

TECHNICAL UNIVERSITY OF DENMARK

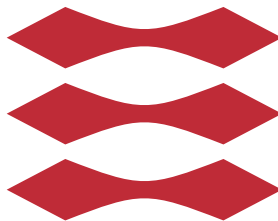
PHD

**Virtual Reality-Based Visualization of
Large Geometric Data**

By:

Mark Bo Jensen

DTU



October 31, 2022

Virtual Reality-Based Visualization of Large Geometric Data

October 31, 2022

Author:

Mark Bo Jensen

Main supervisor:

Associate Professor J. Andreas Bærentzen
Technical University of Denmark

Co-supervisor:

Associate Professor Jeppe Revall Frisvad
Technical University of Denmark

©2022 Mark Bo Jensen

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Understanding and forming new hypotheses about data are two fundamental aspects shared by many scientific disciplines. When working with surface-based data such as geometric data, these aspects are largely facilitated by visualization. Visualization however, becomes more challenging as data complexity increases. To tackle this complexity we need new ways of visualization that simplify data exploration and thereby arguably aid inductive reasoning. Virtual Reality has the potential to be one such way.

When visualizing complex geometric datasets in Virtual Reality they often end up being simplified. It is perhaps because of the performance requirements of Virtual Reality and the use of existing visualization tools, such as game engines. Game engines are evolving to become more general-purpose tools but have previously prioritized visualizing many objects rather than large objects. This has led to a workflow that includes a data simplification step, creating a paradox for data visualization, where the data we wish to inspect needs to be simplified first.

In this thesis, we explore the landscape of existing visualization tools, game engines, and graphics application programming interfaces. Based on our findings we put forth principles and recommendations for the development of new Virtual Reality-based visualization tools.

In this process, we developed a bespoke tool for Virtual Reality-based visualization. We show how this tool can be used with large geometric datasets, forgoing the simplification step. We likewise explore the perceived difficulties of creating bespoke Virtual Reality-based visualization tools. In doing so we demonstrate that Virtual Reality has become a mature alternative to existing solutions for visualization.

An advantage of Virtual Reality is the 6 Degrees of Freedom interface that it provides. The interface allows for a more intuitive way of interaction but requires new interaction paradigms if we wish to take full advantage of it. We discuss how to use the affordances of portals to take advantage of this interface. As well as explore how precise interaction in Virtual Reality is when compared to desktop solutions.

Finally, we present a novel mesh processing algorithm that is simple to implement and results in high performance when used in practice on new hardware.

Danish summary

At forstå og udforme nye hypoteser er to fundamentale aspekter som mange videnskabelige discipliner har til fælles. Når man arbejder med overflade-baserede datasæt som geometriske datasæt, er disse aspekter primært faciliteret via visualisering. Visualiseringen i sig selv bliver mere udfordrende i takt med at datasættene bliver mere komplekse. For at kunne håndtere dette har vi brug for nye måder at visualisere datasæt på. Måder som simplificerer udforskningen af datasættene, hvilket kan siges at underbygge en logisk induktiv tænkemåde. Virtual Reality har potentialet til at blive en sådan ny måde.

Når man visualiserer komplekse geometriske datasæt i Virtual Reality ender det ofte med at man er nødt til at simplificere datasættene. Det kan blandt andet være fordi der er strenge krav til ydeevnen i Virtual Reality og på grund af brugen af eksisterende visualiseringsværktøjer, så som spilmotorer. Spilmotorer er ved at udvikle sig til at blive mere generelt anvendelige, men har tidligere prioriteret at visualisere mange objekter frem for et stort, som netop data visualisering har behov for. Det har ført til en proces inkluderer et data-simplificerings-trin. Dette skaber et paradoks for data-visualisering, hvor man er nødt til at simplificere den data man gerne vil inspicere.

I denne afhandling udforsker vi landskabet af eksisterende visualiseringsværktøjer, spilmotorer, og grafik applikations programmerings grænseflader. Vores mål er at bruge resultaterne til at identificere principper og anbefalinger til at guide udviklingen af Virtual Reality-baserede visualiseringsværktøjer.

Som en del af den proces har vi også udviklet vores eget værktøj til visualisering i Virtual Reality. Vi viser hvordan dette værktøj kan bruges til at visualisere store geometriske datasæt, og samtidig springe simplificerings-trinet over. Dermed har vi anskueliggjort at forskere med god grund kan anvende Virtual Reality som et alternativ til eksisterende visualiseringsværktøjer.

En fordel ved Virtual Reality er, at det tilbyder en brugergrænseflade med 6 frihedsgrader. Denne brugergrænseflade tillader en mere intuitiv måde at interagere med data på. For at udnytte de 6 frihedsgrader fuldt ud har vi brug for nye interaktionsteknikker. Vi diskuterer derfor i denne afhandling, hvordan portalers egenskaber kan anvendes til at skabe en sådan ny interaktionsteknik til Virtual Reality-baseret interaktion. Vi udforsker også hvor præcis interaktion er i Virtual Reality sammenlignet med interaktion på en computer.

Til sidst præsenterer vi en ny algoritme til processering af geometriske datasæt, som er simpel at implementere og resulterer i en høj ydeevne, når den bliver brugt i praksis på nyt hardware.

Preface

This Ph.D. thesis has been submitted to the faculty of the Technical University of Denmark (DTU) in partial fulfillment for acquiring the degree of Doctor Philosophy in Computer Graphics. The work presented in this thesis has been carried out from September 2019 to October 2022. It has been funded by Advokat Bent Thorbergs Fond(ref. 66.531) and the section for Visual Computing at DTU.

This thesis has been supervised by Associate Professor J. Andreas Bærentzen and co-supervised by Associate Professor Jeppe Revall Frisvad, both from DTU. All research activities have taken place at the section for Visual Computing at DTU, except for an external stay which was graciously hosted by Professor Karan Singh at the Dynamic Graphics Project laboratory at the University of Toronto (UofT).

Kongens Lyngby, 31th October 2022
Mark Bo Jensen

List of contributions

The list of publications below has been submitted as part of this thesis.

Peer Reviewed

- Paper I Mark Bo Jensen, Egil I. Jacobsen, Jeppe Revall Frisvad, and Jakob Andreas Bærentzen. 2021. **Tools for Virtual Reality Visualization of Highly Detailed Meshes**. *VisGap - The Gap between Visualization Research and Visualization Software*. Eurographics Association, pp 19-26. <https://doi.org/10.2312/visgap.20211088>
- Paper II Dolores Messer, Michael Atchapero, Mark Bo Jensen, Michelle S. Svendsen, Anders Galatius, Morten T. Olsen, Jeppe Revall Frisvad, Vedrana A. Dahl, Knut Conradsen, Anders B. Dahl, and Jakob Andreas Bærentzen. 2022. **Using virtual reality for anatomical landmark annotation in geometric morphometrics**. *PeerJ* 10:e12869, 23 pages. <https://doi.org/10.7717/peerj.12869>

Under Peer Review

- Paper III Mark Bo Jensen, Jeppe Reval Frisvad, and Jakob Andreas Bærentzen. 2022. **Efficient Rendering of Large-Scale Geometric Data using Meshlets**. *The Journal of Computer Graphics Tools*, 21 pages.

To my Daughter, Emma Linea Jensen.

Acknowledgement

This dissertation has been three years in the making. Those three years would not have been the same without the many people in my life that have helped, motivated, and challenged me to produce the best possible results. I would like to acknowledge the most important people.

To my supervisor, Jakob Andreas Bærentzen, thank you for the many great discussions, insights, and guidance throughout this project without which I would not have been able to produce the results that I have.

To my co-supervisor Jeppe Revall Frisvad, thank you for your enthusiasm, encouragement, discussions, and the many hours of work you have put into this project.

A big thank you goes out to my colleagues at the section for Visual Computing. You have made my time working in the section delightful. I have enjoyed all the ritualistic coffee expeditions, the small talk in the hallways, the banter in our (very)open floor plan office, and the thoughtful discussions.

I would like to thank Karan Singh for supervising my external stay at DGP in UofT, and the fine folks at DGP. Especially Karthik Mahadevan, Arnav Verma, Bingjian Huang, Jiannan Li, Maurício Sousa, and Majeed Kazemitabaar for the warm welcome, the many good discussions, and the trips to the nearby coffee shops.

I would like to thank Mads, and Anders for taking the time to proofread this dissertation, for your friendship, and for our many chinwags. I am grateful to my family, for supporting me through this project. Especially Ida, who has been the backbone of our family while I was far off in Virtual Reality for days on end.

Contents

Summary	i
Danish summary	iii
Preface	v
1 Introduction	1
1.1 Motivation	1
1.2 Scope	2
1.3 Goal of the thesis	3
1.4 Outcome of the thesis	4
1.5 Structure of the thesis	5
2 Background	7
2.1 Virtual Reality	7
2.2 Virtual Reality Systems	7
2.3 Virtual Environments	9
2.4 Spatial Presence and Immersion	9
2.5 Interacting with Virtual Environments	10
2.6 Visually Induced Motion Sickness	15
2.7 Motion-to-Photon Latency	17
2.8 Geometric Data	18
2.8.1 Real-time Rendering	20
2.8.2 A Brief History of Real-Time Graphics	23
2.8.3 Game Engines and Visualization Tools	24
2.9 Scientific Visualization of Geometric Data	25
2.10 Scientific Visualization in Virtual Reality	31
3 Annotation in Virtual Reality	35
3.1 Introduction	36
3.2 Materials and Methods	39
3.3 Results	46
3.4 Discussion	51
3.5 Conclusions	54
3.6 Future work	55
3.7 Appendix	56
3.8 Retrospective	56

4	Tools for Virtual Reality Visualization of Highly Detailed Meshes	59
4.1	Introduction	60
4.2	The Graphics Pipeline	62
4.3	VR Visualization Tools	64
4.3.1	Auxiliary Tools	64
4.4	Experiment Setup and Results	65
4.4.1	Vertex Shading vs Mesh Shading	68
4.4.2	Index Buffer order and Mesh shaders	69
4.5	Ray tracing	69
4.6	Discussion and Conclusion	71
4.7	Retrospective	73
5	Jinsoku: A bespoke Visualization Platform for Virtual Reality	75
5.1	Introduction	75
5.2	Jinsoku	76
5.2.1	Architecture	77
5.2.2	General optimizations	79
5.2.3	Static Registration	81
5.2.4	Scripting	82
5.3	Rendering For VR	82
5.3.1	Variable Rate Shading	83
5.3.2	Multi-View Rendering	84
5.3.3	Performance	85
5.3.4	Conclusion	86
6	Efficient Rendering of Large-Scale Geometric Data using Meshlets.	89
6.1	Introduction	91
6.1.1	Related Work	92
6.2	Meshlets Descriptors	94
6.3	Meshlet Clustering Methods	97
6.4	Experimental Setup	99
6.5	Results	100
6.6	Discussion	108
6.7	Conclusion	110
6.8	Retrospective	110
7	Portals: A Swiss Army Knife for Scientific Visualization	111
7.1	Introduction	111
7.2	Portals	112
7.3	Portal-Based Interaction with Scientific Data in VR	113
7.4	Case Study	115
7.5	Discussion	117
7.6	Conclusion	120

8 Conclusion	121
I Tools for Virtual Reality Visualization of Highly Detailed Meshes	123
II Using virtual reality for anatomical landmark annotation in geometric morphometrics	133
III Efficient Rendering of Large-Scale Geometric Data using Meshlets	157
A Different appendix	183
Bibliography	185

CHAPTER 1

Introduction

1.1 Motivation

Visualization is an integral part of almost every step of the scientific method. It allows us to leverage the visual system to see and explore data. Inspecting data facilitates inductive reasoning and exploring it is a crucial part of forming new hypotheses about the underlying phenomena represented by the data.

As the datasets we wish to visualize grow in size and complexity, visualization itself becomes increasingly demanding. This forces researchers to not only create more efficient visualization tools, but also explore different ways of visualizing data. Virtual Reality (VR)-based visualization is one such way, that actually becomes more useful as the complexity of the data increases (Elden, 2017).

In 1996 Bryson (1996) published a paper on the use of VR for the visualization of scientific data, showing the potential of VR. Three years later F. Brooks (1999) proclaimed that VR barely worked and had only found application in a few niche industries. In other words, the computational power of the hardware used for the VR technology was not powerful enough to make good on the promise of VR. Hardware has continued to develop, as stated by Koomey's Law (Koomey et al., 2011), doubling its processing power every 30 months between 1948 and 2010. Resulting in many times more powerful hardware. The same can be said for VR technology, as large companies have invested heavily in it. Head-mounted displays (HMD) are now being sold at consumer prices with features significantly better than the elaborate setups of the early 2000s. Modern HMDs utilize advances in hardware, computer vision, and machine learning to provide the user with higher resolution, better refresh rates, and more precise and faster head and controller tracking of both position and orientation. This development has matured VR, and we believe that it is at a point where it can live up to its potential.

With the vast improvements of modern HMDs, interaction and navigation with VR becomes much more effortless and direct. The six Degrees of Freedom (DoF) tracking of the controllers and headset can be mapped directly to the six DoF required to navigate in a 3D Virtual Environment (VE). Because of the fast head tracking the view of the VE is updated seemingly as the user moves the head, creating a very convincing experience. Walking in the real world or reaching out with a controller to grab and rotate an object is directly reflected in the VE, making the interaction much more intuitive than using a mouse and keyboard. This makes it easy and fast to get the desired view in VR.

The process is similar to how we inspect and look at objects in the real world, making VR a good choice for visualization.

Combining this intuitiveness of VR with a HMDs ability to isolate the user in a VE with no distractions can lead to less mental effort being spent on interfacing with the application and more mental effort being available for inspection and exploration (Hutchins, Hollan and Norman, 1985).

Using VR however, does not come without constraints. VR experiences need to maintain low latency between movement tracking and displaying an updated image, reflecting tracked movement in the HMD. This is further exacerbated by the need to render two new images every time, one for each eye. If the latency, often referred to as the Motion-to-Photon latency, becomes too great the experience becomes unusable and the user might experience symptoms of motion sickness. To guarantee low Motion-to-Photon latency, complex models and datasets are often simplified when used in conjunction with VR. Jiménez Fernández-Palacios, Morabito and Remondino (2017) present a visualization pipeline for complex heritage 3D models in VR. The pipeline has an incorporated step that simplifies the models, reducing models of 1-3 million triangles to less than 1 million triangles each. The high-frequency details of the models are instead baked into texture maps. These texture maps can then be used to give the illusion of a more detailed model. Because the textures do not alter the silhouette of the model, the illusion breaks down when a model is viewed up close or at odd angles, which is exactly the type of interaction that VR affords. Furthermore, when working with scientific data, the visualization becomes pointless if the data we wish to inspect has to be simplified first.

VR technology, in other words, is a great tool for the visualization of large complex datasets. But because of the hard constraints imposed on VR by the Motion-to-Photon latency, we cannot take full advantage of VR. For VR to become a viable tool for the visualization of scientific data we need to have more focus on building visualization platforms centered around high performance. Only then can we start exploring all the affordances that VR-based visualization promises.

1.2 Scope

As data can be collected with increasingly finer details, the resulting datasets grow larger and more complex. Our goal is to be capable of visualizing such large datasets. Large datasets require longer processing time, which can be challenging when working with VR. Because of this, we must align ourselves with the core specialty of Graphics Processing Unit (GPU) hardware, namely with datasets consisting of geometric data. That way we can leverage the fast processing times afforded by modern GPUs for processing. Geometric data is of course not the only way to represent data, and not

always the best. It has certain drawbacks. Geometric data, or surface-based data, is more often than not a discrete approximation of the underlying continuous phenomenon that is being measured. Section 2.8 gives a more thorough introduction to geometric data. Another reason for this choice is that when working with geometric data there is a tendency to decimate the data before showing it, which is not ideal when visualization is the goal. Often when geometric data is used, the visualization tool is built on an existing platform, such as a game engine, which works natively with geometric data. In section Section 2.10 we explore a cross-section of different VR-based visualization tools, and 13 out of 15 visualization tools were built on existing platforms. Out of those 13, four tools work with geometric data. They are all built on a game engine. Three of them had to decimate the data to achieve good performance in VR. The last expresses a desire to move to a bespoke visualization tool in the future. The practice of decimating meshes comes from limitations and game engines being built to handle many meshes rather than a single big one. This is apparent from the fact that Unity did not introduce the ability to use large meshes until an update at the end of 2017 (Unity, 2017). We aim to show that the decimation process is not required until meshes become much larger than what is currently being decimated, and with that, widen the usability of VR.

Section 2.2 gives an introduction to different VR systems. These systems vary in cost and convenience. In this project, we strictly use consumer-grade HMDs. They are affordable, easy to set up, and widely used. The VR experience afforded by a HMD is quite intuitive. Modern HMDs often feature headset tracking in the actual headset meaning that it is convenient to use even without a dedicated VR space, and because we aim to collaborate with other scientists, this allows us to bring a headset with us for easy evaluation. While it is possible to create multi-user VR experiences, we also restrict this project to single-user VR experiences. We provide the ability to see the VR user's point of view on a regular computer screen.

We use geometric data from two different scientific fields. The first is digital heritage where using scanned 3D models makes it possible to perform what would otherwise be destructive processes. An example of such a process is landmark annotation which is done by marking the object. A process that is very reliant on the ability to visualize the objects with as many details as possible. The other scientific field is shape and topology optimization. Here advanced computer-driven optimization is used to derive load-bearing structures. These structures, while difficult to manufacture, require inspection for the scientists to identify key areas that can drive future research.

1.3 Goal of the thesis

The goal of this thesis is to investigate the viability of VR-based visualization in the context of large and complex datasets. Due to the heterogeneity of scientific visualiza-

tion, we do not constrain ourselves to one field, instead, we focus on geometric data. Because we find that when working with geometric data, it is quite common to use existing graphics engines as these widely support geometric data. We aim to investigate whether existing graphics engines can keep up with the increasingly large and more complex datasets that emerge, or whether it is necessary to build bespoke VR applications to ensure a better experience. We aim to investigate this by exploring the following questions:

- What is the best foundation for a VR-based visualization platform?
- What challenges and benefits are gained by building a bespoke VR-based visualization platform?

Interaction is another crucial part of visualization in VR. If the user is not presented with an interface to the Virtual Environment that is easy to understand, the user might end up spending more time getting accustomed to said interface instead of interacting with the data. Another, and perhaps more interesting aspect of interaction, is the more practical approach. Namely how well does it work? Can we perform reliable, precise, and reproducible manipulations of the data in VR, assuring that the platform can help aid researchers in the visual explorative analysis of their data? This leads to the following question regarding interaction with large datasets in VR:

- What interaction modalities work best for the purpose of understanding geometric data in VR?

1.4 Outcome of the thesis

This thesis has had a large focus on investigating the technical feasibility of visualization, as well as some preliminary experiments carried out on interaction in VR. This has led to several academic papers, of which an overview can be found on page vii. Each paper has been included as a chapter.

The outcome of this thesis can be divided into two parts. The first part focuses on the technical challenges surrounding the visualization of large geometric datasets. Paper I dives deeper into the importance of choosing the right platform. Paper III shows how leveraging state-of-the-art graphics processing hardware has enabled us to widen the scope of VR applications, namely by allowing visualization of larger geometric data sets. We show that great performance gains can be had by building bespoke platforms, but also show that game engines can be utilized to a big extent if care is taken to optimize the application.

The second outcome of this thesis focuses on exploring the challenges of interaction

in VR. In Paper II we have conducted experiments that compare how precise a VR controller is compared to a mouse and find that annotation in VR is indeed no less precise. This opens up human-in-the-loop annotation-based tasks which are used in several fields. In Chapter 7 we show our initial findings of how VR interaction can be done more immersively, directly, and intuitively with the use of portals. We also discuss how portal-based interaction is a promising technique that can be implemented for interaction regardless of the data being visualized. This would allow VR users to rely on previous and conceptual experience with portals when using different VR applications. Having previous experience to rely on can greatly help increase the learning rate of new VR experiences.

Overall, our work has widened the application space of VR. Together these two parts show that VR-based visualization holds great promise to change the way we explore and visualize geometric data.

1.5 Structure of the thesis

We have divided the thesis into chapters that build on each other, creating a common thread throughout. In Chapter 2 we go through the background knowledge that is necessary to understand the different parts that go into building VR-based experiences. The chapter also introduces all the prerequisites for understanding the contributions of this thesis. Chapter 3, Chapter 4, and Chapter 6 are the 3 papers that have been submitted to peer-reviewed research outlets as a part of this thesis. Chapter 5 Introduces the visualization platform that has been built for conducting the experiments presented in this thesis. Chapter 7 introduces portals as a comprehensive tool for VR-based visualization and interaction.

CHAPTER 2

Background

2.1 Virtual Reality

Ivan Sutherland (1965) defined the ultimate display as the following: "The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked."

Sutherland's room, wherein a computer manipulates matter, would be real and not virtual. Instead, a virtual room would be in essence but not in fact. That would make the bullet harmless. It would however be possible to make the virtual bullet seem real, by making it look like a bullet, sound like a bullet, and submit to physics in approximately the same way as a real bullet. Despite not being able to manipulate matter, modern computers can simulate not only a virtual bullet but entire environments. Virtual environments simulating everything from the familiar forces of the physical world to the unfamiliar uncertainties of quantum mechanics. Merriam-Webster (2022) defines virtual reality as "an artificial environment which is experienced through sensory stimuli (such as sights and sounds) provided by a computer and in which one's actions partially determine what happens in the environment". In this thesis, these virtual environments are what constitute the virtual reality.

2.2 Virtual Reality Systems

There are different ways of presenting the user with a virtual environment. Typically these come with different advantages and disadvantages. One of the first VR systems was created in 1967 by the same Ivan Sutherland (1968). The system was capable of showing the user wire-frame objects. The images of these were displayed in stereo on an optical system attached to the user's head. The optical system used two cathode-ray tubes, miniature versions of the screen used in old television sets and computer screens, for displaying the images. Because of this, the background was transparent. The perspective view of the objects was updated based on tracking of the users' head movement. The headset was rather heavy, resulting in the need for attaching it to a big robotic arm suspended from the ceiling. The system provided the user with the freedom

to move around freely, to the extent allowed by the robotic arm. While being far from the ultimate display, this prototype showed that it was possible to start exploring novel setups for presenting virtual environments to the user.

Later another approach that sacrificed the freedom of movement for higher fidelity was the fish tank VR system. The fish tank VR system provided a seated experience in front of a computer screen, but produced high-resolution images and allowed interaction through mouse and keyboard. The stereoscopic effect was achieved by using a lightweight active shutter headset, that switched between blocking one of the eyes in synchronisation with the screen's refresh rate (M. Deering, 1992; Ware, Arthur and Booth, 1993). The stationary aspect of the fish tank VR system allowed for precise head tracking.

The CAVE Automatic Virtual Environment(CAVE) setup, provides a more captivating, but also more circumstantial VR setup, than the fish tank VR system. It does so by combining the freedom to move around, with a high fidelity system. The CAVE projects between 4 and 6 views on the insides of a cube. The user, who is inside the cube, can then freely roam around the cube. Other than requiring projectors and screens, the user also needs to wear an active shutter headset that is synchronised with the refresh rate of the projectors (Cruz-Neira et al., 1992). the CAVE system allows for multi-user VR experiences if the projectors have a high refresh rate and the users are equipped with active shutter headsets that are synchronized with the projectors refresh rate at a fixed offset with each other. Unlike the fish tank VR system the CAVE system is still widely used. Figure 2.1b shows a CAVE setup.

The Fakespace labs Binocular Omni-Orientation Monitor(BOOM) is a way to create a more seamless multi-user experience without requiring an elaborate setup or a lot of headset switching, making it more suited for an office-type environment. The BOOM is a counterbalanced box that contains an immersive stereo display. It is easy to move in place with the hands, and it is more comfortable to use as it does not require the user to wear a headset (Bolas, 1994).

The HMD most commonly used for VR today combines the advantages of several of these predecessors into one system. They provide the freedom to move around from the CAVE, the high-resolution images and precise tracking of the fish tank. All with a form factor and portability that makes it almost as comfortable and office-friendly as the BOOM. Figure 2.1a shows an example of a modern HMD with controllers.



(a) A picture of a HMD called the VALVE index



(b) A picture of a CAVE setup(Courtesy of Davepape)

Figure 2.1: Examples of two different VR systems

2.3 Virtual Environments

A Virtual Environment(VE) is a synthetically created environment that is presented to the user through a virtual reality system. A synthetic reality is a more natural form of human-computer interaction. One that has the potential to fundamentally change the way a person works with a computer because it allows the user to interact with the computer in a more intuitive and direct format that can potentially increase the interactions per unit of time (Foley, 1987). A VE can be considered a type of user interface. Similar to the Graphical User Interfaces(GUI) that desktop applications have. The big difference between these two types of interfaces is their affordances. The VE is a direct user interface, whereas the GUI is more often considered an indirect interface. A direct user interface allows the user to directly interact with the data. An indirect user interface is one where the user indirectly manipulates the data. This could be through manipulating values in boxes that then affect the data. Designing good VEs can be difficult, not only because of their direct nature but also because the user is left to explore them on their own. This demands some intuitiveness of the environment as the user is often enclosed in the virtual environment which makes it hard to receive guidance.

2.4 Spatial Presence and Immersion

Plato's cave analogy has served as inspiration for some of the earliest philosophical explorations of perception and has also lent its name to some of the first endeavours into virtual reality. The CAVE system, which mimics the idea of projecting the virtual world onto screens that surround the user (Cruz-Neira et al., 1992), takes its name from Plato's allegory of the cave. Even though VR systems have come a long way since a lot of the research could be said to address some of the philosophical questions that were raised

by Plato. This is primarily the research that surrounds presence and creating immersive VEs that try to erase the boundary between the real and virtual. Presence in virtual reality is a big enough research topic to have had its own journal aptly titled *Presence: Teleoperators and Virtual Environments*. Presence in general is often described as the feeling of "being there" (STEUER, 1992). K. M. Lee (2004) decoupled presence from technology and has established a more abstract explication of presence, intending to create a common ground for all presence research. Wirth et al. (2007) further extend this and argue that a commonly accepted model of spatial presence is the only solution to further interdisciplinary presence research. Spatial presence is then defined by Wirth et al. as a binary experience, during which perceived self-location and, in most cases, perceived action possibilities are connected to a mediated spatial environment, and mental capacities are bound by the mediated environment instead of reality (Wirth et al., 2007).

It is however quite hard to objectively measure how present the user is. This also goes for presence in a VE. Because of this a widely adopted view is that a higher feeling of presence can be achieved through a higher level of immersion. Which has led to a lot of research into creating more immersive experiences. Slater and Wilbur (1997) define immersion as something that can be objectively assessed through how *Inclusive*, *Extensive*, *Surrounding*, and *Vivid* it is. Cummings and Bailenson (2016) have performed a meta-analysis on studies that test the effect of immersion on spatial presence, and found a medium-sized effect. All studies in the meta-analysis have been done using self-reporting through questionnaires. Different questionnaires have been developed in an attempt to tease out the level of presence experienced by the user, among these are the Slater-Usuh-Steed Questionnaire (Slater, Usuh and Steed, 1994), the MEC Spatial Presence Questionnaire (Vorderer et al., 2004), and the Temple Presence Inventory (Lombard, Ditton and Weinstein, 2009).

2.5 Interacting with Virtual Environments

If we look at the VE relative to the user, we can divide it up into three egocentric circular volumes, the personal space, the action space, and the vista space (Cutting and Vishton, 1995). The personal space has a radius of about 2 meters and is the natural working volume. The action space starts at around 2 meters and extends to 20 meters. This volume is the public space that we can easily move to, toss objects to, and interact with others. From 20 meters and beyond we have the vista space, where we have little control, and binocular vision is almost non-existent (Jerald, 2015). This can be a good starting point both for VE design but also for discussing interaction with VEs. The personal space, within arms reach, has many advantages such as proprioceptive information, direct mapping between movement in the physical and virtual world, strong depth perception and displacement cues due to head-movement, as well as small re-

quirements to the physical VR space (Mark R. Mine, F. P. Brooks and Sequin, 1997). Because of this we limit the interaction abilities discussed here to those that work in the personal space.

Mark R Mine (1995) describes three categories within which the fundamental forms of interaction with a VE fall. *Direct User Interaction* where the normal movement and action of the user is mapped to actions in the virtual world in an intuitive manner. *Physical Controls* where the user uses a physical interface such as controllers, joysticks, or a steering wheel. These physical interfaces provide the user with passive or active haptic feedback that can result in an immersive experience if they are well integrated into the VE. *Virtual Controls*, which facilitate interaction through widgets or other virtual objects that the user can interact with. Since these lack the haptic feedback of physical controls, they can be harder to interact with. Different fundamental forms of interaction, such as manipulation, selection, movement, etc. can then be implemented in a way that fits into one of those categories.

We are especially interested in the *Direct User Interaction* category. Because we want to take advantage of the intuitiveness that VR affords when it comes to mapping user movement in the physical world to navigation in the virtual world. Moving from an indirect to a direct interface, such as a VE, we can create interactions that require the commitment of fewer cognitive resources to the interface. Instead, these can be committed to the actual interaction with the data (Hutchins, Hollan and Norman, 1985). *Direct User Interaction* requires that we have the ability to track the user's movement. We can do this by using a HMD and a controller attached to each hand. Modern HMDs and controllers have six DoF tracking. Head movement maps to moving the virtual camera, while the controllers can move and rotate the object and serve as an annotation tool as well. Controllers fit into both the *Direct User Interaction* and *Physical Controls* categories. We can track the controllers to get the physical hand position, and we can make use of the buttons on the controller. The fact that the user has two controllers also opens up for a bimodal interaction modality that is not really possible when using a mouse and keyboard. This type of interaction is very straightforward since it for instance allows you to rotate and manipulate an object with one hand while painting or annotating it with the other hand. If the same interaction were to be done with a mouse and keyboard it would turn from a simultaneous interaction into a sequential interaction where the user first manipulates the camera to find the the desired view point, and then paints or annotates the object from that view point. See Section 2.9 for more detail on the navigation techniques and challenges of using a keyboard and mouse for interaction with a 3D environment.

Portals, two interconnected window-like gateways that exist in the VE as tangible objects, are an interesting way of facilitating interaction in VR. The user can move between the two gateways discreetly by entering either portal, as well as see through them. In Section 2.9 we explore navigation of geometric data with a keyboard and mouse. In that setting the user is often presented with more viewports, or windows, that show the

mesh. The viewport-grid mimics the multi-screen setup that has become the standard on almost all modern desktop computer stations. Affording the user several different views of the data. This is a quite efficient way of gaining an overview and understanding of the data. In VR, where the camera is directly tied to the user's head, we cannot in the same way utilize this multi-screen setup. Portals can however, be utilized in a similar way. *Poros* is a very interesting implementation of portals that does this by allowing the user to interact with distant objects through portals without moving entirely through the portals, mimicking a multi-screen setup in VR quite well. They also allow the user to perform the same action through several portals at the same time (Pohl et al., 2021). *Photoportals* is a reference-based interaction tool that allows users to collaboratively investigate a 3D scene in VR (Kunert et al., 2014). *Photoportals* can be used either as flat or volumetric portals, and work as a general purpose tool for inspection, teleportation, and manipulation in a 3D VE. *Worlds-in-Wedges* introduces comparative analysis of spatial data to VR with an extension to volumetric portals, where the hemisphere around the user is divided into wedges. Each wedge is then a portal into a world that can be visually compared to the worlds in the other wedges (Nam et al., 2019). Instead of teleportation, the wedges allow the user to move the viewpoints around which allows exploration of the VE contained within a wedge. Virtual portals have been used for architectural visualization to create an immersive way of moving between a miniature and the real world scale of architectural 3D models (Bruder, Steinicke and Hinrichs, 2009). *Reality Portals* have been used for augmented VR, where a real world video feed is shown in a small monitor in a VE (Åkesson and Simsarian, 1999). The robot that the camera is attached to can be controlled from the VE. Section 7.2 elaborates more on Portals.

Interaction can also be done through navigation, and in the setting of data visualization, navigation becomes really important. Navigating in VEs is largely based on a combination of moving around in a small physical space and moving around in a wide-reaching virtual environment, putting a large demand on good navigation. In this section we explore different techniques that can bridge that gap by moving the user from the personal space to the surrounding action space. Navigation can have three different purposes. *Exploration* which is navigating without an explicit goal, *Search* which is navigation to an explicit goal, and *Maneuvering* which is short-range high-precision movement used to position the viewpoint better (D. A. Bowman, Kruijff et al., 2001). Because the camera position is tied to the head movement in VR *Maneuvering* becomes effortless. *Exploration* and *Search* on the other hand do not.

Navigation can be broken down into two components, *Travel* and *Wayfinding*. *Wayfinding* combines the immersive characteristics of the VR system with environmental cues, maps, and compasses that are used in real-world wayfinding. *Wayfinding* is crucial for *Search*-based navigation, but we are more interested in *Exploration*-based navigation, which is more reliant on the actual mechanics of travel. So in this section we focus on the *Travel* component of navigation. We explore this by going through the five categories that D. A. Bowman, Kruijff et al. (2001) break *Travel* down into.

Physical Movement

Physical Movement is where the user moves through the VE by physically moving the body. This type of movement is by far the most immersive way of travel within a VE (Usoh et al., 1999; Wilson et al., 2016), but is often dramatically constrained by the physical space of the VR system. To circumvent this limitation, without drastically expanding the physical space, elaborate locomotion interfaces can be used to create a setup where the user can essentially walk infinitely. The Cyberith Virtualizer omni-directional treadmill uses a low friction base and a belt that is fixed at the users hip (Cakmak and Hager, 2014). The user can then lean into the belt at the hip and start walking. Another locomotion interface is a walking-pad, a flat platform which the user stands on, and by stepping, and jumping on it the user can travel around the VE (Bouguila et al., 2005). This can also be achieved by simply walking in place (Slater, Usoh and Steed, 1995). Another even more elaborate locomotion interface is the Torus Treadmill (Iwata, 1999). The Torus Treadmill is a 2 meter by 1.8 meter assembly of 12 treadmills, each driven by its own motor. The treadmills are mounted on an endless rail to allow the user to walk both along and across them. Instead of using locomotion interfaces, the VR system can also utilize the limited physical space better by employing redirected walking where the user is walking straight in the VE but due to visual cues walk in circles in the real world (Razzaque, 2005).

Manual Viewpoint Manipulation

Manual Viewpoint Manipulation is where the users hand motion is used for travel. Grab & Drag is one way where the viewpoint is moved by grabbing and dragging the VE around (C. H. Lee, A. Liu and Caudell, 2009). Another way is to perform arm swings (McCullough et al., 2015), or hand flapping (Bozgeyikli et al., 2019). Two approaches that bridge the gap between interaction and navigation by allowing the user to stay in the personal space but interact with objects in the action space are; *the go-go interaction* technique where the user can stretch their arms to grab objects and pull them close (Poupyrev et al., 1996), and *scale-world grab* in which the VE scales down around the hand of the user, by a factor that is determined by how extended the users arm is. So that when the user has his arm fully extended the objects furthest away can be grabbed, pulled in, and manipulated (Mark R. Mine, F. P. Brooks and Sequin, 1997). While not technically travel techniques they accomplish the same task by way of hand motion, which is why they are described in this section. Manual viewpoint manipulation-based travel does have a higher chance of exerting fatigue in the user.

Steering

Steering is where the user moves continuously in a direction. A controller can be used to manipulate translation, independently of the head orientation (Buttussi and Chittaro, 2021). The velocity can be determined simply by the direction the user is leaning in as well as the extend of the lean (Fairchild et al., 1993). A similar result can be achieved with a Wii Fit Balance Board (Harris et al., 2014). The steering can also be based on the orientation of a controller used to point or the orientation of the head (Christou and Aristidou, 2017). Flying can also be performed along the orientation of the head by raising an arm (Bozgeyikli et al., 2019). This type of movement runs a large risk of inducing vection. Vection is an illusion of self-motion, and can cause motion sickness. Motion sickness will be explained in more detail in section Section 2.6.

Target-Based Travel

Target-Based Travel is where the user specifies the target, and the system then handles the actual movement. One implementation of this allows the user to move their own miniature in a World-in-miniature(WIM) model, and upon releasing the miniature the users position is updated with a transition into the miniature (Pausch et al., 1995). Teleportation, which instantly and discreetly moves the user from one position to another, is another way. Teleportation can be done by having the user point at a location with their hand for two seconds (Bozgeyikli et al., 2016) or by pointing a controller and pressing a button on it (Schnack, Wright and Holdershaw, 2021). Both solutions show the user a circle of where they will end up. Portals or Orbs can be used to create a transition between the start and end point (Husung and Langbehn, 2019). Orbs are similar to portals, but instead of being windows, they are small spheres making it possible to inspect them from any angle providing a 360 degree view of the "other end". Similarly the Jumper Metaphor allows the user to combine walking with target-based travel, the target to jump to is automatically detected via the headset orientation, and the jump animation is activated if the user moves towards the target with a reasonable speed (Bolte, Steinicke and Bruder, 2011). An elevator metaphor can be used for vertical movement (Vasylevska, Kaufmann and Khrystyna, 2014).

Target-Based Traveling is very popular in VR because it bridges the gap between the personal and action spaces well. It also induces less vection than other ways of navigation. Instant teleportation is often used, but this travel techniques has some drawbacks too. Not only does it break immersion, because of the abrupt change in location and its unrealistic nature, the abrupt change of view is also disorienting (D. Bowman, Koller and Hodges, 1997). Portals and orbs can make the teleportation effect more immersive and less disorienting, by allowing the user to see the destination through the portal and by actually walking into the portal combining the locomotion of walking in the physical

and virtual environment. Walking being an immersive way of travelling in VR (Usuh et al., 1999). Portals and Orbs scored highest in both presence retention and likability when compared to other transitions (Husung and Langbehn, 2019). Transitioning by moving an orb to the headset has even been used in several VR games already, such as The Lab, Google Earth VR, Accounting, Budget Cuts, and with *Photoportals* (Kunert et al., 2014). Portals even facilitate directed movement that allows the user to better take advantage of the limited size of their physical surroundings (Freitag, Rausch and Kuhlen, 2014).

Route-Planning

Route-Planning is where the user manipulates icons, maps or other virtual objects to plan out the route through the VE beforehand. D. A. Bowman, Davis et al. (1999) show the user a 3D map of the VE and allowing them to place points on it, before they become immersed in the same VE and automatically move in a straight line between the placed points. Route-planning for VR experiences is not that useful a tool for the exploration of scientific data. Mainly because the visualization process is used to explore the data in search of regions of interest that are unknown before exploration. Similar techniques could however be used to present the results of exploration to other users.

2.6 Visually Induced Motion Sickness

Motion sickness is the most adverse health effect that virtual reality systems can induce. Motion sickness can include symptoms such as nausea, dizziness, general discomfort, disorientation, and headaches. Motion sickness induced by apparent motion, can also be referred to as cybersickness. Vection can be the result of the travel interfaces used, such as steering-based techniques, but it can also be induced through miscalibration or latencies in the VR system. Vection is a major contributor to motion sickness as well (Jerald, 2015).

If a VR experience induces motion sickness, cybersickness or too much vection it is quite frankly unusable. Because of this, we should be very mindful of creating experiences in VR that specifically avoid exposing users to conditions that can potentially cause motion sickness. Here I present a short overview of the theories of motion sickness. For a much more in-depth exploration of the underlying biological processes and organs that help us perceive motion as well as theories of motion sickness I recommend Sharif Razzaque's thesis (Razzaque, 2005) and The VR Book by Jason Jerald (2015).

The *Sensory Conflict Theory* (Reason and Brand, 1975) revolves around a disagreement

in the perceived motion across different sensory modalities. Primarily the disagreement between the visual stimuli and the vestibular system located in the inner ear. Inside the bony labyrinth of the vestibular system sits the semicircular canals. There are three canals, sitting at right angles to each other. These are used to sense angular acceleration or rotation of the head. The utricle and saccule, also in the bony labyrinth of the vestibular system, both house what is called an otolith. These are used to sense linear acceleration, similarly to a three-axis accelerometer (Khan and Chang, 2013). When in VR the visual and auditory stimuli are coming from the virtual world, but vestibular and proprioceptive cues come from how the body moves in the real world. Proprioception is the body's ability to sense movement, action, and location. This disagreement can result in motion sickness.

The *Evolutionary Theory* (Treisman, 1977) explains that this happens because of an evolutionary trait, namely that a conflict among senses is the result of the body having been poisoned. So because our survival depends on our ability to trust our senses, the body induces motion sickness to discourage movement while it rids itself of the toxins through sweating and vomiting. A general feeling of discomfort is then felt to discourage the incident from happening again.

Three other theories focus more on the comparison of the expected state and the actual state of self-motion and the world around us. Having a lot of experience interacting with the real world has conditioned us to expect that things are in a certain way, and when this is not the case it can cause motion sickness. The *Eye Movement Theory* (Jerald, 2015) argues that motion sickness arises from the unnatural eye movement that is required to stabilize an image in VR on the retina. The visual and vestibular systems are tightly coupled, and the vestibulo-ocular reflex is a reflex that stabilizes eye gaze during head movement due to activation of the vestibular system. The *Postural Instability* (Riccio and Stoffregen, 1991) theory argues that motion sickness can be a result of a user not maintaining postural stability. Because the virtual environment provides stimuli that is different from the real world, it forces our body to adjust the small muscles to keep the user in balance. We can see an effect of this when the user is moving forward in a virtual scene, then the user tends to lean forward in the real world as well. In fact, the vestibular system also plays a role in keeping our head and body in balance (Khan and Chang, 2013). Lastly, we have the *Rest Frame Hypothesis* (Prothero and D. E. Parker, 2003), which assumes that the brain has an internal model of its surroundings, including which objects are stationary. Movement then becomes relative to these stationary objects, and when we perceive the stationary objects as moving, it results in motion sickness.

These theories can be combined into a unified model for VR-induced motion sickness, which can be seen in figure 2.2. Essentially, it shows how we constantly compare expectations, predicted changes, and sensory input with a mental model of self-motion and the world. Central processing takes bottom-up input from the senses and combines this with top-down processing from our mental model of how the world works. It also

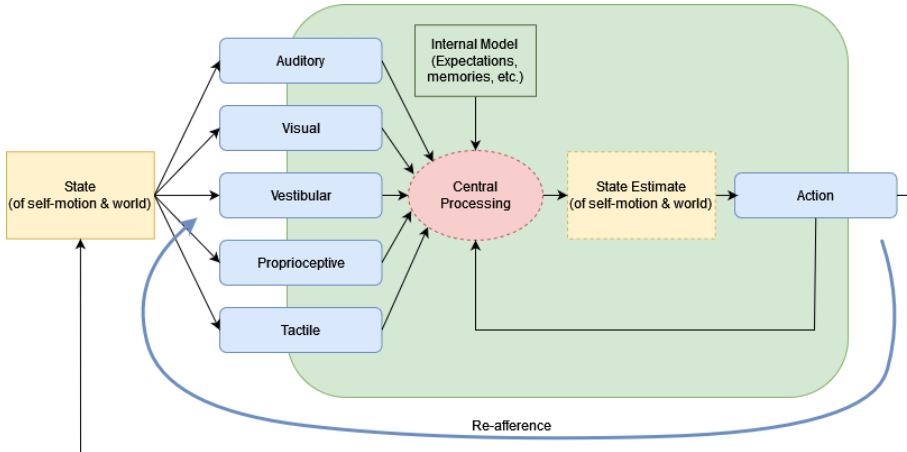


Figure 2.2: A model showing human perception of motion (Adapted from Razzaque, 2005 and Jerald, 2015)

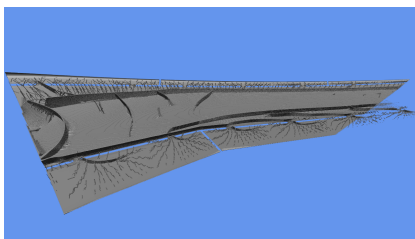
takes into account the predicted change based on our actions. The resulting state is compared to our mental model. If these are out of sync it results in motion sickness.

