between rasterization and ray tracing. Un-

real Engine 4 integrated the ray tracing functionality in DirectX 12 (which is similar to the one in Vulkan) in combination with learning-based denoising into their rasterization-based framework to enable real-time rendering of cinematic quality [LLCK19]. This is an indicator that a hybrid of rasterization and ray tracing will likely become an option in the VR graphics engines of the future. Since Jinsoku is based on the Vulkan API, it directly supports extension to include ray traced shading effects that can potentially enhance the inspection of geometric details.

6. Discussion and Conclusion

Unsurprisingly, our tests show that performance is very dependent on mesh connectivity. This lends a great advantage to Unity in comparison to Jinsoku when rendering an unoptimized mesh, since Unity's proprietary optimization step seems to greatly improve performance. This is particularly true for the Nobby mesh.

Thus, while ParaView is the easiest way to get started on inspection of meshes in VR, ParaView only supports flat shading and lacks the straight forward programmatic extensibility of Unity. Perhaps the biggest limitation of Unity is the lack of support (so far) for the latest features of graphics hardware. The mesh shading pipeline has two vast advantages, namely frustum and backface culling on the granularity of meshlets. Having the ability to only process the parts of the mesh that can be seen by the camera can be really powerful when dealing with large amounts of data and when zooming in on models. Figure 2e shows this clearly. In fact, this indicates that a mesh shading pipeline could very well be the best choice for visualization of large and complex meshes in VR. For the skull, our tests show that a largely unoptimized Jinsoku is capable of performing on par with an optimized version of Unity, and that optimizing the mesh further increases performance while decreasing variance in the render time.

Unfortunately, reaping the benefits of the mesh shading pipeline is largely contingent on having cullable meshlets, and our tools for mesh optimization (e.g. Tootle) are generally still aimed at the vertex shading pipeline. This means that the methods for optimization largely aim to structure the output such that it is suitable for a global cache as opposed to a parallel architecture where vertex locality is made explicit. Moreover, we face the problem that meshes are very different. Given a naïve optimization, the Nobby mesh would contain no meshlets cullable by backface culling for instance. Thus, going forward, a key to good VR performance on arbitrary geometry seems to be mesh pre-processing algorithms which analyze and adapt to the particular inputs.

In conclusion, our paper compared a minimal Vulkan render engine (Jinsoku) with Unity and ParaView. Jinsoku used little optimization but managed to keep up with an optimized Unity application in some of the more interesting cases. Moreover, the mesh shading pipeline is very flexible which can be utilized to gain performance in some of the situations explored in this paper. We admit that this comes at the cost of some additional development time compared to Unity, but the mesh shading pipeline is in itself a compelling argument for building an engine when performance is an overriding concern. More research is needed to quantify the potential performance gains from using mesh optimization algorithms that are specifically tailored to the mesh shading pipeline.

Combining a well optimized engine with a mesh optimization algorithm for a mesh shading pipeline holds a lot of promise for a VR-based visualization platform. In fact, we have seen in our study that it is possible to visualize a mesh containing more than 14.5 million triangles while still achieving render times of 222-370 FPS. This is significantly more than the required 80+ FPS. Not only this, but when investigating a mesh containing more than 38.6 million triangles, we are just around the 80 FPS, and while investigating details, the FPS climbs as high as 204 when using a mesh shading pipeline. With numbers like these, it is safe to say that VR should more often be considered a viable modality for visualization, even of large datasets.

In this paper, our focus has been on rendering efficiency since efficiency limits what data sets we can effectively investigate in VR. As discussed above, we are able to visualize geometric data sets on the order of tens of millions of triangles with a frame rate sufficient for VR if we make the right technical choices. With this in place, we plan to turn our investigations to more application specific problems pertaining to the visualization of large geometric data sets. Tools for explorative analysis of geometric data would appear to benefit from a greater use of virtual reality platforms, but, in many cases, these types of data are either hard to simplify effectively, or important information would be lost by doing so. Thus, going forward, we hope this investigation, and specifically the Jinsoku engine, will be helpful in facilitating the use of VR as a tool for visualization and exploration of these types of geometric data.

7. Acknowledgments

We would like to thank Michelle Strecker Svendsen who scanned the seal skull and Luxion ApS for collaboration on VR ray tracing. The rocking horse model is provided courtesy of INRIA by the AIM@SHAPE-VISIONAIR Shape Repository. This research was supported by Advokat Bent Thorbergs Fond (ref. 66.531).

References

[AALS17] AAGE, NIELS, ANDREASSEN, ERIK, LAZAROV, BOYAN S., and SIGMUND, OLE. "Giga-voxel computational morphogenesis for structural design". *Nature* 550.7674 (2017), 84–86. DOI: 10.1038/nature23911 1, 2.

[AGL05] AHRENS, JAMES, GEVECI, BERK, and LAW, CHARLES. "ParaView: An end-user tool for large data visualization". *The Visualization Handbook* 717–731 (2005). DOI: 10.1016/B978-012387582-2/50038-1 3.