2.7 Motion-to-Photon Latency

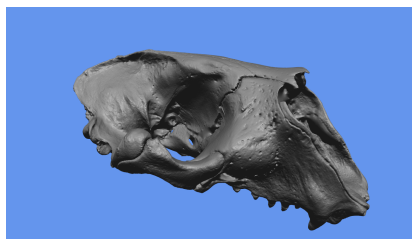
VR systems are made up of many different components and parts that have to work in conjunction with each other. We have some form of movement tracking system, that works at a certain frequency. We have a real-time system that generates images based on the positions and predicted movement of the user. These often work at different frequencies, as well as a computer and cables that transmit these to a display that yet again updates at its own frequency. This can make it quite difficult to measure how fast a VR system can actually display images. Instead of measuring independent components, we can measure the end-to-end time from a user moving their head, to a new image being presented to the HMD which reflects that movement. This is called the *Motion-to-Photon* latency (Jingbo Zhao et al., 2017). For most of our research, we can narrow this down a bit as we keep the entire VR system constant, and only replace the render engine.

2.8 Geometric Data

Throughout this project, we restricted ourselves to the use of geometric data. What we understand by geometric data, is data with inherent spatial information, i.e. meaningful width, height, and depth, where the relative size, distance, length, and shape is part of the data. more importantly we require the data to be "surface-based".



(a) A triangle mesh based on a FEM simulation of a Boeing wing. The mesh consists of 30 670 121 vertices and 38 629 758 triangles.



(b) A triangle mesh based on a 3D scan of a seal skull. The mesh consists of 7 252 445 vertices and 14 504 882 triangles.

Figure 2.3: Two examples of geometric data

The surface that is associated with a given dataset is often approximated based on underlying data, rather than being directly present in the data. If we, for instance, look at the process of 3D scanning, the resulting data is in the form of a point cloud. The acquisition of this point cloud can be through *Structure from Motion* (Schonberger and Frahm, 2016), *Structured Light Scanning* (McPherron, Gernat and Hublin, 2009), or a similar process. Afterwards the surface is then approximated through a surface reconstruction algorithm such as *Poisson* (Kazhdan, Bolitho and Hoppe, 2006) or *Voronoi*-based (Amenta, Bern and Kamvysselis, 1998) surface reconstruction. Figure 2.3b shows the surface reconstruction of a 3D scanned seal skull. Similarly when we look at cross-sectional images as acquired through different types of scanning we end up with a stack of images that we can turn into a volume represented by a voxel grid. We can approximate the surface of a volume by firstly turning the volume into a voxel grid and secondly extracting the isosurfaces of each voxel, by using an algorithm such as *Marching Cubes* (Lorensen and Cline, 1987). This approach also holds for large voxel-based simulation methods such as *Finite Element Modeling* (FEM) (Reddy, 2019). Figure 2.3a shows an aeroplane wing which is the result of a giga-voxel FEM simulation. So working with surface-based data often means working with some approximation of the actual surface.

However, many methods exist for turning most data acquisition methods into a surface based geometric dataset, and there are many good reasons to be working with surface-based data, or 3D models. Most real-time rendering applications work with 3D models. The surface of these models, or meshes, could be formed of triangles. Triangles consist of three vertices which is the minimum required to define a plane in 3D space. In other

words, a triangle is always planar. Being planar is incredibly helpful, as it lets us define a normal vector to the triangle (plane), which is essential when working with surface representations. Furthermore, it also means that the vertices are coplanar. Using this, we can define a winding of the triangle, which can be used to determine if we are viewing it from the front or the back. If we were to use other more complex surfaces we would lose the coplanarity of the shape, which means we lose the ability to compute one normal, as well as use winding information to figure out if the shape is visible. Triangles can also be represented in a great many ways that help reduce the overall memory footprint of the mesh. Lastly and maybe most importantly GPUs are incredibly fast at processing triangles.

Working with triangle-based surface meshes is closely tied to how the GPU works, and is largely motivated by the GPU's ability to hardware accelerate the processing of triangles, resulting in a high triangle throughput. Because of this, and because Paper III depends on a deeper understanding of the graphics pipeline, we now dive deeper into how we can represent triangle meshes in an efficient way, as well as how the real-time rendering pipeline actually consumes and processes the triangle meshes. Then we briefly cover the history of real-time computer graphics to gain an understanding of how the pipeline has evolved in the way that it has, and lastly we briefly explore some of the existing platforms built on this pipeline that enable us to make VR-based experiences.

One way of representing a triangle mesh can be seen in Figure 2.4. Essentially the meshes consist of vertices that are stored in the Vertex Buffer. Another buffer, called the Index Buffer, is used to store the order and indices of vertices that make up the triangles. Each triangle T in the figure is found by advancing along the index buffer. The first triangle is made up of the first three indices, the next triangle is made up of the two last processed indices, and the next index in the index buffer. That way it is possible to draw new triangles by just processing one more vertex each time. For more detail on how the GPU and mesh representations have evolved over the years see subsection 6.1.1. Each vertex at the very least needs to contain a position in 3D space, but more often than not each vertex contains a lot more vertex data. Equation 2.1 shows a vertex that, besides its position, also contains a directional vector, which is a weighted average of the normals of all the triangles that the given vertex is part of called a normal vector. Oftentimes, a vertex also contains a texture coordinate.

$$Vertex_n = \begin{bmatrix} Position_x & Normal_x & TextureCoordinate_x \\ Position_y & Normal_y & TextureCoordinate_y \\ Position_z & Normal_z & \end{bmatrix} \quad (2.1)$$

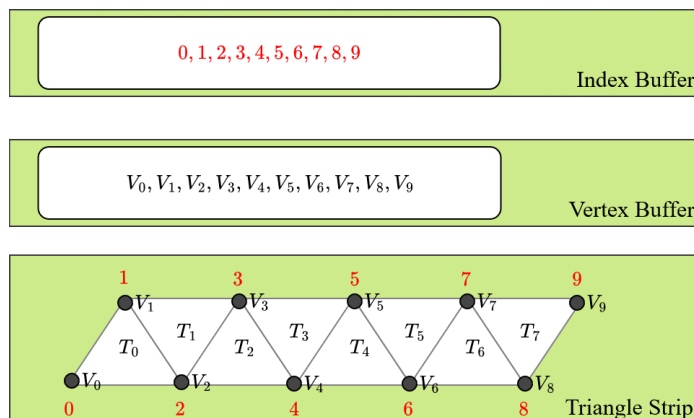


Figure 2.4: This diagram shows how a triangle strip can be converted to an index and vertex buffer. The index buffer holds indices into the vertex buffer. The vertices are used for processing triangles. The triangles are processed three vertices at a time, starting with indices 0,1,2. Then only one new vertex needs to be processed(3) for the next triangle, which is made up of indices 1,2,3.

2.8.1 Real-time Rendering

Real-time rendering is about generating new images fast. Interaction is a big part of real-time rendering, simply because a high frame rate allows for the perception of interaction, by having the subsequent images be affected by the user's actions and intention immediately. In fact, introducing as little as a 15 milliseconds delay between image updates slows and causes errors in interaction (Watson and D. Luebke, 2005), suggesting that we should aim to be at least above 66 frames per second(FPS). In the context of VR, we are then interested in creating images at a high enough frame rate to ensure fluid interaction with the VE, as well as a proper reflection of the user's head movement, and avoid inducing motion sickness. This means that we often require as much as 90 FPS when developing VR experiences (Akenine-Möller, Haines and Hoffman, 2018). Such a high FPS becomes even more challenging when we take into account rendering a new image per eye. This doubles the FPS needed to a staggering 180 FPS.

Real-time rendering and computer graphics have since the early 1990s taken advantage of the GPU for hardware acceleration to increase the frame rate. With modern GPU architectures, several different approaches to creating real-time computer graphics applications have become more viable. Hardware rasterization of triangles is however by far the most common approach when we are aiming at rendering objects at the high frame rates required by VR. Rasterization is the projection of the triangles in a triangle mesh to an image plane in which a rectangle is divided into pixels. After projection, the triangle is rasterized into fragments that each cover a pixel (if any). Another approach is *Ray Tracing* (RT). RT is a rendering technique where we trace one or more

rays through each pixel. Rays can also reflect and refract around the scene and collect colour information for their pixel of origin. GPU RT provides a hierarchical mesh representation through hardware accelerated *Bounding Volume Hierarchy* (BVH) that scales well with large models. Lastly, a combination of either rasterization or RT can be made in combination with machine learning techniques to increase resolution, fidelity, or frame rates. A graphics rendering pipeline, or rendering pipeline, is a conceptual model which is used to describe the different stages that are used for turning a 3D scene, which is viewed from a virtual camera, into a 2D image. In this thesis, we primarily use the rasterization pipeline. Two variations of the rasterization pipeline exist, but common for both is that they consist of a combination of programmable and fixed function stages. The programmable stages are referred to as shaders. In essence, shaders are small self-contained programs that are executed by the GPU hardware. Depending on the stage in the sequence different types of data may be exposed to that stage which can be manipulated before it is pushed further down the pipeline. The fixed function stages of the pipeline have different functionality depending on where in the pipeline it is, but equal for all of the stages is that they are not programmable. The rasterization pipeline can be used in conjunction with a primitive type, which is either triangles, lines, or points. There are two variations of the rasterization pipeline, which are described below.

Vertex Shading Pipeline

The *Vertex Shading Pipeline* is the original rasterization pipeline. It has been in use since before the first GPU was created. In fact, the old GPUs were made to hardware accelerate the *Vertex Shading Pipeline*. A diagram of the pipeline can be seen in figure 2.5. The diagram shows different stages, which vary between being fixed function, programmable and optional. For a given collection of vertices that is processed with this pipeline, it is first specified by the programmer how the mesh data is made available to the pipeline. This can be done in a couple of different ways. For triangle meshes, it can be done with a vertex and index buffer. Vertices are then processed one at the time. The *Vertex Shader Stage* computes the position of the vertices as well as propagates any desired vertex information further down the pipeline. After processing the vertices they can be sent through two optional stages in the pipeline. Because these stages are not generally used, nor supported, on all GPUs they are optional. The first is the tessellation stage. The stage is used to divide a primitive into smaller primitives. This can be helpful for automatically increasing or decreasing the complexity of objects depending on their distance to the virtual camera. The other optional stage is the geometry shader. The geometry shader takes a primitive as an input, and outputs between zero or more primitives. This can for instance be used for turning vertices into a quad of two triangles for particle simulation. In the *Post-Vertex Processing stage*, clipping is performed. Here it is determined whether a primitive is visible. The vertices of visible primitives are passed on, while partially visible primitives are clipped before being passed on.

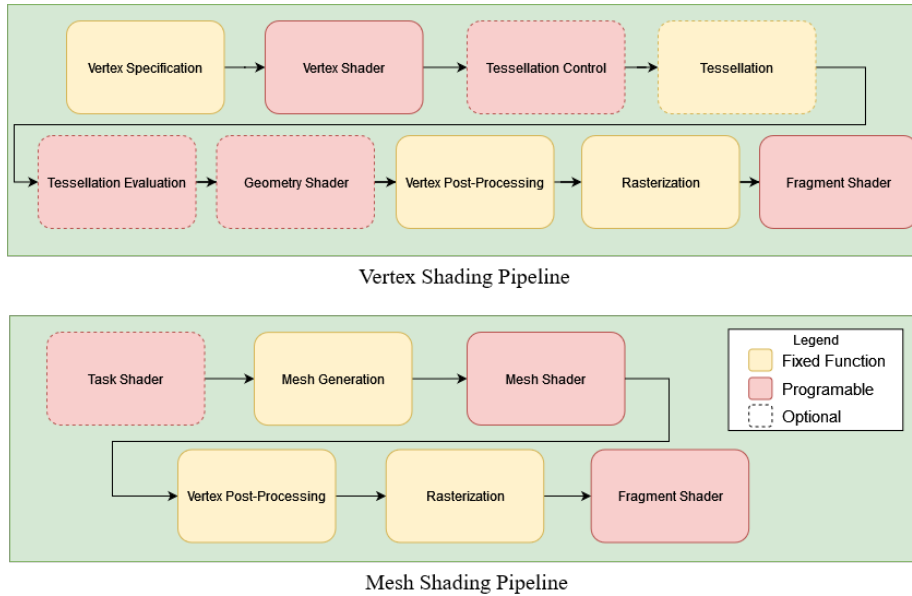


Figure 2.5: A diagram showing the Vertex(top) and Mesh(bottom) shading pipelines

Clipping is done by creating new vertices as the intersection between the visible part of the scene and the primitive. These new vertices then replace the vertices that are not visible from the camera. After all the vertices that are not visible are filtered away, the pipeline starts rasterization. Rasterization is divided into two steps. First, we have the primitive assembly, where the primitives are formed from the vertices. Afterward, for each pixel, the primitives are checked, and if they overlap a fragment, is generated. This all happens in fixed function hardware before the *Fragment Shader Stage*, where the pixel is shaded by a fragment shader.

Mesh Shading Pipeline

The *Mesh Shading Pipeline* was first introduced in 2018 with NVIDIA's Turing GPU architecture. The pipeline can be seen in the bottom diagram in figure 2.5. *Mesh Shading Pipeline* differs from the *Vertex Shading Pipeline* in that it does away with the vertex processing stages, and instead processes *Meshlets*. *Meshlets* are small clusters of primitives. It does this by introducing two new shader stages, namely the *Task* and *Mesh Shader Stages*. These two shader stages are more dynamic than any of the shader stages that are associated with the *Vertex Shading Pipeline*. Meaning that instead of relying on fixed function hardware for processing the vertex specification, the programmer is free to define the mesh input to the *Mesh Shader Stage*. The *Mesh Shader*

Stage expects processed vertices, as well as an index buffer for the *Meshlet* as output that can be passed on to the *Fragment Shader Stage*. The pipeline is restricted to the same primitive types as the *Vertex Shading Pipeline*. Another welcome addition to this pipeline is the *Task Shader*. The *Task Shader* and *Work Generation Stage* are optional stages that can be used to cull entire *Meshlets* and make new work groups based on the *Meshlets* that are passed down the pipeline for processing. Meaning that it is no longer necessary to process an entire mesh if only part of it is visible. The *Mesh Shader Stage* handles all the processing of the vertices and the propagation of the vertex data. It can also be used for tessellation and other manipulation of the *Meshlet*, such as level of detail decisions, and even triangle-based culling.

The biggest difference between these two pipelines is the granularity at which the shader stages work. For the *Vertex Pipeline*, the *Vertex Shader Stage* acts upon a single vertex, whereas the *Mesh Shader Stage* acts on a cluster of vertices. In other words, the vertex shader program is executed on a single thread on a GPU processor, while the task and mesh shader stage is executed on the entire processor with explicit control over the threads within that processor. It is this paradigm shift that makes the mesh shading pipeline much more powerful. It allows the programmer much more freedom in how the processing is done.

2.8.2 A Brief History of Real-Time Graphics

Very often, bespoke hardware has failed because of Moore's law which, for a period of time, almost ensured that any special purpose hardware solution would be overtaken by general purpose hardware (Leiserson et al., 2020). However, graphics hardware has been a tremendous success, largely because of the massive parallelism offered by the task of triangle rasterization is simply not a good match for a traditional CPU, and later, because the graphics hardware itself turned into a general purpose solution, now widely used for other tasks than graphics.

There is no doubt that computer games were the main driver of the development of graphics hardware, and the release of Doom by Id Software in 1993 is an important milestone. Ironically, while Doom did not exploit hardware acceleration, its popularity persuaded many consumers to upgrade their computers, making them ready for the first hardware accelerated 3D graphics PCI expansion card. This card, which was released toward the end of 1996, was the Voodoo Graphics PCI card created by 3dfx Interactive. The affordable price of the Voodoo card and the popularity of Doom fueled the advent of dedicated graphics processing add-on cards (McClanahan, 2010). The Voodoo only accelerated rasterization, meaning that the programs still did the vertex transformations on the CPU, and then the graphics card rasterized the triangles into pixels (Haines, 2006).

While the Voodoo II improved on the performance of its predecessor, the next major innovation came from NVIDIA which launched the Riva TNT in 1998. This GPU provided more flexibility during the fragment shading process, allowing for more inputs that in turn meant better-looking images (Zeller et al., 2004). Only a year later, NVIDIA released the GeForce 256, and ATI released the Radeon 7500. These two GPUs introduced the hardware accelerated vertex processing pipeline and provided more freedom in the fragment processing stage (McClanahan, 2010). Full programmability came in 2001 when NVIDIA and ATI released the NVIDIA GeForce 3 and the Radeon 8500, respectively. At this juncture, GPUs offered programmability in both the vertex and the fragment (pixel) stage but using distinct processing units on the GPU (Zeller et al., 2004).

The unification of the processing of vertices and fragments came a few years later in 2006 with the release of the Geforce 8000 series and Radeon HD 2000s. This meant that now the vertex shader and fragment shader both used the same streaming multiprocessors on the chip (Lindholm et al., 2008). The unification of the vertex and fragment shaders increased the GFLOPS of GPUs because the entire GPU hardware could be used at all times. This increase, along with the flexibility of more general-purpose streaming multiprocessors, meant that the computing power of the GPU surpassed that of the CPU for tasks that could exploit the massive parallelism. For this reason, the GPU started to be used for other types of computation besides generating images of 3D geometry (Wu and Y. Liu, 2008). To harness the GPU for compute tasks, NVIDIA also released a new programming API called CUDA (Compute Unified Device Architecture), and a couple of years later OpenCL was released. These two new APIs relieved specialists in scientific computing from the arduous task of using a graphics API such as OpenGL for their numerical tasks (Fang, Varbanescu and Sips, 2011).

In 2018 another major shift occurred with the NVIDIA RTX series and the ATI Radeon RX 5000 series. A large demand for the acceleration of convolutional neural networks for deep learning made NVIDIA and ATI create support for it through hardware accelerated tensor operations. Moreover, hardware acceleration for RT was introduced in this generation (Sanzharov et al., 2019) marking the first time a different mode of rendering than rasterization was directly supported by graphics hardware.

2.8.3 Game Engines and Visualization Tools

Early games were largely written from scratch with very little content and few roles in the development process other than that of the programmer. However, since games generally require many similar facilities, software layers emerged between the bespoke code for the individual game and the APIs for graphics, sound, etc. These layers are generally called game engines (Mishra and Shrawankar, 2016). Perhaps unsurprisingly, the first 3D game engine that was made publicly available is associated with the game

Doom mentioned above. The game was based on the engine later labelled as id Tech 1 and written in large part by John Carmack. It has since been made publicly available under the GPL license (<https://github.com/id-Software/DOOM>).

Apart from the id Tech engines, a number of commercial engines have been very influential. Best known are probably the Unreal engines from Epic Games, e.g. UE4 which is frequently used for developing VR applications. However, in this thesis, we will focus on another popular engine, namely Unity3D (<https://unity3d.com>) which was one of the first engines that ran well within a web browser as a plugin. Later, Unity became the go-to tool for developing games on a variety of platforms, recently including VR games.

Valve is another important contender. They run the biggest online game store, called Steam, maintain the Source Engine, and have created their own HMD. Valve has also developed OpenVR and SteamVR which is the current standard used for communicating with different HMDs (Ripton and Prasuethsut, 2015). OpenXR is a Khronos Group alternative to OpenVR which promises a unification of both AR and VR into one API. Most of the available game development has also added VR support through the integration of OpenVR. This support is in the shape of plugins that can be used on top of the existing engines.

Several different tools, besides game engines, can be used for creating and viewing geometric datasets. Digital content creation tools such as *Blender* (<https://www.blender.org>), many of Autodesk's software packages (<https://www.autodesk.dk/>), and *KeyShot* (<https://www.keyshot.com>) allow users to create view and edit geometric datasets. These are more used in the visual effects and film industry to create computationally heavy high-quality images offline. In the scientific community a lot of researchers work with *ParaView* (<https://www.paraview.org>) or *MeshLab* (<https://www.meshlab.net/>), two open source visualization platform that supports many different data formats. ParaView also has a plugin that enables VR-based data visualization.

2.9 Scientific Visualization of Geometric Data

Visualization of large and complex datasets are essential for scientists and engineers to understand, analysis and communicate. Kehrer and Helwig Hauser (2013) made a survey on visual analysis of multifaceted scientific data. Multifaceted scientific data considers everything from spatiotemporal to multivariate to multirun data. This survey proposes six categories for visualization of scientific data, based on a literature review of 200 papers. The six categories reflect different visualization, analysis, and interaction methods that have been used together with multifaceted scientific data. The six categories are:

Visual Data Fusion which fuses different facets of the data into a single visualization with a common frame of reference. *Relation and Comparison* which is in essence comparative analysis, where similarities and differences of the data are investigated. *Navigation*, which pertains to exploring the data through moving around and within it. *Focus+Context and Overview+Detail*, This category can be divided into two, *Focus+Context*, which is defined by leveraging computer processing to visualize a region of interest in more detail with the surrounding context being represent in a more simplistic manner, whereas *Overview+Detail* is the separation of details and overview into different views. *Interactive Feature Specification* enable the user to interactively select regions of interest and interact with the data. *Data Abstraction and Aggregation* which is a collection of regions of interests, or annotations based on algorithmic analysis of the data.

To gain a better understanding of how these categories can be applied to geometric data we go through them one at a time. This is by no means an exhaustive list of every scientific visualization paper of geometric data but rather a dive into the techniques that can be used in conjunction with geometric data and how they fit into the categories. Hopefully this will provide the reader with an idea of how visualization tools for geometric data utilizes aspects from each category.

Visual Data Fusion

When working with explicit surface data, visual data fusion can be done through the use of texture mapping and surface colouring. Color maps can be used to encode different scalar values that can be applied onto the surface, and isoluminant maps can be used to make the 3D structure of the surface easier to perceive (Moreland, 2009). Rocha et al. (2017) introduce *decal-maps*. Decal-maps consists of a 2D glyph, pattern, symbol or text that can be transferred to a surface upon contact. This can be used to map multivariate data onto arbitrary surfaces, such as rock type, porosity, direction and magnitude for oil and water flow into a single texture that can be applied to a petroleum reservoir model. Tang et al. (2006) use textures to map weather data onto a surface model of China, where the texture encodes temperature, wind speed, precipitation, and pressure.

Textures can also be used in conjunction with geometric objects for *Flow Visualization* (FlowVis) which covers a wide spectrum of industries, such as visualizing aerodynamics, climate and weather simulation, medical visualization and water flow. Typically the data takes on the shape of a multivariate vector field, and the goal is to visualize this. One way of visualizing it is through Line Integral Convolution(LIC). LIC creates a texture-based visualization of a vector field by performing a convolution of the vector field on a white noise image at each pixel, blurring it in the direction of the vector field's integral curves (Cabral and Leedom, 1993). This can be extended from 2D to an arbi-

rary surface, through a combination of 3D noise textures and the idea of projecting a piece of the resulting texture onto each triangle, taking advantage of the triangles being planar (Stalling and Hege, 1997; Battke, Stalling and Hege, 1997). Toledo and Celes (2011) use this technique to map the 3D flow of oil onto the surface of a black-oil reservoir, where they also add colors to the texture depending magnitude and normal vector of the surface model. Flow can also be visualized as a streamline texture on geometric objects (R. S. Laramee et al., 2004), while image based flow visualization techniques can be used to advect a triangle surface along an unsteady vector field (Wijk, 2002; R. Laramee, Jobard and H. Hauser, 2003).

Relation and Comparison

The general idea of relation and comparison is very similar to that of comparative analysis. I.e. that data from two different sources are visualized with the intend to either show similarities or differences (Pagendarm and Post, 1995). Gleicher et al. (2011) present a taxonomy of visual design for comparison based on the general issues related to the Information Visualization(InfoVis) community, across different domains. It divides comparative design into 3 categories. *Juxtaposition* where two objects are presented separately. *Superposition* or superimposition, where two objects are overlaid in the same space. *Explicit Encoding* where some visual encoding is used to compute and visualize the relationship between the objects. In the taxonomy by Gleicher et al. (2011), 111 visualization systems from the InfoVis community were surveyed. Out of these only three supported 3D geometric data, while two of these three system performed a dimensional reduction on the data before visualization. This is maybe not so surprising as abstract data is quite common in InfoVis. The three categories however can also be used when analysing geometric data, as is more commonly seen in scientific visualization. Of the 200 visualizations that they survey only 25 fall into the category of comparative analysis. Out of the 25, eight use some spatial 3D and time-varying 3D data and only two of these are related to geometric 3D data. Weigle and Taylor (2005) conduct two experiments to explore how an intersection of two meshes are best visualized. The visualization is rendered offline. Busking et al. (2011) present an image-based extension to Weigle and Taylor's intersection visualization, which makes it applicable for dynamic comparisons. Both of these methods fall into the superposition category. Kim, Carlis and D. F. Keefe (2017) have made a survey on visual analysis on spatial 3D and 4D data. They survey visualization tools that support spatial 3D and 4D data, and explore whether or not those tools support comparative analysis through *Juxtaposition*, *Superimposition*, *Interchangeable*, and *Explicit Encoding*. The survey consists of 41 tools that include comparative visualization designs. Out of these 21 use spatial 3D data. Out of those 21, eight use geometric data. Six of these focus on visual encoding. Three of these are on nested-surface visualization. Two of these are the same that are present in (Kehrer and Helwig Hauser, 2013), and the last explores the use of glyphs in combination with transparency (Interrante, Fuchs and Pizer, 1997).

Two of these present tools that are used for merging different version of the same 3D models (Doboš and Steed, 2012), and highlighting the differences resulting from different surface reconstruction algorithms of point clouds (Schmidt et al., 2014), effectively using texture to highlight differences. The last paper presents a method called Ensemble Surface Slicing which is used to slice different objects and present them as one image that it stitches together from slices pertaining to different objects (Alabi et al., 2012). Another paper presents the user with an interactive interface that allows the user to edit a mesh resulting from Finite Element Modeling, either by directly manipulating the geometry or tuning the parameters used to simulate the mesh (Coffey et al., 2013). The last paper presents a tool for validating a machine learning based triangle segmentation against the ground truth for medical image segmentation (Landesberger, Basgier and Becker, 2016).

Navigation

The desktop computer which is used to visualize and inspect the 3D data always comes with a keyboard and mouse. The user interface, keyboard, and mouse are the result of over 30 years of combined efforts from computer scientists in universities, government laboratories and corporate research groups working to create the perfect 2D interface for the desktop computer (Perry and Voelcker, 1989). Because the mouse, specialized in 2D interfaces, only has two DoF (Three with the mouse wheel), it becomes challenging to properly navigate in a 3D environment that requires six DoF.

The simplest six DoF interface is to present the user with a 2D interface where they can manipulate six sliders with the mouse (Zhai, 1998). This does however not work out very well because people cannot mentally decompose orientation into separate rotational axes (Parsons, 1995). Instead M. Chen, Mountford and Sellen (1988) laid the foundation for the interaction design that has become the standard, through the adoption in many *Digital Content Creation* (DCC), *Computer-Aided Design* (CAD), and game making software packages, by introducing a virtual sphere around the object of interest. The mouse can then be used to rotate the sphere by clicking and dragging. Dragging vertically rotates around the Y-axis, while dragging horizontally, rotates around the X-axis. By dragging along, or outside the edge of the circle it rotates around the Z-axis. The rotation can be constrained to different axes via a keyboard press. Shoemaker (1992) introduces the Archball. A similar approach that instead of allowing the user to click outside the sphere, maps the mouse movement to the ball, rotating the object based on the arch drawn by the mouse. While a keyboard press for constrained axis rotation is also required, it produces a visual cue on the sphere. Namely three mutually perpendicular half circles on the sphere that, when clicked, constrain rotation. But even with the Archball and Virtual Sphere, the mouse is still inferior when compared with a six DoF interaction device when rotating 3D objects (Hinckley et al., 1997). This covers finding the desired orientation, but the user can move around. This is typically

done through direct control of the camera. Two different approaches prevail in scientific visualization. *Eyeball in Hand*, where the user directly manipulates the camera as if it was in the hand of the user, and *World in Hand*, where the camera is held still and the data is manipulated (Christie, Olivier and Normand, 2008).

When working with geometric data in 3D the user is often given control of a virtual camera that can be used to change the direction from which the data is viewed. Since the camera is virtual, it can freely fly around the data. This can easily be extended to several cameras, which allows the user to view the data from different angles at the same time. The camera is controlled with a combination of inputs from the mouse and keyboard. Good camera controls are important because geometric data more often than not are solid, and thus require some maneuvering to get around occluding parts. The movement required to effectively inspect datasets that have both low and high frequency details needs to include the ability to change the velocity at which the camera flies. High velocities are required to quickly cover big distances when inspecting the data from afar, but at the same time low velocity is required when inspecting the data up close. Mackinlay, Card and Robertson (1990) implement rapid controlled movement by having the user pick a target which is used to scale the velocity. Ware and Fleet (1997) use *Depth Modulated Flying* (DMF) to adjust velocity. DMF changes the velocity depending on the distance to the nearest point on screen.

Focus+Context and Overview+Detail

In Section 2.9 we explored the superimposition of different models, but sometimes the geometric models themselves are made up of many distinct parts occluding each other. Such models can for instance be found in manufacturing, engineering and medicine. To visualize these models and explore the distinct parts, inspiration can be taken from technical drawings that often provide a curated view containing important details, such illustration use transparency and phantom lines to give context, while focusing on the inner parts (Diepstraten, Weiskopf and Ertl, 2002). A silhouette can also be used to outline the transparent objects for context (Gooch et al., 1998). Diepstraten, Weiskopf and Ertl (2003) show a texture-based approach mimicking cutout and breakaway illustration. W. Li, Ritter et al. (2007) use *Constructive Solid Geometry* (CSG) to make cutting volumes for creating interactive cutaway illustrations. Another approach is the *exploded view* diagrams, where the distinct parts of the model are moved outside of the shell to show where they fit and the shell is kept for context (W. Li, Agrawala et al., 2008). Yang, J. Chen and Beheshti (2005) use a camera model that creates a nonlinear magnifying glass like effect on the screen to enhance part of the image. Depth of field can be used to blur out parts of the resulting image, while keeping the region of interest in focus (Miksch and Helwig Hauser, 2001). *Level of Detail* (LoD) strategies can be used to give focus and computing power to regions of interest on a geometric level. Many different ways of including an LoD version of geometric data exist, from

real-time adaptive sampling to precomputed versions of the data that are exchanged depending on the distance to the camera (David Luebke et al., 2003).

When working with *Overview and Detail*, the idea is to separate information into two distinct views. With 3D geometric data, we can think of this as having multiple virtual cameras with their own associated viewports. DCC software always has more than one virtual camera. In fact the user is often presented with four virtual cameras in a grid and the ability to create new virtual cameras. The cameras can freely be manipulated independently of each other, providing a good Overview+Detail of the models being visualized. The use of orthographic projection further helps give an overview, and simplifies the interaction with a two DoF mouse (Mendes et al., 2019). Another example of this is the YMCA application by (Schmidt et al., 2014), which can be used to compare different mesh reconstruction algorithms. Here the user is presented with small views that show details of the regions of interest with the most divergence between the algorithms next to a large view of the entire model.

Interactive Feature Specification

Working with geometric data on a desktop computer typically means using a 2DoF mouse for navigation, as previously described, and also for interaction. Selection with a mouse is typically done by ray-casting from the pixel position that the user clicks and into the 3D scene (Bleisch and Nebiker, 2008). This allows the user to select an object, a triangle or even a vertex. D. Keefe et al. (2009) use this to create tracers on a geometric model of a skull used for exploring the biomechanics of chewing in pigs. Creating a trace on the skull is linked to a 2D view that shows how that point moves throughout the chewing movement. Extending this to multi-selection can be done by allowing the user to draw a rectangle, or free-form lasso on the display (North et al., 2009) which can then be projected out into the 3D scene. These projection-based selection tools are not always the best, because they also select occluded geometry, so when selecting vasculature based geometry with a lot of overlap, instead the user can click on a vessel, and the entire sub-tree is selected (Preim and Oeltze, 2008). A volumetric brush, in the shape of a box or ellipsoid can also be used for selecting the intersection of the data and the brush (Weyrich et al., 2004).

Other input devices can also be used for interaction with geometric data, such as gesture based selection. Bacim, Nabiyouni and D. A. Bowman (2014) use free-hand gesture tracking to *slice-n-swipe* a selection. The method works by progressively refining the selection, by slicing the data, and swiping away the unwanted part until only the desired region of interest is left. Ulinski et al. (2007) use a bimodal interaction interface, where the user manipulates a box volume and fits it over the region of interest.

Data Abstraction and Aggregation

Manually selecting a large number of vertices or triangles to mark a region of interest quickly becomes a tedious task on large datasets and models. Instead automatic methods can be used. Segmentation of a geometric model is one way of dividing the model into several regions by breaking a model down into smaller element or segments. M. Attene et al. (2006) evaluate several algorithms that can be used for mesh segmentation. In cultural heritage segmentation can be helpful for comparing different archaeological finds which may help relate them to each other and history (Manferdini and Remondino, 2010). Segmentation also makes it possible to make more precise queries when searching through 3D shape repositories (Marco Attene et al., 2007).

Geometric Deep Learning is a field focusing on using deep learning for shape segmentation and annotation. Deep learning methods can be used to automatically find regions of interest in new datasets, once trained on previously annotated data. Cao et al. (2020) have made a comprehensive survey on geometric deep learning. The input to these models can be in the form of 2D images of the desired features (Liebelt and Schmid, 2010), 3D annotation points on the surface (Paulsen et al., 2019), or entire point clouds of vertices that make up a region or object of interest (Koo, Jung and Y. Yu, 2021).

Yi et al. (2016) use active learning to annotate regions in large 3D shape databases, by manually annotating a small sample of the database they train a neural network through human supervision to extend the work to the remaining network. *Active Learning*, or *Interactive Machine Learning* is making deep learning a viable method for small datasets such as is often found in the biomedical field (Holzinger, 2016; Yimam et al., 2015). One benefit of this is that it allows deep learning to be used on increasingly smaller datasets. Another is that it takes the variance out of the annotation and segmentation process that could otherwise arise if several human annotators carry out the work instead.

2.10 Scientific Visualization in Virtual Reality

In 1999 Brooks assessed that while VR almost worked in 1994, it now barely works, and is used more throughout different production applications (F. Brooks, 1999). He surveyed different applications of VR from training astronauts and pilots and psychiatric treatment, to design reviews of automobiles and electric boats. In 2000 Dam et al. (2000) made a paper detailing some of the unresolved technical challenges that keep VR from becoming more used in scientific visualization. In order for VR to become more integrated in scientific visualization the display resolution needs to increase, the interaction needs to be more fast and effective, and the graphics processing capabilities

need to increase. In essence the *Motion-to-Photon* latency needs to be dramatically decreased.

These challenges are being overcome today, by consumer grade VR headsets. With large companies pushing low latency, high resolution consumer grade HMDs and graphics processing hardware seeing big performance increases. With this democratization of VR systems and graphics hardware the use of VR for scientific visualization has once again started to gain traction. In this section we go through some scientific visualization tools that have been built for VR. We aim to provide the reader with an overview of different tools and applications. This is not meant to be an exhaustive literature review of all applications, but to rather give an idea of what is possible when working with scientific visualization in VR.

Data from many different data sources is being explored in VR, and while 3D data seems like the ideal candidate to be treated with VR, higher dimensional abstract data can also be explored in VR. Donalek et al. (2014) have built a VR-based visualization tool called *iViz* to help support the discovery process in exploration of big datasets. The Tool encodes high-dimensions data into a 3D coordinate system, by utilizing opacity, shape and color to encode dimensions beyond the XYZ-axis. This approach can help users explore and look for groupings of data that might not otherwise be apparent. *SphereViz* is another VR-based visualization tool for searching through multi-dimensional image data sets. The user is immersed in a sphere where the images are embedded into the interior of the sphere (Soldati, Doulis and Csillaghy, 2007). Moran et al. (2015) built a VR-based visualization tool that juxtaposes geo-tagged Twitter posts onto a 3D model of MIT's campus. The user can explore these Twitter posts and their connections to other Twitter posts.

Cross-sectional imaging techniques are widely used in medical and biomedical imaging, as well as neuroscience. These techniques are used to capture internal organs, bones and vasculature information. This results in a model with many nested surfaces. Explicit surface representations of nested surfaces are not ideal because of occlusion. Because of this and the difficulties of handling many semi-transparent surfaces in rasterization volume rendering is frequently used with this type of data. So it is not surprising the VR visualization platforms that work with medical image data are based on different volume representations. *DIVA* is a visualization platform for microscopy data (El Beheiry et al., 2020). *DIVA* equips the user with a controller manipulated clipping plane that can be used in real-time to remove portions of the displayed data, making it easier to extract information from the data. Like *DIVA* *ConfocalVR* is a visualization tool by Stefani, Lacy-Hulbert and Skillman (2018) which is used for visualizing image stack data from confocal microscopes. It is used for exploring and understanding the 3D structure of cell architecture. *TeraVR* is a VR-based visualization tool used to explore and annotate teravoxel-scale whole brain imaging data (Yimin Wang et al., 2019). This has successfully shown that tracing intermingled axon clusters was 50-80% faster in *TeraVR*, when compared to a non-VR approach. Usher et al.

(2018) have also built a VR-based visualization tool for tracing neurons in large-scale microscopy data, and even find that neuroanatomists become less physically and mentally fatigued when performing the tracing in VR, when compared to the none-VR state of the art.

VR has been used in design and manufacturing since before the turn of the century as surveyed by F. Brooks (1999). This is still the case today. VR allows people to find more faults when presented with the design of power units in VR, but more importantly allowing people who are not well-versed in computer aided inspection to rely on real world experience (Wolfartsberger, 2019). Marks, Estevez and Connor (2016) did a case study where a designer from a company called *Stimson Yachts* inspected his yacht design in a VE with wave sounds and a realistic ocean surface. The experience has the potential to replace a process where clients inspect fully built prototypes of yachts that often end up being scrapped as changes are made. *NOMAD VR* is a bespoke VR visualization tool which is based on a the largest database of materials science compounds called NOMAD García-Hernández and Kranzlmüller (2019). It is an elegant solution that scales well across different VR systems, from a CAVE setup to a mobile phone. They explore a couple of different datasets, and find that VR promotes an inside-out style exploration of data that leads to faster detection of the space between chemical groups in drugs, mistakes in nanoparticles simulation and the evolution of propene as it travels through porous material.

Cultural heritage is another good use case for visualization VR, where fragile artifacts and objects can be digitized and then explored in a more natural setting within a VE. It also makes it possible to relive ancient times. Kim, Jackson et al. (2015) made a visualization tool called *Bema* that together with a CAVE Setup allows the user to visit ancient Greece. More specifically the hill of Pnyx, which was used for political assemblies. The user can explore the hill at three different points in time, with as many as 14,000 Athenian citizens in attendance and explore how well they hear and see the speakers. Gonizzi Barsanti et al. (2015) scan old Egyptian artefacts to make them more widely available for inspection and analysis without risking damage to the original artefacts. Similarly Jiménez Fernández-Palacios, Morabito and Remondino (2017) built a pipeline for visualizing and exploring large and complex cultural heritage sites in VR. They add hotspots to indicate regions of interest in the form of text that help translate old text and describe the region. These can be found while exploring different case studies.

Scientific visualization in a multi user setting is also possible in VR. Either through CAVE systems where more people can be in the CAVE together. With *NOMAD VR* the researchers observed that some people need some guidance to navigate to the regions of interest, and more optimal areas of exploration. Or where each user has their own HMD. Both *ConfocalVR* and *TeraVR* allows multiple users each with their own HMD, to collaborate on annotation tasks via the internet. A test conducted in *TeraVR* had three annotators in three different cities(two different countries) co-annotate in real-

time. When compared to a single annotator it reduced the annotation time to 20% while not sacrificing much in terms of the quality of the annotation. *NOMAD VR* also found great success in experienced users guiding new users while exploring data in a CAVE setting.

Annotation in Virtual Reality

When working with scientific visualization, it is typically desired to not only inspect but also interact with the data. With the keyboard and mouse, interaction is typically comprised of a combination of direct and indirect manipulation. The user can indirectly manipulate the data through a graphical user interface, or directly manipulate it with the mouse. The mouse is held in a precision grip, meaning that the finer motor skills of the fingers can be used quite precisely to manipulate the position of the mouse on the screen. Furthermore, the user can let go of the mouse, leaving it in a specific onscreen position. Mapping the three DoF afforded by the mouse to the six DoF required in VR is not easy nor desirable. Instead, either hand-tracking or tracked controllers are used for interaction in VR. This is an entirely different modality when compared to using the mouse. Not only because the tracking itself can introduce artifacts or latency that can reduce precision, but also because the interaction with the VE is tied to the users' hands. This can introduce fatigue in the user as they might need to keep their hands and arms elevated for prolonged times because it is not possible to leave the virtual controllers at a fixed position in the virtual space and let go of the physical controllers to get a brief respite. Controllers are held in the palm and thus held in a power grip instead of a precision grip. In the power grip, most movement of the controller is performed by moving the wrist. The tripod finger position of the precision grip affords more precision than the power grip (Batmaz, Mutasim and Stuerzlinger, 2020).

The differences between mouse-based and controller-based interaction are quite big, so while inspection in VR might in many cases be superior to desktop-based inspection, it is not entirely clear if interaction can be made at the same precision in VR. When interaction is used to annotate data it is typically done with the desire to use the annotation points for further analysis of the data, and if VR-based interaction is indeed less precise than desktop-based precision, then that process will introduce unnecessary noise into the subsequent analysis. VR however allows for a much simpler way of interacting with data, because of its 6 DoF tracking. Because manipulation of the data is very straightforward it might be the case that the inspection in VR allows the user to place the annotation points more easily. As such it is not clear which interaction modality is best suited for annotation. This raises a quite vital question for interaction in VR, does visualization in VR come at the cost of precision?

In this chapter, we investigate this in the setting of geometric morphometrics, where the shape variation of biological objects is studied through the use of landmark annotation. We compare landmarks placed in state-of-the-art software with *Unity3D*-based VR software. In the pilot study, five operators place landmarks using both types of soft-

ware on a series of 3D scanned seal skulls. I took part in designing the *Unity3D*-based VR software, writing the paper, the design of the pilot study as well as being the first participant in the study. Since the study aims to analyze the precision and accuracy of the annotations, it serves well as a use case that can be generalized to other forms of interaction in VR that require precision.

3.1 Introduction

Geometric morphometrics is a powerful approach to study shape and is widely used to capture and quantify shape variation on biological objects such as skulls (F. Rohlf and Marcus, 1993; Bookstein, 1998; Dean C. Adams, F. J. Rohlf and Slice, 2004; Slice, 2005; P. Mitteroecker and Gunz, 2009). In geometric morphometrics, shapes can be described by locating points of correspondences in anatomically meaningful landmark positions that are easily identifiable (Bookstein, 1991). An acknowledged and widely used practice is to obtain landmarks directly from a physical specimen through a tactile 3D digitizer arm (Sholts et al., 2011; Waltenberger, Rebay-Salisbury and Philipp Mitteroecker, 2021). However, collecting landmarks digitally from a 3D scanned model of the physical specimen is also a viable and widely accepted alternative that has found increased use in recent times (Sholts et al., 2011; Robinson and Terhune, 2017; Bastir et al., 2019; Messer et al., 2021; Waltenberger, Rebay-Salisbury and Philipp Mitteroecker, 2021). Annotation of a 3D digital model is typically done using a software tool based on mouse, keyboard, and 2D display (Bastir et al., 2019), and hence placing landmarks can be a tedious task since the perception of shape in a 2D environment is limited as compared with the real world. When a landmark is placed, e.g., a landmark on a 3D tip, small rotations are needed to verify the position, otherwise there might be a significant distance between the actual and the desired landmark coordinates. Annotation of 3D landmarks on a 2D display is more time-consuming than when using a digitizer arm (Messer et al., 2021). On the other hand, having landmarks placed on a 3D scan of a model carries a number of advantages in terms of data sharing and repositioning of landmarks compared to stand-alone landmarks from a 3D digitizer (Waltenberger, Rebay-Salisbury and Philipp Mitteroecker, 2021). We argue that virtual reality (VR) provides a closer-to-real-world alternative to desktop annotation that retains the multiple benefits of having the landmarks on a 3D scanned model, including the ability to easily share the digital 3D model, examine it from all angles and accurately place landmarks. The user interaction afforded by the VR head-mounted display allows navigators to move the virtual camera while the controllers can move and rotate the object and serve as an annotation tool as well. All in all, the head-mounted display and controllers exhibit six degrees of freedom (DOF), which map directly to the six DOF needed to intuitively navigate in a 3D environment. This should significantly ease the annotation process and hence make 3D models more useful in biological studies that often require large sample sizes to obtain robust statistics. Indeed, the process of annotating in VR

might ultimately be as fast as or faster than desktop annotation. To investigate the use of VR for digitally annotating landmarks on animal 3D models, we present a prototype VR annotation system and study the impact of VR on annotation performance as compared with a traditional system using 2D display and user interaction by mouse and keyboard.

When comparing a desktop interface to a VR interface, some aspects of VR should be considered. Both latency and tracking noise is higher in VR than with a standard computer mouse. This can degrade performance and precision (Teather et al., 2009). Furthermore, most VR controllers are held in a power grip (clutching the fingers around the object, thereby using the strength of the wrist), as opposed to holding a mouse in a precision grip (holding an object with the fingertips, such as when using a pen, thereby enabling the finer motor skills of the fingers). This makes it more difficult to be precise when annotating objects using most VR controllers (Pham and Stuerzlinger, 2019). Using the mouse on the other hand allows for more controlled and more precise movements, all while allowing the user to let go of the mouse without losing the position on the 3D model. While the mouse only has three DOF that need to be mapped to the six DOF, it still allows for direct manipulation of either the object or the camera.

The benefits of digital 3D representation of biological specimens (such as skulls) was discovered more than two decades ago (Recheis et al., 1999). This developed into the more inclusive field of digital or virtual morphology (Weber, 2015), and the workflows in a virtual morphology lab is now a topic of considerable interest (Bastir et al., 2019). Bastir et al. (2019) discuss the various databases and the key software tools available for geometric morphometrics. One of the discussed software tools is Landmark editor (Wiley et al., 2005), which is the predecessor of Stratovan Checkpoint. It seems that none of the software tools in this area employ virtual reality.

VR allows an operator to virtually annotate landmarks in 3D models in a way that resembles real-world annotation of physical specimens (D. A. Bowman, McMahan and Ragan, 2012; Mendes et al., 2019). The directness of this interaction produces a short distance between thought and physical action, making for a simple and straightforward interaction modality. More direct interaction demands a lower cognitive load (Hutchins, Hollan and Norman, 1985). More cognitive effort can then be invested in understanding and interacting with the data that are presented. Jang et al. (2017) showed that direct manipulation in VR provides a better understanding, and that it benefited students with low spatial ability the most. Bouaoud et al. (2020) found that students gain a better understanding of craniofacial fractures by inspecting 3D models based on CT scans in VR.

In a recent study, Cai et al. (2020) found that using VR to teach about deformities in craniovertebral junctions would improve the ability of the students when afterwards placing landmarks in radiographs of craniovertebral junctions with deformities. This was an improvement as compared with students receiving teaching with physical mod-

els. We consider this an indicator that perhaps the act of placing landmarks in digital morphology could be improved too if performed in VR. We follow up on this indication and compare precision and accuracy in placing landmarks on virtual 3D representations of skulls when using our VR system and when using the traditional 2D display and mouse interface.

The possibility of *haptic* feedback is an important aspect of VR input devices. Haptic pertains to the sense of touch, and we can broadly distinguish between two types of haptic feedback. Purely tactile feedback simply means that nerves in your skin are stimulated when you touch something. Standard VR controllers support this through the expedient of vibration, and this is often called *vibrotactile* feedback. The word *kinesthetic* is used about the sense of how limbs of a person's body are positioned in space. Thus, a device which provides force feedback, thereby preventing your hand from going through a virtual surface, is often described as kinesthetic.

Within the area of placing medical landmarks, Z. Li et al. (2021) performed a comparison of the traditional 2D display and mouse interface with two variants of the VR interface, one using standard VR controllers held in a power grip, and one using kinesthetic controllers held in a precision grip. Note that this study differs from ours in another important respect: They show markers which the participants are supposed to target when annotating, whereas we consider the task of deciding where to place the point to be integral to the annotation task (i.e. there is no ground truth). The improvement in marking accuracy was found to be statistically significant when the kinesthetic input device was employed. The task completion time and difficulty of use was however higher for the kinesthetic VR device as compared with the standard vibrotactile controller. The latter was thus easier to use and had as good overall accuracy as the 2D display and mouse interface. Interestingly, the results of Z. Li et al. (2021) suggest that marking accuracy in VR is less affected by marking difficulty than when using a 2D interface. The task performance was thus more stable in VR than when using the traditional 2D tool. This is another motivation behind our test of the performance of placing landmarks in VR as compared with a traditional 2D tool. Z. Li et al. (2021) show that when mouse and VR interfaces are used in a similar way, the haptic feedback helps improving marking accuracy. They do so by having the users interact with the virtual world through an asymmetric bimanual (i.e. using both hands) interface, where one hand holds the controller or mouse which is used to both manipulate and annotate the virtual objects, while the other hand can press the spacebar to place the annotation point. Because the same hand is used both for manipulation and marker placement, their interface does not follow the theoretical framework for designing an asymmetric bimanual interface by Guiard (1987). Kabbash, Buxton and Sellen (1994) show that carefully designed asymmetric bimanual interfaces can improve task performance, while inappropriately designed interfaces lower performance. Balakrishnan and Kurtenbach (1999) show that using an asymmetric bimanual interface designed with the theoretical framework proposed by Guiard (1987) leads to a 20% performance increase over a unimanual interface. Our study employs an asymmetric bimanual inter-

face that follows the theoretical framework by Guiard (1987), and investigates whether combining this interface with regular VR controllers will lead to similar improvements in annotation performance, saving the users from having to acquire special purpose haptic devices.

In geometric morphometrics studies, the presence of measurement error can influence the results of the performed analysis by increasing the level of noise, which can obscure the biological signal, and/or by introducing bias (Fruciano, 2016). Fruciano (2016) discusses the different sources of measurement error. Several studies in geometric morphometrics quantified measurement error in a situation where landmark data were collected using different devices, and 3D capture modalities (e.g. micro CT, surface scanner, photogrammetry), involving several operators (Robinson and Terhune, 2017; Shearer et al., 2017; Fruciano et al., 2017; Giacomini et al., 2019; Messer et al., 2021). Different systems for annotation of digital 3D models were however not compared in these studies.

To investigate whether annotation in VR is a viable alternative to mouse and keyboard for digital annotation of landmarks on 3D models, we compare our VR prototype to Stratovan Checkpoint (Stratovan Corporation, Davis, CA, USA; <https://www.stratovan.com/products/checkpoint>), a commonly used software for digitally annotating landmarks on 3D models using mouse and keyboard. We note that Stratovan Checkpoint comes with many features, but we focus only on the placing of anatomical landmarks. We study the impact of VR on annotation performance. In a first step, we assess overall and landmark-wise precision and accuracy. Moreover, we investigate different sources of measurement error (between systems, between and within operators) in an overall and landmark-wise explorative analysis. Finally, we investigate differences in annotation time between systems and operators, and over time.

3.2 Materials and Methods

VR annotation system

The VR annotation system was developed at the Technical University of Denmark using Unity 2019 (Unity Technologies, San Francisco, CA, USA; <https://unity.com>) and the Oculus Rift hardware released in 2016 (Facebook Technologies, LLC, Menlo Park, CA, USA; <https://www.oculus.com>). Users in the virtual environment are presented with an asymmetric bimanual interface, which adheres to Guiard (1987)'s three high-order principles: Assuming a right-handed subject, 1) The system uses a *right-to-left reference* where motion of the right hand finds its spatial reference relative to the left hand. The user manipulates and orients the skull by grabbing it with the left

controller. The skull can be scaled by pressing buttons on the left controller. The right controller is then used to place the annotation point, this is done with a ray-gun. The ray-gun shoots out a virtual red laser-line that is intersecting with the surface of the 3D model. Landmark annotation is mimicking the gesture of shooting a gun: The user aims by pointing the controller at the landmark location, and presses the index trigger of the controller. An annotation gun is chosen since it fits well with the power grip that the VR controllers are held in. 2) The actions of the left and right hand are on *asymmetric scales of motion* where the left hand performs large scale movements adjusting the skull and the right hand performs small scale movements to set the annotation point. 3) This workflow means that the left hand moves before the right hand, adhering to the principle of *left-hand precedence*. The VR annotation system supports both right- and left-handed subjects.

3D models are rendered opaque. It is possible to move the viewpoint such that the inside of a 3D model is visible. As opposed to the desktop software Stratovan Checkpoint, the inside of a 3D model is not rendered. Only 3D models are rendered, with no additional information being shown, i.e. unlike other systems, we do not show cross sections.

A comprehensive description of the VR annotation system, and a detailed comparison of the VR system to the desktop software Stratovan Checkpoint and a 3D digitizer arm are provided in the Supplemental Article S1. A demonstration of the VR annotation system, and Stratovan Checkpoint, are shown in Supplemental Videos S1, and S2, respectively.

Landmark data collection

Our study investigates how precisely, accurately and fast a user can place landmarks using our VR annotation system compared to Stratovan Checkpoint, a mouse and keyboard based desktop system. To carry out this experiment, we scanned, reconstructed, and annotated grey seal skulls.

Sample

The sample consisted of six grey seal (*Halichoerus grypus*) skulls. Of these, five skulls were held by the Natural History Museum of Denmark and originated from the Baltic Sea population, whereas one skull originated from the western North Atlantic population and was held by the Finnish Museum of Natural History in Helsinki (Table A1). We selected the skulls based on size in order to cover a large span: In our sample, skull length ranges from about 18 to 28 centimetres. All the selected specimens were

intact, and did not have any abnormalities in size, shape or colour variation. We did not include mandibles.

Generation of 3D models

The 3D models of the specimens were generated as previously described in Messer et al. (2021). In a first step, the skulls were 3D scanned in four positions using a 3D structured light scanning setup (SeeMaLab (Eiriksson et al., 2016)). On the basis of geometric features, the point clouds from the four positions of a given skull were then globally aligned using the Open3D library (Q.-Y. Zhou, Park and Koltun, 2018), followed by non-rigid alignment as suggested by Gawrilowicz and J. A. Bærentzen (2019). The final 3D model was reconstructed on the basis of Poisson surface reconstruction (Kazhdan, Bolitho and Hoppe, 2006; Kazhdan and Hoppe, 2013) using the Adaptive Multigrid Solvers software, version 12.00, by Kazhdan (Johns Hopkins University, Baltimore, MA, USA; <https://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version12.00>).

In a last step, we used the *Decimate* function with a ratio of 0.1 in the Blender software, version 2.91.2, (Blender Institute B.V., Amsterdam, the Netherlands; <https://www.blender.org>) to downsample the final meshes of all specimens, thereby reducing the number of faces to about 1.5 million. The average edge length, which is a measure of 3D model resolution, is between 0.312 millimetres (smallest skull) and 0.499 millimetres (largest skull). Original meshes consist of about 15 million faces, and we performed downsampling to ensure that our hardware could render the meshes with a frame rate suitable for VR. By using the same resolution 3D model for both the VR and traditional desktop system, we ensured that observed differences in precision and accuracy were not due to differences in resolution. Supplemental Fig. S1 shows the six final 3D models. A comparison of the original with the downsampled 3D model is illustrated in Supplemental Fig. S2 using specimen C7.

Annotation of landmarks

For each of the six skulls, Cartesian coordinates of six fixed anatomical landmarks (Fig. 3.1; Table A2) were recorded by four operators. Each operator applied two different systems to place the landmarks on the reconstructed 3D digital models of the grey seal skulls: 1) Stratovan Checkpoint software, version 2018.08.07, (Stratovan Corporation, Davis, CA, USA; <https://www.stratovan.com/products/checkpoint>), without actively using the simultaneous view three perpendicular cross-sections, and 2) our own virtual reality tool (Supplemental Article S1). To assess within-operator error, each operator annotated the same skull six times with both systems. In total, 288

landmark configurations were collected.

Our choice of landmarks is a subset of six out of 31 previously defined anatomical landmarks on grey seal skulls (Messer et al., 2021). We chose the landmarks with indices 2, 3, 11, 18, 24, 28 to have landmarks both of Type I (three structures meet, e.g. intersections of sutures) and Type II (maxima of curvature) (Bookstein, 1991; Brombin and Salmaso, 2013). Moreover, we excluded symmetric landmarks, and landmarks not well defined on all skulls. We selected landmarks located at points that are spread over the whole skull while exhibiting different characteristics.

Experience in placing landmarks, and experience in annotating digital models are two important factors that are likely to influence operator measurement error. Thus, our chosen operators differ from each other with respect to these two relevant factors: Two operators (A and D) were biologists, both of them having experience annotating landmarks using a Microscribe® digitizer, but only operator D had experience placing landmarks on 3D models using Stratovan Checkpoint. The other two operators (B and C) had a background in virtual reality. Operator B had previously annotated landmarks on one grey seal specimen in both systems and was the developer of the virtual reality annotation tool presented in this study. Operator C was the only one having no experience in placing landmarks and was collecting the landmark data in a test run prior to other data collection. Operators A, B and D spread data collection over two to three days, whereas operator C annotated all 3D models on the same day.

All operators recorded the landmark configurations in the same pre-defined, randomized order (Supplemental Table S1), making them switch between systems and specimens. The primary goal was to prevent the operators from memorizing where they previously had placed the landmarks on a particular skull using a specific system. Operator A slightly changed the order by swapping specimen 223 and 96 using the virtual reality tool, and specimen 323 and 664 using Stratovan Checkpoint, both for the third replica. Moreover, Operator A accidentally skipped skull 323 once (virtual reality, third replica) during data collection, and thus annotated this skull three months later. In each annotation round, operators sequentially placed the landmarks in the order (2, 3, 11, 18, 24, 28). Operator C, however, collected landmark coordinates in a different order (18, 2, 3, 24, 28, 11).

In addition to collecting landmark coordinates, the annotation time was recorded for each measurement of six landmarks. In case of the virtual reality tool, annotation time was automatically recorded, whereas the operators had to manually record the time using a small stopwatch program that ran in a console window when they annotated landmarks in Stratovan Checkpoint. The operators were instructed to use as much time as they needed for a satisfactory annotation.

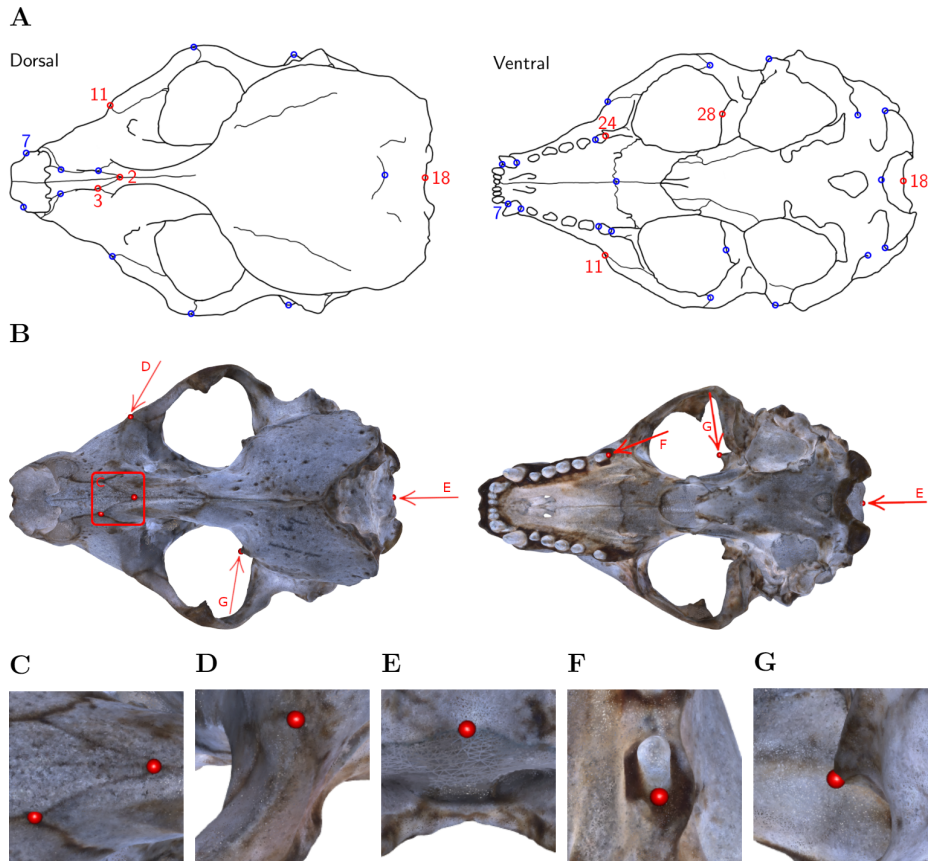


Figure 3.1: Landmark definition. (A) For our study, we selected the six landmarks marked in red based on the set of 31 anatomical landmarks on grey seal skulls as defined by Messer et al. (2021). *Note:* Adapted from Messer et al. (2021). Reprinted with permission. (B) The six landmarks placed on the 3D model of skull 96. Square/Arrows indicate the camera view for taking images (C) – (G), which show the placed landmarks on the 3D model of skull 96 and highlight their respective features. (C) Landmarks 2 and 3: Intersections of sutures; (D) Landmark 11: Apex along a margin; (E) Landmark 18: Medial point of a margin; (F) Landmark 24: Posterior/saddle point; (G) Landmark 28: 3D tip.

Statistical data analysis and outliers

All statistical analyses were conducted in the R software version 3.5.3 (R Core Team, 2020). For geometric morphometric analyses, we used the package *geomorph* (Dean C. Adams and Otárola-Castillo, 2013).

There were three different types of outliers present in the raw landmark coordinates data: 1) swapped landmarks, 2) obviously wrongly placed landmarks (4-9 mm away from all corresponding replicas; mean Euclidean distance between corresponding replicas was 0.02-0.31 millimetres)¹, and 3) landmarks localized at two distinct points, for several replicas at each point, or distributed between two distinct points. We put swapped landmarks into the correct order, and replaced the two obviously wrongly placed landmarks by an estimate based on the remaining five replicas using *geomorph*'s function `estimate.missing` (thin-plate spline approach). There were two cases of outliers of type 3): Landmark 18 annotated by operator C on specimen C7, and landmark 28 annotated by operator A on specimen 42.11, in both cases when using Stratovan Checkpoint as well as the VR annotation system (Supplemental Fig. S3). Assuming that in these two cases, the operators were in doubt where to clearly place the landmarks, we decided to include outliers of type 3) in all our analyses to not confound the results.

In our design, the repetitions were performed in a randomized order to avoid obvious sources of autocorrelations between repeated measurements on the same landmarks. This justifies ignoring the longitudinal aspects of the landmarking by modelling deviations between measurements as random errors.

For a specific specimen, annotation both in Stratovan Checkpoint and VR is based on the same 3D model, and digitization happened in the same local reference frame. Thus, the 48 landmark coordinate sets measured on the same specimen were directly comparable without first having to align them.

We further note that there is no ground truth landmark position in geometric morphometrics. For this reason, we focused on consistent landmark placement in our data analysis.

Annotation time

We sorted the recorded annotation times by system and operator in the order (Supplemental Table S1) the operators were annotating the skulls, and computed trend lines

¹The two outliers of Type 2) were replica 3 of landmark 11 on NHMD specimen 664, and replica 6 of landmark 28 on FMNH specimen C7, both placed in VR by operator C.

using a linear model including quadratic terms. This allowed us to investigate differences in annotation time between systems and operators, and over time.

Landmark-wise measurement error

We assessed landmark-wise annotation precision by computing the Euclidean distance between single landmark measurements and the corresponding landmark mean. In a first step, we visually compared the precision between systems and operators for each landmark. For that purpose, we computed the landmark means by averaging over replicas. In order to test landmark-wise whether medians across operators within one system were significantly different from each other, we used the `pairwisePercentileTest` function in the R package `rcompanion` (Mangiafico, 2021) to perform pairwise permutation tests based on 10'000 permutations. Additionally, we compared landmark-wise median overall precision between the two systems.

In a second step, to test whether the landmark-wise precision depends on the annotation system, we performed a three-way exploratory Analysis of Variance (ANOVA) with the factors *System*, *Operator* and *Specimen* separately for each landmark. Since we have a crossed data structure, we included all interaction terms. We note that our data are balanced, and that we only considered fixed effects models. Precision was computed from landmark means, which were obtained by averaging over replicas, systems and operators. Since the Euclidean distances between landmark measurements and means had right-skewed distributions, Euclidean distances were log-transformed to approximate a Gaussian distribution.

Since there is no ground truth involved in geometric morphometrics, we assessed accuracy by investigating how closely replicate landmarks were placed in VR compared to the traditional desktop system. For this purpose, we computed landmark means by averaging over replicas, followed by computing Euclidean distances between landmark means obtained from the two systems (for given operators and specimens). This allowed us to visually compare landmark-wise overall accuracy, and investigate differences in accuracy between operators for each landmark. For each landmark, we tested differences in operator median accuracy by performing pairwise permutation tests. We note, however, that the statistical power of these tests is limited due to the small sample size of six annotated specimens per operator.

Overall measurement error

We assessed overall measurement error similarly as previously described in Messer et al. (2021). In a first step, we computed Procrustes distances between devices, be-

tween operators, and within operators (i.e. between landmark replica) to investigate overall measurement error. Here, we define Procrustes distance as the sum of distances between corresponding landmarks of two aligned shapes. This allowed us to investigate the extent of differences in the total shape of the same specimen in various ways: measurement by (a) the same operator using a different system (between-system error), (b) different operators using the same system (between-operator error), and (c) the same operator using the same system (within-operator error). Since all measurements from a specific specimen were in the exact same coordinate system, we did not have to align the landmark coordinates prior to the computation of Procrustes distances. Note that we computed Procrustes distances between all possible combinations, which introduces pseudoreplicates. For each error source, we tested differences in median Procrustes distances between operators, systems, or system-and-operator by performing pairwise permutation tests. We also compared median Procrustes distance between the error sources.

In a second step, we ran a Procrustes ANOVA (Goodall, 1991; Klingenberg and McIntyre, 1998; Klingenberg, Barluenga and Meyer, 2002; Collyer, Sekora and D. C. Adams, 2015) to assess the relative amount of measurement error resulting from the different error sources *System*, *Operator*, and *Specimen* simultaneously. With this approach, Procrustes distances among specimens are used to statistically assess the model, and the sum-of-squared Procrustes distances are used as a measure of the sum of squares (Goodall, 1991). As opposed to a classical ANOVA, which is based on explained covariance matrices, Procrustes ANOVA allowed us to estimate the relative contribution of each factor to total shape variation, which is given by the R-squared value. Prior to Procrustes ANOVA, the landmark configurations had to be aligned to a common frame of reference using a generalized Procrustes analysis (GPA) (Gower, 1975; Ten Berge, 1977; Goodall, 1991), in which the configurations were scaled to unit centroid size, followed by projection to tangent space. GPA eliminates all geometric information (size, position, and orientation) that is not related to shape. Since we have a crossed data structure, we used the following crossed model for the full Procrustes ANOVA: $Coordinates \sim Specimen \times System \times Operator$.

3.3 Results

Annotation time

Fig. 3.2 shows that all operators became faster at annotating a specimen over time, especially during the initial period of data collection. We observe that the two trend lines representing operator C's annotation times are exhibiting a minimum around annotation 20 (VR) and 25 (Stratovan Checkpoint). We point out that this is not an artefact

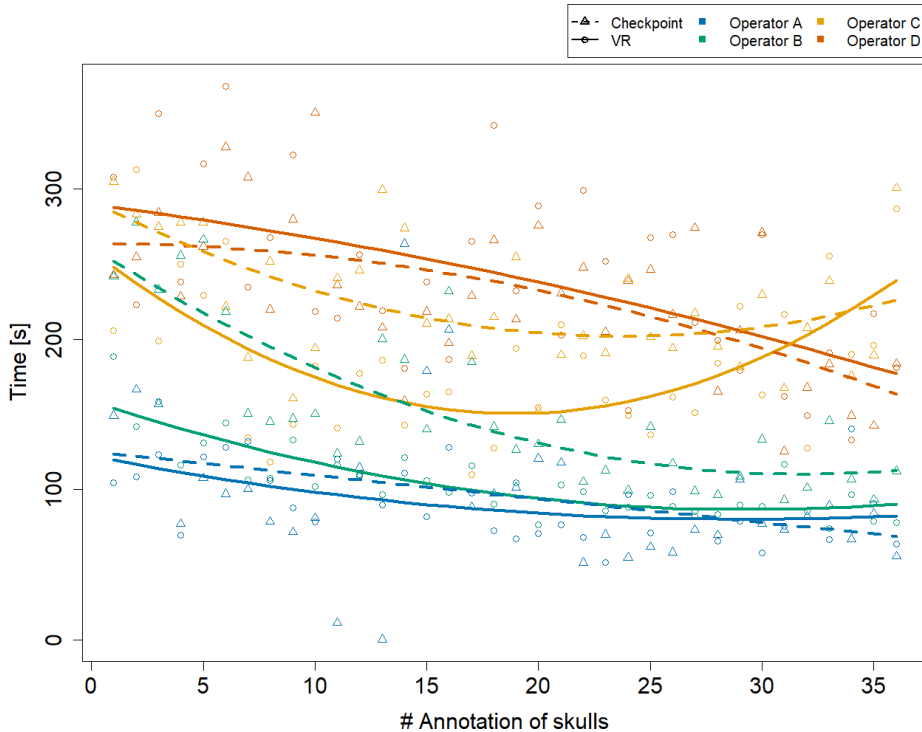


Figure 3.2: Annotation time by system and operator over time. The x-axis represents the skulls in the order the operators were annotating them. Trend lines suggesting a learning effect were estimated using a linear model including quadratic terms. Operator A's annotation time in Stratovan Checkpoint was not recorded properly for annotation 11 and 13.

due to the quadratic trend, but that operator C's annotation times were actually increasing towards the end of data collection. This might be explained by the fact that C was the only operator collecting all data on the same day. The biologists A and D seemed to be equally fast in both systems over the whole data collection period. Operators B and C, that have a background in virtual reality, started out being much faster in VR, but were approaching VR annotation times in Stratovan Checkpoint over time.

Landmark-wise measurement error

The boxplots of Euclidean distances between single landmark measurements and landmark means (Fig. 3.3) reveal that on an overall basis, a similar annotation precision was obtained for all six landmarks in VR compared to Stratovan Checkpoint. There were substantial differences between operators: Operator A, for example, was generally significantly less precise than the other operators. This might be explained by the

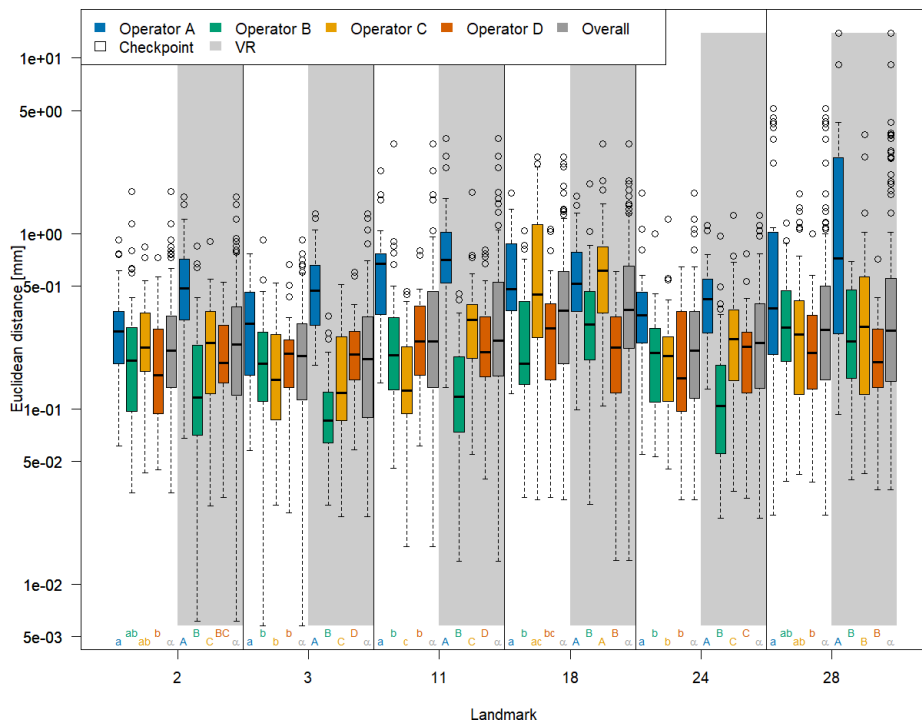


Figure 3.3: Landmark-wise precision. Precision is measured as the Euclidean distance between single landmark measurements and the operator landmark mean. For each landmark, we compared median overall precision between systems, and median precision between operators within each system for test of significance. For each of these 18 comparison rounds, groups sharing the same letter are not significantly different. The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range. Circles represent outliers. Note that the vertical axis is logarithmic.

fact that operator A, who is experienced in physical annotation, was not zooming in as much on the 3D models as the other operators during data collection. Moreover, operator B was significantly more precise in VR than other operators. Operators A and C appeared to be more precise in Stratovan Checkpoint compared to VR. Finally, operator D was much more consistent than the other operators, which is demonstrated by a similar obtained precision for all landmarks.

We obtained corresponding results in our ANOVA, which we ran separately for each landmark (Table 3.1): The factor *System* was only significant in case of landmark 11, but not for the five other landmarks. Computing the means of the Euclidean distance between measurements and mean of landmark 11 separately for each system (VR: 0.57 mm; Stratovan Checkpoint: 0.69 mm) revealed that annotation of landmark 11 was more precise in VR compared to Stratovan Checkpoint. There was no strongly significant interaction between *System* and *Operator*, nor between *System* and *Spec-*

Table 3.1: ANOVA, separately for each landmark. Dependent variable is log-transformed Euclidean distance between single landmark measurements and landmark means. We applied the following crossed structure: System \times Operator \times Specimen. Residuals reflect landmark replica, and have 240 degrees of freedom.

Variables	Df	F	Pr(>F)	F	Pr(>F)	F	Pr(>F)
		LM 2		LM 3		LM 11	
System	1	2.32	0.129	0.54	0.461	9.53	0.002
Operator	3	21.15	0.000	76.02	0.000	57.56	0.000
Specimen	5	2.36	0.041	42.54	0.000	9.95	0.000
System:Operator	3	0.89	0.445	1.34	0.261	3.65	0.013
System:Specimen	5	2.16	0.059	0.43	0.825	3.01	0.012
Operator:Specimen	15	1.73	0.047	4.20	0.000	5.87	0.000
System:Operator:Specimen	15	0.82	0.657	2.45	0.002	3.01	0.000
		LM 18		LM 24		LM 28	
System	1	1.73	0.190	0.98	0.324	0.23	0.635
Operator	3	31.91	0.000	51.97	0.000	17.27	0.000
Specimen	5	21.60	0.000	12.32	0.000	183.35	0.000
System:Operator	3	0.42	0.741	2.01	0.113	7.69	0.000
System:Specimen	5	0.91	0.476	4.24	0.001	1.84	0.105
Operator:Specimen	15	2.64	0.001	6.18	0.000	8.01	0.000
System:Operator:Specimen	15	1.59	0.077	3.42	0.000	1.56	0.086

imen for five landmarks (*System* and *Operator*: 2,3,11,18,24; *System* and *Specimen*: 2,3,11,18,28). As in Fig. 3.3, we detected major, significant differences between operators for all landmarks, which were expressed in large F-values. This was also true for interaction terms involving the factor *Operator*. Finally, we found that the variability in precision was larger between operators than between specimens, except for landmark 28. This exception can be explained by the fact that landmark 28 was subject to large outliers of type 3), measured by one operator (A) on one specimen, which were not excluded from the analysis. A similar effect is observed in Fig. 3.3. Examination of Q-Q-plots of the residuals showed that for most of the landmarks, the distribution of the residuals has heavier tails than the Gaussian distribution. However, since balanced ANOVAs are fairly robust to deviations from the Gaussian distribution, we decided not to investigate this further in this explorative study.

We note that we obtained corresponding F-values and significance levels when including outliers of type 2) in our ANOVA (Supplemental Table S2). However, annotation of landmark 11 seemed to be more precise in Stratovan Checkpoint compared to VR (VR: 0.93 mm; Stratovan Checkpoint: 0.77 mm), which can be explained by the fact that the outlier of type 2) at landmark 11 was placed in VR.

With respect to accuracy (Fig. 3.4), we found that landmarks were generally placed at similar coordinates in both VR and Stratovan Checkpoint, for all landmarks. For our sample, the Euclidean distance between system means, averaged over specimens and operators, ranged from 0.165 millimetres (LM 3) to 0.465 millimetres (LM 28), which

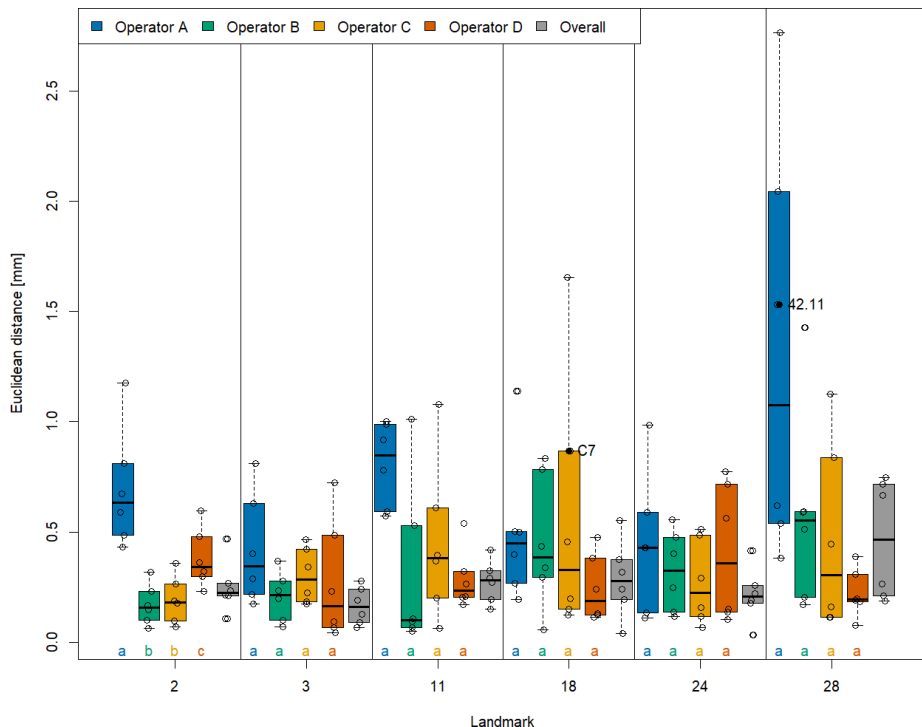


Figure 3.4: Landmark-wise accuracy. Accuracy is measured as the Euclidean distance between system means. For each landmark, median accuracies of operators sharing the same letter are not significantly different. Jittered data points correspond to the six specimens. The labelled two specimens correspond to the outliers of type 3). The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range.

is of the same magnitude as the resolution of the 3D models. Similarly to precision, accuracy varied substantially between operators: In particular for landmarks 2, 11, and 28, operator A was less accurate than the other operators. However, this was only significant in case of landmark 2 when comparing the medians (based on the limited sample size of six specimens per operator). We note that the two specimens for which we had outliers of type 3), did not show the lowest accuracies for that particular landmark.

Overall measurement error

The permutation significance test revealed that the median of the Procrustes distances within operators was not significantly different for both systems (Fig. 3.5), providing evidence that Stratovan Checkpoint and the VR annotation system exhibited similar

precision for the group of operators participating in this study. The distribution of Procrustes distances between systems is comparable to that within operators, however, the median of the Procrustes distances between systems is significantly larger than that within operators. In general, Procrustes distances between operators exhibited larger values than those between systems or within operators, with a significant difference in median, which validates the landmark-wise analysis. The median Procrustes distance between operators using the VR annotation system was significantly smaller than when using Stratovan Checkpoint. As in the landmark-wise analysis, we observed significant systematic differences between operators: For operators B and D, who were the only operators having experience in digitally placing landmarks, we found smaller measurement differences between systems than for operators A and C. A similar pattern was observed for within-operator error. Moreover, operator A was more precise in Stratovan Checkpoint, whereas operators B and D were more precise using the VR annotation system.

An analysis of the outliers revealed that they were almost exclusively measured on specimens 42.11 and C7, where we observed outliers of type 3) (Supplemental Fig. S3), and on specimen 664. The largest outliers are connected to measurements by operator A, and measurements in VR. Landmark 28 contributed substantially to the large Procrustes distances. This is line with the results on landmark-wise annotation precision (Fig. 3.3).

Running a Procrustes ANOVA on the whole dataset, again, validated the landmark-wise analysis (Table 3.2). The factor *System* did not seem to contribute much to total shape variation (0.04%), and a comparable result was obtained for the interaction terms involving the factor *System*. The main contributing factor of the two measurement error sources was *Operator*, which accounted for 1.6% of total shape variation. As in the landmark-wise ANOVAs, the interaction between *Operator* and *Specimen* is of importance and explained 1.7% of total shape variation, indicating that the operators were not experienced. As in Fig. 3.5, the results indicate that between-operator error was larger than between-system error. Most of the total shape variation (94.2%) was explained by biological variation among grey seal specimens. We note that our findings do not change when including outliers of type 2) in our Procrustes ANOVA (Supplemental Table S3).

3.4 Discussion

We developed a VR-based annotation software to investigate whether VR is a viable alternative to mouse and keyboard for digital annotation of landmarks on 3D models. For this purpose, the VR annotation system was compared to the desktop program Stratovan Checkpoint as a tool for placing landmarks on 3D models of grey seal skulls.

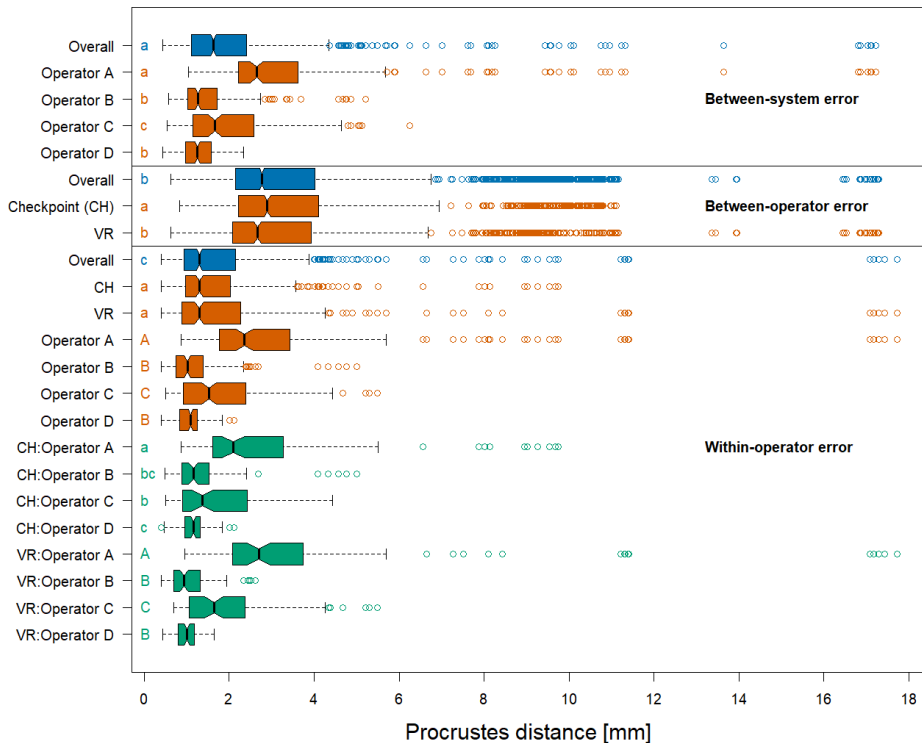


Figure 3.5: Boxplots of Procrustes distances. Computation of Procrustes distances between systems; between systems for a given operator; between operators; between operators for a given system; within operators; and within operators for a given system/operator/system-and-operator. Within an error source, we compared operator/system/system-and-operator medians with significance tests. Moreover, we compared median overall Procrustes distance between error sources. For each of these six comparison rounds, groups sharing the same letter are not significantly different. The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range. Outliers are represented by circles. The boxplot colours indicate whether a boxplot is based on all Procrustes distances for a given error source (blue), on a subset (red), or on a subset of a subset (green).

Table 3.2: Procrustes ANOVA on shape. We applied the following crossed structure: System \times Operator \times Specimen. Residuals reflect landmark replica. The R-squared values (Rsq) give estimates of the relative contribution of each factor to total shape variation.

Variables	Df	MS	Rsq	F	Pr(>F)
System	1	0.00035	0.0004	3.909	0.013
Operator	3	0.00527	0.0163	59.592	0.001
Specimen	5	0.18275	0.9421	2065.941	0.001
System:Operator	3	0.00022	0.0007	2.526	0.003
System:Specimen	5	0.00012	0.0006	1.372	0.140
Operator:Specimen	15	0.00107	0.0166	12.114	0.001
System:Operator:Specimen	15	0.00010	0.0015	1.101	0.281
Residuals	240	0.00009	0.0219		

The two systems were compared by means of overall and landmark-wise precision and accuracy, as well as annotation time. We used a carefully chosen setup, where four operators were placing six well-defined anatomical landmarks on six skulls in six trials, which allowed the investigation of multiple sources of measurement error (between systems, within and between operators, and between specimens).

On a desktop computer, an operator is forced to place landmarks through the point-of-view of their display. This is in contrast to VR, where an operator may annotate landmarks from angles different than their point-of-view, since the point-of-view is tracked using the head-mounted display and the controllers can be used to annotate landmarks from an arbitrary direction.

Another benefit of the VR system compared to Stratovan Checkpoint is that it allows the user to scale the specimen. Hence the application is agnostic to specimen size, which is especially helpful in annotating smaller specimens. In Stratovan Checkpoint, the specimen cannot be resized, but the camera can be placed closer. However, when placed too closely, the camera's near-plane will clip the specimen, thereby setting a limit on how closely one can view the specimen. In real-time rendering, two clipping planes are used to delimit the part of the scene that is drawn. The depth buffer has limited precision, and the greater the distance between these two planes, the more imprecise the depth buffer and the greater the risk of incorrectly depth sorted pixels. Unfortunately, the need to move the near plane away from the eye entails that if we move the camera very close to an object, it may be partially or entirely clipped by the near plane.

All in all, our analysis showed that annotation in VR is a promising alternative to desktop annotation. We found that landmark coordinates in VR were close to landmark coordinates in Stratovan Checkpoint. Taking mouse and keyboard annotation as the reference, this implies that landmark annotation in VR is accurate, which is in line with the findings of Z. Li et al. (2021). The accuracy achieved is of the same magnitude as the resolution of the 3D models. Furthermore, when investigating precision, both in landmark-wise ANOVAs, and a Procrustes ANOVA involving all landmarks at once, the factor *System* was not significant, in contrast to the factors *Operator* and *Specimen*. This demonstrated that the measured annotation precision in VR was comparable to mouse and keyboard annotation, whereas precision significantly differed between operators and specimens. These results are in line with previous studies on measurement error which found a larger between-operator compared to between-system or within-operator error (e.g., Shearer et al., 2017; Robinson and Terhune, 2017; Messer et al., 2021). However, we obtained a much smaller between-system error when comparing annotation in VR with desktop annotation than Messer et al. (2021), who compared physical and digital landmark placement on grey seal skulls.