[ASLA20] AAGE, NIELS, SIGMUND, OLE, LAZAROV, BOYAN B., and ANDREASSEN, ERIK. *TopWingData*. Dataset. Technical University of Denmark, 2020. DOI: 10.11583/dtu.12581615.v1 2.

[BS14] BUNSCH, ERYK and SITNIK, ROBERT. "Method for visualization and presentation of priceless old prints based on precise 3D scan". *Measuring, Modeling, and Reproducing Material Appearance*. Vol. 9018. SPIE, 2014, 90180Q. DOI: 10.1117/12.2042635 1.

[BSM11] BUNSCH, ERYK, SITNIK, ROBERT, and MICHONSKI, JAKUB. "Art documentation quality in function of 3D scanning resolution and precision". *Computer Vision and Image Analysis of Art II*. Vol. 7869. Proceedings of SPIE. 2011, 78690D. DOI: 10.1117/12.876647 1.

[Bul21] BULAO, JACQUELYN. *How Much Data Is Created Every Day in 2020?* TechJury Blog. 2021. URL: https://techjury.net/blog/how-much-data-is-created-every-day/ 1.

[Bur20] BURGESS, JOHN. "RTX on–The NVIDIA Turing GPU". *IEEE Micro* 40.2 (2020), 36–44. DOI: 10.1109/MM.2020.2971677 3, 6.

[CCB*19] CORDEIL, MAXIME, CUNNINGHAM, ANDREW, BACH, BENJAMIN, HURTER, CHRISTOPHE, THOMAS, BRUCE H, MARRIOTT, KIM, and DWYER, TIM. "IATK: An immersive analytics toolkit". *IEEE Conference on Virtual Reality and 3D User Interfaces (VR 2019)*. 2019, 200–209. DOI: 10.1109/VR.2019.8797978 2, 3.

[CCC*15] CHANDLER, TOM, CORDEIL, MAXIME, CZAUDERNA, TOBIAS, DWYER, TIM, GLOWACKI, JAROSLAW, GONCU, CAGATAY, KLAPPERSTUECK, MATTHIAS, KLEIN, KARSTEN, MARRIOTT, KIM, SCHREIBER, FALK, and WILSON, ELLIOT. "Immersive analytics". *Big Data Visual Analytics (BDVA 2015)*. IEEE, 2015, 1–8. DOI: 10.1109/BDVA.2015.7314296 1.

[DDC*14] DONALEK, CIRO, DJORGOVSKI, S. G., CIOC, ALEX, WANG, ANWELL, ZHANG, JERRY, LAWLER, ELIZABETH, YEH, STACY, MAHABAL, ASHISH, GRAHAM, MATTHEW, DRAKE, ANDREW, DAVIDOFF, SCOTT, NORRIS, JEFFREY S., and LONGO, GIUSEPPE. "Immersive and collaborative data visualization using virtual reality platforms". *IEEE International Conference on Big Data*. 2014, 609–614. DOI: 10.1109/BigData.2014.7004282 2.

[For06] FORSYTH, TOM. *Linear-Speed Vertex Cache Optimisation*. Sept. 2006. URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html 3.

[Hai06] HAINES, ERIC. "An introductory tour of interactive rendering". *IEEE Computer Graphics and Applications* 26.1 (2006), 76–87. DOI: 10.1109/MCG.2006.9 2.

[HMS*20] HASSELGREN, J., MUNKBERG, J., SALVI, M., PATNEY, A., and LEFOHN, A. "Neural temporal adaptive sampling and denoising". *Computer Graphics Forum* 39.2 (2020), 147–155. DOI: 10.1111/cgf.13919 6.

[KBB*06] KREYLOS, OLIVER, BAWDEN, GERALD, BERNARDIN, TONY, BILLEN, MAGALI I., COWGILL, ERIC S., GOLD, RYAN D., HAMANN, BERND, JADAMEC, MARGARETE, KELLOGG, LOUISE H., STAADT, OLIVER G., and SUMNER, DAWN Y. "Enabling scientific workflows in virtual reality". *International Conference on Virtual Reality Continuum and Its Applications (VRCIA 2006)*. 2006, 155–162. DOI: 10.1145/1128923.1128948 1.

[KHBW20] KOCH, DANIEL, HECTOR, TOBIAS, BARCZAK, JOSHUA, and WERNESS, ERIC. *Ray Tracing in Vulkan*. Khronos Blog. Mar. 2020. URL: https://www.khronos.org/blog/ray-tracing-in-vulkan 3, 6.

[KKI*18] KERBL, BERNHARD, KENZEL, MICHAEL, IVANCHENKO, ELENA, SCHMALSTIEG, DIETER, and STEINBERGER, MARKUS. "Revisiting the vertex cache: Understanding and optimizing vertex processing on the modern GPU". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2 (2018), 29:1–29:16. DOI: 10.1145/3233302 3.

[Kub17] KUBISCH, CHRISTOPH. *Introduction to Turing Mesh Shaders*. NVIDIA Developer Blog. Sept. 2017. URL: https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/ 3.