Our results revealed that VR was significantly more precise than Stratovan Checkpoint for one landmark. A possible explanation is that the location of this landmark (no 11)

on an apex along a margin (Fig. 3.1D) required an operator to observe the approximate location on a skull from various angles to decide on the landmark position. This might have been easier in VR because of the direct mapping between head and camera movement. The operators might also have benefited from the ability to look at landmark 11 independently of the angle used for landmark placement, allowing the operators to view the silhouette of the apex while pointing the annotation gun at the apex, perpendicular to the camera direction.

We found a weak indication that both precision in VR, and precision in general seemed to be positively linked to an operator's experience in placing landmarks on 3D models, and not necessarily to an operator's knowledge of VR or experience in placing landmarks on physical skulls. This result highlights the importance of annotation training on 3D models prior to the digital annotation process in order to obtain a higher precision. However, we have to keep in mind that the group of operators participating in this study is not a representative sample of professional annotators. Some of the operators in this study were not experienced in annotating landmarks. This was reflected in the interaction between *Operator* and *Specimen*, which was significant for all landmark-wise ANOVAs and in the Procrustes ANOVA.

We did not find any evidence that annotation in VR is faster compared to desktop annotation, which is in line with Z. Li et al. (2021). However, we did not include difficult to place landmarks, for which Li et al. found a significantly shorter annotation time in VR than on the desktop. Even though Z. Li et al. (2021) conducted a similar experiment, there are three major differences to our setup: 1) In their study, the point the user was supposed to annotate was actually shown during data collection. This is different from the real-life annotation of landmarks we simulate, as the latter includes interpretation of the specimen's anatomy under the respective constraints and advantages of the two interfaces. 2) They used an in-house 2D annotation tool, whereas our study involves an industry standard 2D annotation software. 3) In our study, the user manipulates the model with the non-dominant hand and annotates with the dominant hand. This is similar to how one adjusts the paper with the non-dominant hand and writes on it with the dominant hand. We compare this to Stratovan Checkpoint's unimanual interface where the mouse is used both for manipulation and annotation and the keyboard is used to change the mode of the mouse.

3.5 Conclusions

To sum up, annotation in VR is a promising approach, and there is potential for further investigation. The current implementation of the VR annotation system is a basic prototype, as opposed to Stratovan Checkpoint, which is a commercial desktop software. Nevertheless, we did not find significant differences in precision between the

VR annotation system and Stratovan Checkpoint. For one landmark, annotation in VR was even superior with respect to precision compared to mouse and keyboard annotation. Accuracy of the VR annotation system, which was measured relative to Stratovan Checkpoint, was of the same magnitude as the resolution of the 3D models. In addition, our study is based on a non-representative sample of operators, and did not involve any operator with a background both in biology and VR.

3.6 Future work

The VR controllers in our study are held in a power grip and do not provide haptic feedback although they can provide vibrotactile feedback (i.e. vibrate) when the user touches a surface. Z. Li et al. (2021) employ the Geomagic Touch X which, as mentioned, is a precision grip kinesthetic device that does provide haptic feedback. Unfortunately, the haptic feedback comes at the expense of quite limited range since the pen is attached to an articulated arm. On the other hand, there are precision grip controllers which are not haptic devices and hence not attached to an arm. Thus, a future study comparing power grip, precision grip, and the combination of precision grip and haptic feedback would be feasible. Such a study might illuminate whether the precision grip or the haptic feedback is more important.

An interesting extension of the current VR system would be to add a kind of nudging to help the user make small adjustments to the placed landmarks. A further extension could be use of shape information through differential geometry to help guide annotation points toward local extrema.

Improvement of the VR system in terms of rendering performance would be beneficial as it would enable display of the 3D scan in all details (Jensen et al., 2021). This would potentially improve the user's precision when placing landmarks. Further improvements of the VR annotation system could be customizable control schemes, camera shortcuts to reduce annotation time, manipulation of the clipping plane to see hidden surfaces, rendering cross sections, and more UIs with information on the current annotation session.

In this study, we focused on clearly defined landmarks. It would be interesting to investigate (as in the work of Z. Li et al. (2021)) whether VR might be superior to desktop annotation in the case of landmarks that are more difficult to place. Furthermore, most operators had no experience with one or both types of software. It would be very interesting with a longer term study to clarify the difference between the learning curves associated with the different annotation systems: how quickly does proficiency increase and when does it plateau? Such a study might also help illuminate whether habitually wearing an head mounted display is problematic. There is some fatigue associated

with usage of a head mounted display, and this could become either exacerbated or ameliorated with daily use, something we could not address in this study.

3.7 Appendix

Table A1: List of original 3D models of grey seal (*Halichoerus grypus*) skulls used in this study and their source. NHMD: Natural History Museum of Denmark; FMNH: Finnish Museum of Natural History. Skull length was approximated by the average Euclidean distance between landmarks 7 and 18 (Fig. 3.1) based on eight repeated measurements by Messer et al. (2021).

Institution	Specimen	Skull length [cm]	Source (MorphoSource identifiers)
NHMD	42.11	19.3	https://doi.org/10.17602/M2/M357658
NHMD	96	23.8	https://doi.org/10.17602/M2/M364247
NHMD	223	21.3	https://doi.org/10.17602/M2/M364279
NHMD	323	18.2	https://doi.org/10.17602/M2/M364263
NHMD	664	21.5	https://doi.org/10.17602/M2/M364287
FMNH	C7-98	28.1	https://doi.org/10.17602/M2/M364293

Table A2: List of the six anatomical landmarks used in this study (L = left, R = right). Four landmarks are of Type I, and two of Type II.

Landmark description	Name	Type
Caudal apex of nasal	2	I
Intersection of maxillofrontal suture and nasal (L)	3	I
Anterior apex of jugal (R)	11	II
Dorsal apex of foramen magnum	18	II
Posterior point of last molar (L)	24	II
Ventral apex of orbital socket (L)	28	II

3.8 Retrospective

The experiment that was carried out in this paper was telling. Different operators with different backgrounds adopt quite distinct workflows in VR, despite being given the same instructions and introduction to the application before the start of the experiment. So it is important to consider the workflow of VR-based applications as the affordances of VR can have a big impact on how the user acts when immersed in a VE. Operator A, who was generally significantly less precise than the other operators, and had a background in physical annotation was a good example of this. Since the VE presented him with an interface that was similar enough to the physical annotation process, he fell back to that workflow forging the ability to zoom in completely. In Chapter 7 we build on this realization and discuss one way of creating an interaction metaphor that integrates the affordances of VR with a workflow that helps guide the user when interacting with geometric data.

The VR prototype developed for this paper, ran into performance issues when using the full-resolution versions of the Seal Skulls. Leading to the realization that it was probably not as simple as relying on existing tools if we wanted to keep up with big and complex datasets. Marking a shift in focus from interaction to developing a high-performance VR-based visualization platform. The following three chapters detail the exploration, development, and optimization of that platform.

CHAPTER 4

Tools for Virtual Reality Visualization of Highly Detailed Meshes

In the Annotation paper II presented in chapter 3, we worked with decimated versions of the Seal Skulls. We did this because the combination of hardware and *Unity3D* out-of-the-box did not produce high enough frame rates for VR. We required a high frame rate to avoid motion sickness, so the choice of decimating the meshes made sense. It did however lead to a general concern about visualization in VR. Decimating the meshes removed the high-frequency details. Such details could be of great importance when doing annotation, and of great interest when being visualized. It is exactly those high-frequency details that we hoped to understand better by using VR.

When building VR-based annotation applications, and real-time rendering applications in general, we realized that this could be accomplished in many different ways. We could rely on existing real-time rendering engines, such as *Unity3D* or *Unreal Engine*. We could also code up our own using one of the many different graphics APIs, such as *Vulkan*, *OpenGL*, *Metal*, or *DirectX12*. Another possibility still was to use an existing visualization tool such as *ParaView*. The ramifications of picking one over the other were not entirely clear to us, and we went with *Unity3D* because of our previous experience with it, and its accessibility to all the functionality that we might require when building our prototype. It turned out however that *Unity3D*, by default, is not set up to prioritize high performance, but instead to allow the user to easily implement everything from global illumination to physics.

There is no doubt that we are constrained by performance when developing for VR, meaning that we cannot attain the same performance that we can on desktop. As data collection and generation are getting more detailed the resulting models become bigger and more complex. So for a VR-based visualization tool to stay relevant, it needs to consider this. Whether the best performance is achieved by building on top of an existing game engine or starting from scratch with a graphics API is not entirely clear. Because of that, we set out to explore this in Paper I, which is presented in this chapter.

We aim to build a good starting point for a platform that can later be adapted to specific tasks. So in this chapter, we investigate the different starting points for building a visualization platform that has high performance at its core. The results of the paper help shed some light on which approach to building a visualization application for VR

is the most promising. I worked on every aspect of this paper, except for the *Ray Tracing* section.

4.1 Introduction

As of 2020, more than 2.5 quintillion (10^{18}) bytes of data are generated daily (Bulao, 2021). 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 (Thomas and Cook, 2006) and Virtual Reality (VR) has been explored for better dealing with and analyzing big data (Moran et al., 2015). The use of extended reality for visual analytics has led to the notion of immersive analytics (Chandler et al., 2015), where a head-mounted display (HMD) offers many exploration modes that can improve task performance (Wagner, Stuerzlinger and Nedel, 2021). However, this comes at the cost of significant rendering performance requirements (80+ frames per second) to avoid cybersickness issues (Wagner, Stuerzlinger and Nedel, 2021). 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 (Jiayan Zhao et al., 2019).

Apart from use in visualization of geospatial data (Kreylos et al., 2006; Jiayan Zhao et al., 2019), 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 a 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 (Jingbo Zhao et al., 2017)) and this carries a real risk that users become cybersick (Stauffert, Niebling and Latoschik, 2020). 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.

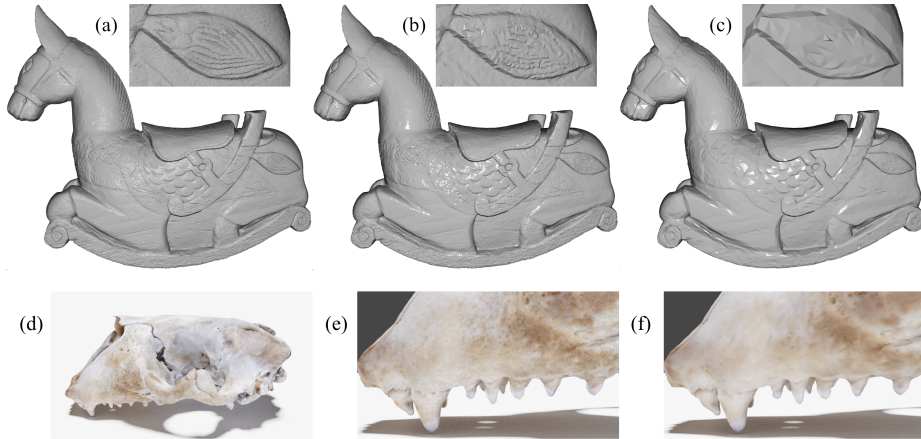


Figure 4.1: The rocking horse (a) consists of 2.2 million triangles. We reduce it to 10% of the original number of triangles (b) and further to 1% (c). While this fairly large reduction has almost no effect on the silhouette, the fine scale geometric details are clearly impacted by the reduction to 10% 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.

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 (Aage, Andreassen et al., 2017). In 3D scanning, object surfaces can be scanned with a measurement sampling density (MSD) of 10,000 points per square millimeter (Bunsch, Sitnik and Michonski, 2011), and scanning a $39.3 \times 28 \text{ cm}^2$ woodcut with a MSD at just 2500 points per square millimeter resulted in 277 GB of data (Bunsch and Sitnik, 2014). 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 (Bunsch, Sitnik and Michonski, 2011). Some examples of meshes with details at varying scales are shown in Figure 4.1. The seal skull (4.1d–4.1f) 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, 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 (4.1a–4.1c). 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

Table 4.1: 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

the most popular visualization tools; and Unity, which is a game engine and a popular tool for VR-based visualization (Donalek et al., 2014; Sicat et al., 2019; Cordeil et al., 2019). 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 4.2, the tested platforms in Section 4.3, and we present and analyze our results in Section 4.4.

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 4.1 provides some mesh complexity info for the three models and example visualizations are in Figure 4.2 (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 (Aage, Andreassen et al., 2017; Aage, Sigmund et al., 2020) 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.

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 4.5. While all the results are discussed in Section 4.6.

4.2 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 (Haines, 2006). 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 (Sander, Nehab and Barczak, 2007). 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 (Forsyth, 2006). 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 (Kerbl et al., 2018). 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* (Kubisch, 2018a; Kubisch, 2020). 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 (Burgess, 2020). It has also recently become possible to mix ray tracing and rasterization using the Vulkan API (Koch et al., 2020). 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.

4.3 VR Visualization Tools

ParaView is a tool designed for visualization and analysis of extremely large datasets (Ahrens, Geveci and Law, 2005). Paraview is built on the Visualization Toolkit (VTK), and it includes easy-to-use VR-based visualization (Martin et al., 2016, updated 2018), 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” (Cordeil et al., 2019). 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 (Unity Graphics Team, 2020).

We compare these two solutions to our own (bespoke) VR visualization application implemented in C++ using the Vulkan API (Sellers and Kessenich, 2017). 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.

4.3.1 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 4.5). This means that applications are subject to the same asynchronous time warping implementation.

Table 4.2: 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

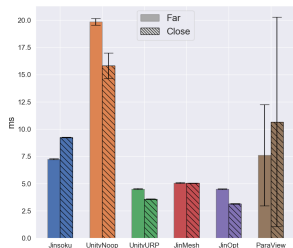
The three test meshes have an increasing number of triangles and vertices. The Seal Skull mesh and the Nobby mesh were optimized using Tootle (https://github.com/GPUOpen-Archive/amd_tootle). This program greedily reorganizes the mesh so that triangles using a given vertex are as close as possible in the list of triangles. Tootle was created for the vertex shader pipeline where locality is useful for vertex caching (Nehab, Barczak and Sander, 2006), 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 have changed the vertex order. Table 4.2 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.

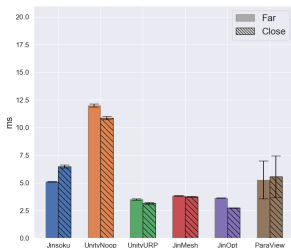
4.4 Experiment Setup and Results

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

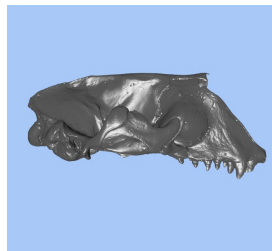
- In Jinsoku, we used Phong shading with a fixed light position. Texture mapping was not employed. Hence, each vertex carries only one attribute in addition to its position, namely the surface normal.
- In Unity (UnityURP in Figure 4.1), we also used Phong shading with a fixed light position. The Phong 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 4.1) uses a deferred rendering pipeline and includes shadows.
- ParaView uses flat shading and has no options for changing this in VR.



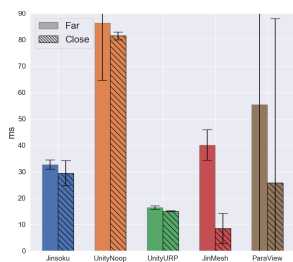
(a) Render times for Seal Skull on Quest.



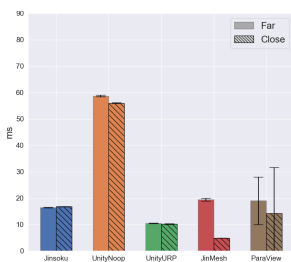
(b) Render times for Seal Skull on Index.



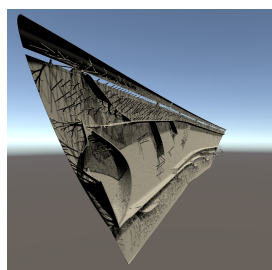
(c) Seal Skull visualized in Jinsoku.



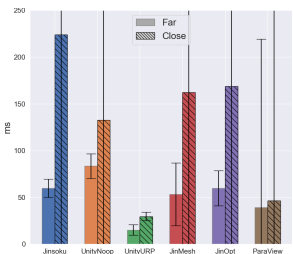
(d) Render times for Wing on Quest.



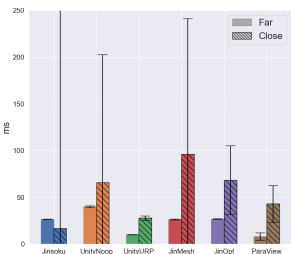
(e) Render times for Wing on Index.



(f) Wing visualized in Unity.



(g) Render times for Nobby on Quest.



(h) Render times for Nobby on Index.



(i) Nobby visualized in ParaView.

Figure 4.2: 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-colored 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.

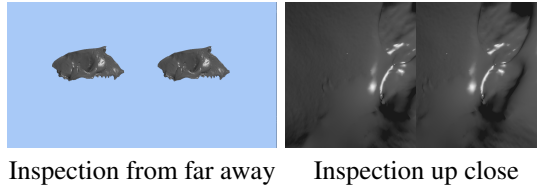


Figure 4.3: The two test conditions.

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 4.2. 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 4.3).

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 inspected 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 (Wagner, Stuerzlinger and Nedel, 2021), 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

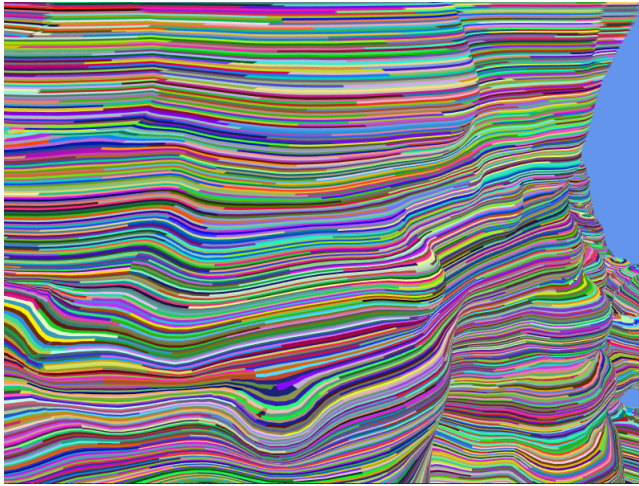


Figure 4.4: Nobby meshlets.

from afar. All other tests show average rendering times from 16–223.9 ms (\sim 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.

4.4.1 Vertex Shading vs Mesh Shading

We can compare the vertex and mesh shading pipeline by inspecting the blue and red bars in Figures 4.2a, 4.2b, 4.2d, 4.2e, 4.2g, 4.2h. 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 of 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 4.2g and 4.2h. Figure 4.4 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 4.2e and 4.2d show that the quest and index mesh shader pipeline produces the smallest average render times across headsets when inspecting the mesh up close.

4.4.2 Index Buffer order and Mesh shaders

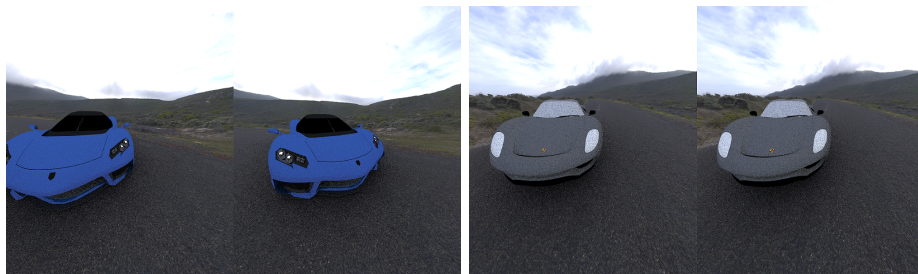
Allowing Unity to optimize the mesh is in part what resulted in the performance that can be seen in Figure 4.2. This motivated us to try and see if the mesh shading pipeline would also benefit from similar treatment. It is clear that meshlets also benefit from locality optimizing the index buffer, not only does it produce more cullable meshlets but it also decrease the total number of meshlets and the variance in the render time. This indicates that less vertices are shared across meshlets. Figures 4.2a and 4.2b 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.

4.5 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 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 (S. G. Parker et al., 2010; Wald and S. G. Parker, 2019): 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



\triangle : 300,603, t: 9.43 ms

\triangle : 15,740,813, t: 9.80 ms

Figure 4.5: VR ray tracing with one sample ambient occlusion (reason for the noise) rendered using an NVIDIA RTX 2080 graphics card. Here, \triangle is number of triangles and t is render time.

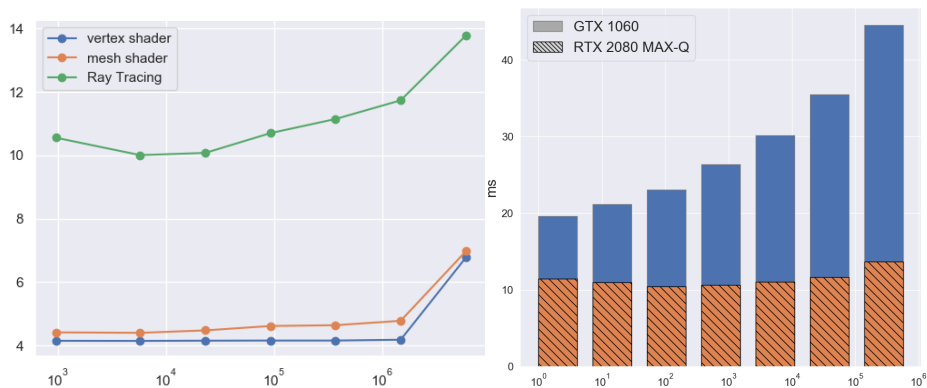


Figure 4.6: Performance of our GPU VR ray tracer when rendering the Blender monkey (Wikipedia, 2021) with increasing number of subdivisions. We compare with the two shading pipelines in Jinsoku (left) and with a GPU architecture from before RTX (right). The horizontal axes are logarithmic, meaning that the development in performance should be a straight line for logarithmic time complexity. This is not quite obtained, but RTX is getting there.

tracing in Figure 4.2. 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 4.5) and on two GPUs on laptop computers (Figure 4.6) with models of different complexity (numbers of triangles). We tested performance for GPUs with different hardware architectures. The ones called RTX have special RT cores dedicated to hardware acceleration of ray tracing (Burgess, 2020).

The RTX graphics card almost achieves the logarithmic time complexity with increas-

ing number of triangles (Figure 4.6). The difference in performance as a function of the number of triangles is very small across several orders of magnitude (Figures 4.5 and 4.6). 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 4.6). RTX cards for stationary computers are fast enough to support the frame rates needed for ray traced virtual reality (Figure 4.5). 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 (Burgess, 2020), the future will see very efficient denoising that can also exploit temporal correspondences between frames (Hasselgren et al., 2020). 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 (Koch et al., 2020) (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. Unreal 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 (E. Liu et al., 2019). 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.

4.6 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 4.2e shows this clearly. In fact, this indicates that a mesh shad-

ing 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.

4.7 Retrospective

The experiments carried out in this paper were done early in the project. They reflect the start of the project well. We started this project with no Vulkan experience and no visualization platform to build our experiments on. So we explored our options and found out that there were indeed many. We decided to compare a bespoke Vulkan engine with different existing solutions, creating a good cross-section of the options. The earliest version of Jinsoku was the culmination of a lot of reverse engineering of other Vulkan-based applications, and the performance numbers in this paper reflect that. Parallel to the publication of this paper, Jinsoku was refined with some VR-specific features and some general optimization. In the next chapter, we present these features, as well as new measurements that are more indicative of Jinsoku's actual performance.

CHAPTER 5

Jinsoku: A bespoke Visualization Platform for Virtual Reality

We started this project with the hypothesis that it would be possible to get better performance from a bespoke visualization engine compared to using a standard game engine. Simply because the game engine needs to pack more functionality to make games, than we require for building a visualization tool. More functionality adds complexity which can affect performance. We only had two requirements when starting out, namely that the underlying platform needed to support VR and rasterization, with the goal being to deliver the best performance possible. In Paper I we found that a bespoke Vulkan-based visualization platform, using the *Mesh Shading Pipeline*, held the biggest potential. This chapter details the further development and optimization of that bespoke Vulkan-based visualization platform, named Jinsoku.

5.1 Introduction

Throughout the evolution of graphics hardware, a few APIs have afforded access to the resources provided by the GPUs. OpenGL, which was originally based on Silicon Graphics' IRIS GL API, was one of the early contenders and, thanks to its far-sighted extension mechanism, managed to keep up for many years. However, with the compounding changes, it became apparent that OpenGL did not reflect the operation of the GPU hardware, and concerns about driver overhead emerged. This led to the development of modern graphics APIs, specifically Apple's Metal API, Microsoft's DirectX 12, and Vulkan. Vulkan being the appointed successor to OpenGL. Because these APIs better reflect the GPU hardware, they pose considerable challenges for the aspiring graphics developer. Vulkan, in particular, requires numerous lines of code devoted to the marshalling and management of GPU resources, making it much more verbose and explicit, while also providing less hand-holding (Bailey, 2018). In light of this, it is perhaps unsurprising that APIs which operate largely in the same way as traditional OpenGL still remain. Indeed, OpenGL itself is still widely used. As is OpenGL ES for embedded systems which is also used on the web via the WebGL Javascript bindings.

With modern graphics APIs being more verbose and challenging to work with, other

solutions have gained in popularity. Instead of working directly with graphics APIs, graphics engines and visualization tools that abstract these APIs are being favored. This has led to a decrease in knowledge, and an increase in the perceived difficulty, of working with modern graphics APIs. While it no doubt has become more challenging to build bespoke graphics and visualization applications, it is perhaps not as challenging as it is perceived to be. It is however exacerbated by the increasingly fragmented graphics space, with many different APIs and approaches to choose from. This makes it more difficult for scientists, requiring them to spend increasingly more time formulating requirements, exploring the available options, and developing the platform. In this chapter, we discuss our experiences and provide an overview of our bespoke VR-based visualization engine. Some of the features that have helped make it perform well are highlighted. Lastly, we discuss the perceived and actual difficulty of working with Vulkan.

5.2 Jinsoku

Jinsoku is a real-time visualization engine built from scratch using the 1.1 version of the Vulkan API. It is built with performance in mind, so it is fitting that it is named after the Japanese word for swiftness and speed. Vulkan has become notorious for requiring 1000 lines of code "just" to be able to rasterize a triangle. This is no exaggeration, but there is more to the story. Those 1000 lines serve as boilerplate code which not only goes into rasterizing the triangle but also makes it possible to expand on the engine more easily. Every subsequent feature that is implemented in a Vulkan engine can take advantage of the boilerplate code. In the end, the codebase for two equivalent rendering engines in OpenGL and Vulkan is roughly at parity. In a presentation from GDC, titled "Getting explicit: How hard is Vulkan Really?"¹, the codebase of an OpenGL version and Vulkan version of a game called Doom 3 is compared in terms of lines of code. The Vulkan version has around 5000 lines of code compared to the 7000 lines of code in the OpenGL version. The catch is that the Vulkan version is missing debug functionality that is present in the OpenGL version.

The benefits of Vulkan come through its explicit programming style which allows the programmer to micromanage the GPU to a greater extent than OpenGL. This means that the programmer can make more informed decisions concerning the specific application. These decisions can result in better performance. This also means there are more opportunities for making poor decisions when using Vulkan. Too many poor decisions and the potential performance increase gained by the explicit programming style will vanish.

¹<https://www.khronos.org/developers/library/2018-gdc>

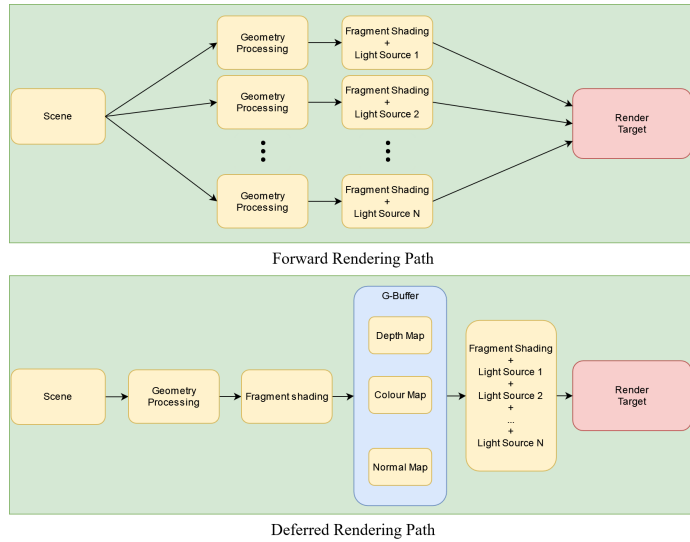


Figure 5.1: This diagram shows the difference between deferred and forward rendering.

5.2.1 Architecture

Some architectural design choices were influenced by Jinsoku being built with performance in mind. The first is that we decided to go with a *forward rendering path*. The *forward rendering path* processes each object that we wish to render, one at a time, for each light source in the scene. For most visualization tools one light source is enough, and it is possible to access more light sources than in the fragment shader. Too many light sources, however, can become a bottleneck when rendering a scene with many light sources. When a scene has many light sources it makes sense to defer the light calculations instead. This is what a *deferred rendering path* does. It goes through all the objects and creates a depth map, normal map, and color map which it stores in a G-buffer. Then it goes through each pixel and uses the information from the G-buffer and the light sources to calculate the lighting. Since Jinsoku only has one light source, we stick to the simpler and faster *forward rendering path*.

The second choice is to use simple surface illumination models. Jinsoku only uses two different surface illumination models. The first is a Phong shader. The Phong lighting model is a very commonly used surface lighting model. It approximates a diffuse material with a specular highlight, according to Lambert's cosine law and the law of reflection (Phong, 1975). The second is a *Material Capture* (MatCap) shader. Based on *The lit sphere* by Sloan et al. (2001), a method developed for non-photorealistic rendering. *The lit sphere*, allows artists to create textures that focus on conveying form

without having to take material or lighting properties into account. The created texture is used for the full appearance of a mode. These textures are in the shape of a circle, as a sphere viewed from any angle resembles a circle, which allows for a texture look-up that is based on the screen-space normal of the object, which essentially projects it into the texture. MatCaps is an extension of the lit sphere that is more focused on capturing plausible materials rather than artistic textures. With a MatCap shader, there is no need for lighting the scene. The light is already captured, or baked, into the texture. It fits very well with Jinsokus *Forward Rendering Path*. This shader also allows the user to pick the materials that they want to use themselves. Figure 5.2 shows a MatCap texture and how it looks when applied to the seal skull in Jinsoku.

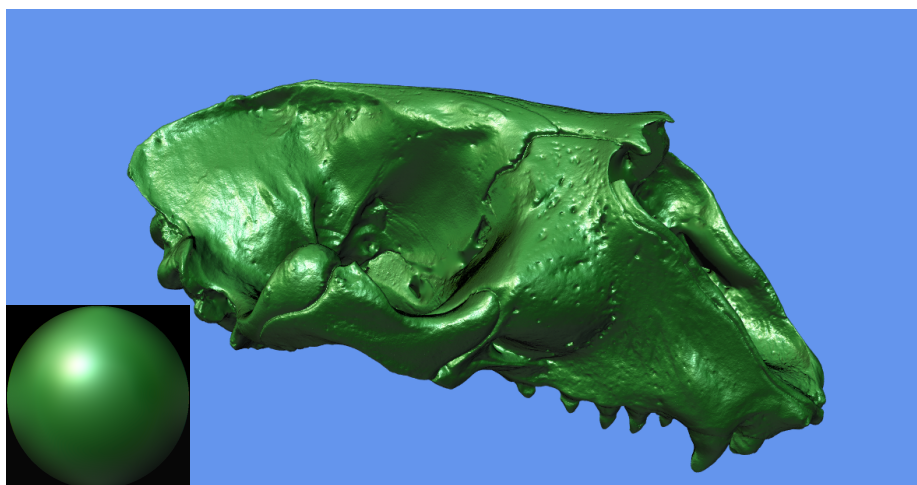


Figure 5.2: A visualization of the seal skull with a MatCap shader. The actual MatCap texture is made with(<https://cables.gl/p/pDCOCw>) and shown in the left corner.

The actual architecture of Jinsoku is presented in figure 5.3. It gives an overview of the components that make up Jinsoku, and how they fit together. The architecture is made around the Vulkan API's ability to enable extensions. The user can request extensions that the user wants Jinsoku to enable. Jinsoku then checks if these are available on the hardware platform, and enables them if they are. Examples of such features are the *Mesh Shading Pipeline*, *Multi-View Rendering*, and *Variable Rate Shading*. The two latter are described in detail in Section 5.3. If the user requests extensions that are not supported then Jinsoku falls back to a setup that is supported. Moving through the diagram in figure 5.3, we start with the Main function which executes Jinsoku. It loads a

configuration file that is passed to a Jinsoku Factory that, based on the settings in the file instantiates the desired instance of Jinsoku. Several different instances exist, *Compute pipeline only*, *Desktop only*, and an *OpenVR* version. Each instance of Jinsoku can use between zero and two presentation platforms. The Jinsoku and presentation instances are implemented with an interface-style program to assure compatibility. In C++ we implement interfaces via abstract classes. Each instance of Jinsoku also has a *Vulkan Context*. This handles all Vulkan logic. Based on the Configurations passed to Jinsoku the Vulkan Context enables the desired extensions, and sets up the correct resource strategy, renderer, and rendering context. The *Scene Logic* contains collision detection between controllers and objects as well as portals which are described more in Chapter 7. *Input Handler* handles the inputs either from the keyboard or the controllers used in VR. Jinsoku is built with a couple of external dependencies. These include code from the NVIDIA mesh shader code example (Kubisch, 2018b), Arseny Kapoulkine (2017) library, and Valve’s OpenVR API. *GLFW* and *OpenVR* is used to present images to desktop and HMDs. *GLM* and *Eigen* are used for the calculations, processing, and data representation that mirrors the datatypes used in shader programs on the GPU. *Lua* is used for scripting. *STB* and *TinyObjLoader* are used for texture and mesh loading. *MeshletMaker* is a library that we have developed. It is used to load meshes from disk, and process them into one of the many different meshlet generation strategies that we explore more in-depth in Chapter 6 and Paper III.

The development of Jinsoku has followed this project closely. It has been refactored twice. It has been used for a couple of student projects. It has evolved with this project, and it is as much a product of this project as the published papers are. Jinsoku contains 15.415 lines of code, and while that is not a particularly meaningful metric, it reflects the time that has gone into its development. Jinsoku currently has eight different resource strategies. These are all around 1200-1400 lines of code and contain duplicate code since only one of them is used at runtime. So the actual codebase is more likely around 10.000 lines of code.

5.2.2 General optimizations

Jinsoku follows the best practices of programming with Vulkan (Subtil, Rusch and Fedorov, 2019). The first one involves reducing the number of small allocations on the GPU in favor of a few big allocations, which can then be sub-allocated for the desired resources. Jinsoku does this by first collecting all the resources and calculating their memory requirements before performing any allocations on the GPU side. It is also necessary to distinguish between what type of memory is needed, and whether it needs to be visible on the host (CPU), device (GPU), or both. So one memory block of each type is allocated according to the needed size.

When the actual rendering is performed each object is associated with a pipeline, a

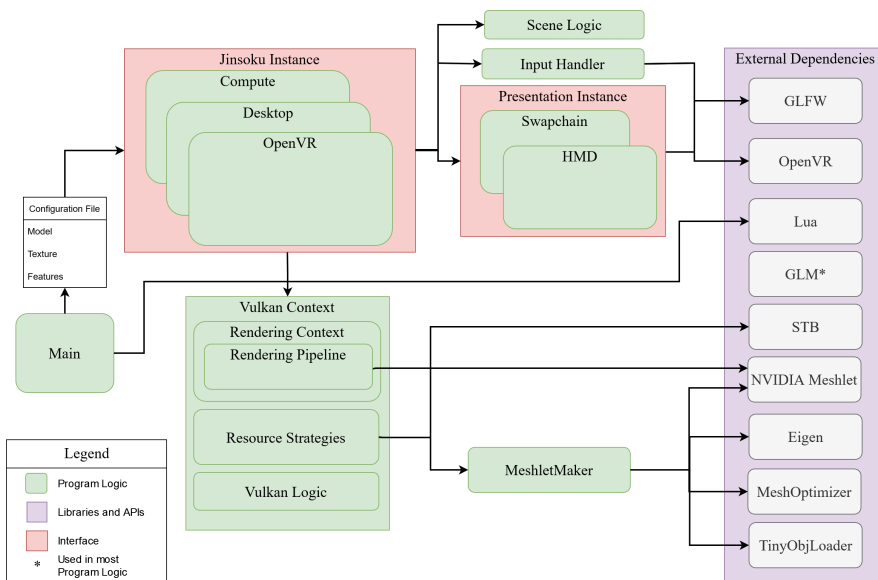


Figure 5.3: This diagram shows an overview of Jinsoku’s architecture. The Green block shows the program logic, being the actual functionality of Jinsoku. The purple block contains all the external dependencies and libraries that were used in the development of Jinsoku. The two red blocks show the parts of Jinsoku that are built as interfaces. *Jinsoku Instance* is a factory that spits out the required instance of Jinsoku, containing the features requested in the configuration file. The *Presentation Instance* provides an interface that defines where the rendered images are presented and depends on the instance of Jinsoku is running.

shader layout, and some data. This all needs to be bound before the draw call is made. If the objects are rendered in a random order this can lead to a lot of binding and unbinding of resources. The best practice is to group objects together based on pipeline and layouts to minimize rebinding. Which we do in Jinsoku.

All the instructions for the GPU are recorded into static *Command Buffers*, which are created once at start-up. *Command Buffers* are essentially small collections of prerecorded instructions that are recorded on the CPU side, before being sent to the GPU for execution. Two kinds of *Command Buffers* exist, secondary and primary buffers. A primary buffer can include secondary buffers. In Jinsoku we have one dynamically created primary *Command Buffer*, which we use to include static secondary buffers. That way we only have to record our drawing instructions once, at start-up into secondary buffers, and then dynamically add them at runtime. With this method, we can add more secondary buffers to the primary as new features are added to Jinsoku. This functionality can for instance be used to prerecord portals into buffers but only add the visible portals' buffers to the primary buffer at render time.

5.2.3 Static Registration

Since Jinsoku is made first and foremost for testing the performance of different rasterization pipelines and optimization strategies, it is important that it works in a modular way. We need to be able to ask for different pipelines and optimization strategies at start up, allowing us to record their performance and compare them to each other. We do this by use of the *Static Registration* programming pattern. This is a pattern that is often used when building plugin architectures, where it is impossible to predict how many plugins might be created during the life cycle of the program. In our case we have two types of "plugins". We have rasterization pipelines and resource allocation strategies.

We implement the *Static Registration* pattern by taking advantage of the fact that static variables are initialized before the main function is run in C++. For each rasterization pipeline and resource allocation strategy we instantiate a static variable that holds an instance of a *Registry Class* that is associated with it. On instantiation, this class calls a static method which adds it to the C++ equivalent of a list. This approach is heavily inspired by NVIDIA's Mesh Shader demo on github (Kubisch, 2018b). That way, when setting up Jinsoku, we can go through this list and pick a rasterization pipeline, and based on that, a resource allocation strategy. Each registry has a priority and a list of requirements that need to be fulfilled for it to work on the current hardware. This allows Jinsoku to automatically pick the right combination of rasterization pipeline and resource allocation strategy for getting the best performance possible given the hardware that it is running on. Each renderer needs to support at least one resource strategy. Currently, we have two different rasterization pipelines, we have a pipeline

that implements the *Vertex Shading Pipeline*, and one that implements the *Mesh Shading Pipeline*. We have a couple of resource allocation strategy permutations for the *Mesh Shading Pipeline* that implements different storage strategies. The *Mesh Shading Pipeline* allows the rasterization pipeline to process *Meshlets* instead of individual vertices (Section 2.8). Chapter 4 and Chapter 6 go into more details on *Meshlets* and the *Mesh Shading Pipeline*. *Meshlets* are small clusters of triangles. The *Task Shader Stage* allows the pipeline to cull invisible *Meshlets* before they are processed. The input to the *Mesh Shading Pipeline* is definable by the programmer. This is the reason for our many resource strategies. These resource strategies have been used when doing research for Paper III.

5.2.4 Scripting

To make Jinsoku more accessible and usable for scientists that wish to use it for visualization we have set up a configuration file that is read when Jinsoku is executed. The configuration file is written in a scripting language called Lua². The configuration file can be used to set the model, texture, pipeline, and resource strategy that one wishes to use with Jinsoku. We have decided to use Lua instead of a text file because enforcing syntax on the configuration file minimizes the risk of errors being introduced when editing the file. When Jinsoku starts up it feeds the configuration file to a Lua virtual machine. Lua is a powerful scripting language that can interact with Jinsoku at runtime and potentially develop into a way of scripting scenes for Jinsoku.

5.3 Rendering For VR

Making Jinsoku VR-based means that we need to use an API for interfacing with a HMD. It just so happens that almost all HMD manufacturers have an API, and using these either ties you to one company or forces you to implement different APIs if you wish to support more than one HMD manufacturer. Valve has made an API, called OpenVR, which is somewhat cross-headset, and works with many of the big HMD manufacturers. The downside is that the documentation of the API is scarce, and gaining familiarity with it is primarily done by reverse-engineering a demo made by Valve. Despite this, we have implemented the OpenVR API.

Jinsoku is VR first. This means that Jinsoku runs naively in a headless mode. Headless means that we do not waste processing power presenting images to the desktop screen. Of course, it can be handy to present the images shown in VR to the desktop in certain scenarios, so we have included a setting in the configuration file that can be used to

²<https://www.lua.org/>

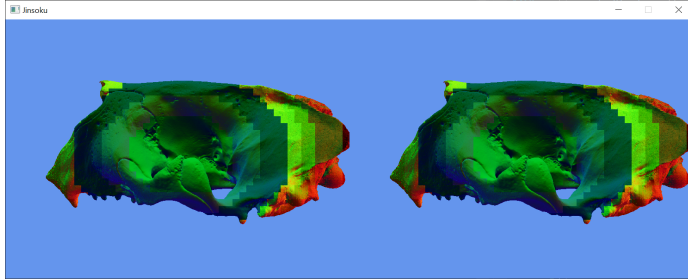


Figure 5.4: A render of the seal skull with VRS enabled. The VRS pattern is circular and moves outwards from the middle of the image, reducing the number of fragment calls as we move further out. This is indicated by the change in color.

enable presentation to the desktop. Other than this, arguably small, optimization we have implemented two big VR-specific features to optimize performance. These are detailed in the following subsections

5.3.1 Variable Rate Shading

We have implemented *Variable Rate Shading* (VRS) in our engine, this feature essentially allows us to render images to a frame buffer that emulates a varying pixel resolution. Ideally, this feature would be used together with eye tracking so that the part of the image that the user looks at is always rendered at the correct resolution, with a circular fall-off as we move out towards the periphery of the eye. None of our HMDs had eye tracking so instead, we made the circle at the center of the image, expecting users to always look straight ahead, and move the head instead of the eyes when exploring the VE. That way we focus our rendering efforts on the foveal area and spend less computational power on the part of the image that is only visible to the peripheral vision. We can get away with this because the peripheral vision is less acute. If the eyes are moved instead of the head, then the effect breaks down. Figure 5.4 shows an example of the skull rendered with VRS. We have overlaid a color onto the geometry to show how we have created different zones in the image that vary the number of fragment calls per pixel. In Figure 5.5 we see the Valve Index controllers in VR, where one controller stands out sharply in the center of the image, and the other is more jagged and pixelated. This is a result of the VRS pattern.

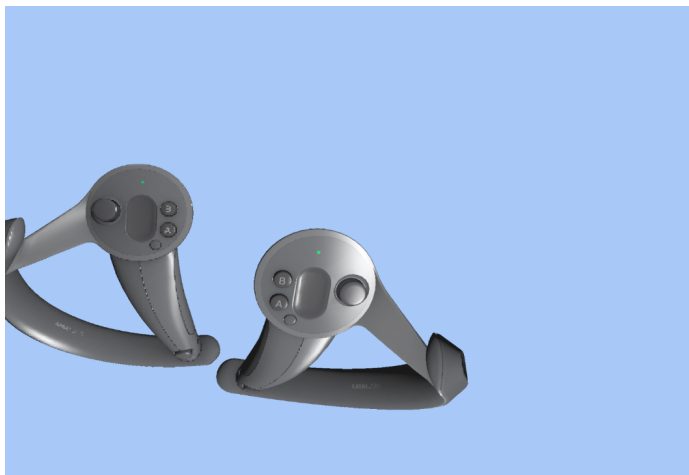


Figure 5.5: Two Valve Index controllers viewed in VR with VRS turned on. Notice how the left controller has become more pixelated due to the lower resolution in the peripheral area of the image

5.3.2 Multi-View Rendering

Multi-View Rendering (MVR) is an optimization strategy that lets the graphics pipeline change the order in which vertex and fragment shaders are called, for consecutive views. The normal way to render two views is to go through the pipeline twice i.e. calling the vertex shader and then fragment shader for the first view, and then again for the second view. When MVR is enabled, the pipeline runs the vertex shader for both views at the same time, and then the fragment shader for each view. This reduces some overhead when loading shader programs but more importantly, it allows the pipeline to take advantage of reusing vertices that are already processed and in memory. This can provide a good performance increase when a lot of the geometry that is being rendered is visible in both views. This is an ideal feature for increased performance in VR, since we always need to render two views instead of one, and the views are almost identical meaning that almost all of the visible geometry is visible in both views.

Figure 5.6 shows how MVR works. For the *Mesh Shading Pipeline* we have to take into account that we only call the Task Shader once at the start, so we need to expand the meshlet culling abilities to take both views into account.

An interesting little hack that we had to make use of is that when you work with MVR you essentially only use one image buffer twice as wide so that it can hold two images next to each other. This is clever because by having the two images in the same memory

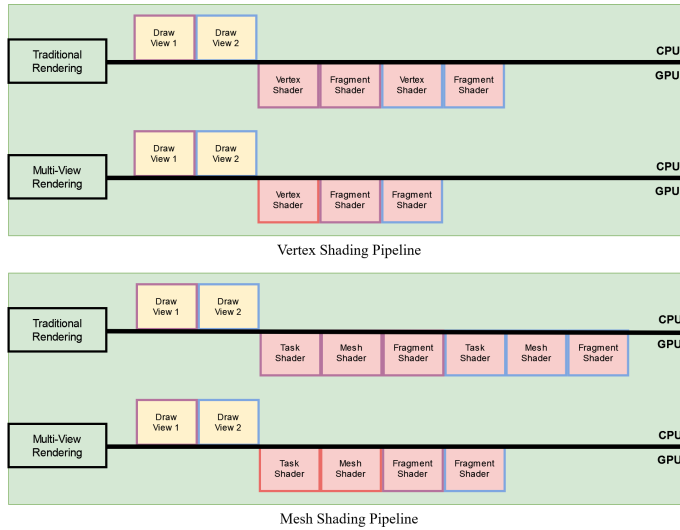


Figure 5.6: The difference between rendering a frame with and without multi-view enabled for the vertex pipeline and the mesh shading pipeline.

block it becomes more efficient when writing to it at render time. The OpenVR API however expects you to deliver two distinct images every time, so you cannot just give it one large image instead. These images are Vulkan objects. Each *Image Object* is first created and then backed by memory via an allocation process. To get around this you first create one *Image Object*, and back it with a memory allocation for the entire multi-view image. For the second *Image Object* you create it, but do not allocate any memory for it. Instead, you let it point to the middle of the memory allocation of the first *Image Object*.

5.3.3 Performance

We recreated the experiment that we performed for Paper I, with a version of Jinsoku that has been optimized after the publication of the paper. This version of Jinsoku features a *Manual Viewpoint Manipulation* interface where the user can drag themselves around the VE. Rotation around a controller can be done simply by grabbing the air and rotating the physical controller. Scaling the VE up or down is done by grabbing the air with both controllers and moving them closer together or further apart. We use four different versions of Jinsoku. The first just has the general optimizations, and refactoring of the engine, the second adds MVR, the third adds MVR and VRS, and the fourth one adds MVR, VRS, and our best-performing meshlet clustering algorithm

from Paper III. We use the *Valve Index* and desktop setup in this experiment. All the versions of Jinsoku that are used in this figure use the *Mesh Shading Pipeline*. The results are measured in milliseconds and can be seen in figure 5.7. We see that the general optimizations bring the average render time when inspecting the wing in VR, down when compared to the version of Jinsoku used for Paper I. The average render times from the version of Jinsoku used in the paper, for inspecting the wing from afar is ~ 20 ms and ~ 5.0 ms when inspecting it up close (Paper I). When using MVR and inspecting the wing from afar we decrease the render time by roughly ~ 2.0 ms compared to the general optimizations. When inspecting the wing up close we drop down to an average render time of ~ 2.5 ms from ~ 4.0 ms. Adding VRS provides a small decrease in render time, when inspecting the mesh from far away, which is not enough to justify the deterioration in the image quality. Of course, the improvement may be larger if the surface illumination model used is more complex than the Phong shader. Lastly, we add our novel clustering algorithm to shave another ~ 0.5 ms of the average render time for inspecting the wing from afar, bringing it down to ~ 7.5 ms. The optimized version of Unity had an average render time of ~ 10.0 ms for inspecting the wing from both afar and close (Paper I). We now outperform both Unity and our previous version of Jinsoku. When inspecting the wing from afar we render the images $\sim 25\%$ faster than Unity and $\sim 62.5\%$ faster than the previous version of Jinsoku. When inspecting the wing from up close we render the images $\sim 75\%$ faster than Unity and $\sim 50\%$ faster than our previous version of Jinsoku. We observe that the render time does not decrease when inspecting the mesh up close for any of our three most optimized solutions. This is most likely because we have managed to shift the bottleneck in rendering elsewhere in the pipeline.

5.3.4 Conclusion

In this chapter, we have highlighted some of the optimizations and features that have been implemented in Jinsoku, as well as shown the performance that we can get out of a bespoke visualization engine. While working with Vulkan is indeed more time-consuming than working with an existing visualization platform, it has allowed us to leverage state-of-the-art hardware features, such as the *Mesh Shading Pipeline*, VRS, and MVR. These features have made it possible for us to gain a big boost in performance, where we not only outperform the other solutions that we compared it with but also shift the performance bottleneck to another part of the pipeline when inspecting the mesh up close. It is also worth noting that beyond implementing the *Mesh Shading Pipeline*, we have mostly adhered to best practices and reverse-engineered existing solutions when creating Jinsoku. The learning curve was steep in the beginning, but not as steep as it may be perceived to be, so if state-of-the-art features are desired then it might well be worthwhile to build a bespoke solution. The *Mesh Shading Pipeline*, despite being introduced in hardware four years ago, has not been available in commercial game engines before Unreal Engine 5 which has come out this year. Control

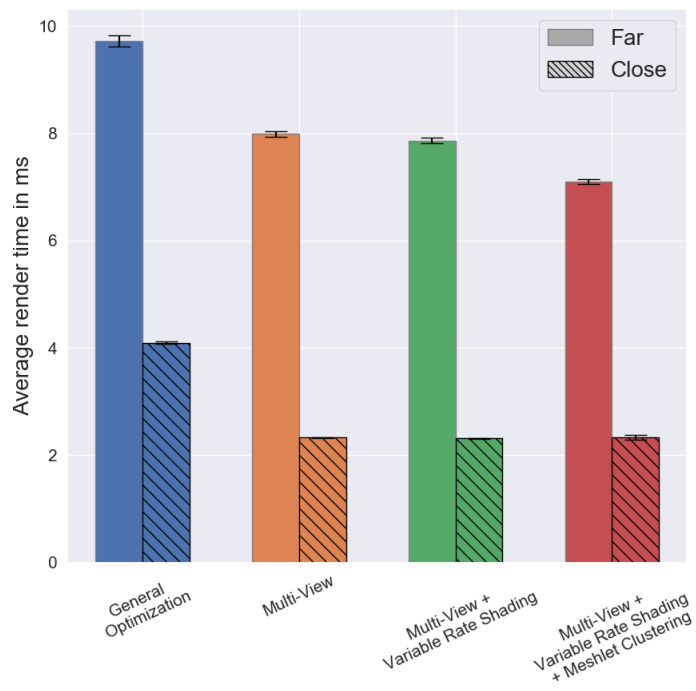


Figure 5.7: Comparison of rendering performance of Jinsoku when rendering the wing model, with different optimization strategies implemented.

over which features are implemented first, instead of having to wait for a third party to prioritize and implement them, can make a big difference in the final application.

CHAPTER 6

Efficient Rendering of Large-Scale Geometric Data using Meshlets.

In chapter 4, based on paper I, we compared different starting points for a VR-based visualization platform. We identified the bespoke Vulkan engine, based on the *Mesh Shading Pipeline*, to hold the most promise. The fact that the *Mesh Shading Pipeline* allowed us to only process the visible parts of the mesh, by using the *Task Shader* to cull *Meshlets* before processing, was indeed very convenient when working with very large meshes. Especially in VR where we would expect the user to get close to the mesh. While the *Vertex Shading Pipeline* also culled triangles that were not visible, it did not do so until after the vertices were processed.

Meshlet generation is essentially just clustering of triangles. The formation of these clusters is of great interest. Everything from the compactness and shape of the *Meshlet*, to the direction, that the triangles within the *Meshlets* are facing has an impact on how easy they are to cull. Not only this, but it also has an impact on how many *Meshlets* a mesh is divided into. It is however not entirely clear how big the impact of different meshlet collections is on the rendering performance. So to investigate if we ought to optimize after one of these parameters, all of them, or none of them we explore different ways of building meshlet collections and compare their render times. Clustering is a well-established method for exploratory analysis, so we take inspiration from the existing literature and compare existing methods against novel methods. This is done in our bespoke rendering engine, Jinsoku, as it gives us full control over the entire process. This work has resulted in paper III, currently in review, which is presented in this chapter.

I have worked on all aspects of this paper. Some of the initial investigations of using the *k*-medoids method were made by two master's students, but in the end, I had to re-implement their work as we started working with bigger meshes.

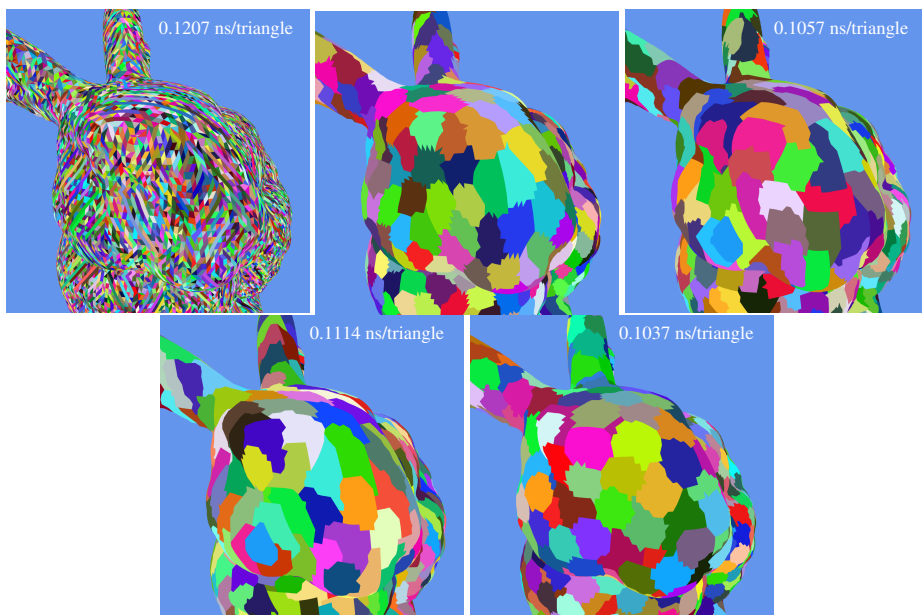


Figure 6.1: Different methods for organizing the triangles of the Stanford Bunny into meshlets.

Each colored patch is a meshlet. From top left to bottom right: NVIDIA (Kubisch, 2018b), k -medoids (Kaufman and Rousseeuw, 1990), greedy (ours), bounding sphere (ours), Kapoulkine (2017). We describe the details of the methods in Section 6.3. Each image shows the render time in nanoseconds per triangle. The time is based on a linear regression fitted to the render time of six meshes as a function of their triangle count. Because k -medoids has too few datapoints we omit its time.

6.1 Introduction

Rasterization is fast and highly parallelized on the graphics processing unit (GPU). In extended reality (xR) applications, where too low a frame rate breaks the immersion and potentially causes motion sickness (Rebenitsch and Owen, 2016), rasterization is the method of choice. Rasterization is however triangle bound, which means that every triangle must be processed for every frame. This can be prohibitively expensive if we want to visualize massive triangle meshes in xR applications. On the other hand, it is especially in xR applications that we need massive triangle meshes, because the user is free to closely inspect the geometry from arbitrary points of view.

To facilitate a higher triangle throughput, which helps uphold high frame rates even for massive meshes, the rasterization pipeline was recently modified to enable clustering of triangles into meshlets (Kubisch, 2018a; Kubisch, 2020). Meshlets improve performance by enabling us to process and cull geometry at a coarser level of granularity than triangles (Jensen et al., 2021). This relaxes the triangle boundedness, because the pipeline no longer needs to process all the triangles that are submitted to it. This modified pipeline is called the mesh shading pipeline.

Mesh shading is now directly exposed in Vulkan, DirectX 12, and OpenGL (Kubisch, 2018a). This gives rise to the question of how to best create the meshlets, i.e. the triangle clusters. Some developers, notably Kapoulkine (2017) and NVIDIA (Kubisch, 2018b), have released code for organizing triangle meshes into meshlets, but the question of how to form meshlets that deliver good rendering performance has received limited attention. In this paper, we evaluate the rendering performance when using different approaches for organizing triangle meshes into meshlets. Our tests include six different meshes consisting of 70 thousand to 39 million triangles. We evaluate performance by rendering the meshes from many randomly selected views while measuring render time per triangle. To our surprise, we find that meshlet collections produce lower render times when using local and greedy algorithms.

We also conduct a small explorative study into different meshlet descriptors in order to investigate how they affect render times. A meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. Apart from describing different algorithms for forming meshlet collections and reporting their rendering times, our main contribution is to identify the most important metrics to consider when assessing the quality of a meshlet collection.

6.1.1 Related Work

The GPU was originally introduced as special purpose hardware for triangle rasterization. Over the past few decades, GPUs have evolved into highly efficient and very general architectures for parallel computation (Haines, 2006; Dally, Keckler and Kirk, 2021). GPUs are discrete cards, which means that all data needs to be sent to the GPU if it is to be processed on the GPU. This can become a bottleneck (Hoppe, 1999) when working with large datasets, such as very big triangle meshes. A mitigation strategy for big triangle meshes is to use mesh representations that minimize the data footprint. One such widely used mesh representation is *triangle strips*. Triangle strips minimize the data footprint by feeding triangle strips to the GPU with consecutive triangles sharing an edge. In this way, the next triangle is simply described by processing one more vertex, as the two vertices from the shared edge with the previous triangle have already been processed. An index buffer can be used to represent the triangle strip. This is filled with indices to a vertex buffer and replaces vertex duplication with the less memory consuming duplication of vertex indices.

To organize a mesh into triangle strips, we need a path through the mesh where each triangle is only visited once. This is equivalent to finding a Hamiltonian circle in the dual graph of the mesh, which is an NP-complete problem (Dillencourt, 1996). As a result, greedy approaches for creating triangle strips have been explored instead. Arkin et al. (1996) generate triangle strips by greedily adding triangles with fewest adjacent triangles to the strip. This approach avoids leaving behind isolated triangles (triangle islands). The algorithm is made for a graphics API that predates OpenGL, called Iris GL. Iris GL has a command that makes it possible to change the vertex order of the last processed triangle. That makes it possible to change the direction of a triangle strip. Since OpenGL does not have this command degenerate triangles are added to the triangle strip in order to stitch strips together, at the cost of one extra vertex. Evans, Skiena and Varshney (1996) seek to minimize this use of degenerate triangles by generating triangle strips based on a global heuristic that looks for large patches that can easily be converted into large strips.

The *generalized triangle mesh* introduced by M. Deering (1995) relies on a special purpose hardware accelerated cache called the mesh buffer. This buffer stores vertices through explicit commands. Using a mesh buffer makes it possible to exploit that vertices are on average connected to six triangles, which is hard to fully utilize with triangle strips (M. F. Deering and Nelson, 1993). Chow (1997) introduces an algorithm for converting meshes into generalized triangle meshes.

Hoppe (1999) relies on the post transform and lighting cache (post-T&L cache). The post-T&L cache is part of the vertex shading pipeline. The vertex shading pipeline is the traditional rasterization pipeline which is used to process geometric data and turn it into rasterized images. The post-T&L cache holds the most recently processed vertices

that have not yet been converted into primitives. Using this, he optimizes triangle strips by reusing the vertices in the cache as much as possible. Several others have built on this principle to further improve rendering performance (Lin and T.-Y. Yu, 2006; Forsyth, 2006; Sander, Nehab and Barczak, 2007). In 2006, with the introduction of the unified shader model (Lindholm et al., 2008), the GPU became massively parallel, allowing for all shader stages to be run on all the generic processors on the GPU. This led Kerbl et al. (2018) to question whether the post-T&L cache is still a part of the GPU architecture. Based on empirical evidence obtained through vertex shader invocations, they show that modern GPUs turn the index buffer into smaller batches and process these in parallel.

A new rasterization pipeline called the *mesh shading pipeline* was introduced with NVIDIA's Turing architecture (Kubisch, 2018a). This pipeline lets the GPU process small parts of the mesh called meshlets instead of individual triangles. The pipeline no longer has the fixed function batching that Kerbl et al. (2018) found in the vertex shading pipeline. Instead, this is done by the programmer, providing the opportunity to make more informed decisions on how the mesh is batched into meshlets. Since each meshlet is processed in parallel, there is no longer a post-T&L cache to hold processed vertices, instead each processor has a cache of shared memory that all the threads on that processor can access. Since the batching is done before rendering, it does not need to take place again every time a new frame is rendered, removing some overhead. The pipeline expects a local index buffer for each meshlet as an output from the mesh shader stage, so this can either be precomputed or generated in the mesh shader. An optional task shader stage can run before the mesh shader to control culling, tessellation, and other things before it dispatches meshlets. The fragment shader stage is unchanged. Kubisch (2018a) provides an excellent overview of the hardware limits, built-in variables, and recommendations for the mesh shading pipeline.

The mesh shading pipeline has received surprisingly little academic attention. This is arguably because both the vertex and mesh shading pipeline benefit from triangle strips that are arranged spatially to increase vertex locality. This becomes quite apparent knowing that batching takes place on both pipelines, and both the post-T&L cache as well as the processor shared memory can take advantage of vertex reuse. Furthermore, real-time graphics is no longer exclusively about rasterizing triangles as efficiently as possible since real-time ray tracing and methods based on machine learning have become hardware accelerated and – often in combination – lead to a more diverse set of viable methods for efficient, high quality graphics.

Wihlidal (2016) shows how the graphics pipeline can benefit from clustering of triangles, and compute-based culling of these clusters. Jensen et al. (2021) show that the mesh shading pipeline has great potential for visualizing large geometric datasets, an Unterguggenberger et al. (2021) show how the mesh shading pipeline can be used for dynamic meshes. Mesh shaders work well for rendering large terrain (Santerre, Abe and Watanabe, 2020), and can be used for continuous level of detail (Englert, 2020). In

the gaming industry the mesh shading pipeline has been adopted and is now part of Unreal 5's virtualized geometry pipeline called Nanite (Karis, Stubbe and Wihlidal, 2021). It is also possible to find Github repositories with mesh processing tools for the mesh shading pipeline (Walbourn, 2014; Kapoulkine, 2017; Lempiainen, 2020). Neff et al. (2022) investigate texture atlases to reduce meshlet overdraw. In this paper, we explore different clustering strategies for meshlet generation and distill two key principles that lead to better real-time rendering performance when generating meshlets.

6.2 Meshlets Descriptors

The buffer setup that we use with the mesh shading pipeline has three buffers, see Figure 6.2. A local index buffer is divided into one section for each meshlet, and the local indices start from 0 in each section. The indices are all 8-bit because they refer to the local indices within a single meshlet. The hardware limit for vertices in a single meshlet is 256, so 8 bits suffice. The global index buffer is also divided into sections, one for each meshlet. This buffer differs from the traditional index buffer in the sense that index duplication is reduced. If one meshlet uses a vertex several times, the local index that points to the same global index is duplicated instead. The last buffer is simply the vertex buffer, which is the same as the one used for the vertex shading pipeline. With these buffers, all we need is a small descriptor for each meshlet providing information about it for the multiprocessor. NVIDIA suggests keeping the size of the meshlet descriptors to 128 bits which, on their hardware, is equivalent to the minimum amount of data that is fetched on a GPU-side load instruction. The meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. It needs to at least hold offsets into the global and local index buffers, as well as the number of primitives and vertices used in the meshlet. Other than this, the descriptor can also store a bounding box, an average normal for the meshlet, or any other information that the programmer wants to have associated with a meshlet.

The layouts of four different descriptors are in Tables 6.1 and 6.2. All descriptors use at most 128 bits. All descriptors pack a bounding box into 48 bits, namely 8 bits for the minimum and maximum coordinate on the x-, y- and z-axis. The bounding box coordinates are relative to the extent of the mesh bounding box. They all use 8 bits for describing the number of primitives and vertices in the meshlet. The normal cone is represented by a normal and an angle packed into 24 bits. The normal and cone angles are mapped into octants based on Cigolle et al. (2014). All data in a descriptor is packed into four 32-bit unsigned integers. The NVIDIA descriptor A packs the 8 bit cone angle partially into two 32 bit unsigned integers. The 4 upper bits in one and the 4 lower bits in the other. The remaining 3 descriptors pack the 8 bit cone angle together, which saves some unpacking within the mesh shader. The biggest point of divergence between the 4 descriptors lies in how they store the offsets required for the global and

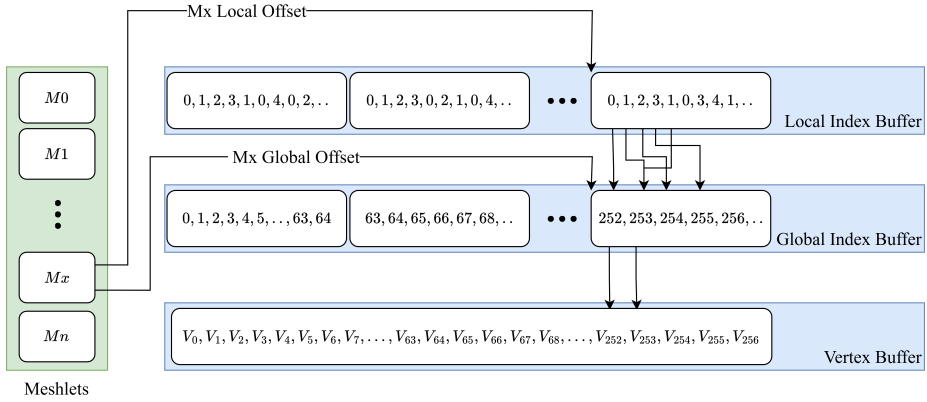


Figure 6.2: The three buffers used by the GPU when processing meshlets: local index buffer, global index buffer, and vertex buffer. The meshlet descriptor has offsets into these buffers. Note that global indices (re)appear in all meshlets they are used in.

Table 6.1: The memory layout of two meshlet descriptors proposed by NVIDIA (Kubisch, 2018b). Meshlet descriptors are 128 bit data structures that are used in task and mesh shaders.

NVIDIA descriptor A		NVIDIA descriptor B	
	Bits		Bits
Bounding Box	48	Bounding Box	48
No. Vertices	8	No. Vertices	8
No. Primitives	8	No. Primitives	8
Global idx offset	20	vertexPack	8
Local idx offset	20	Index buffer offset	32
Normal Cone	24	Normal Cone	24

local index buffers.

The NVIDIA descriptor A has 20 bits left for indexing into both the local and the global index buffer. This means that meshes that require an offset which is larger than 2^{20} will need to be broken into several draw calls.

The NVIDIA descriptor B takes these same 40 bits and uses 32 of them for offsetting which allows for much larger meshes. The downside of this is that the offsets into the global and local index buffers need to be aligned, as the same offset is used in both buffers. The remaining 8 bits are used to describe how the global indices are packed, i.e. if they are 16 bit or 32 bit numbers. This effectively means that the global indices can be packed into 16 bits for meshlets that only use global indices that are smaller than 2^{16} .

The third descriptor separates the task and meshlet descriptors, this means that it uses 256 bits for each meshlet instead of 128. But it only loads 128 bits per shader stage. By

Table 6.2: A descriptor for the task shader stage (left) and another descriptor for the mesh shader stage (right). Use of different descriptors for task and mesh shaders is an alternative to using the same descriptor for both shaders.

Task Shader meshlet descriptor	
	Bits
Bounding Box	48
Normal Cone	24

Mesh Shader meshlet descriptor	
	Bits
No. Vertices	8
No. Primitives	8
Global idx offset	32
Local idx offset	32

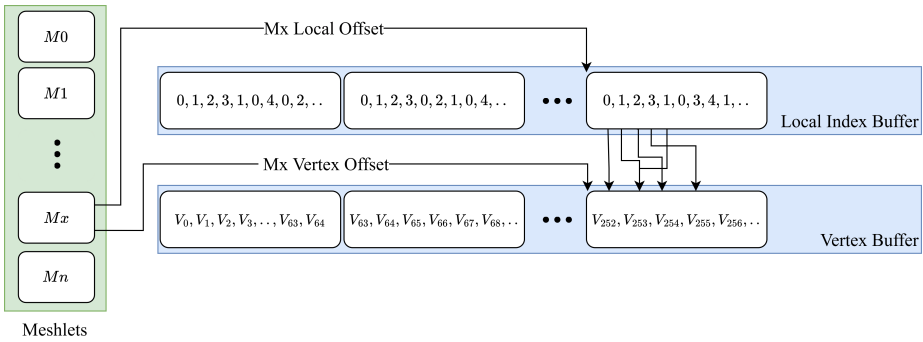


Figure 6.3: A monolithic version of the buffer setup used by the GPU when processing meshlets. Using only two buffers: local index buffer and vertex buffer. The monolithic meshlet descriptor has offsets into these. Note that vertices (re)appear in all meshlets they are used in.

doing that we can get rid of the task shader related data in the mesh shader descriptor and vice-versa. That way we can allow 32 bits for both the global and local index buffer offsets. So here we require no alignment between the buffers. We refer to this as the split descriptor.

Figure 6.3 shows an alternative buffer setup for a monolithic meshlet descriptor. The monolithic descriptor is also divided into two descriptors, to allow for 2x32 bits offsetting. One offsets into the local index buffer, and instead of using a global index buffer, the second offsets directly into the vertex buffer, which is divided into sections for each meshlet. The trade off here is memory, since some vertices will be duplicated and appear in several sections. On the other hand, no global index buffer is needed. The duplication is required for all vertices that live on the border of a meshlet. So, the four different descriptors all come with different memory footprints as well as some variations in how much GPU side unpacking they require.

Each meshlet can only contain a certain number of vertices and primitives. These numbers dependent on the GPU hardware. In the case of NVIDIA’s 2000 RTX series, the hardware limits are 256 vertices and 256 primitives. Lower values can be set as well. NVIDIA suggests using either 32 or 64 vertices and 40, 84 or 126 primitives for

each meshlet. In this paper, we use 64 vertices and 126 primitives throughout, which is the same as NVIDIA use in their meshlet sample (Kubisch, 2018b).

We use NVIDIA descriptor B when comparing the rendering performance of different meshlet generation methods because it allows us to process large meshes with one draw call. For our descriptor comparison, we compare all four descriptors while using the meshlet clustering method with best performance.

6.3 Meshlet Clustering Methods

The following paragraphs describe the different methods for organizing a mesh into meshlet collections (clusters) that we compare. Figure 6.1 exemplifies the differences between the meshlets generated by the different methods.

NVIDIA On behalf of NVIDIA, Kubisch (2018b) provides an example of organizing a mesh into meshlets. The meshlets are created one at the time by going through the index buffer. New primitives and vertices are added to the current meshlets as long as there is room for more. When it is full, a new meshlet is created. This process is repeated until the algorithm has gone through the entire index buffer. Every time a primitive is added to a meshlet it generates local 8-bit indices for the vertices, or reuses existing local indices if the vertices are already in the meshlet. It de-duplicates the global vertex indices, meaning that the global index of a vertex is only stored once, in each meshlet that uses the vertex, instead of being stored once for each triangle that it is part of. Instead the local indices are stored for each triangle. Because the local index buffer is 8-bit and the global index buffer is 16-or 32-bit this save spaces. The approach has a dependency on the original connectivity of the index buffer, and the resulting number of meshlets, as well as the vertex reuse within the meshlets is highly dependant on the structure of the index buffer. Figure 6.1 shows an example of the resulting meshlets. The index buffer appears to not be very optimized, which results in a lot meshlets being generated.

Kapoulkine Arseny Kapoulkine (2017) maintains a widely used and popular library called meshOptimizer. The library has several functions that improve, pack, and optimize meshes for better render performance, and it includes a meshlet generation strategy. First, the library creates a data structure based on triangle and vertex adjacency. A centroid and a normal is then calculated for each triangle, and the area of the mesh is also calculated. The area is used to create an expected meshlet area, assuming square flat patches. In addition, a k d tree is created from the triangle structure. All this is used to create the meshlets. The k d tree is used to pick the starting triangle for a meshlet,

and the adjacency structure is then used to look up the nearest triangles. Each triangle gets two ratings: one based on vertex reuse, another based on how much it increases the area of the meshlet. Regarding triangle reuse, triangles that already have vertices in the meshlet get a higher rating. Triangles islands also get higher importance. Should it happen that there is room for more triangles in the meshlet but none available on the border, the algorithm uses the *k*d tree to look up the nearest available triangle. The meshlet generation algorithm allows one to set a weight for the triangle normals, that will make it weigh these more when picking the next triangle for the current meshlet. We set it to 0.0, 0.5 and 1.0, and found that 0.0 produced the best results for the large meshes while the difference between the weights only had a very small impact on the small meshes. Because of this, we report our results with the weight set to 0.0.

Greedy We have developed a greedy algorithm that uses a list of vertices, where each vertex contains information about which triangles it is part of. The algorithm takes the first vertex, and then from that, grows out the triangle cluster until a meshlet is full. If a meshlet hits the vertex max before the primitive max, we look at the border of the meshlet for triangles that already have all vertices in the meshlet, and add these. A new meshlet is then started from a vertex on the border of the meshlet that was just completed, and the process is repeated. If a meshlet runs out of available triangles on its border, we go back to the list and picks the next available one. Because of this, the algorithm is sensitive to the order of the vertex list. We therefore use a heuristic to sort the list before running the algorithm. We find that half the time sorting according to the biggest bounding box axis length gives the best result. In particular, this is the case for the three biggest meshes. We also developed a version using a triangle list instead of a vertex list, but found that the vertex based algorithm always outperformed the triangle based. This is most likely because the meshlet border for vertices is based on all the triangles that the vertices in the meshlet touch, while the border in the triangle version is based on all triangles that share an edge with triangles that are already in the meshlet. This effectively means that the border is "larger" for the vertex version which results in fewer meshlets overall. Moving forward we only report on the vertex based algorithms, and use the heuristic of sorting the vertex list based on the longest bounding box axis of the mesh, from low to high.

Bounding sphere Our more advanced strategy is similar to the greedy one, except we here grow a bounding sphere around the starting vertex and use an algorithm by A. Bærentzen and Rotenberg (2021) to add triangles that minimize the radius of this bounding sphere. In addition to striving for a minimal bounding sphere radius, we also (inspired by Kapoulkine) prioritize triangles with vertices already in the meshlet and triangle islands.

***k*-medoids** One way to create clusters of triangles is by turning a mesh into smaller partitions using *k*-medoids (Kaufman and Rousseeuw, 1990). While this is an algorithm normally used for unsupervised learning, to investigate if and how many clusters a dataset might have, we use it to obtain balanced clusters. We chose the *k*-medoids approach because it works along the mesh surface, whereas the more commonly known *k*-means clustering would use a centroid, the cluster mean, to represent a cluster. A centroid detached from the surface easily results in clusters with triangles that are not connected. A medoid on the other hand is an actual datapoint within the cluster that is most suited to represent that cluster. These can be found by minimizing the dissimilarity within a partition. The *k*-medoids method partitions the mesh into *k* clusters and finds the medoid for each of these clusters. The medoid is the triangle with the shortest distance to all other triangles in the cluster. The algorithm runs in two steps after creating an initial clustering of the mesh. First the medoids of all clusters are found. All triangles are then compared to these medoids and assigned to the cluster with the most similar medoid. These two steps are repeated until convergence (Kaufman and Rousseeuw, 1990). The dissimilarity can be expressed through a distance metric between triangles. We run the algorithm on a triangle data structure, where the distance between two triangles is equal to the number of adjacent triangles we have to walk through to get from one to the other. The convergence criterion is to have an average distance close to zero between the new and old cluster centers, meaning that cluster centers moved very little in the last iteration. We start the algorithm with a number of clusters found by dividing the total number of triangles with the maximum number of triangles in a meshlet. After convergence we check if the clusters fit into meshlets. If not, then we add one new cluster and repeat. By only adding one new cluster we minimize the total number of clusters at the cost of longer processing times.

The five methods just mentioned vary quite a bit in implementation complexity. With NVIDIA's algorithm arguably being the simplest to implement, as it just directly works on the index buffer. After this comes the greedy algorithm that uses a triangle and vertex adjacency structure in a sorted list instead of the index buffer, with the bounding sphere version adding a little complexity in terms of a triangle scoring function. Then we have Kapoulkine's which requires both a triangle and vertex adjacency structure, a *kd* tree, and two scoring functions. Lastly, we have the *k*-medoids algorithm which not only requires a triangle adjacency structure, but also two iterative steps based on the breadth first algorithm, and to even be applicable it needs to be optimized and parallelized.

6.4 Experimental Setup

We compare the five different algorithms to see which one performs best, and why. Our hope is that this comparison allows us to distil more general principles for mesh-




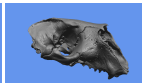

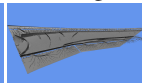
Bunny	Happy	Dragon	Skull	Nobby	Wing
					
V: 34 817 T: 69 630	V: 543 652 T: 1 087 716	V: 3 609 600 T: 7 219 045	V: 7 252 445 T: 14 504 882	V: 16 960 045 T: 32 905 214	V: 30 670 121 T: 38 629 758

Figure 6.4: The six meshes used in our experiment and their numbers of vertices (V) and triangles (T).

let generation that transcend the specific hardware, and numbers used. To make sure that no bias is introduced into the experimental process we have set up a Vulkan visualization engine which visualizes all the objects from a new random point in space each frame. Our efforts to randomize the view point is to average out the effect of overdraw. By setting the random seed we make sure that all algorithms are tested with the same sequence of view points, we do this for a total of 100.000 frames and record different statistics for each method that will be presented below. The frames are rendered at a resolution of 1280x720 pixels. We perform the analysis on 99.999 of the 100.000 frames. The first frame shows a significantly higher render time, presumably because of some data transfer between the CPU and the GPU, which is not evident for the subsequent 99.999 frames. All experiments were run on a desktop with an Intel Core i9-9900k, 64GB of DDR4-2666 RAM, and one NVIDIA GeForce RTX 2080 Ti Turbo OC with 11GB of GDDR6 RAM. We report our results in average render time per frame in milliseconds, while also exploring different other metrics surrounding the meshlets that impact the render timer. We use five different models for our tests in this paper. The vertex and triangle count of each model can be seen in Figure 6.4. The Stanford Bunny, Happy Buddha and Asian Dragon are from The Stanford 3D Scanning Repository (<https://graphics.stanford.edu/data/3Dscanrep/>). The Seal Skull has been 3D scanned into a point cloud and digitally reconstructed as a triangle mesh (<https://www.morphosource.org/projects/000355763>). The topology optimized airplane wing (Aage, Andreassen et al., 2017; Aage, Sigmund et al., 2020) is the largest model in our comparisons. The last mesh has been created with PrusaSlicer (<https://www.prusa3d.com/>) using a model called Nobby (<https://www.prusaprinters.org/prints/35338-nobby-octopus-sculpt>). We use the same experimental set up when testing the different meshlet descriptors, using the best performing meshlet generation algorithm.

6.5 Results

We are interested in finding a good clustering algorithm for meshlet generation. To investigate this, we plot the render times of the different algorithms as a function of triangle count in Figure 6.5. We see a fairly linear trend. The solid lines show render

Table 6.3: Render times.

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	0.15	0.28	0.87	1.70	4.21	4.68
Kapoulkine	0.13	0.22	0.79	1.33	4.09	4.10
greedy	0.13	0.23	0.76	1.36	4.11	3.74
bounding sphere	0.13	0.22	0.74	1.25	4.15	3.58
<i>k</i> -medoids	0.13	0.23	0.77	N/A	N/A	N/A

Table 6.4: Slope of a linear regression fitted to the six mesh render times based on the four different algorithms with and without culling. The slope shows how much an algorithm increases in render time as more triangles are rendered. The time is given in nanoseconds.

Method	without culling	with culling
NVIDIA	0.1246	0.1207
Kapoulkine	0.1179	0.1114
greedy	0.1109	0.1057
bounding sphere	0.1091	0.1037

times without meshlet culling, while the dashed lines include meshlet culling. Figure 6.6 shows how many percent of the meshlets are culled on average, each frame. The actual render times can be seen in Table 6.3. Here it becomes evident that for the two smallest meshes there is not really any difference in performance between the best performing algorithms, but clearly, for the larger methods there is a difference in performance. Given the linear trend we also fit a regression line to each algorithm, and report the resulting slope in Table 6.4. The slopes are reported in nanoseconds per triangle, with and without culling, and we consider these slopes an overall measure of the performance of the different methods.. The *k*-medoids method is omitted in this table due to the few data points. The smaller the slope is the less an algorithm grows in render time as more triangles are rendered. The bounding sphere algorithm achieves the smallest slope, so extrapolating from our six meshes, it increases the least in render time as the number of triangles grow. Since the difference between the algorithms is evident both with and without culling of meshlets, it means that the clustering within the meshlets themselves also contribute to the difference in render times. When we compare the render times to the implementation complexity of the algorithm, we have NVIDIA’s algorithm which is the simplest to implement, but this comes with a performance hit. On the other hand we have Kapoulkine’s algorithm which achieved good render times but is rather complicated to implement. Right in the middle we have the greedy algorithm. This has the second smallest slope while also being quite simple to implement.

Each meshlet has a maximum number of vertices and a maximum number of primitives that it can contain. We find that all methods (except *k*-medoids) have a very high average vertex count. For each meshlet collection, we find the average vertex fill (ratio

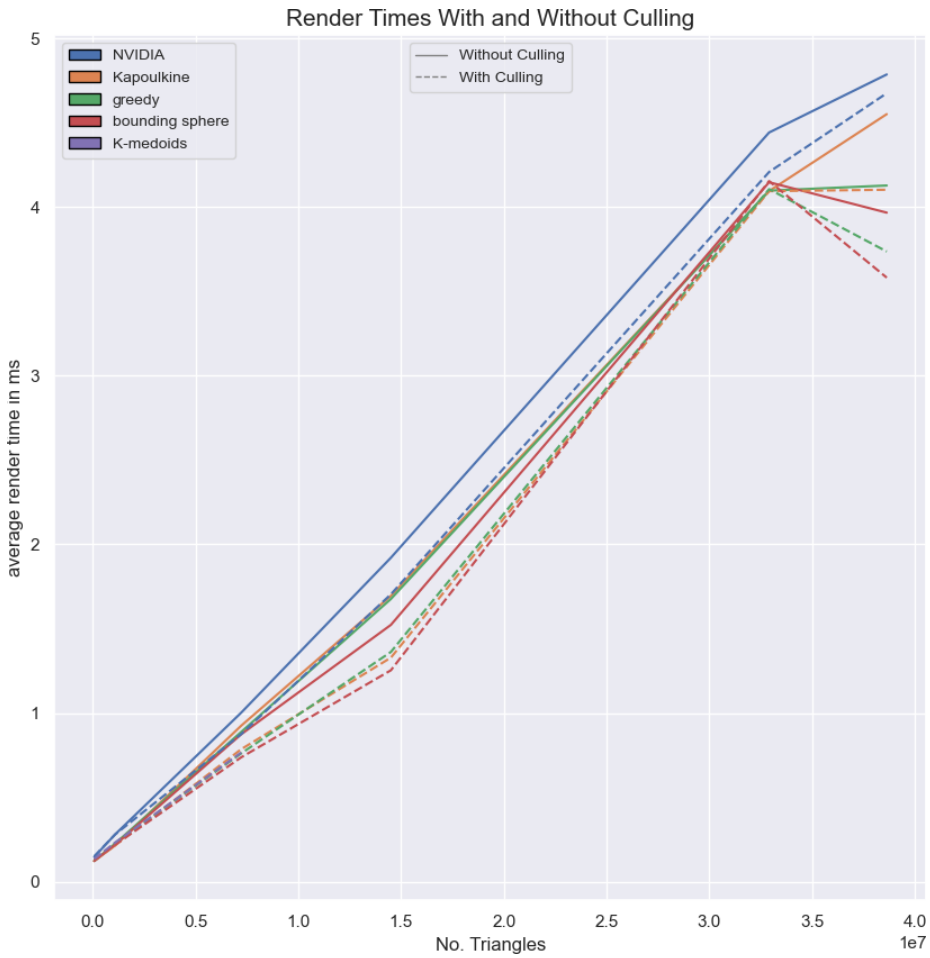


Figure 6.5: Average render time as a result of triangles based on the six meshes. Render times with meshlet culling are presented with a dashed line and render times without culling are presented with an opaque line.

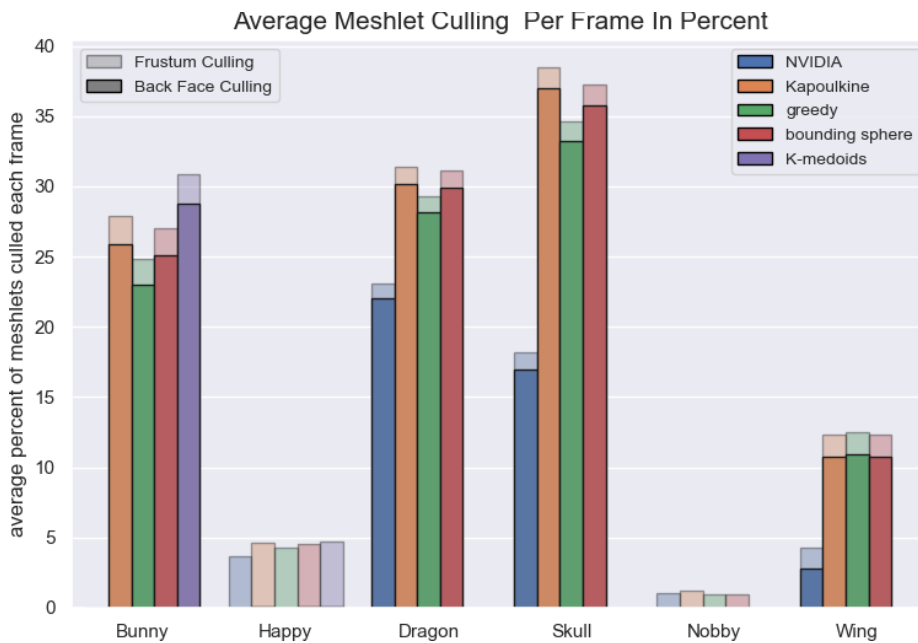


Figure 6.6: The average percent of meshlets that are culled for each frame when using the five different clustering algorithms. The culled meshlets are divided into two, the back face culled meshlets are represented by the fully opaque bars, while the frustum culled meshlets are represented by the semi-transparent bars.