[Kub20] KUBISCH, CHRISTOPH. *Using Mesh Shaders for Professional Graphics*. Dec. 2020. URL: https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/ 3.

[LLCK19] LIU, EDWARD, LLAMAS, IGNACIO, CAÑADA, JUAN, and KELLY, PATRICK. "Cinematic rendering in UE4 with real-time ray tracing and denoising". *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Ed. by HAINES, ERIC and AKENINE-MÖLLER, TOMAS. Apress, 2019, 289–319. DOI: 10.1007/978-1-4842-4427-2_19 7.

[MDJA18] MARTIN, KEN, DEMARLE, DAVID, JHAVERI, SANKHESH, and AYACHIT, UTKARSH. *Taking ParaView into Virtual Reality*. Kitware Blog. 2016, updated 2018. URL: https://blog.kitware.com/taking-paraview-into-virtual-reality/ 3.

[MGHK15] MORAN, A., GADEPALLY, V., HUBBELL, M., and KEPNER, J. "Improving Big Data visual analytics with interactive virtual reality". *IEEE High Performance Extreme Computing Conference (HPEC 2015)*. 2015, 1–6. DOI: 10.1109/HPEC.2015.7322473 1.

[NBS06] NEHAB, DIEGO, BARCZAK, JOSHUA, and SANDER, PEDRO V. "Triangle order optimization for graphics hardware computation culling". *Symposium on Interactive 3D Graphics and Games (I3D '06)*. ACM, 2006, 207–211. DOI: 10.1145/1111411.1111448 3.

[PBD*10] PARKER, STEVEN G., BIGLER, JAMES, DIETRICH, ANDREAS, FRIEDRICH, HEIKO, HOBEROCK, JARED, LUEBKE, DAVID, MCALLISTER, DAVID, MCGUIRE, MORGAN, MORLEY, KEITH, ROBISON, AUSTIN, and STICH, MARTIN. "OptiX: A general purpose ray tracing engine". *ACM Transactions on Graphics* 29.4 (2010). DOI: 10.1145/1778765.1778803 6.

[SK17] SELLERS, GRAHAM and KESSENICH, JOHN. *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley, 2017 3.

[SLC*19] SICAT, RONELL, LI, JIABAO, CHOI, JUNYOUNG, CORDEIL, MAXIME, JEONG, WON KI, BACH, BENJAMIN, and PFISTER, HANSPETER. "DXR: A toolkit for building immersive data visualizations". *IEEE Transactions on Visualization and Computer Graphics* 25.1 (2019), 8440858. DOI: 10.1109/TVCG.2018.2865152 2.

[SNB07] SANDER, PEDRO V., NEHAB, DIEGO, and BARCZAK, JOSHUA. "Fast triangle reordering for vertex locality and reduced overdraw". *ACM Transactions on Graphics* 26.3 (July 2007), 89:1–89:9. DOI: 10.1145/1276377.1276489 3.

[SNL20] STAUFFERT, JAN-PHILIPP, NIEBLING, FLORIAN, and LATOSCHIK, MARC ERICH. "Latency and cybersickness: Impact, causes and measures. A review". *Frontiers in Virtual Reality* 1 (2020), 31. DOI: 10.3389/frvir.2020.582204 1.

[TC06] THOMAS, J. J. and COOK, K. A. "A visual analytics agenda". *IEEE Computer Graphics and Applications* 26.1 (2006), 10–13. DOI: 10.1109/MCG.2006.5 1.

[Uni20] UNITY GRAPHICS TEAM. *Personal communications*. 2020 3.

[Wik21] WIKIPEDIA. *Blender (Software) - Suzanne*. 2021. URL: https://en.wikipedia.org/wiki/Blender_(software)#Suzanne 6.

[WP19] WALD, INGO and PARKER, STEVEN G. "RTX Accelerated Ray Tracing with OptiX". *ACM SIGGRAPH 2019 Courses*. 2019. URL: https://sites.google.com/view/rtx-acc-ray-tracing-with-optix 6.

[WSN21] WAGNER, JORGE, STUERZLINGER, WOLFGANG, and NEDEL, LUCIANA. "The effect of exploration mode and frame of reference in immersive analytics". *IEEE Transactions on Visualization and Computer Graphics* (2021). To appear. DOI: 10.1109/TVCG.2021.3060666 1, 4.

[ZAVJ17] ZHAO, JINGBO, ALLISON, ROBERT S., VINNIKOV, MARGARITA, and JENNINGS, SION. "Estimating the motion-to-photon latency in head mounted displays". *IEEE Virtual Reality (VR 2017)*. 2017, 313–314. DOI: 10.1109/VR.2017.7892302 1.

[ZWL*19] ZHAO, JIAYAN, WALLGRÜN, JAN OLIVER, LAFEMINA, PETER C., NORMANDEAU, JIM, and KLIPPEL, ALEXANDER. "Harnessing the power of immersive virtual reality-visualization and analysis of 3D earth science data sets". *Geo-spatial Information Science* 22.4 (2019), 237–250. DOI: 10.1080/10095020.2019.1621544 1.