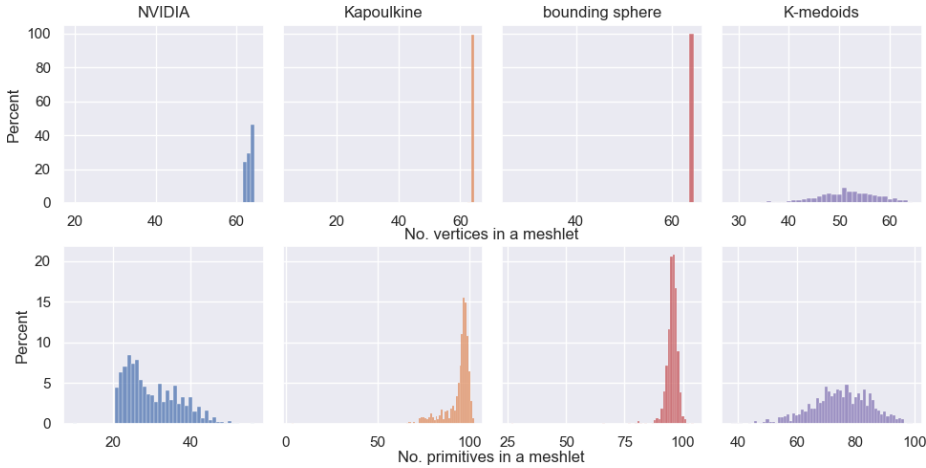


Figure 6.7: The distribution of the number of vertices and the number of triangles in each meshlet across four meshlet generation algorithms. The top row shows the vertices and bottom row is triangles. The meshlet collections are based on the Stanford Bunny mesh.

of vertices in a meshlet to the maximum number it can hold). All other collections have an average above 0.99 (except for k -medoids with Bunny: 0.812, Happy: 0.770, Dragon: 0.811). With all algorithms achieving close to vertex-complete meshlets, i.e. meshlets that are filled with vertices to the limit, the vertex completeness does not help us explain the differences in render times.

To see why k -medoids generates meshlet collections with a lower average vertex completeness, we compare its distributions to the other algorithms in Figure 6.7. Since the nature of the k -medoids algorithm is to balance out the clusters we get a distribution of the number of vertices with two fat tails. This means that we will always be below capacity, and when we compare it to NVIDIA’s, and especially Kapoulkine’s, we see high peaks and only a tail to one side. Kapoulkine’s algorithm performs better than both NVIDIA’s and the k -medoids, and produces quite few meshlets when compared to the two. The numbers of meshlets produced by the different methods for the different meshes are listed in 6.6. Since the k -medoids algorithm is trying to distribute the triangles and not the vertices the distribution of the number of triangles show the same two tailed distribution. NVIDIA’s and Kapoulkine’s distributions are more interesting. Kapoulkine’s has a peak at a high number of triangles, and a tail that falls off towards smaller numbers, while NVIDIA’s is opposite. This is most likely because of the index buffer, and how it does not promote locality as well as Kapoulkine’s adjacency based method, resulting in less locality and more unique vertices. These results informed us that greedy strategies ensure more vertex- and triangle-complete meshlets.

Since vertex completeness did not help differentiate the algorithms, we instead inspect triangle completeness. Table 6.5 shows the average primitive fill (ratio of primitives

Table 6.5: The average primitive fill for meshlet collections.

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	0.238	0.370	0.459	0.503	0.849	0.488
Kapoulkine	0.747	0.767	0.753	0.756	0.906	0.699
greedy	0.731	0.706	0.739	0.718	0.910	0.723
bounding sphere	0.756	0.744	0.759	0.755	0.911	0.751
k -medoids	0.600	0.568	0.597	N/A	N/A	N/A

to the maximum number of primitives). Unlike the vertex count, the primitive count varies quite a bit more across the different algorithms and meshes. If we compare this Table 6.3, we see a correlation between the methods that perform the best and their primitive fill being high (although not as simple as saying that the highest primitive fill yields the best render times). The primitive fill number also explains the variance in the meshlet collection sizes. If we look at NVIDIA’s algorithm for instance, it produces more meshlets than the other algorithms. Since each meshlet holds fewer primitives, we need more meshlets to represent the meshes. The k -medoids algorithm does not achieve a high primitive fill for any of its three meshes. Since it fails to produce high vertex fill, it becomes even more difficult to achieve a high primitive fill. NVIDIA’s algorithm has the lowest primitive fill, and also performs the worst, which indicates that it is difficult to build meshlets directly from the index buffer.

The NVIDIA and k -medoids algorithms both generate meshlet collections with a somewhat wide distribution of vertices and primitives (Figure 6.7). To investigate how this impacts the performance of meshlet collections, we sort the meshlets with respect to number of vertices and number of primitives. We only do this for the NVIDIA-based meshlet collections as the other algorithms generate more uniform meshlets. As seen in Figure 6.8, the order of the meshlets do play a role. The plot shows the render time when not culling any meshlets and using the NVIDIA descriptor B without index packing. We clearly see that sorting after vertex fill yields the best results. This is most likely due to the fact that vertices need to be loaded and transformed in the mesh shader, whereas primitives are represented by an index list that just requires loading in data. The reason why the render times are affected is that the GPU resources are used better. Meshlets are dispatched in groups to be processed in parallel, and if these groups are done processing at the same time, a new group can be dispatched without idle time. If the meshlets are of varying sizes, some will finish before others and will end up having to wait for the biggest meshlet to finish processing before a new group can be dispatched.

Since cullability increases performance of the meshlet collections, we find it interesting to explore the importance of the cullability of the meshlets. To test this we tweak our bounding sphere technique for generating meshlets. When a meshlet runs out of new triangles to add from its border, we finish the meshlet instead of going back to the vertex

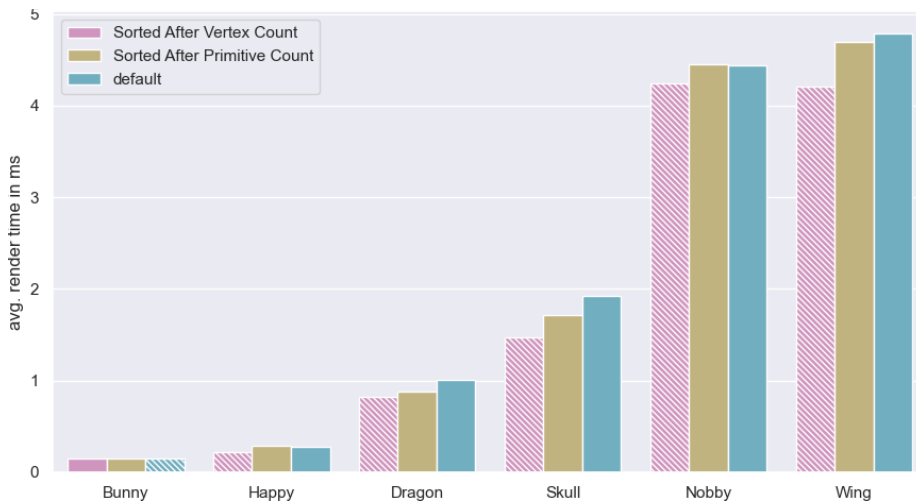


Figure 6.8: The average render times for the NVIDIA meshlet collections for each mesh as a result of sorting the meshlet list that is send to the GPU. The list is sorted based on number of vertices and primitives. The resulting render times are compared to sending the meshlet list as is. The hatched bar for each mesh show the best performing ordering.

Table 6.6: Number of meshlets.

method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	2 321	23 321	124 797	229 043	307 774	628 686
Kapoulkine	740	11 246	76 127	152 261	288 230	438 582
greedy	756	12 231	77 538	160 436	286 956	424 325
bounding sphere	731	11 605	75 457	152 501	286 767	408 302
k -medoids	921	15 210	95 953	N/A	N/A	N/A

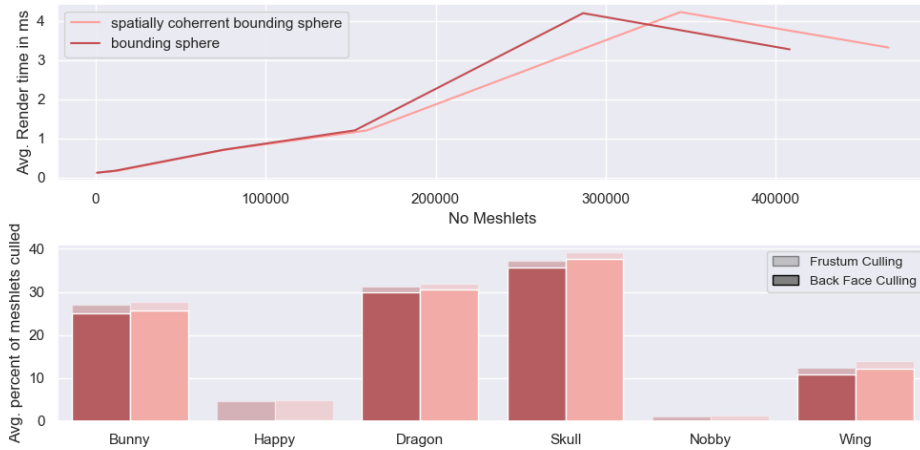


Figure 6.9: Comparison between the bounding sphere vertex meshlet collections with and without spatially coherent meshlets. The top plot shows the average render time, as a function of the size of the meshlet collections. The bottom plot shows the average percent of meshlets that are being culled per frame for each method.

list to look for new candidates. This enforces spatially coherent meshlets. By doing this we create more compact meshlets, making them more likely to be frustum culled. This also reduces the chance of adding a triangle with a normal that deviates too much from the meshlet normal. The increased cullability comes at the cost of a larger meshlet collection. In Figure 6.9, we see that the more cullable spatially coherent meshlet collections are offset to the right of the normal meshlet collection because they contain more meshlets. For smaller meshes, the spatially coherent meshlet collections show better performance, despite having more meshlets. The increased number of meshlets seems to be offset by the larger amount of culling. The increased culling is however not sufficient to hide the larger loading and processing times for the big meshes. Here, the difference in render times between the two meshlet collections is small.

Meshlet Descriptor Comparison We use our bounding sphere algorithm to test the four different meshlet descriptors described in Section 6.2. Results are in Figure 6.10. The type of descriptor that has the best performance varies from mesh to mesh. Only for Nobby we see a really big difference in render times. Here, the NVIDIA pack descriptor outperforms the other descriptors with as much as 1 ms. The Nobby model is a representation of a 3D print, because of this it consists of tubes. These tubes will have normals that point in all directions making it impossible to form meshlets with well defined normal cones, meaning that no or very little back face culling is taking place. Because of this, all visible meshlets are processed which gives an interesting insight into how much the meshlet culling affects performance. The high average render time for the NVIDIA descriptor A is most likely a result of overdraw, because the mesh has

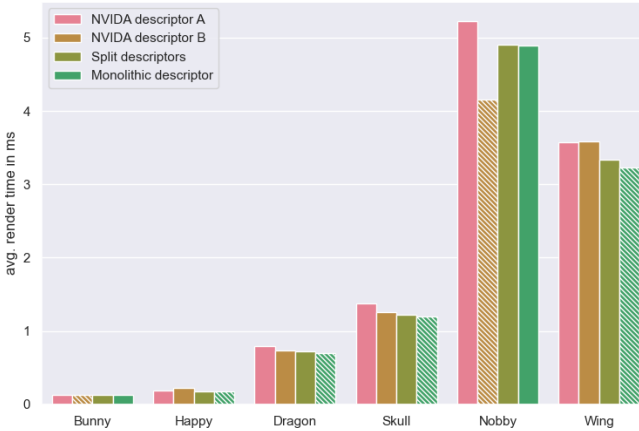


Figure 6.10: Performance comparison across the six meshes for the four different meshlet descriptors described in Section 6.2. For each mesh, the hatched bars highlight the descriptor with best performance.

almost no cullable meshlets, and is divided into several chunks. NVIDIA descriptor B has some (680) meshlets that can be compressed.

6.6 Discussion

Most of our experiments show that vertex completeness is important. Exploring the meshlets generated from k -medoids show this the best. The distributions from Figure 6.7 and render times from Table 6.3 shows that one should prioritize vertex complete meshlets over balanced meshlets. Our investigation into spatially coherent meshlets show the same, albeit with a weaker signal. Spatially coherent meshlets results in better cullable meshlets at the cost of generating more meshlets. Generating more meshlets means having a bigger distribution of vertices and primitives. The differences here are small when compared to the k -medoids results because the portion of meshlets with lower vertex completeness is small, but for bigger meshes it starts to affect performance more. More vertex complete meshlets also means more uniform meshlets, and more uniform meshlets reduce render times. We saw this when sorting the NVIDIA meshlet collections in Figure 6.8.

Inspecting Table 6.3 in conjunction with Table 6.5 revealed the correlation between high primitive fill and better performance. It is interesting to explore the interaction between average primitive fill and vertex completeness by inspecting the k -medoids and the NVIDIA meshlet collections. For the Bunny mesh, we see an example where the average primitive fill on the NVIDIA meshlet collection is so low that the high

average vertex fill cannot compensate for it. This demonstrates that one should not only optimize around one heuristic but take both into account. For the Happy mesh, the NVIDIA collection performs better than k -medoids, showing that vertex completeness is more important. For the Dragon mesh, the tables have turned and the k -medoids collection, with a better balance between the two, performs best. This interaction tells us that it is important to prioritize both vertex completeness and primitive fill. We also observe that it is hard to have a high primitive fill without having nearly vertex complete meshlets.

Striving for cullable meshlets is the third heuristic. Our experiments show that cullable meshlets can help balance out larger meshlet collections. Figure 6.9 exemplifies how meshlet collections slightly enlarged to increase cullability can indeed result in better performance. It does however not seem to affect performance as much as vertex completeness or maximizing the primitive fill.

The Skull and Nobby meshes produce some surprising results for some of the meshlet generation strategies. It is surprising that Kapoulkine's algorithm does not perform best on the Skull, as the data show more culling, and less meshlets. Perhaps the difference is that our method builds meshlets along the z -axis of the skull as opposed to from the middle and out, this could affect vertex loading, overdraw, and cache misses on the GPU.

Nobby shows that some meshes will be exceptions to the rule. It will be possible to find meshes where these heuristics and metrics break down. In fact, tuning one aspect of meshlet generation affects all the other aspects. The metrics, and indeed most of the factors we explore in this paper are highly correlated, and this can make it hard to isolate different aspects as they affect each other. Two collections of meshlets might differ in efficiency even if almost all meshlets are packed to capacity in both collections. Because of this, it becomes even more desirable to have an algorithm that is simple to implement. The greedy algorithm proves to be quite useful in practice as it achieves good render times across the meshes while also being simple to implement.

Lastly we conducted a small exploratory experiment which compared different ways of packing the meshlet descriptor data. Interestingly, we find that the monolithic descriptor performs quite well. This is certainly interesting. The monolithic descriptor uses a simpler buffer setup, and by using one descriptor per shader stage, it becomes possible to add more meta data if desired.

6.7 Conclusion

We find, quite simply, that the more compact the meshlet collections become the better they perform. Meshlets have vertex and primitive limits, in this paper we used the suggested 64 vertices and 126 triangles. Since each triangle requires 3 vertices, the meshlets always hits the vertex limit before the triangle limit. In other words, it is absolutely paramount that a meshlet collection achieves a high average vertex fill. In this way, it becomes possible to also have a high average primitive fill. Because of this, we recommend the following strategy for optimizing meshlet generation: *make the meshlets vertex complete first and then maximize the primitive fill*. The combination of these two will create meshlets with large vertex reuse and locality, while also minimizing the total number of meshlets that are required to represent a mesh. Finally, we of course recommend to *strive for cullable meshlets*, but not at the cost of a too big increase in meshlet collection size. We found that performance rather quickly drops when the meshlet collections grow in size.

We also explored other properties of both the mesh shading pipeline and the meshlet collections. We found that high uniformity in the meshlet collections promotes even workload across processors on the GPU which yields better render times. Different meshlet descriptors do not have the biggest impact on render times, so working with monolithic meshlets could prove to be a good choice for scientific visualization where rendering is done on distributed systems. As an interesting topic for future work, descriptors that require less data unpacking in the mesh shader could yield improved render performance, and since dividing descriptors into two also did not affect performance too much, it could be interesting to explore whether new useful meta data could be added.

6.8 Retrospective

The results of Paper I made it quite clear that our bespoke Vulkan-based visualization engine attained the upper hand when inspecting meshes up close by leveraging the *Mesh Shading Pipeline* and its ability to cull meshlets before processing them. An important feature for exploring the use of portals for interaction in VR.

Simply put, one portal in VR requires the visualization engine to render two new images, one for each eye. Meaning that for each portal that is added, the entire VE needs to be rendered two more times. That would simply not be possible while maintaining a high frame rate without the results of this paper. Without the results of this paper, we would probably have abandoned the portal-based interaction prototype that is explored more in the next chapter.

Portals: A Swiss Army Knife for Scientific Visualization

The work presented in this chapter is based on the work made during the external stay in the Dynamic Graphics Project laboratory at the University of Toronto. The work ties into the second part of the thesis and lays the foundation for a way to better interact with complex geometric datasets. During the external stay, a prototype of a portal-based navigation tool was made. The tool was implemented in our bespoke engine, Jinsoku. Because the prototype does not include all the desired functionality, part of this chapter outlines that missing functionality. As well as argue how that would not only improve the prototype but make portal-based interaction a welcome addition to the visualization of scientific data in VR. We strongly believe that portal-based interaction can become the standard interaction tool for scientific visualization in VR. We are also aware that what we present in this chapter is our reasoning for why we believe that portals have the potential to become the standard interaction tools rather than a complete solution that is ready to be used. Deciding and refining the interface and input devices for the desktop computer took more than 30 years of collaborative effort across different industries (Perry and Voelcker, 1989). Similarly, so will the establishment of portals as a standard interaction tool take time and require collaboration. We hope that our efforts can mark the starting point for finding a standard way of interacting in VR, that will help make VR more accessible.

7.1 Introduction

Scientific Visualization is a largely heterogeneous discipline, with specialized visualization tools being tailor-made for each project to support specific data types and interaction modalities (Kehrer and Helwig Hauser, 2013). This means that switching from one visualization tool to another requires learning a new interaction modality. This creates a barrier. Desktop visualization tools have lowered this barrier because *Digital Content Creation* (DCC) and *Computer-Aided Design* (CAD) software, which is used for making visual effects, construction planning, and video games, has driven a default way of interacting with 3D data. VR-based scientific visualization, however, is a newer field and does not enjoy the same standardized interaction interface. So for VR-based visualization tools to become more widely adopted they need to break down this barrier through a standardized interaction interface (El Beheiry et al., 2019).

We propose the use of portals as a standardized tool for the immersive visualization of scientific data. Portals hold the potential of becoming the only interface required for all our needs, lowering the barrier of switching to new visualization tools. We believe that VR applications can use portals similarly to how DCC and CAD software uses the same navigation and interaction modalities across application, allowing users to rely on previous experiences when trying out new software. Drawing further inspiration from people's understanding of portals, as well as their representation in media, can help further lower the barrier. Permitting users who have previous experience with portals to rely on that to faster and more efficiently adapt to VR applications. Just as good gameplay and tight controls are dependent on a great implementation, so will the successful use and adaptation of portals depend on it.

7.2 Portals

When we hear the word portal, we think of two connected doorway- or window-like gateways. These gateways are not physically connected but instead create a discontinuity in space-time that allows for things to pass discretely from one gateway to the other, and back again. The movement between the two gateways happens instantaneously, no matter the distance between them, and can be seen as a form of teleportation. It is possible to see through the portals. One gateway will show what is visible at the other gateway.

The window-like metaphor for portals is very reminiscent of the multi-view setup that has become omnipresent in desktop-based 3D software. When working with 3D data in a multi-view grid setup, the user interacts with the data through one window while inspecting it through all of the windows. Inspecting the data through several windows at the same time helps understand the shape and curvature of the data, ultimately guiding the interaction and making it more precise. Portals are quite frankly the natural extension of this setup, from desktop to VR. Instead of having a grid of windows, we have portals, physical objects that exist in the VE, with the same affordances, but better. Better because the ability to perform *Maneuvering* in VR is effortless, allowing the user to make small viewpoint corrections by moving their head. What makes portals better is that these small movements are reflected in the portals, changing the view of the data with the head movement revealing different details. The analogy is easy to understand because it follows so naturally. We find that existing literature fails to make this observation, perhaps because it follows so naturally, but this is exactly why portals hold the potential to become a unified interaction metaphor for VR-based scientific visualization.

7.3 Portal-Based Interaction with Scientific Data in VR

In Section 2.10 we went through the different VR-based visualization tools, and in Section 2.9 we explored the different techniques that could be used for desktop-based scientific visualization of geometric data. We also explored the use of portals in VR in Section 2.5.

We now aim to combine the knowledge that we have gained from exploring these three sections, to build a portal-based interaction metaphor for VR-based scientific visualization of geometric data. One that can combine the required interaction and visualization techniques from scientific visualization of geometric data with the definition of a portal that we made in the previous section. To keep consistent with Section 2.9, we divide this section into the same six categories.

Visual Data Fusion

Visual Data Fusion can clutter up the geometric data, making it difficult to comprehend the underlying shape. Here the window analogy of portals can help users inspect a version of the geometric data with the abstract information encoded into the texture, next to a version of the geometric data with a surface shading that highlights its shape.

Relation and Comparison

Portals can be used interchangeably to look through one, and then another behind it, either by moving the portals around or by changing the opacity of the portal in front, making it transparent. One could also perform post-processing on the portal images to facilitate the superimposition of objects onto each other, but we do not find it wise to support this because it inherently changes how the portal works by essentially merging different worlds and portals into one. It would be more appropriate to show portal *Slices* as that will most likely cause less cognitive dissonance in the user. Inspired by vertically slicing objects in a dataset and subsequently stitching the slices, one from each object in the dataset, together for comparison (Alabi et al., 2012). In fact portal *Slices* may enhance the experience of using *Slices* because *Maneuvering* means that the visible view through the portal slices simply moves with the head.

Navigation

Since the camera is attached to the head in VR, we cannot rely on the techniques for changing the velocity of the camera. That and flying around in VR is very prone to cause motion sickness, so instead, the primary mode of transportation in VR is teleportation. As we can see from the literature on portals in VR, navigation is the primary use for them, and here portals truly do shine. They allow for teleportation in a more immersive manner, that does not require the user to reorient themselves after teleporting. They allow for a more immersive way of navigating a vast VE, by supporting redirection to utilize the small physical space better. They can even be used to curate routes through the data by being placed in advance, allowing the user to move from portal to portal experiencing just the regions of interest. A WIM-like view can be achieved with the use of portals by adding a transfer function to the portals that affect the scale of the VE. Allowing the user to get an overview of the model or inspecting it up close. Simply by moving through portals.

Focus+Context and Overview+Detail

Viewing geometric data through a multi-view grid is a common part of the workflow when working with a desktop setup. As discussed earlier, portals are a very natural extension of the multi-view grid to VR. This allows for a multi-view setup in VR that can easily be used for *Focus+Context*, by having the view through the portal show the data in a higher resolution, or with a more complex surface representation for a better understanding of the geometry, while also having the context available. Essentially using portals identical to how one would use a magnifying glass. Similarly, a multi-view portal setup can be used to zoom in on details in one portal, while the other shows the entire data from far away, allowing for a very intuitive way to facilitate *Overview+Detail*.

Interactive Feature Specification

The user might be able to interact with the data too, either by changing it, annotating it, or marking regions of interest. This type of interaction is necessary if the user wishes to share their findings with other scientists or perform subsequent analysis on the data. The view into the data that is visible through a portal can be used for this. Not only that, but by presenting all of the portals that previous users have placed, subsequent users can get a quick overview of all regions of interest, and even navigate to them. It is further possible to extend this region of interest by associating an annotation point on the data with a portal.

Another way to handle Annotation could be by first placing a portal and then annotating through it. Annotation through a portal, while looking at the data, is an interesting use case that might be more cognitively demanding, depending on how this is implemented. One way would be to rely on partial teleportation and allow the user to put one controller through a portal while viewing the object. This might cause some confusion due to the conflict between proprioception and visual stimuli of a hand or controller floating detached in space. Another way to handle this could be through action at a distance, where some item held by the user extends into the portal and gets teleported (Singh, 2021).

Data Abstraction and Aggregation

The portal itself can be the annotation. The visible vertices, the image from the portal, or even the center point of the portal projected onto the geometric data can act as the annotation. As the portal can be seen as a virtual camera, the surface shader or other rendering parameters can be changed post-annotation to produce different images. These different ways of representing the same annotation point can be helpful in *Geometric Deep Learning* (GDL) because they can be used as different input formats. This can help explore the landscape of formats for inputs to a network, without requiring annotators to create new annotation points. As GDL is being used for more complex geometric datasets, we also need better annotations, because the neural network cannot get rid of the variance of the same landmark across different objects and annotators, already present in the dataset. Lifting the multi-view interface into VR could be one way of reducing that variance, especially if combined with *Interactive Machine Learning*.

If desired, the portal views can also be used to export images of the regions of interest to show others an aggregation of the most interesting parts of the data. This would also allow collaborators who do not have access to VR to see the discoveries that have been made.

7.4 Case Study

We developed a prototype in Jinsoku that uses portals for navigation. It does this by presenting the user with two portals at the center of the VE. The user can see the data that has been loaded into the VE through the two portals. The *Overview Portal* has a transfer function applied to it that scales the data down when the user steps through it, allowing the user to get an overview of the data. The *Details Portal* has a transfer function applied to it that scales the data up when the user steps through it, allowing

the user to inspect the details of the data up close. If the user goes back through either *Portal End* the scale is reset. The user can move and rotate the *Portal Ends* simply by moving them.

This facilitates a WIM-inspired workflow, where the user can walk through the *Overview Portal*, to get an overview of the data, and then move the *Portal End* belonging to the *Details Portal* to a region of interest before moving back through the *Portal End* they came from and into the *Details Portal* to explore the region of interest. Figure 7.1 shows the *Overview Portal* and the *Details Portal*.

While developing the prototype we had to consider a couple of portal attributes. We could have made the portals unidirectional, which has the major benefit that we only need to implement the portal entrance. However, making the portals bidirectional makes it easy to use them for navigation. To save on computational power, we disallow portals from being visible through portals. This means that we do not have to deal with the potential of wasting resources on rendering many recursive images for scenarios where one portal entrance is visible through the other portal entrance.

We allow portals to represent the VE seen through them in a variety of different ways. As such we can think of stepping through a portal as applying a transfer function that can change the VE, or connect different VEs. The change will be visible through the portal, so it will not be confusing to the user. In the scenario where two VEs are connected, a portal may lead to a VE with a different dataset. This can for instance be a different simulation of the same data or a different object within the same dataset. That way the user can have several portals that show different datasets, which allows them to step through a portal and inspect one of the datasets in more detail. It can also be helpful to display the same data through a portal with a different surface representation, or at a different scale.

We tested this prototype with one of the creators of the giga-voxel simulation-based wing (Aage, Sigmund et al., 2020). It is immediately apparent that this type of navigation has the advantage that the user does not need to constantly scale the mesh up and down to navigate around it. The imprecision that comes with a direct interaction interface, such as VR, can make it a chore to constantly scale the model up and down, simply because it is hard to hit the same scale that you were previously inspecting the mesh at. Being able to instead travel through portals that teleport the user to VEs that are already scaled to the appropriate sizes greatly smooths out this part of the workflow. It was also less fatiguing compared to the previous drag-and-grab interface. The navigation felt very intuitive and straightforward. Going into the *Overview Portal* with the miniature version of the wing, allowed the user to grab the *Details Portals Portal End* and move it around to the desired position before moving back through the *Overview Portal* and into the *Details Portal* with the scaled-up version of the wing. This type of navigation quite easily bridged the gap between the large space of the VE and the small physical space allowing the user to freely explore the wing, rather than constantly be

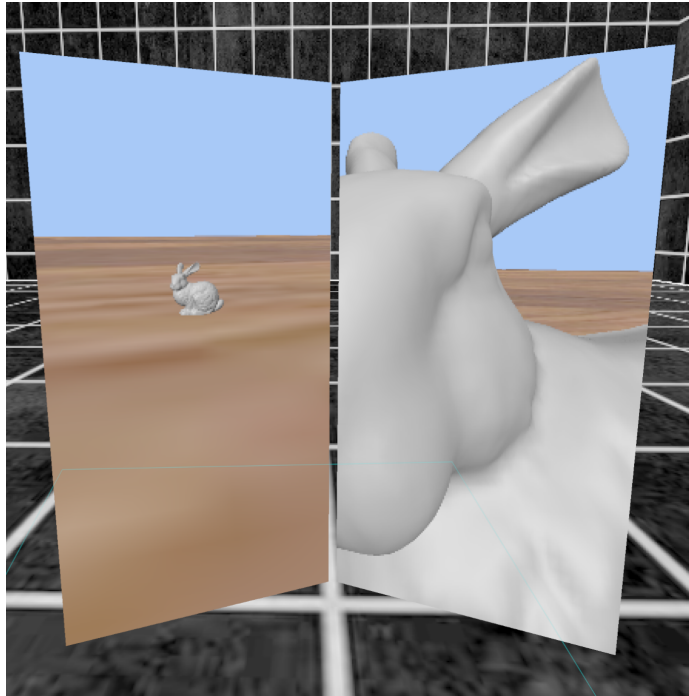


Figure 7.1: The *Overview Portal* (left) and the *Details Portal* (right), that allow the user to enter two different worlds. One where the Stanford Bunny is very large, and one where it is very small.

wary of moving beyond the physical space dedicated to the VR setup. All in all, it was the best way the creators had ever inspected their wing (Aage, 2022).

7.5 Discussion

Several aspects of the portal-based interaction tool have yet to be implemented, and this means that there are still several questions that need to be explored and addressed in future work, these questions are discussed here, together with a re-envisioning of how the experiment in Paper II could have been made with the use of portals.

Shape Morphology

When looking back on the experiment we carried out in Paper II, and the VR prototype that was used for annotation, some interesting observations can be made. The prototype

facilitated the placement of annotation points on the surface of seal skulls with an annotation gun. The annotation gun had a red laser sight to help the user aim. The annotation points were to be placed based on a description given to the participants. For most participants, the annotation workflow in VR included rotating and moving the skull around until the desired viewpoint was found. From there the skull was scaled up to help place the annotation point. After placing the annotation point the participants would inspect it to make sure that they were happy with the placement. Sometimes this would lead to a refinement of the placed annotation point. Post-placement refinement happened more often in VR, simply because it was easy for the participants to find a new vantage point that showed that the annotation point was slightly off from the local extrema. From this, we can see that placing annotation points at local extrema often requires viewing the mesh from different vantage points. Portals facilitate this by allowing the user to inspect the mesh from one vantage point and also see it through a portal from another vantage point. Portals also allow the user to place the annotation points independently of the viewing directions.

Using portals allows for other changes to the annotation process, such as facilitating automatic scaling of the mesh. One of the participants did not scale the skull up at all, which greatly affected his precision when working with the VR prototype. By having two portals dedicated to a WIM-inspired workflow, working with the skull at different scales would automatically have been integrated into his workflow. Another way to automatically enforce scaling could be to have users first place a portal before putting their hand through the portal to annotate the mesh. The placed portal would then show a scaled-up version of the mesh.

Since portals are essentially different virtual cameras, one of the portals can also be used to render the seal skull with a different shader that helps highlight curvature, or, even better, a more detailed version of the mesh could be annotate-able and viewable through a portal by combining portals with the mesh shading pipeline.

The Portal Room

Placing many portals begs the question of how to keep an overview of all the portals. The overview of the placed portals could be given to the user through a *Portal Room* that exists independently of what else is being visualized. The user can then walk through a *Portal End* to reach the room, inspect all portals and pick one that leads back to the data or even change the data that is being visualized from that room. A room could also allow for a small "holographic" version of the data to be displayed, which could include small representations of portals that have been placed. In fact, this room could very well tie into the presentation of the visualization application, always starting the user in familiar surroundings. From a scientific standpoint, there are some interesting benefits from using a *Portal Room*. Users have a tendency to underestimate distances in the personal space while immersed in a VE (Bruder, Steinicke, Wieland

et al., 2012). This can to some extent be minimized by starting off the user in a room that is a replica of the physical environment that the user is located in (Interrante, Ries and Anderson, 2006). Another added benefit is that this way of starting off a VR-based application also increases presence (Steinicke et al., 2009). It seems plausible that a more correct estimation of distance, combined with being more present in the virtual environment, allows for better analysis of spatial 3D data.

The Portal Compass

Another implementation that allows for a different experience, is to have an object at hand which has all portal entrances projected onto it. Similar to a compass the user can take the *Portal Compass* "out of the pocket" and inspect the portals. This object can then be left hanging in the air at which point it can be used as a viewport that shows the model from another angle. By utilizing a 3D object such as a sphere, the portals can be projected onto it based on their location relative to the model, which would reside in the middle of the sphere. This would also automatically disallow the user to view all portals at once, which could potentially put a lot of stress on the frame rate. The solution could be a combination, where the *Portal Room* is used to start the application and to facilitate the WIM-based navigation, while the *Portal Compass* contains the portals that are placed throughout the inspection of the data.

Partial Teleportation

When working through a grid of viewports in DCC and CAD applications, the user can manipulate data through the different viewports without "going" through them. They can choose to maximize one viewport, which could be considered "going through" that viewport, but often the desire to do so depends on the task being carried out. This task-dependent interaction poses an interesting question for our portal implementation. Namely, should it be possible to only move the users' hands through a portal to interact with the data being visualized, while remaining in a position where the user can see all portals? Or, should the user carry around some device that shows the portals and only be allowed to fully teleport through portals? If the physical space is much smaller than the space from the portal to the data that the user wishes to interact with, then it would not work to simply teleport the hand. This should be explored more. Another approach could be to provide the user with tools that let them interact at a distance, forgoing the entire discussion, such as the annotation gun in our previous study. This could work well through a portal without teleporting anything through it.

Portal Placement

Another very important aspect is how the user interacts with the portals. More specifically how the user places and manipulates the portals. This is no trivial task in VR. Direct interaction suffers from a lack of precision compared to indirect interaction where numbers can be tweaked in boxes. Here ray casting gives a very strong visual indication of where a user is pointing, but portal placement along the ray needs to be enabled through some input. Placement along the ray could be done in a stepped manner, to increase precision, while the user observes a small preview of the final view through the portal. Another approach could be to directly tie portals to annotation points, allowing the user to move them along a line perpendicular to the surface that the annotation is placed on, or along the surface of a sphere with the annotation point at the center.

7.6 Conclusion

Portals have the potential to be much more than just a single-purpose navigational tool for moving about the VE. Portals are a great match for VR because they act as a direct tangible interface that exists in the environment. The successful adaptation of portals as a direct interaction and navigation interface for VR is however reliant on marrying the expectations they elicit with mechanics that stay consistent across different applications. This is no easy task and will require a lot of research and exploration. Contradictions or ambiguous functionality can result in a frustrated or confused user. If however, the underlying functionality is well defined it becomes possible to use portals for a wide variety of tasks in VR without having to change them in a context-aware way, which could lead to portals becoming an omnipresent addition to our VR toolset.

CHAPTER 8

Conclusion

The work in this thesis has led to several scientific publications. The publications aim to help guide the process of building VR-based visualization tools. They widen the scope of applications that can use VR by enabling the visualization of more complex geometric datasets. The results show that VR is a mature technology and that it, in combination with modern hardware, should be considered more often for visualization. Using VR can lead to discoveries within many scientific fields that rely on visualization.

Setting out to build a VR-based visualization tool for scientific data formed the base of the research carried out in this project. It has resulted in a common thread throughout the project. The obstacles we faced along the way became the questions we addressed in our publications. Starting with a blank slate, but a specific use-case for annotation, posed the simple yet important question of whether precision would be affected by using VR. Building the prototype for annotation revealed that *Unity3D* had trouble working with the full resolution of the seal skulls. Begging the question of what a good starting point for a high-performance VR-based visualization platform would be. A natural extension of this was to explore more hardware-oriented optimization strategies for better performance in VR. Having built a high-performance VR-based visualization platform allowed us to explore interaction more in detail through portals, a very computationally heavy method.

In Paper II, we compared annotation for morphology in VR and on desktop. The comparison was between the state-of-the-art desktop-based solution and a VR-based prototype. We conducted a pilot study with participants that covered the entire spectrum of potential users well, with expert users well-versed in using desktop annotation and VR applications. We found no loss in precision when going from desktop-based to VR-based annotation. We also discovered performance issues when visualizing the seal skulls in VR, forcing us to decimate them in the pilot study to ensure that the VR experience did not cause motion sickness.

We were confronted with the need for high performance because we had to decimate the seal skulls. This led to us exploring the different approaches to building VR applications: Paper I. It was evident that there were many different ways to accomplish this. We addressed this by picking a cross-section of different approaches and comparing them in terms of performance with a bespoke VR-based visualization platform named *Jinsoku*. We found that *Jinsoku* held the most potential because it could leverage new hardware features. It used the *Mesh Shading Pipeline* to cull large parts of the mesh when viewing it up close, resulting in high performance.

In Chapter 5, we continue the work on Jinsoku because we did not manage to take full advantage of its potential in Paper I. We further optimize and implement the necessary features to make Jinsoku performant enough to support further research. Part of this process was published in Paper III, where we explored different clustering methods for *Meshlets*. We found that compact clusters that minimized the cluster boundary resulted in high performance while being simple to implement.

Having used a *Manual Viewpoint Manipulation* interface for both Paper I and Paper II, we experienced the fatigue of navigating the VE by dragging and scaling. In Chapter 7, we explored a different way of navigation. One that could be more beneficial for visualizing geometric data in VR, facilitating navigation and interaction. We built a prototype that used portals, a *Target-Based Travel* interface. Using portals was made possible by the time spent optimizing Jinsoku for performance because portals are computationally heavy when used in VR. We further discussed how a portal-based interaction metaphor covers all the types of interactions we require.

At the beginning of the project, we set out to answer three questions with our work. We have investigated and explored different venues that help answer these questions. Below we sum up our contributions:

- We have shown how different VR-based visualization platforms can be optimized to yield better performance in VR.
- We implemented a bespoke Vulkan engine to tackle the perceived difficulty of working with Vulkan.
- We showed that direct access to state-of-the-art GPU features can be worth the time spend building bespoke solutions.
- We have shown that the need for decimating large geometric datasets in VR is no longer required.
- We have shown that a user working with controllers in VR can be just as precise as a user working with a mouse on a desktop.
- We have explored a portal-based interaction metaphor that works well for understanding geometric data.
- We have presented a novel meshlet clustering algorithm that is both fast and simple to implement.

These contributions underpin the story that larger geometric datasets can comfortably be inspected in VR. That in itself has helped increase the usability of VR. Scientific fields with larger geometric datasets can now, based on our results, consider VR-based visualization as a possible venue to explore.

PAPER I

Tools for Virtual Reality Visualization of Highly Detailed Meshes

Mark Bo Jensen, Egil I. Jacobsen, Jeppe Reval Frisvad, and Jakob Andreas Bærentzen.
2021.

VisGap - The Gap between Visualization Research and Visualization Software. Gillmann, C., Krone, M., Reina, G. & Wischgoll, T. (eds.). Eurographics Association, 8 pages. <https://doi.org/10.2312/visgap.20211088>

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 a 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 [ZAV17]) 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 super-computer) 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 [BSM1], and scanning a 39.3×28 cm² 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 [BSM1]. 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,

© 2021 The Author(s)
Eurographics Proceedings © 2021 The Eurographics Association.

DOI: 10.2312/visgap.20211088



20

M. B. Jensen et al. / Tools for Virtual Reality Visualization of Highly Detailed Meshes

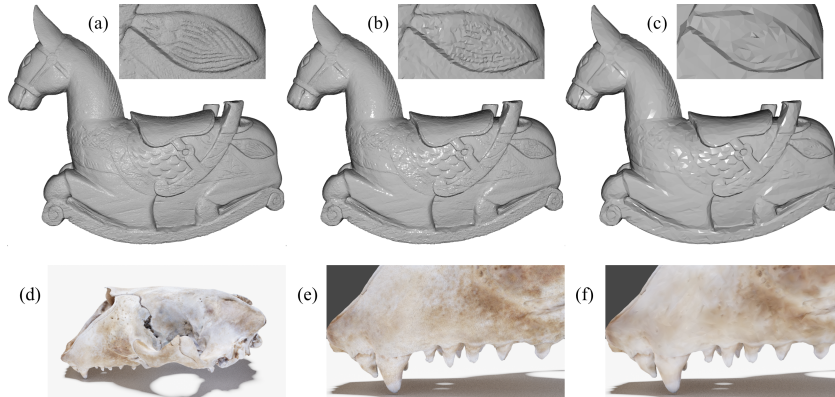


Figure 1: The rocking horse (a) consists of 2.2 million triangles. We reduce it to 10% of the original number of triangles (b) and further to 1% (c). While this fairly large reduction has almost no effect on the silhouette, the fine scale geometric details are clearly impacted by the reduction to 10% 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 cre-

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

ated 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.

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 Visualiza-

tion 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 inspected 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 of 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

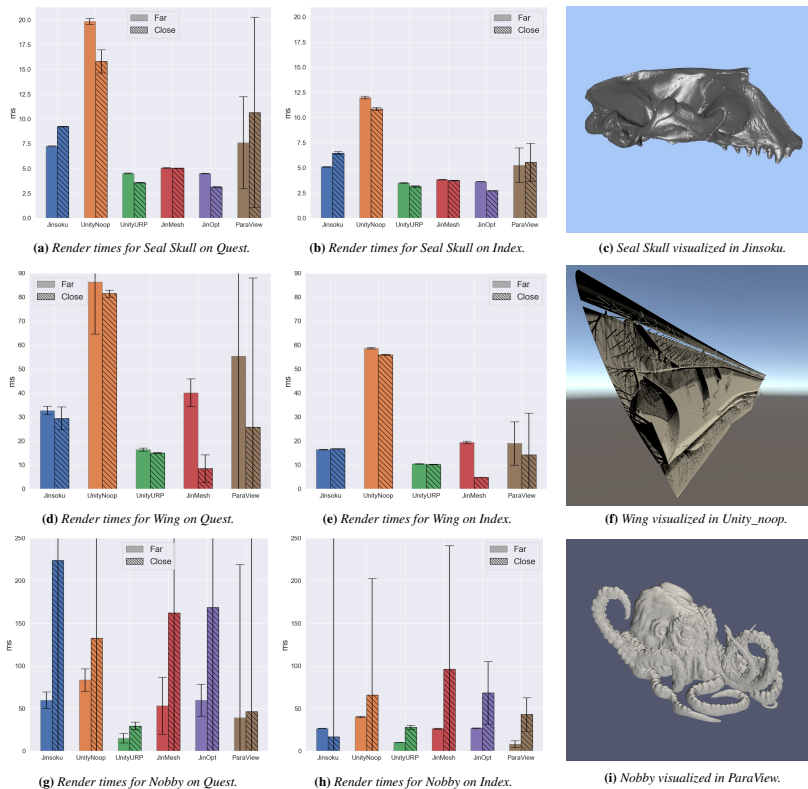
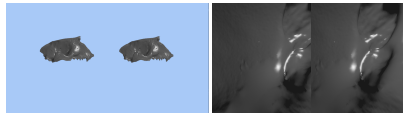


Figure 2: Test results as bar plots (a,b,d,e,g,h). Each bar plot has render time in milliseconds on the vertical axis and shows two test cases for one mesh on one platform. The crosshatched bar is for close-up inspection while the flat-coloured bar is for far-away inspection. The whiskers show the variance 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.

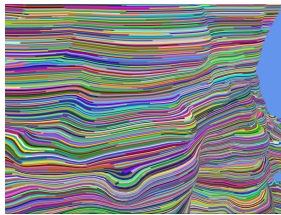


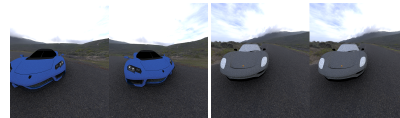
Figure 4: Nobby meshlets.

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).



Δ : 300,603, t: 9.43 ms Δ : 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, Δ is number of triangles and t is render time.

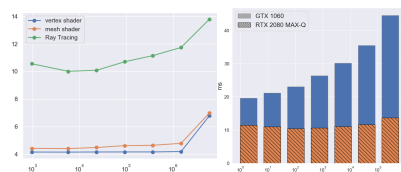


Figure 6: Performance of our GPU VR ray tracer when rendering the Blender monkey [Wik21] with increasing number of subdivisions. We compare with the two shading pipelines in Jinsoku (left) and with a GPU architecture from before RTX (right). The horizontal axes are logarithmic, meaning that the development 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 2c 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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/10.1109/VR.2019.8797978). 2, 3.
- [CC*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](https://doi.org/10.1109/BDVA.2015.7314296). 1.
- [DDC*14] DONALEK, CIRO, DJORGOVSKI, S. G., CIOC, ALEX, WANG, ANWELL, ZHANG, JERRY, LAWLER, ELIZABETH, YE, 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](https://doi.org/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.2006.92](https://doi.org/10.1109/MCG.2006.92).
- [HMS*20] HASSELGREN, J., MUNKBERG, J., SALVI, M., PATNEY, A., and LEFONH, A. "Neural temporal adaptive sampling and denoising". *Computer Graphics Forum* 39.2 (2020), 147–155. doi: [10.1111/cgft.13919](https://doi.org/10.1111/cgft.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](https://doi.org/10.1145/1128923.1128948). 1.
- [KHBW20] KOCH, DANIEL, HECTOR, TOBIAS, BARCZAK, JOSHUA, and WERNES, 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](https://doi.org/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](https://doi.org/10.1007/978-1-4842-4427-2_19). 7.
- [MDJA18] MARTIN, KEN, DEMARLE, DAVID, JHAVERI, SANKHESH, and AYACHTI, 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](https://doi.org/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 (ID '06)*. ACM, 2006, 207–211. doi: [10.1145/1111411.1111448](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](https://doi.org/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](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](https://doi.org/10.1109/TVCG.2021.3060666). 1, 4.
- [ZAVJ17] ZHAO, JINGBO, ALLISON, ROBERT S., VINNIKOV, MARGARITA, and JENNINGS, STION. "Estimating the motion-to-photon latency in head mounted displays". *IEEE Virtual Reality (VR 2017)*. 2017, 313–314. doi: [10.1109/VR.2017.7892302](https://doi.org/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](https://doi.org/10.1080/10095020.2019.1621544). 1.

PAPER II

Using virtual reality for anatomical landmark annotation in geometric morphometrics

Dolores Messer, Michael Atchapero, Mark Bo Jensen, Michelle S. Svendsen, Anders Galatius, Morten T. Olsen, Jeppe Reval Frisvad, Vedrana A. Dahl, Knut Conradsen, Anders B. Dahl, and Jakob Andreas Bærentzen. 2022.

PeerJ. 23 pages. <https://doi.org/10.7717/peerj.12869>



Using virtual reality for anatomical landmark annotation in geometric morphometrics

Dolores Messer^{1,*}, Michael Atchapero^{2,*}, Mark B. Jensen¹, Michelle S. Svendsen³, Anders Galatius⁴, Morten T. Olsen³, Jeppe R. Frisvad¹, Vedrana A. Dahl¹, Knut Conradsen¹, Anders B. Dahl¹ and Andreas Bærentzen¹

¹ Department of Applied Mathematics and Computer Science, Technical University of Denmark, Kgs. Lyngby, Denmark

² Department of Psychology, University of Copenhagen, Copenhagen, Denmark

³ Globe Institute, University of Copenhagen, Copenhagen, Denmark

⁴ Department of Ecoscience, Aarhus University, Roskilde, Denmark

* These authors contributed equally to this work.

ABSTRACT

To study the shape of objects using geometric morphometrics, landmarks are oftentimes collected digitally from a 3D scanned model. The expert may annotate landmarks using software that visualizes the 3D model on a flat screen, and interaction is achieved with a mouse and a keyboard. However, landmark annotation of a 3D model on a 2D display is a tedious process and potentially introduces error due to the perception and interaction limitations of the flat interface. In addition, digital landmark placement can be more time-consuming than direct annotation on the physical object using a tactile digitizer arm. Since virtual reality (VR) is designed to more closely resemble the real world, we present a VR prototype for annotating landmarks on 3D models. We study the impact of VR on annotation performance by comparing our VR prototype to Stratovan Checkpoint, a commonly used commercial desktop software. We use an experimental setup, where four operators placed six landmarks on six grey seal (*Halichoerus grypus*) skulls in six trials for both systems. This enables us to investigate multiple sources of measurement error. We analyse both for the configuration and for single landmarks. Our analysis shows that annotation in VR is a promising alternative to desktop annotation. We find that annotation precision is comparable between the two systems, with VR being significantly more precise for one of the landmarks. We do not find evidence that annotation in VR is faster than on the desktop, but it is accurate.

Subjects Zoology, Data Science

Keywords Virtual morphology, Virtual reality, Stratovan checkpoint, 3D annotation, Geometric morphometrics, Measurement error, *Halichoerus grypus*

INTRODUCTION

Geometric morphometrics is a powerful approach to study shape and is widely used to capture and quantify shape variation of biological objects such as skulls (Rohlf & Marcus, 1993; Bookstein, 1998; Adams, Rohlf & Slice, 2004; Slice, 2005; Mitteroecker & Gunz,

Submitted 27 October 2021

Accepted 10 January 2022

Published 7 February 2022

Corresponding author
Dolores Messer, dolmes@dtu.dk

Academic editor
Yilun Shang

Additional Information and
Declarations can be found on
page 20

DOI 10.7717/peerj.12869

© Copyright
2022 Messer et al.

Distributed under
Creative Commons CC-BY 4.0

OPEN ACCESS

How to cite this article Messer D, Atchapero M, Jensen MB, Svendsen MS, Galatius A, Olsen MT, Frisvad JR, Dahl VA, Conradsen K, Dahl AB, Bærentzen A. 2022. Using virtual reality for anatomical landmark annotation in geometric morphometrics. *PeerJ* 10:e12869
DOI 10.7717/peerj.12869

2009). In geometric morphometrics, shapes can be described by locating points of correspondences in anatomically meaningful landmark positions that are easily identifiable (Bookstein, 1991). An acknowledged and widely used practice is to obtain landmarks directly from a physical specimen through a tactile 3D digitizer arm (Sholts et al., 2011; Waltenberger, Rebay-Salisbury & Mitteroecker, 2021). However, collecting landmarks digitally from a 3D scanned model of the physical specimen is also a viable and widely accepted alternative that has found increased use in recent times (Sholts et al., 2011; Robinson & Terhune, 2017; Bastir et al., 2019; Messer et al., 2021; Waltenberger, Rebay-Salisbury & Mitteroecker, 2021). Annotation of a digital 3D model is typically done using a software tool based on mouse, keyboard, and 2D display (Bastir et al., 2019), and hence placing landmarks can be a tedious task since the perception of shape in a 2D environment is limited as compared with the real world. When a landmark is placed, e.g., a landmark on a 3D tip, small rotations are needed to verify the position, otherwise there might be a significant distance between the actual and the desired landmark coordinates. Annotation of 3D landmarks on a 2D display is more time-consuming than when using a digitizer arm (Messer et al., 2021). On the other hand, having landmarks placed on a 3D scan of a model carries a number of advantages in terms of data sharing and repositioning of landmarks compared to stand-alone landmarks from a 3D digitizer (Waltenberger, Rebay-Salisbury & Mitteroecker, 2021). We argue that virtual reality (VR) provides a closer-to-real-world alternative to desktop annotation that retains the multiple benefits of having the landmarks on a 3D scanned model, including the ability to easily share the digital 3D model, examine it from all angles and accurately place landmarks. The user interaction afforded by the VR head-mounted display allows navigators to move the virtual camera while the controllers can move and rotate the object and serve as an annotation tool as well. All in all, the head-mounted display and controllers exhibit six degrees of freedom (DOF), which map directly to the six DOF needed to intuitively navigate in a 3D environment. This should significantly ease the annotation process and hence make 3D models more useful in biological studies that often require large sample sizes to obtain robust statistics. To investigate the use of VR for digitally annotating landmarks on animal 3D models, we present a prototype VR annotation system and study the impact of VR on annotation performance as compared with a traditional system using 2D display and user interaction by mouse and keyboard.

When comparing a desktop interface to a VR interface, some aspects of VR should be considered. Both latency and tracking noise is higher in VR than with a standard computer mouse. This can degrade performance and precision (Teather et al., 2009). Furthermore, most VR controllers are held in a power grip (clutching the fingers around the object, thereby using the strength of the wrist), as opposed to holding a mouse in a precision grip (holding an object with the fingertips, such as when using a pen, thereby enabling the finer motor skills of the fingers). This makes it more difficult to be precise when annotating objects using most VR controllers (Pham & Stuerzlinger, 2019). Using the mouse on the other hand allows for more controlled and more precise movements, all while allowing the user to let go of the mouse without losing the position on the 3D

model. While the mouse only has three DOF that need to be mapped to the six DOF, it still allows for direct manipulation of either the object or the camera.

The benefits of digital 3D representation of biological specimens (such as skulls) was discovered more than two decades ago ([Recheis et al., 1999](#)). This developed into the more inclusive field of digital or virtual morphology ([Weber, 2015](#)), and the workflows in a virtual morphology lab is now a topic of considerable interest ([Bastir et al., 2019](#)). [Bastir et al. \(2019\)](#) discuss the various databases and the key software tools available for geometric morphometrics. One of the discussed software tools is Landmark editor ([Wiley et al., 2005](#)), which is the predecessor of Stratovan Checkpoint. It seems that none of the software tools in this area employ virtual reality.

VR allows an operator to virtually annotate landmarks in 3D models in a way that resembles real-world annotation of physical specimens ([Bowman, McMahan & Ragan, 2012](#); [Mendes et al., 2019](#)). The directness of this interaction produces a short distance between thought and physical action, making for a simple and straightforward interaction modality. More direct interaction demands a lower cognitive load ([Hutchins, Hollan & Norman, 1985](#)). More cognitive effort can then be invested in understanding and interacting with the data that are presented. [Jang et al. \(2017\)](#) showed that direct manipulation in VR provides a better understanding, and that it benefited students with low spatial ability the most. [Bouaoud et al. \(2020\)](#) found that students gain a better understanding of craniofacial fractures by inspecting 3D models based on CT scans in VR.

In a recent study, [Cai et al. \(2020\)](#) found that using VR to teach about deformities in craniovertebral junctions would improve the ability of the students when afterwards placing landmarks in radiographs of craniovertebral junctions with deformities. This was an improvement as compared with students receiving teaching with physical models. We consider this an indicator that perhaps the act of placing landmarks in digital morphology could be improved too if performed in VR. We follow up on this indication and compare precision and accuracy in placing landmarks on virtual 3D representations of skulls when using our VR system and when using the traditional 2D display and mouse interface.

The possibility of *haptic* feedback is an important aspect of VR input devices. Haptic pertains to the sense of touch, and we can broadly distinguish between two types of haptic feedback. Purely tactile feedback simply means that nerves in your skin are stimulated when you touch something. Standard VR controllers support this through the expedient of vibration, and this is often called *vibrotactile* feedback. The word *kinesthetic* is used about the sense of how limbs of a person's body are positioned in space. Thus, a device which provides force feedback, thereby preventing your hand from going through a virtual surface, is often described as kinesthetic.

Within the area of placing medical landmarks, [Li et al. \(2021\)](#) performed a comparison of the traditional 2D display and mouse interface with two variants of the VR interface, one using standard VR controllers held in a power grip, and one using kinesthetic controllers held in a precision grip. Note that this study differs from ours in another important respect: They show markers which the participants are supposed to target when annotating, whereas we consider the task of deciding where to place the point to be integral

to the annotation task (*i.e.* there is no ground truth). The improvement in marking accuracy was found to be statistically significant when the kinesthetic input device was employed. The task completion time and difficulty of use was however higher for the kinesthetic VR device as compared with the standard vibrotactile controller. The latter was thus easier to use and had as good overall accuracy as the 2D display and mouse interface. Interestingly, the results of [Li et al. \(2021\)](#) suggest that marking accuracy in VR is less affected by marking difficulty than when using a 2D interface. The task performance was thus more stable in VR than when using the traditional 2D tool. This is another motivation behind our test of the performance of placing landmarks in VR as compared with a traditional 2D tool. [Li et al. \(2021\)](#) show that when mouse and VR interfaces are used in a similar way, the haptic feedback helps improving marking accuracy. They do so by having the users interact with the virtual world through an asymmetric bimanual (*i.e.* using both hands) interface, where one hand holds the controller or mouse which is used to both manipulate and annotate the virtual objects, while the other hand can press the spacebar to place the annotation point. Because the same hand is used both for manipulation and marker placement, their interface does not follow the theoretical framework for designing an asymmetric bimanual interface by [Guiard \(1987\)](#). [Kabbash, Buxton & Sellen, 1994](#) show that carefully designed asymmetric bimanual interfaces can improve task performance, while inappropriately designed interfaces lower performance. [Balakrishnan & Kurtenbach \(1999\)](#) show that using an asymmetric bimanual interface designed with the theoretical framework proposed by [Guiard \(1987\)](#) leads to a 20% performance increase over a unimanual interface. Our study employs an asymmetric bimanual interface that follows the theoretical framework by [Guiard \(1987\)](#), and investigates whether combining this interface with regular VR controllers will lead to similar improvements in annotation performance, saving the users from having to acquire special purpose haptic devices.

In geometric morphometrics studies, the presence of measurement error can influence the results of the performed analysis by increasing the level of noise, which can obscure the biological signal, and/or by introducing bias ([Fruciano, 2016](#)). [Fruciano \(2016\)](#) discusses the different sources of measurement error. Several studies in geometric morphometrics quantified measurement error in a situation where landmark data were collected using different devices, and 3D capture modalities (*e.g.* micro CT, surface scanner, photogrammetry), involving several operators ([Robinson & Terhune, 2017](#); [Shearer et al., 2017](#); [Fruciano et al., 2017](#); [Giacomini et al., 2019](#); [Messer et al., 2021](#)). Different systems for annotation of digital 3D models were however not compared in these studies.

To investigate whether annotation in VR is a viable alternative to mouse and keyboard for digital annotation of landmarks on 3D models, we compare our VR prototype to Stratovan Checkpoint (Stratovan Corporation, Davis, CA, USA; <https://www.stratovan.com/products/checkpoint>), a commonly used software for digitally annotating landmarks on 3D models using mouse and keyboard. We note that Stratovan Checkpoint comes with many features, but we focus only on the placing of anatomical landmarks. We study the impact of VR on annotation performance. In a first step, we assess overall and landmark-wise precision and accuracy. Moreover, we investigate different sources of

measurement error (between systems, between and within operators) in an overall and landmark-wise explorative analysis. Finally, we investigate differences in annotation time between systems and operators, and over time.

MATERIALS AND METHODS

VR annotation system

The VR annotation system was developed at the Technical University of Denmark using Unity 2019 (Unity Technologies, San Francisco, CA, USA; <https://unity.com>) and the Oculus Rift hardware released in 2016 (Facebook Technologies, LLC, Menlo Park, CA, USA; <https://www.oculus.com>). Users in the virtual environment are presented with an asymmetric bimanual interface, which adheres to *Guiard (1987)*'s three high-order principles: Assuming a right-handed subject, (1) The system uses a *right-to-left reference* where motion of the right hand finds its spatial reference relative to the left hand. The user manipulates and orients the skull by grabbing it with the left controller. The skull can be scaled by pressing buttons on the left controller. The right controller is then used to place the annotation point, this is done with a ray-gun. The ray-gun shoots out a virtual red laser-line that is intersecting with the surface of the 3D model. Landmark annotation is mimicking the gesture of shooting a gun: The user aims by pointing the controller at the landmark location, and presses the index trigger of the controller. An annotation gun is chosen since it fits well with the power grip that the VR controllers are held in. (2) The actions of the left and right hand are on *asymmetric scales of motion* where the left hand performs large scale movements adjusting the skull and the right hand performs small scale movements to set the annotation point. (3) This workflow means that the left hand moves before the right hand, adhering to the principle of *left-hand precedence*. The VR annotation system supports both right- and left-handed subjects.

3D models are rendered opaque. It is possible to move the viewpoint such that the inside of a 3D model is visible. As opposed to the desktop software Stratovan Checkpoint, the inside of a 3D model is not rendered. Only 3D models are rendered, with no additional information being shown, *i.e.* unlike other systems, we do not show cross sections.

A comprehensive description of the VR annotation system, and a detailed comparison of the VR system to the desktop software Stratovan Checkpoint and a 3D digitizer arm are provided in the [Article S1](#). A demonstration of the VR annotation system, and Stratovan Checkpoint, are shown in [Videos S1](#) and [S2](#), respectively.

Landmark data collection

Our study investigates how precisely, accurately and fast a user can place landmarks using our VR annotation system compared to Stratovan Checkpoint, a mouse and keyboard based desktop system. To carry out this experiment, we scanned, reconstructed, and annotated grey seal skulls.

Sample

The sample consisted of six grey seal (*Halichoerus grypus*) skulls. Of these, five skulls were held by the Natural History Museum of Denmark (NHMD) and originated from the Baltic

Sea population, whereas one skull originated from the western North Atlantic population and was held by the Finnish Museum of Natural History in Helsinki (FMNH) (Table A1). We selected the skulls based on size in order to cover a large span: In our sample, skull length ranges from about 18 to 28 cm. All the selected specimens were intact, and did not have any abnormalities in size, shape or colour variation. We did not include mandibles.

Generation of 3D models

The 3D models of the specimens were generated as previously described in Messer *et al.* (2021). In a first step, the skulls were 3D scanned in four positions using a 3D structured light scanning setup (SeeMaLab (Eiriksson *et al.*, 2016)). On the basis of geometric features, the point clouds from the four positions of a given skull were then globally aligned using the Open3D library (Zhou, Park & Koltun, 2018), followed by non-rigid alignment as suggested by Gawrilowicz & Bærentzen (2019). The final 3D model was reconstructed on the basis of Poisson surface reconstruction (Kazhdan, Bolitho & Hoppe, 2006; Kazhdan & Hoppe, 2013) using the Adaptive Multigrid Solvers software, version 12.00, by Kazhdan (Johns Hopkins University, Baltimore, MA, USA; <https://www.cs.jhu.edu/~misha/Code/PoissonRecon/Version12.00>).

In a last step, we used the Decimate function with a ratio of 0.1 in the Blender software, version 2.91.2, (Blender Institute B.V., Amsterdam, the Netherlands; <https://www.blender.org>) to downsample the final meshes of all specimens, thereby reducing the number of faces to about 1.5 million. The average edge length, which is a measure of 3D model resolution, is between 0.312 mm (smallest skull) and 0.499 mm (largest skull). Original meshes consist of about 15 million faces, and we performed downsampling to ensure that our hardware could render the meshes with a frame rate suitable for VR. By using the same resolution 3D model for both the VR and traditional desktop system, we ensured that observed differences in precision and accuracy were not due to differences in resolution. Figure S1 shows the six final 3D models. A comparison of the original with the downsampled 3D model is illustrated in Fig. S2 using FMNH specimen C7-98.

Annotation of landmarks

For each of the six skulls, Cartesian coordinates of six fixed anatomical landmarks (Fig. 1; Table A2) were recorded by four operators. Each operator applied two different systems to place the landmarks on the reconstructed digital 3D models of the grey seal skulls: (1) Stratovan Checkpoint software, version 2018.08.07, (Stratovan Corporation, Davis, CA, USA; <https://www.stratovan.com/products/checkpoint>), without actively using the simultaneous view of three perpendicular cross-sections, and (2) our own virtual reality tool (Article S1). To assess within-operator error, each operator annotated the same skull six times with both systems. In total, 288 landmark configurations were collected.

Our choice of landmarks is a subset of six out of 31 previously defined anatomical landmarks on grey seal skulls (Messer *et al.*, 2021). We chose the landmarks with indices 2, 3, 11, 18, 24, 28 to have landmarks both of Type I (three structures meet, e.g. intersections of sutures) and Type II (maxima of curvature) (Bookstein, 1991; Brombin & Salmasso,

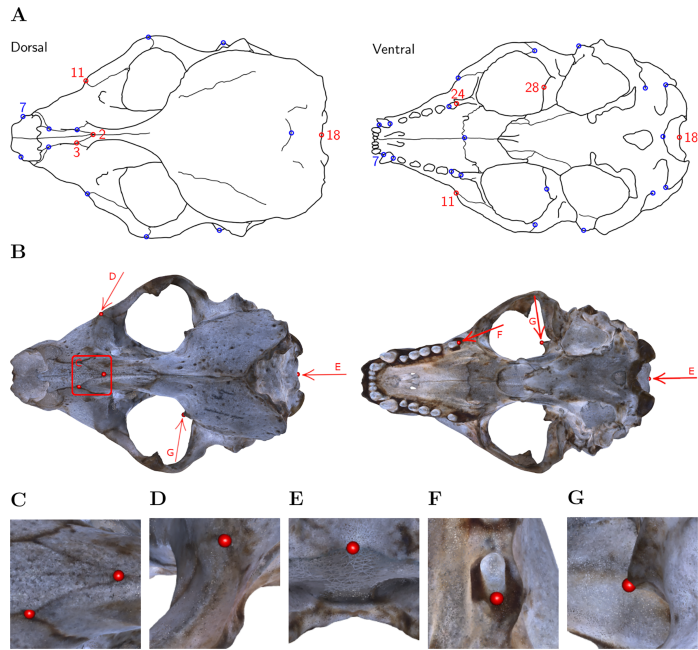


Figure 1 Landmark definition. (A) For our study, we selected the six landmarks marked in red based on the set of 31 anatomical landmarks on grey seal skulls as defined by Messer et al. (2021). Note: Adapted from Messer et al. (2021). Reprinted with permission. (B) The six landmarks placed on the 3D model of NHMD specimen 96. Square/Arrows indicate the camera view for taking images (C)–(G), which show the placed landmarks on the 3D model of NHMD specimen 96 and highlight their respective features. (C) Landmarks 2 and 3: Intersections of sutures; (D) Landmark 11: Apex along a margin; (E) Landmark 18: Medial point of a margin; (F) Landmark 24: Posterior/saddle point; (G) Landmark 28: 3D tip.

Full-size DOI: 10.7717/peerj.12869/fig-1

2013). Moreover, we excluded symmetric landmarks, and landmarks not well defined on all skulls. We selected landmarks located at points that are spread over the whole skull while exhibiting different characteristics.

Experience in placing landmarks, and experience in annotating digital models are two important factors that are likely to influence operator measurement error. Thus, our chosen operators differ from each other with respect to these two relevant factors: Two operators (A and D) were biologists, both of them having experience annotating landmarks using a Microscribe® digitizer, but only operator D had experience placing landmarks on 3D models using Stratovan Checkpoint. The other two operators (B and C) had a background in virtual reality. Operator B had previously annotated landmarks on

one grey seal specimen in both systems and was the developer of the virtual reality annotation tool presented in this study. Operator C was the only one having no experience in placing landmarks and was collecting the landmark data in a test run prior to other data collection. Operators A, B and D spread data collection over 2 to 3 days, whereas operator C annotated all 3D models on the same day.

All operators recorded the landmark configurations in the same pre-defined, randomized order (Table S1), making them switch between systems and specimens. The primary goal was to prevent the operators from memorizing where they previously had placed the landmarks on a particular skull using a specific system. Operator A slightly changed the order by swapping NHMD specimens 223 and 96 using the virtual reality tool, and NHMD specimens 323 and 664 using Stratovan Checkpoint, both for the third replica. Moreover, Operator A accidentally skipped specimen 323 once (virtual reality, third replica) during data collection, and thus annotated this skull 3 months later. In each annotation round, operators sequentially placed the landmarks in the order (2, 3, 11, 18, 24, 28). Operator C, however, collected landmark coordinates in a different order (18, 2, 3, 24, 28, 11).

In addition to collecting landmark coordinates, the annotation time was recorded for each measurement of six landmarks. In case of the virtual reality tool, annotation time was automatically recorded, whereas the operators had to manually record the time using a small stopwatch program that ran in a console window when they annotated landmarks in Stratovan Checkpoint. The operators were instructed to use as much time as they needed for a satisfactory annotation.

Statistical data analysis and outliers

All statistical analyses were conducted in the R software version 3.5.3 (R Core Team, 2020). For geometric morphometric analyses, we used the package *geomorph* (Adams & Otárola-Castillo, 2013).

There were three different types of outliers present in the raw landmark coordinates data: (1) swapped landmarks, (2) obviously wrongly placed landmarks (4–9 mm away from all corresponding replicas; mean Euclidean distance between corresponding replicas was 0.02–0.31 mm)¹, and (3) landmarks localized at two distinct points, for several replicas at each point, or distributed between two distinct points. We put swapped landmarks into the correct order, and replaced the two obviously wrongly placed landmarks by an estimate based on the remaining five replicas using *geomorph*'s function `estimate.missing` (thin-plate spline approach). There were two cases of outliers of type (3): Landmark 18 annotated by operator C on FMNH specimen C7-98, and landmark 28 annotated by operator A on NHMD specimen 42.11, in both cases when using Stratovan Checkpoint as well as the VR annotation system (Fig. S3). Assuming that in these two cases, the operators were in doubt where to clearly place the landmarks, we decided to include outliers of type (3) in all our analyses to not confound the results.

In our design, the repetitions were performed in a randomized order to avoid obvious sources of autocorrelations between repeated measurements on the same landmarks. This

¹ The two outliers of Type (2) were replica 3 of landmark 11 on NHMD specimen 664, and replica 6 of landmark 28 on FMNH specimen C7-98, both placed in VR by operator C.

justifies ignoring the longitudinal aspects of the landmarking by modelling deviations between measurements as random errors.

For a specific specimen, annotation both in Stratovan Checkpoint and VR is based on the same 3D model, and digitization happened in the same local reference frame. Thus, the 48 landmark coordinate sets measured on the same specimen were directly comparable without first having to align them.

We further note that there is no ground truth landmark position in geometric morphometrics. For this reason, we focused on consistent landmark placement in our data analysis.

Annotation time

We sorted the recorded annotation times by system and operator in the order the operators were annotating the skulls (Table S1), and computed trend lines using a linear model including quadratic terms. This allowed us to investigate differences in annotation time between systems and operators, and over time.

Landmark-wise measurement error

We assessed landmark-wise annotation precision by computing the Euclidean distance between single landmark measurements and the corresponding landmark mean. In a first step, we visually compared the precision between systems and operators for each landmark. For that purpose, we computed the landmark means by averaging over replicas. In order to test landmark-wise whether medians across operators within one system were significantly different from each other, we used the `pairwisePercentileTest` function in the R package `rcompanion` (Mangiafico, 2021) to perform pairwise permutation tests based on 10,000 permutations. Additionally, we compared landmark-wise median overall precision between the two systems.

In a second step, to test whether the landmark-wise precision depends on the annotation system, we performed a three-way exploratory Analysis of Variance (ANOVA) with the factors *System*, *Operator* and *Specimen* separately for each landmark. Since we have a crossed data structure, we included all interaction terms. We note that our data are balanced, and that we only considered fixed effects models. Precision was computed from landmark means, which were obtained by averaging over replicas, systems and operators. Since the Euclidean distances between landmark measurements and means had right-skewed distributions, Euclidean distances were log-transformed to approximate a Gaussian distribution.

Since there is no ground truth involved in geometric morphometrics, we assessed accuracy by investigating how closely replicate landmarks were placed in VR compared to the traditional desktop system. For this purpose, we computed landmark means by averaging over replicas, followed by computing Euclidean distances between landmark means obtained from the two systems (for given operators and specimens). This allowed us to visually compare landmark-wise overall accuracy, and investigate differences in accuracy between operators for each landmark. For each landmark, we tested differences in operator median accuracy by performing pairwise permutation tests. We note, however,

that the statistical power of these tests is limited due to the small sample size of six annotated specimens per operator.

Overall measurement error

We assessed overall measurement error similarly as previously described in [Messer et al. \(2021\)](#). In a first step, we computed Procrustes distances between devices, between operators, and within operators (*i.e.* between landmark replica) to investigate overall measurement error. Here, we define Procrustes distance as the sum of distances between corresponding landmarks of two aligned shapes. This allowed us to investigate the extent of differences in the total shape of the same specimen in various ways: measurement by (a) the same operator using a different system (between-system error), (b) different operators using the same system (between-operator error), and (c) the same operator using the same system (within-operator error). Since all measurements from a specific specimen were in the exact same coordinate system, we did not have to align the landmark coordinates prior to the computation of Procrustes distances. Note that we computed Procrustes distances between all possible combinations, which introduces pseudoreplicates. For each error source, we tested differences in median Procrustes distances between operators, systems, or system-and-operator by performing pairwise permutation tests. We also compared median Procrustes distance between the error sources.

In a second step, we ran a Procrustes ANOVA ([Goodall, 1991](#); [Klingenberg & McIntyre, 1998](#); [Klingenberg, Barluenga & Meyer, 2002](#); [Collyer, Sekora & Adams, 2015](#)) to assess the relative amount of measurement error resulting from the different error sources *System*, *Operator*, and *Specimen* simultaneously. With this approach, Procrustes distances among specimens are used to statistically assess the model, and the sum-of-squared Procrustes distances are used as a measure of the sum of squares ([Goodall, 1991](#)). As opposed to a classical ANOVA, which is based on explained covariance matrices, Procrustes ANOVA allowed us to estimate the relative contribution of each factor to total shape variation, which is given by the R-squared value. Prior to Procrustes ANOVA, the landmark configurations had to be aligned to a common frame of reference using a generalized Procrustes analysis (GPA) ([Gower, 1975](#); [Ten Berge, 1977](#); [Goodall, 1991](#)), in which the configurations were scaled to unit centroid size, followed by projection to tangent space. GPA eliminates all geometric information (size, position, and orientation) that is not related to shape. Since we have a crossed data structure, we used the following crossed model for the full Procrustes ANOVA: $Coordinates \sim Specimen \times System \times Operator$.

RESULTS

Annotation time

[Figure 2](#) shows that all operators became faster at annotating a specimen over time, especially during the initial period of data collection. We observe that the two trend lines representing operator C's annotation times are exhibiting a minimum around annotation 20 (VR) and 25 (Stratovan Checkpoint). We point out that this is not an artefact due to

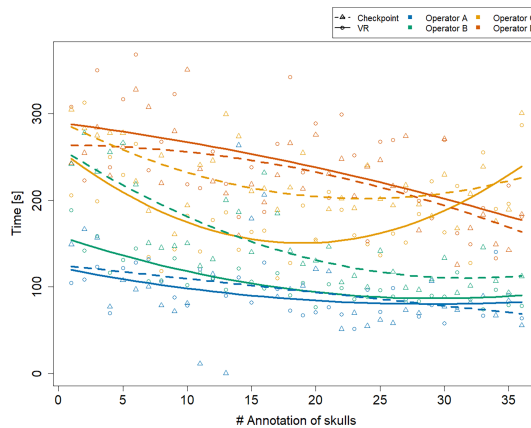


Figure 2 Annotation time by system and operator over time. The x-axis represents the skulls in the order the operators were annotating them. Trend lines suggesting a learning effect were estimated using a linear model including quadratic terms. Operator A's annotation time in Stratovan Checkpoint was not recorded properly for annotation 11 and 13. [Full-size !\[\]\(7c47b229ca7bdb95c18f544ee7ceb332_img.jpg\) DOI: 10.7717/peerj.12869/fig-2](https://doi.org/10.7717/peerj.12869/fig-2)

the quadratic trend, but that operator C's annotation times were actually increasing towards the end of data collection. This might be explained by the fact that C was the only operator collecting all data on the same day. The biologists A and D seemed to be equally fast in both systems over the whole data collection period. Operators B and C, that have a background in virtual reality, started out being much faster in VR, but were approaching VR annotation times in Stratovan Checkpoint over time.

Landmark-wise measurement error

The boxplots of Euclidean distances between single landmark measurements and landmark means (Fig. 3) reveal that on an overall basis, a similar annotation precision was obtained for all six landmarks in VR compared to Stratovan Checkpoint. There were substantial differences between operators: Operator A, for example, was generally significantly less precise than the other operators. This might be explained by the fact that operator A, who is experienced in physical annotation, was not zooming in as much on the 3D models as the other operators during data collection. Moreover, operator B was significantly more precise in VR than other operators. Operators A and C appeared to be more precise in Stratovan Checkpoint compared to VR. Finally, operator D was much more consistent than the other operators, which is demonstrated by a similar obtained precision for all landmarks.

We obtained corresponding results in our ANOVA, which we ran separately for each landmark (Table 1): The factor *System* was only significant in case of landmark 11, but not for the five other landmarks. Computing the means of the Euclidean distance between

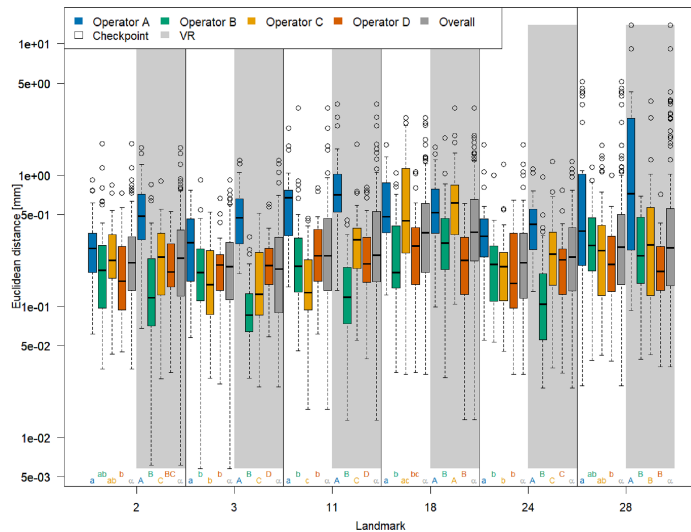


Figure 3 Landmark-wise precision. Precision is measured as the Euclidean distance between single landmark measurements and the operator landmark mean. For each landmark, we compared median overall precision between systems, and median precision between operators within each system for test of significance. For each of these 18 comparison rounds, groups sharing the same letter are not significantly different. The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range. Circles represent outliers. Note that the vertical axis is logarithmic. [Full-size !\[\]\(429fa903b72fda6689f4e2eacafe6305_img.jpg\) DOI: 10.7717/peerj.12869/fig-3](https://doi.org/10.7717/peerj.12869/fig-3)

measurements and mean of landmark 11 separately for each system (VR: 0.57 mm; Stratovan Checkpoint: 0.69 mm) revealed that annotation of landmark 11 was more precise in VR compared to Stratovan Checkpoint. There was no strongly significant interaction between *System* and *Operator*, nor between *System* and *Specimen* for five landmarks (*System* and *Operator*: 2, 3, 11, 18, 24; *System* and *Specimen*: 2, 3, 11, 18, 28). As in Fig. 3, we detected major, significant differences between operators for all landmarks, which were expressed in large *F*-values. This was also true for interaction terms involving the factor *Operator*. Finally, we found that the variability in precision was larger between operators than between specimens, except for landmark 28. This exception can be explained by the fact that landmark 28 was subject to large outliers of type (3), measured by one operator (A) on one specimen, which were not excluded from the analysis. A similar effect is observed in Fig. 3. Examination of Q-Q-plots of the residuals showed that for most of the landmarks, the distribution of the residuals has heavier tails than the Gaussian distribution. However, since balanced ANOVAs are fairly robust to deviations from the Gaussian distribution, we decided not to investigate this further in this explorative study.

Table 1 ANOVA, separately for each landmark. Dependent variable is log-transformed Euclidean distance between single landmark measurements and landmark means. We applied the following crossed structure: System × Operator × Specimen. Residuals reflect landmark replica, and have 240 degrees of freedom.

Variables	Df	F	Pr(>F)	F	Pr(>F)	F	Pr(>F)
				LM 2		LM 3	LM 11
System	1	2.32	0.129	0.54	0.461	9.53	0.002
Operator	3	21.15	0.000	76.02	0.000	57.56	0.000
Specimen	5	2.36	0.041	42.54	0.000	9.95	0.000
System:Operator	3	0.89	0.445	1.34	0.261	3.65	0.013
System:Specimen	5	2.16	0.059	0.43	0.825	3.01	0.012
Operator:Specimen	15	1.73	0.047	4.20	0.000	5.87	0.000
System:Operator:Specimen	15	0.82	0.657	2.45	0.002	3.01	0.000
				LM 18		LM 24	LM 28
System	1	1.73	0.190	0.98	0.324	0.23	0.635
Operator	3	31.91	0.000	51.97	0.000	17.27	0.000
Specimen	5	21.60	0.000	12.32	0.000	183.35	0.000
System:Operator	3	0.42	0.741	2.01	0.113	7.69	0.000
System:Specimen	5	0.91	0.476	4.24	0.001	1.84	0.105
Operator:Specimen	15	2.64	0.001	6.18	0.000	8.01	0.000
System:Operator:Specimen	15	1.59	0.077	3.42	0.000	1.56	0.086

We note that we obtained corresponding *F*-values and significance levels when including outliers of type (2) in our ANOVA (Table S2). However, annotation of landmark 11 seemed to be more precise in Stratovan Checkpoint compared to VR (VR: 0.93 mm; Stratovan Checkpoint: 0.77 mm), which can be explained by the fact that the outlier of type (2) at landmark 11 was placed in VR.

With respect to accuracy (Fig. 4), we found that landmarks were generally placed at similar coordinates in both VR and Stratovan Checkpoint, for all landmarks. For our sample, the Euclidean distance between system means, averaged over specimens and operators, ranged from 0.165 mm (LM 3) to 0.465 mm (LM 28), which is of the same magnitude as the resolution of the 3D models. Similarly to precision, accuracy varied substantially between operators: In particular for landmarks 2, 11, and 28, operator A was less accurate than the other operators. However, this was only significant in case of landmark 2 when comparing the medians (based on the limited sample size of six specimens per operator). We note that the two specimens for which we had outliers of type (3), did not show the lowest accuracies for that particular landmark.

Overall measurement error

The permutation significance test revealed that the median of the Procrustes distances within operators was not significantly different for both systems (Fig. 5), providing evidence that Stratovan Checkpoint and the VR annotation system exhibited similar precision for the group of operators participating in this study. The distribution of Procrustes distances between systems is comparable to that within operators, however, the

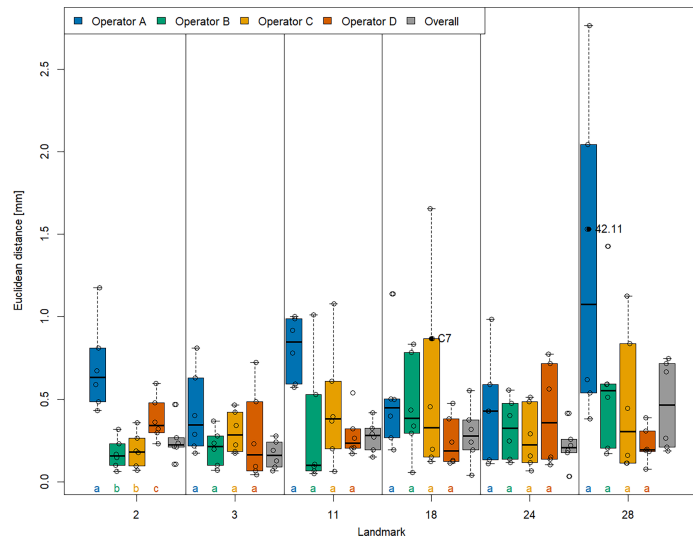


Figure 4 Landmark-wise accuracy. Accuracy is measured as the Euclidean distance between system means. For each landmark, median accuracies of operators sharing the same letter are not significantly different. Jittered data points correspond to the six specimens. The labelled two specimens correspond to the outliers of type (3). The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range. [Full-size !\[\]\(d05e99f54f2116973a3261aa569ffd8a_img.jpg\) DOI: 10.7717/peerj.12869/fig-4](https://doi.org/10.7717/peerj.12869/fig-4)

median of the Procrustes distances between systems is significantly larger than that within operators. In general, Procrustes distances between operators exhibited larger values than those between systems or within operators, with a significant difference in median, which validates the landmark-wise analysis. The median Procrustes distance between operators using the VR annotation system was significantly smaller than when using Stratovan Checkpoint. As in the landmark-wise analysis, we observed significant systematic differences between operators: For operators B and D, who were the only operators having experience in digitally placing landmarks, we found smaller measurement differences between systems than for operators A and C. A similar pattern was observed for within-operator error. Moreover, operator A was more precise in Stratovan Checkpoint, whereas operators B and D were more precise using the VR annotation system.

An analysis of the outliers revealed that they were almost exclusively measured on NHMD specimen 42.11 and FMNH specimen C7-98, where we observed outliers of type (3) (Fig. S3), and on NHMD specimen 664. The largest outliers are connected to measurements by operator A, and measurements in VR. Landmark 28 contributed

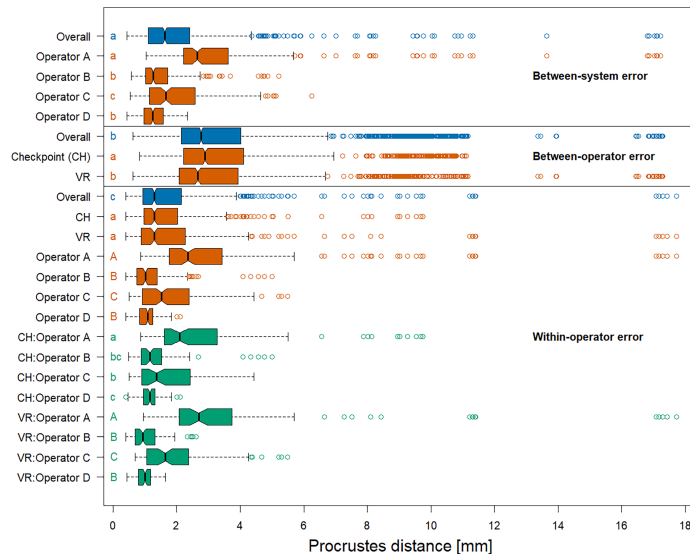


Figure 5 Boxplots of Procrustes distances. Computation of Procrustes distances between systems; between systems for a given operator; between operators; between operators for a given system; within operators; and within operators for a given system/operator/system-and-operator. Within an error source, we compared operator/system/system-and-operator medians with significance tests. Moreover, we compared median overall Procrustes distance between error sources. For each of these six comparison rounds, groups sharing the same letter(s) are not significantly different. The thick bars represent the median, boxes display the interquartile range, and the whiskers extend to 1.5 times the interquartile range. Outliers are represented by circles. The boxplot colours indicate whether a boxplot is based on all Procrustes distances for a given error source (blue), on a subset (red), or on a subset of a subset (green). [Full-size !\[\]\(46cd2931c16a18b659f03caeff1e3155_img.jpg\) DOI: 10.7717/peerj.12869/fig-5](https://doi.org/10.7717/peerj.12869/fig-5)

substantially to the large Procrustes distances. This is in line with the results on landmark-wise annotation precision (Fig. 3).

Running a Procrustes ANOVA on the whole dataset, again, validated the landmark-wise analysis (Table 2). The factor *System* did not seem to contribute much to total shape variation (0.04%), and a comparable result was obtained for the interaction terms involving the factor *System*. The main contributing factor of the two measurement error sources was *Operator*, which accounted for 1.6% of total shape variation. As in the landmark-wise ANOVAs, the interaction between *Operator* and *Specimen* is of importance and explained 1.7% of total shape variation, indicating that the operators were not experienced. As in Fig. 5, the results indicate that between-operator error was larger than between-system error. Most of the total shape variation (94.2%) was explained by

Table 2 Procrustes ANOVA on shape. We applied the following crossed structure: System × Operator × Specimen. Residuals reflect landmark replica. The R-squared values (Rsq) give estimates of the relative contribution of each factor to total shape variation.

Variables	Df	MS	Rsq	F	Pr(>F)
System	1	0.00035	0.0004	3.909	0.013
Operator	3	0.00527	0.0163	59.592	0.001
Specimen	5	0.18275	0.9421	2,065.941	0.001
System:Operator	3	0.00022	0.0007	2.526	0.003
System:Specimen	5	0.00012	0.0006	1.372	0.140
Operator:Specimen	15	0.00107	0.0166	12.114	0.001
System:Operator:Specimen	15	0.00010	0.0015	1.101	0.281
Residuals	240	0.00009	0.0219		

biological variation among grey seal specimens. We note that our findings do not change when including outliers of type (2) in our Procrustes ANOVA (Table S3).

DISCUSSION

We developed a VR-based annotation software to investigate whether VR is a viable alternative to mouse and keyboard for digital annotation of landmarks on 3D models. For this purpose, the VR annotation system was compared to the desktop program Stratovan Checkpoint as a tool for placing landmarks on 3D models of grey seal skulls. The two systems were compared in terms of overall and landmark-wise precision and accuracy, as well as annotation time. We used a carefully chosen setup, where four operators were placing six well-defined anatomical landmarks on six skulls in six trials, which allowed the investigation of multiple sources of measurement error (between systems, within and between operators, and between specimens).

On a desktop computer, an operator is forced to place landmarks through the point-of-view of their display. This is in contrast to VR, where an operator may annotate landmarks from angles different than their point-of-view, since the point-of-view is tracked using the head-mounted display and the controllers can be used to annotate landmarks from an arbitrary direction.

Another benefit of the VR system compared to Stratovan Checkpoint is that it allows the user to scale the specimen. Hence the application is agnostic to specimen size, which is especially helpful in annotating smaller specimens. In Stratovan Checkpoint, the specimen cannot be resized, but the camera can be placed closer. However, when placed too closely, the camera's near-plane will clip the specimen, thereby setting a limit on how closely one can view the specimen. In real-time rendering, two clipping planes are used to delimit the part of the scene that is drawn. The depth buffer has limited precision, and the greater the distance between these two planes, the more imprecise the depth buffer and the greater the risk of incorrectly depth sorted pixels. Unfortunately, the need to move the near plane away from the eye entails that if we move the camera very close to an object, it may be partially or entirely clipped by the near plane.

All in all, our analysis showed that annotation in VR is a promising alternative to desktop annotation. We found that landmark coordinates annotated in VR were close to landmark coordinates annotated in Stratovan Checkpoint. Taking mouse and keyboard annotation as the reference, this implies that landmark annotation in VR is accurate, which is in line with the findings of *Li et al. (2021)*. The accuracy achieved is of the same magnitude as the resolution of the 3D models. Furthermore, when investigating precision, both in landmark-wise ANOVAs, and a Procrustes ANOVA involving all landmarks at once, the factor *System* was not significant, in contrast to the factors *Operator* and *Specimen*. This demonstrated that the measured annotation precision in VR was comparable to mouse and keyboard annotation, whereas precision significantly differed between operators and specimens. These results are in line with previous studies on measurement error which found a larger between-operator compared to between-system or within-operator error (e.g., *Shearer et al., 2017; Robinson & Terhune, 2017; Messer et al., 2021*). However, we obtained a much smaller between-system error when comparing annotation in VR with desktop annotation than *Messer et al. (2021)*, who compared physical and digital landmark placement on grey seal skulls.

Our results revealed that VR was significantly more precise than Stratovan Checkpoint for one landmark. A possible explanation is that the location of this landmark (no 11) on an apex along a margin (*Fig. 1D*) required an operator to observe the approximate location on a skull from various angles to decide on the landmark position. This might have been easier in VR because of the direct mapping between head and camera movement. The operators might also have benefited from the ability to look at landmark 11 independently of the angle used for landmark placement, allowing the operators to view the silhouette of the apex while pointing the annotation gun at the apex, perpendicular to the camera direction.

We found a weak indication that both precision in VR, and precision in general seemed to be positively linked to an operator's experience in placing landmarks on 3D models, and not necessarily to an operator's knowledge of VR or experience in placing landmarks on physical skulls. This result highlights the importance of annotation training on 3D models prior to the digital annotation process in order to obtain a higher precision. However, we have to keep in mind that the group of operators participating in this study is not a representative sample of professional annotators. Some of the operators in this study were not experienced in annotating landmarks. This was reflected in the interaction between *Operator* and *Specimen*, which was significant for all landmark-wise ANOVAs and in the Procrustes ANOVA.

We did not find any evidence that annotation in VR is faster compared to desktop annotation, which is in line with *Li et al. (2021)*. However, we did not include difficult to place landmarks, for which *Li et al. (2021)* found a significantly shorter annotation time in VR than on the desktop. Even though *Li et al. (2021)* conducted a similar experiment, there are three major differences to our setup: (1) In their study, the point the user was supposed to annotate was actually shown during data collection. This is different from the real-life annotation of landmarks we simulate, as the latter includes interpretation of the specimen's anatomy under the respective constraints and advantages of the two

interfaces. (2) They used an in-house 2D annotation tool, whereas our study involves an industry standard 2D annotation software. (3) In our study, the user manipulates the model with the non-dominant hand and annotates with the dominant hand. This is similar to how one adjusts the paper with the non-dominant hand and writes on it with the dominant hand. We compare this to Stratovan Checkpoint's unimanual interface where the mouse is used both for manipulation and annotation and the keyboard is used to change the mode of the mouse.

CONCLUSIONS

To sum up, annotation in VR is a promising approach, and there is potential for further investigation. The current implementation of the VR annotation system is a basic prototype, as opposed to Stratovan Checkpoint, which is a commercial desktop software. Nevertheless, we did not find significant differences in precision between the VR annotation system and Stratovan Checkpoint. For one landmark, annotation in VR was even superior with respect to precision compared to mouse and keyboard annotation. Accuracy of the VR annotation system, which was measured relative to Stratovan Checkpoint, was of the same magnitude as the resolution of the 3D models. In addition, our study is based on a non-representative sample of operators, and did not involve any operator with a background both in biology and VR.

FUTURE WORK

The VR controllers in our study are held in a power grip and do not provide haptic feedback although they can provide vibrotactile feedback (*i.e.* vibrate) when the user touches a surface. *Li et al. (2021)* employ the Geomagic Touch X which, as mentioned, is a precision grip kinesthetic device that does provide haptic feedback. Unfortunately, the haptic feedback comes at the expense of quite limited range since the pen is attached to an articulated arm. On the other hand, there are precision grip controllers which are not haptic devices and hence not attached to an arm. Thus, a future study comparing power grip, precision grip, and the combination of precision grip and haptic feedback would be feasible. Such a study might illuminate whether the precision grip or the haptic feedback is more important.

An interesting extension of the current VR system would be to add a kind of nudging to help the user make small adjustments to the placed landmarks. A further extension could be use of shape information through differential geometry to help guide annotation points toward local extrema.

Improvement of the VR system in terms of rendering performance would be beneficial as it would enable display of the 3D scan in all details (*Jensen et al., 2021*). This would potentially improve the user's precision when placing landmarks. Further improvements of the VR annotation system could be customizable control schemes, camera shortcuts to reduce annotation time, manipulation of the clipping plane to see hidden surfaces, rendering cross sections, and more UIs with information on the current annotation session.

In this study, we focused on clearly defined landmarks. It would be interesting to investigate (as in the work of *Li et al. (2021)*) whether VR might be superior to desktop annotation in the case of landmarks that are more difficult to place. Furthermore, most operators had no experience with one or both types of software. It would be very interesting with a longer term study to clarify the difference between the learning curves associated with the different annotation systems: how quickly does proficiency increase and when does it plateau? Such a study might also help illuminate whether habitually wearing a head mounted display is problematic. There is some fatigue associated with usage of a head mounted display, and this could become either exacerbated or ameliorated with daily use, something we could not address in this study.

APPENDIX

Table A1 List of original 3D models of grey seal (*Halichoerus grypus*) skulls used in this study and their source. NHMD: Natural History Museum of Denmark; FMNH: Finnish Museum of Natural History. Skull length was approximated by the average Euclidean distance between landmarks 7 and 18 (Fig. 1) based on eight repeated measurements by *Messer et al. (2021)*.

Institution	Specimen	Skull length (cm)	Source (MorphoSource identifiers)
NHMD	42.11	19.3	https://doi.org/10.17602/M2/M357658
NHMD	96	23.8	https://doi.org/10.17602/M2/M364247
NHMD	223	21.3	https://doi.org/10.17602/M2/M364279
NHMD	323	18.2	https://doi.org/10.17602/M2/M364263
NHMD	664	21.5	https://doi.org/10.17602/M2/M364287
FMNH	C7-98	28.1	https://doi.org/10.17602/M2/M364293

Table A2 List of the six anatomical landmarks used in this study (L = left, R = right). Four landmarks are of Type I, and two of Type II.

Landmark description	Name	Type
Caudal apex of nasal	2	I
Intersection of maxillofrontal suture and nasal (L)	3	I
Anterior apex of jugal (R)	11	II
Dorsal apex of foramen magnum	18	II
Posterior point of last molar (L)	24	II
Ventral apex of orbital socket (L)	28	II

ACKNOWLEDGEMENTS

The authors would like to thank Daniel Klingberg Johansson from the Natural History Museum of Denmark, and Mia Valtonen from the Institute of Biotechnology at University of Helsinki for giving us access to the grey seal collections. We would further like to thank Florian Gawrilowicz for his assistance with the 3D model creation part, and for providing us with his code for non-rigid point cloud registration.

ADDITIONAL INFORMATION AND DECLARATIONS

Funding

The authors received no funding for this work.

Competing Interests

The authors declare that they have no competing interests.

Author Contributions

- Dolores Messer conceived and designed the experiments, performed the experiments, analyzed the data, prepared figures and/or tables, authored or reviewed drafts of the paper, and approved the final draft.
- Michael Atchapero conceived and designed the experiments, performed the experiments, authored or reviewed drafts of the paper, designed and implemented the VR annotation system, and approved the final draft.
- Mark B. Jensen conceived and designed the experiments, performed the experiments, authored or reviewed drafts of the paper, designed the VR annotation system, and approved the final draft.
- Michelle S. Svendsen performed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Anders Galatius conceived and designed the experiments, performed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Morten T. Olsen conceived and designed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Jeppe R. Frisvad conceived and designed the experiments, authored or reviewed drafts of the paper, designed the VR annotation system, and approved the final draft.
- Vedrana A. Dahl conceived and designed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Knut Conradsen conceived and designed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Anders B. Dahl conceived and designed the experiments, authored or reviewed drafts of the paper, and approved the final draft.
- Andreas Bærentzen conceived and designed the experiments, authored or reviewed drafts of the paper, designed the VR annotation system, and approved the final draft.

Data Availability

The following information was supplied regarding data availability:

The landmark data is available at DTU Data: Messer, Dolores; Atchapero, Michael; Jensen, Mark Bo; Strecker Svendsen, Michelle; Galatius, Anders; Tange Olsen, Morten; et al. (2021): 3D Landmark data on grey seal skull measured on 3D surface models using the Stratovan Checkpoint software, and a VR annotation system. Technical University of Denmark. Dataset. <https://doi.org/10.11583/DTU.14977353.v1>.

The 3D models (not downsampled) are available at MorphoSource (Table A1).

Supplemental Information

Supplemental information for this article can be found online at <http://dx.doi.org/10.7717/peerj.12869#supplemental-information>.

REFERENCES

- Adams DC, Otárola-Castillo E. 2013. Geomorph: an R package for the collection and analysis of geometric morphometric shape data. *Methods in Ecology and Evolution* 4(4):393–399 DOI 10.1111/2041-210X.12035.
- Adams DC, Rohlf FJ, Slice DE. 2004. Geometric morphometrics: ten years of progress following the ‘revolution’. *Italian Journal of Zoology* 71(1):5–16 DOI 10.1080/11250000409356545.
- Balakrishnan R, Kurtenbach G. 1999. Exploring bimanual camera control and object manipulation in 3d graphics interfaces. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '99*, New York: Association for Computing Machinery, 56–62.
- Bastir M, García-Martínez D, Torres-Tamayo N, Palancar CA, Fernández-Pérez FJ, Riesco-López A, Osborne-Márquez P, Ávila M, López-Gallo P. 2019. Workflows in a virtual morphology lab: 3d scanning, measuring, and printing. *Journal of Anthropological Sciences* 97:107–134 DOI 10.4436/JASS.97003.
- Bookstein FL. 1991. *Morphometric tools for landmark data: geometry and biology*. Cambridge: Cambridge University Press.
- Bookstein FL. 1998. A hundred years of morphometrics. *Acta Zoologica Academiae Scientiarum Hungaricae* 44:7–59.
- Bouaoud J, El Beheiry M, Jablon E, Schouman T, Bertolus C, Picard A, Masson J-B, Khonsari RH. 2020. DIVA, a 3D virtual reality platform, improves undergraduate craniofacial trauma education. *Journal of Stomatology, Oral and Maxillofacial Surgery* 122(4):367–371 DOI 10.1016/j.jormas.2020.09.009.
- Bowman DA, McMahan RP, Ragan ED. 2012. Questioning naturalism in 3D user interfaces. *Communications of the ACM* 55(9):78–88 DOI 10.1145/2330667.2330687.
- Brombin C, Salmasso L. 2013. A brief overview on statistical shape analysis. In: *Permutation Tests in Shape Analysis*, New York: Springer, 1–16.
- Cai S, He Y, Cui H, Zhou X, Zhou D, Wang F, Tian Y. 2020. Effectiveness of three-dimensional printed and virtual reality models in learning the morphology of craniovertebral junction deformities: a multicentre, randomised controlled study. *BMJ Open* 10(9):e036853 DOI 10.1136/bmjopen-2020-036853.
- Collyer ML, Sekora DJ, Adams DC. 2015. A method for analysis of phenotypic change for phenotypes described by high-dimensional data. *Heredity* 115(4):357–365 DOI 10.1038/hdy.2014.75.
- Eiriksson ER, Wilm J, Pedersen DB, Aanaes H. 2016. Precision and accuracy parameters in structured light 3-D scanning. *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* W8:7–15 DOI 10.5194/isprs-archives-XL-5-W8-7-2016.
- Fruciano C. 2016. Measurement error in geometric morphometrics. *Development Genes and Evolution* 3(3):139–158 DOI 10.1007/s00427-016-0537-4.
- Fruciano C, Celik MA, Butler K, Dooley T, Weisbecker V, Phillips MJ. 2017. Sharing is caring? Measurement error and the issues arising from combining 3D morphometric datasets. *Ecology and Evolution* 7(17):7034–7046 DOI 10.1002/ece3.3256.

- Gawrilowicz F, Bærentzen JA. 2019.** Optimal, non-rigid alignment for feature-preserving mesh denoising. In: *Proceedings of the International Conference on 3D Vision (3DV)*, Piscataway: IEEE, 415–423.
- Giacomini G, Scaravelli D, Herrel A, Veneziano A, Russo D, Brown RP, Meloro C. 2019.** 3D photogrammetry of bat skulls: perspectives for macro-evolutionary analyses. *Evolutionary Biology* **46**(3):249–259 DOI 10.1007/s11692-019-09478-6.
- Goodall C. 1991.** Procrustes methods in the statistical analysis of shape. *Journal of the Royal Statistical Society. Series B (Methodological)* **53**(2):285–339 DOI 10.1111/j.2517-6161.1991.tb01825.x.
- Gower JC. 1975.** Generalized procrustes analysis. *Psychometrika* **40**(1):33–51 DOI 10.1007/BF02291478.
- Guiard Y. 1987.** Asymmetric division of labor in human skilled bimanual action. *Journal of Motor Behavior* **19**(4):486–517 DOI 10.1080/00222895.1987.10735426.
- Hutchins EL, Hollan JD, Norman DA. 1985.** Direct manipulation interfaces. *Human-Computer Interaction* **1**(4):311–338 DOI 10.1207/s15327051hci0104_2.
- Jang S, Vitale JM, Jyung RW, Black JB. 2017.** Direct manipulation is better than passive viewing for learning anatomy in a three-dimensional virtual reality environment. *Computers & Education* **106**(3):150–165 DOI 10.1016/j.compedu.2016.12.009.
- Jensen MB, Jacobsen EI, Frisvad JR, Bærentzen JA. 2021.** Tools for virtual reality visualization of highly detailed meshes. In: Gillmann C, Krone M, Reina G, Wischgoll T, eds. *VisGap - The Gap between Visualization Research and Visualization Software*. Geneva: The Eurographics Association.
- Kabbash P, Buxton W, Sellen A. 1994.** Two-handed input in a compound task. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '94*, New York: Association for Computing Machinery, 417–423.
- Kazhdan M, Bolitho M, Hoppe H. 2006.** Poisson surface reconstruction. In: *Eurographics Symposium on Geometry Processing, SGP '06*, Geneva: Eurographics Association, 61–70.
- Kazhdan M, Hoppe H. 2013.** Screened poisson surface reconstruction. *ACM Transactions on Graphics* **32**(3):1–13 DOI 10.1145/2487228.2487237.
- Klingenberg CP, Barluenga M, Meyer A. 2002.** Shape analysis of symmetric structures: quantifying variation among individuals and asymmetry. *Evolution* **56**(10):1909–1920 DOI 10.1111/j.0014-3820.2002.tb00117.x.
- Klingenberg CP, McIntyre GS. 1998.** Geometric morphometrics of developmental instability: analyzing patterns of fluctuating asymmetry with procrustes methods. *Evolution* **52**(5):1363–1375 DOI 10.1111/j.1558-5646.1998.tb02018.x.
- Li Z, Kiiveri M, Rantala J, Raisamo R. 2021.** Evaluation of haptic virtual reality user interfaces for medical marking on 3D models. *International Journal of Human-Computer Studies* **147**(1):102561 DOI 10.1016/j.ijhcs.2020.102561.
- Mangiafico S. 2021.** rcompanion: functions to support extension education program evaluation. R package version 2.4.6. Available at <https://cran.r-project.org/web/packages/rcompanion/>.
- Mendes D, Caputo FM, Giachetti A, Ferreira A, Jorge J. 2019.** A survey on 3D virtual object manipulation: from the desktop to immersive virtual environments. *Computer Graphics Forum* **38**(1):21–45 DOI 10.1111/cgf.13390.
- Messer D, Svendsen MS, Galatius A, Olsen MT, Dahl VA, Conradsen K, Dahl AB. 2021.** Measurement error using a SeeMaLab structured light 3D scanner against a Microscribe 3D digitizer. *PeerJ* **9**(4):e11804 DOI 10.7717/peerj.11804.

- Mitteroecker P, Gunz P. 2009. Advances in geometric morphometrics. *Evolutionary Biology* 36(2):235–247 DOI 10.1007/s11692-009-9055-x.
- Pham D-M, Stuerzlinger W. 2019. Is the pen mightier than the controller? A comparison of input devices for selection in virtual and augmented reality. In: *25th ACM Symposium on Virtual Reality Software and Technology, VRST '19*, New York: ACM.
- R Core Team. 2020. R: a language and environment for statistical computing. Vienna: The R Foundation for Statistical Computing. Available at <http://www.R-project.org/>.
- Recheis W, Weber GW, Schäfer K, Knapp R, Seidler H, zur Nedden D. 1999. Virtual reality and anthropology. *European Journal of Radiology* 31(2):88–96 DOI 10.1016/S0720-048X(99)00089-3.
- Robinson C, Terhune CE. 2017. Error in geometric morphometric data collection: combining data from multiple sources. *American Journal of Physical Anthropology* 164(1):62–75 DOI 10.1002/ajpa.23257.
- Rohlf F, Marcus L. 1993. A revolution in morphometrics. *Trends in Ecology & Evolution* 8:129–132 DOI 10.1016/0169-5347(93)90024-J.
- Shearer BM, Cooke SB, Halenar LB, Reber SL, Plummer JE, Delson E, Tallman M. 2017. Evaluating causes of error in landmark-based data collection using scanners. *PLOS ONE* 12(11):1–37 DOI 10.1371/journal.pone.0187452.
- Sholts SB, Flores L, Walker PL, Wärmländer SKTS. 2011. Comparison of coordinate measurement precision of different landmark types on human crania using a 3D laser scanner and a 3D digitiser: implications for applications of digital morphometrics. *International Journal of Osteoarchaeology* 21(5):535–543 DOI 10.1002/oa.1156.
- Slice DE. 2005. Modern morphometrics. In: *Modern Morphometrics in Physical Anthropology*, Boston: Springer US, 1–45.
- Teather R, Pavlovych A, Stuerzlinger W, MacKenzie I. 2009. Effects of tracking technology, latency, and spatial jitter on object movement. In: *Symposium on 3D User Interfaces (3DUI 2009)*, Piscataway: IEEE, 43–50.
- Ten Berge JMF. 1977. Orthogonal procrustes rotation for two or more matrices. *Psychometrika* 42:267–276 DOI 10.1007/BF02294053.
- Waltenberger L, Rebay-Salisbury K, Mitteroecker P. 2021. Three-dimensional surface scanning methods in osteology: a topographical and geometric morphometric comparison. *American Journal of Physical Anthropology* 174(4):846–858 DOI 10.1002/ajpa.24204.
- Weber GW. 2015. Virtual anthropology. *American Journal of Physical Anthropology* 156(Part 2):22–42 DOI 10.1002/ajpa.22658.
- Wiley DF, Amenta N, Alcantara DA, Ghosh D, Kil YJ, Delson E, Harcourt-Smith W, Rohlf FJ, St. John K, Hamann B. 2005. Evolutionary morphing. In: *Visualization Conference (VIS 05)*, Piscataway: IEEE, 431–432.
- Zhou Q-Y, Park J, Koltun V. 2018. Open3D: a modern library for 3D data processing. *ArXiv*. Available at [arXiv:1801.09847v1](https://arxiv.org/abs/1801.09847v1).

PAPER III

Efficient Rendering of Large-Scale Geometric Data using Meshlets

Mark Bo Jensen, Jeppe Reval Frisvad, and Jakob Andreas Bærentzen. 2022.

In review for the Journal of Computer Graphics Tools. 21 pages.

Submitted to the *Journal of Computer Graphics Techniques*

September 27, 2022

Efficient Rendering of Large-Scale Geometric Data Using Meshlets

Mark Bo Jensen
Technical University of Denmark

Jeppe Revall Frisvad
Technical University of Denmark

J. Andreas Bærentzen
Technical University of Denmark

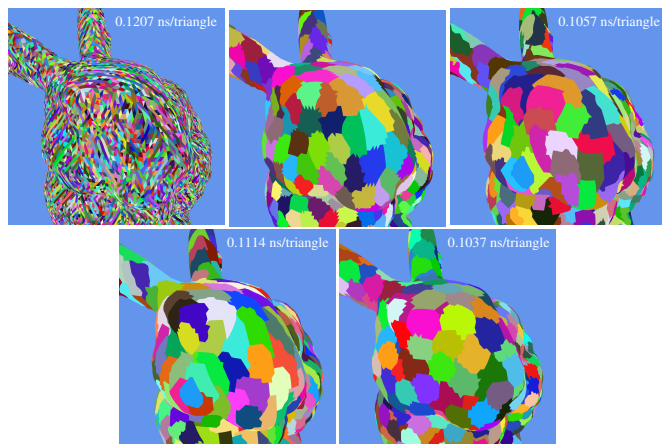


Figure 1. Different methods for organizing the triangles of the Stanford Bunny into meshlets. Each colored patch is a meshlet. From top left to bottom right: NVIDIA [Kubisch 2018b], k -medoids [Kaufman and Rousseeuw 1990], greedy (ours), bounding sphere (ours), Kapoulkine [2017]. We describe the details of the methods in Section 3. Each image shows the render time in nanoseconds per triangle. The time is based on a linear regression fitted to the render time of six meshes as a function of their triangle count. Because k -medoids has too few datapoints we omit its time.

Abstract

Mesh shaders were recently introduced for faster rendering of triangle meshes. Instead of pushing each individual triangle through the rasterization pipeline, we can create triangle clusters called meshlets and perform per-cluster culling operations. This is a great opportunity to efficiently render very large meshes. However, the performance of mesh shaders depends on how we create the meshlets. We test rendering performance after use of different methods for organizing triangle meshes into meshlets. To measure the performance of a method, we render meshes of different complexity from many randomly selected views and measure the render time per triangle. Based on our findings, we suggest guidelines for creation of meshlets. Using our guidelines, we propose two simple methods for generating meshlets with good rendering performance. Our objective is to make it easier for the graphics practitioner to organize a triangle mesh into high performance meshlets.

1. Introduction

Rasterization is fast and highly parallelized on the graphics processing unit (GPU). In extended reality (xR) applications, where too low a frame rate breaks the immersion and potentially causes motion sickness [Rebenitsch and Owen 2016], rasterization is the method of choice. Rasterization is however triangle bound, which means that every triangle must be processed for every frame. This can be prohibitively expensive if we want to visualize massive triangle meshes in xR applications. On the other hand, it is especially in xR applications that we need massive triangle meshes, because the user is free to closely inspect the geometry from arbitrary points of view.

To facilitate a higher triangle throughput, which helps uphold high frame rates even for massive meshes, the rasterization pipeline was recently modified to enable clustering of triangles into meshlets [Kubisch 2018a; Kubisch 2020]. Meshlets improve performance by enabling us to process and cull geometry at a coarser level of granularity than triangles [Jensen et al. 2021]. This relaxes the triangle boundedness, because the pipeline no longer needs to process all the triangles that are submitted to it. This modified pipeline is called the mesh shading pipeline.

Mesh shading is now directly exposed in Vulkan, DirectX 12, and OpenGL [Kubisch 2018a]. This gives rise to the question of how to best create the meshlets, i.e. the triangle clusters. Some developers, notably Kapoulkine [2017] and NVIDIA [Kubisch 2018b], have released code for organizing triangle meshes into meshlets, but the question of how to form meshlets that deliver good rendering performance has received limited attention. In this paper, we evaluate the rendering performance when using different approaches for organizing triangle meshes into meshlets. Our tests include six different meshes consisting of 70 thousand to 39 million triangles. We evaluate performance by rendering the meshes from many randomly selected views while measuring render time per triangle. To our surprise, we find that meshlet col-

lections produce lower render times when using local and greedy algorithms.

We also conduct a small explorative study into different meshlet descriptors in order to investigate how they affect render times. A meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. Apart from describing different algorithms for forming meshlet collections and reporting their rendering times, our main contribution is to identify the most important metrics to consider when assessing the quality of a meshlet collection.

1.1. Related Work

The GPU was originally introduced as special purpose hardware for triangle rasterization. Over the past few decades, GPUs have evolved into highly efficient and very general architectures for parallel computation [Haines 2006; Dally et al. 2021]. GPUs are discrete cards, which means that all data needs to be sent to the GPU if it is to be processed on the GPU. This can become a bottleneck [Hoppe 1999] when working with large datasets, such as very big triangle meshes. A mitigation strategy for big triangle meshes is to use mesh representations that minimize the data footprint. One such widely used mesh representation is *triangle strips*. Triangle strips minimize the data footprint by feeding triangle strips to the GPU with consecutive triangles sharing an edge. In this way, the next triangle is simply described by processing one more vertex, as the two vertices from the shared edge with the previous triangle have already been processed. An index buffer can be used to represent the triangle strip. This is filled with indices to a vertex buffer and replaces vertex duplication with the less memory consuming duplication of vertex indices.

To organize a mesh into triangle strips, we need a path through the mesh where each triangle is only visited once. This is equivalent to finding a Hamiltonian circle in the dual graph of the mesh, which is an NP-complete problem [Dillencourt 1996]. As a result, greedy approaches for creating triangle strips have been explored instead. Arkin et al. [1996] generate triangle strips by greedily adding triangles with fewest adjacent triangles to the strip. This approach avoids leaving behind isolated triangles (triangle islands). The algorithm is made for a graphics API that predates OpenGL, called Iris GL. Iris GL has a command that makes it possible to change the vertex order of the last processed triangle. That makes it possible to change the direction of a triangle strip. Since OpenGL does not have this command degenerate triangles are added to the triangle strip in order to stitch strips together, at the cost of one extra vertex. Evans et al. [1996] seek to minimize this use of degenerate triangles by generating triangle strips based on a global heuristic that looks for large patches that can easily be converted into large strips.

The *generalized triangle mesh* introduced by Deering [1995] relies on a special purpose hardware accelerated cache called the mesh buffer. This buffer stores vertices through explicit commands. Using a mesh buffer makes it possible to exploit that

vertices are on average connected to six triangles, which is hard to fully utilize with triangle strips [Deering and Nelson 1993]. Chow [1997] introduces an algorithm for converting meshes into generalized triangle meshes.

Hoppe [1999] relies on the post transform and lighting cache (post-T&L cache). The post-T&L cache is part of the vertex shading pipeline. The vertex shading pipeline is the traditional rasterization pipeline which is used to process geometric data and turn it into rasterized images. The post-T&L cache holds the most recently processed vertices that have not yet been converted into primitives. Using this, he optimizes triangle strips by reusing the vertices in the cache as much as possible. Several others have built on this principle to further improve rendering performance [Lin and Yu 2006; Forsyth 2006; Sander et al. 2007]. In 2006, with the introduction of the unified shader model [Lindholm et al. 2008], the GPU became massively parallel, allowing for all shader stages to be run on all the generic processors on the GPU. This led Kerbl et al. [2018] to question whether the post-T&L cache is still a part of the GPU architecture. Based on empirical evidence obtained through vertex shader invocations, they show that modern GPUs turn the index buffer into smaller batches and process these in parallel.

A new rasterization pipeline called the *mesh shading pipeline* was introduced with NVIDIA's Turing architecture [Kubisch 2018a]. This pipeline lets the GPU process small parts of the mesh called meshlets instead of individual triangles. The pipeline no longer has the fixed function batching that Kerbl et al. [2018] found in the vertex shading pipeline. Instead, this is done by the programmer, providing the opportunity to make more informed decisions on how the mesh is batched into meshlets. Since each meshlet is processed in parallel, there is no longer a post-T&L cache to hold processed vertices, instead each processor has a cache of shared memory that all the threads on that processor can access. Since the batching is done before rendering, it does not need to take place again every time a new frame is rendered, removing some overhead. The pipeline expects a local index buffer for each meshlet as an output from the mesh shader stage, so this can either be precomputed or generated in the mesh shader. An optional task shader stage can run before the mesh shader to control culling, tessellation, and other things before it dispatches meshlets. The fragment shader stage is unchanged. Kubisch [2018a] provides an excellent overview of the hardware limits, built-in variables, and recommendations for the mesh shading pipeline.

The mesh shading pipeline has received surprisingly little academic attention. This is arguably because both the vertex and mesh shading pipeline benefit from triangle strips that are arranged spatially to increase vertex locality. This becomes quite apparent knowing that batching takes place on both pipelines, and both the post-T&L cache as well as the processor shared memory can take advantage of vertex reuse. Furthermore, real-time graphics is no longer exclusively about rasterizing triangles

as efficiently as possible since real-time ray tracing and methods based on machine learning have become hardware accelerated and – often in combination – lead to a more diverse set of viable methods for efficient, high quality graphics.

Wihlidal [2016] shows how the graphics pipeline can benefit from clustering of triangles, and compute-based culling of these clusters. Jensen et al. [2021] show that the mesh shading pipeline has great potential for visualizing large geometric datasets, and Unterguggenberger et al. [2021] show how the mesh shading pipeline can be used for dynamic meshes. Mesh shaders work well for rendering large terrain [Santerre et al. 2020], and can be used for continuous level of detail [Englert 2020]. In the gaming industry the mesh shading pipeline has been adopted and is now part of Unreal 5’s virtualized geometry pipeline called Nanite [Karis et al. 2021]. It is also possible to find Github repositories with mesh processing tools for the mesh shading pipeline [Walbourn 2014; Kapoulkine 2017; Lempainen 2020]. Neff et al. [2022] investigate texture atlases to reduce meshlet overdraw. In this paper, we explore different clustering strategies for meshlet generation and distill two key principles that lead to better real-time rendering performance when generating meshlets.

2. Meshlets Descriptors

The buffer setup that we use with the mesh shading pipeline has three buffers, see Figure 2. A local index buffer is divided into one section for each meshlet, and the local indices start from 0 in each section. The indices are all 8-bit because they refer to the local indices within a single meshlet. The hardware limit for vertices in a single meshlet is 256, so 8 bits suffice. The global index buffer is also divided into sections, one for each meshlet. This buffer differs from the traditional index buffer in the sense that index duplication is reduced. If one meshlet uses a vertex several times, the local index that points to the same global index is duplicated instead. The last buffer is simply the vertex buffer, which is the same as the one used for the vertex shading pipeline. With these buffers, all we need is a small descriptor for each meshlet providing information about it for the multiprocessor. NVIDIA suggests keeping the size of the meshlet descriptors to 128 bits which, on their hardware, is equivalent to the minimum amount of data that is fetched on a GPU-side load instruction. The meshlet descriptor is a small structure that keeps track of the meta data surrounding a meshlet. It needs to at least hold offsets into the global and local index buffers, as well as the number of primitives and vertices used in the meshlet. Other than this, the descriptor can also store a bounding box, an average normal for the meshlet, or any other information that the programmer wants to have associated with a meshlet.

The layouts of four different descriptors are in Tables 1 and 2. All descriptors use at most 128 bits. All descriptors pack a bounding box into 48 bits, namely 8 bits for the minimum and maximum coordinate on the x-, y- and z-axis. The bounding box

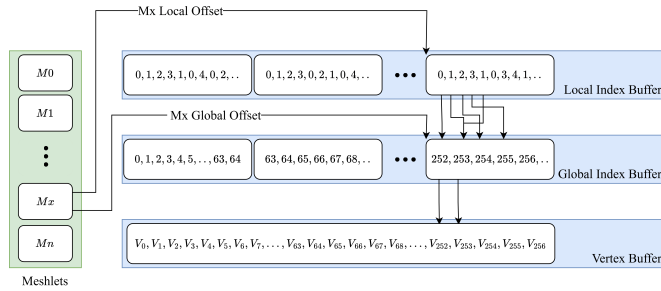


Figure 2. The three buffers used by the GPU when processing meshlets: local index buffer, global index buffer, and vertex buffer. The meshlet descriptor has offsets into these buffers. Note that global indices (re)appear in all meshlets they are used in.

Table 1. The memory layout of two meshlet descriptors proposed by NVIDIA [Kubisch 2018b]. Meshlet descriptors are 128 bit data structures that are used in task and mesh shaders.

NVIDIA descriptor A		NVIDIA descriptor B	
	Bits		Bits
Bounding Box	48	Bounding Box	48
No. Vertices	8	No. Vertices	8
No. Primitives	8	No. Primitives	8
Global idx offset	20	vertexPack	8
Local idx offset	20	Index buffer offset	32
Normal Cone	24	Normal Cone	24

coordinates are relative to the extent of the mesh bounding box. They all use 8 bits for describing the number of primitives and vertices in the meshlet. The normal cone is represented by a normal and an angle packed into 24 bits. The normal and cone angles are mapped into octants based on Cigolle et al. [2014]. All data in a descriptor is packed into four 32-bit unsigned integers. The NVIDIA descriptor A packs the 8 bit cone angle partially into two 32 bit unsigned integers. The 4 upper bits in one and the 4 lower bits in the other. The remaining 3 descriptors pack the 8 bit cone angle together, which saves some unpacking within the mesh shader. The biggest point of divergence between the 4 descriptors lies in how they store the offsets required for the global and local index buffers.

The NVIDIA descriptor A has 20 bits left for indexing into both the local and the global index buffer. This means that meshes that require an offset which is larger than 2^{20} will need to be broken into several draw calls.

Table 2. A descriptor for the task shader stage (left) and another descriptor for the mesh shader stage (right). Use of different descriptors for task and mesh shaders is an alternative to using the same descriptor for both shaders.

Task Shader meshlet descriptor		Mesh Shader meshlet descriptor	
	Bits		Bits
Bounding Box	48	No. Vertices	8
Normal Cone	24	No. Primitives	8
		Global idx offset	32
		Local idx offset	32

The NVIDIA descriptor B takes these same 40 bits and uses 32 of them for offsetting which allows for much larger meshes. The downside of this is that the offsets into the global and local index buffers need to be aligned, as the same offset is used in both buffers. The remaining 8 bits are used to describe how the global indices are packed, i.e. if they are 16 bit or 32 bit numbers. This effectively means that the global indices can be packed into 16 bits for meshlets that only use global indices that are smaller than 2^{16} .

The third descriptor separates the task and meshlet descriptors, this means that it uses 256 bits for each meshlet instead of 128. But it only loads 128 bits per shader stage. By doing that we can get rid of the task shader related data in the mesh shader descriptor and vice-versa. That way we can allow 32 bits for both the global and local index buffer offsets. So here we require no alignment between the buffers. We refer to this as the split descriptor.

Figure 3 shows an alternative buffer setup for a monolithic meshlet descriptor. The monolithic descriptor is also divided into two descriptors, to allow for 2x32 bits offsetting. One offsets into the local index buffer, and instead of using a global index buffer, the second offsets directly into the vertex buffer, which is divided into sections for each meshlet. The trade off here is memory, since some vertices will be duplicated and appear in several sections. On the other hand, no global index buffer is needed. The duplication is required for all vertices that live on the border of a meshlet. So, the four different descriptors all come with different memory footprints as well as some variations in how much GPU side unpacking they require.

Each meshlet can only contain a certain number of vertices and primitives. These numbers dependent on the GPU hardware. In the case of NVIDIA's 2000 RTX series, the hardware limits are 256 vertices and 256 primitives. Lower values can be set as well. NVIDIA suggests using either 32 or 64 vertices and 40, 84 or 126 primitives for each meshlet. In this paper, we use 64 vertices and 126 primitives throughout, which is the same as NVIDIA use in their meshlet sample [Kubisch 2018b].

We use NVIDIA descriptor B when comparing the rendering performance of dif-

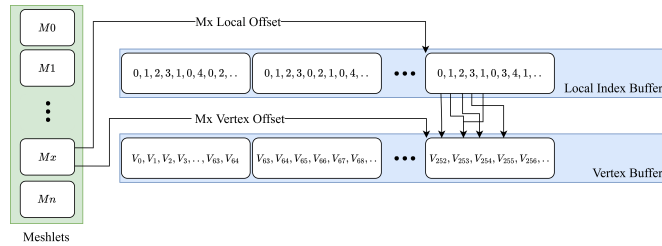


Figure 3. A monolithic version of the buffer setup used by the GPU when processing meshlets. Using only two buffers: local index buffer and vertex buffer. The monolithic meshlet descriptor has offsets into these. Note that vertices (re)appear in all meshlets they are used in.

ferent meshlet generation methods because it allows us to process large meshes with one draw call. For our descriptor comparison, we compare all four descriptors while using the meshlet clustering method with best performance.

3. Meshlet Clustering Methods

The following paragraphs describe the different methods for organizing a mesh into meshlet collections (clusters) that we compare. Figure 1 exemplifies the differences between the meshlets generated by the different methods.

NVIDIA On behalf of *NVIDIA*, Kubisch [2018b] provides an example of organizing a mesh into meshlets. The meshlets are created one at a time by going through the index buffer. New primitives and vertices are added to the current meshlets as long as there is room for more. When it is full, a new meshlet is created. This process is repeated until the algorithm has gone through the entire index buffer. Every time a primitive is added to a meshlet it generates local 8-bit indices for the vertices, or reuses existing local indices if the vertices are already in the meshlet. It de-duplicates the global vertex indices, meaning that the global index of a vertex is only stored once, in each meshlet that uses the vertex, instead of being stored once for each triangle that it is part of. Instead the local indices are stored for each triangle. Because the local index buffer is 8-bit and the global index buffer is 16-or 32-bit this save spaces. The approach has a dependency on the original connectivity of the index buffer, and the resulting number of meshlets, as well as the vertex reuse within the meshlets is highly dependant on the structure of the index buffer. Figure 1 shows an example of the resulting meshlets. The index buffer appears to not be very optimized, which results in a lot meshlets being generated.

Kapoulkine Arseny Kapoulkine [2017] maintains a widely used and popular library called meshOptimizer. The library has several functions that improve, pack, and optimize meshes for better render performance, and it includes a meshlet generation strategy. First, the library creates a data structure based on triangle and vertex adjacency. A centroid and a normal is then calculated for each triangle, and the area of the mesh is also calculated. The area is used to create an expected meshlet area, assuming square flat patches. In addition, a *kd* tree is created from the triangle structure. All this is used to create the meshlets. The *kd* tree is used to pick the starting triangle for a meshlet, and the adjacency structure is then used to look up the nearest triangles. Each triangle gets two ratings: one based on vertex reuse, another based on how much it increases the area of the meshlet. Regarding triangle reuse, triangles that already have vertices in the meshlet get a higher rating. Triangles islands also get higher importance. Should it happen that there is room for more triangles in the meshlet but none available on the border, the algorithm uses the *kd* tree to look up the nearest available triangle. The meshlet generation algorithm allows one to set a weight for the triangle normals, that will make it weigh these more when picking the next triangle for the current meshlet. We set it to 0.0, 0.5 and 1.0, and found that 0.0 produced the best results for the large meshes while the difference between the weights only had a very small impact on the small meshes. Because of this, we report our results with the weight set to 0.0.

Greedy We have developed a greedy algorithm that uses a list of vertices, where each vertex contains information about which triangles it is part of. The algorithm takes the first vertex, and then from that, grows out the triangle cluster until a meshlet is full. If a meshlet hits the vertex max before the primitive max, we look at the border of the meshlet for triangles that already have all vertices in the meshlet, and add these. A new meshlet is then started from a vertex on the border of the meshlet that was just completed, and the process is repeated. If a meshlet runs out of available triangles on its border, we go back to the list and picks the next available one. Because of this, the algorithm is sensitive to the order of the vertex list. We therefore use a heuristic to sort the list before running the algorithm. We find that half the time sorting according to the biggest bounding box axis length gives the best result. In particular, this is the case for the three biggest meshes. We also developed a version using a triangle list instead of a vertex list, but found that the vertex based algorithm always outperformed the triangle based. This is most likely because the meshlet border for vertices is based on all the triangles that the vertices in the meshlet touch, while the border in the triangle version is based on all triangles that share an edge with triangles that are already in the meshlet. This effectively means that the border is "larger" for the vertex version which results in fewer meshlets overall. Moving forward we only report on the vertex based algorithms, and use the heuristic of sorting the vertex list based on the longest

bounding box axis of the mesh, from low to high.

Bounding sphere Our more advanced strategy is similar to the greedy one, except we here grow a bounding sphere around the starting vertex and use an algorithm by Bærentzen and Rotenberg [2021] to add triangles that minimize the radius of this bounding sphere. In addition to striving for a minimal bounding sphere radius, we also (inspired by Kapoulkine) prioritize triangles with vertices already in the meshlet and triangle islands.

k-medoids One way to create clusters of triangles is by turning a mesh into smaller partitions using *k*-medoids [Kaufman and Rousseeuw 1990]. While this is an algorithm normally used for unsupervised learning, to investigate if and how many clusters a dataset might have, we use it to obtain balanced clusters. We chose the *k*-medoids approach because it works along the mesh surface, whereas the more commonly known *k*-means clustering would use a centroid, the cluster mean, to represent a cluster. A centroid detached from the surface easily results in clusters with triangles that are not connected. A medoid on the other hand is an actual datapoint within the cluster that is most suited to represent that cluster. These can be found by minimizing the dissimilarity within a partition. The *k*-medoids method partitions the mesh into *k* clusters and finds the medoid for each of these clusters. The medoid is the triangle with the shortest distance to all other triangles in the cluster. The algorithm runs in two steps after creating an initial clustering of the mesh. First the medoids of all clusters are found. All triangles are then compared to these medoids and assigned to the cluster with the most similar medoid. These two steps are repeated until convergence [Kaufman and Rousseeuw 1990]. The dissimilarity can be expressed through a distance metric between triangles. We run the algorithm on a triangle data structure, where the distance between two triangles is equal to the number of adjacent triangles we have to walk through to get from one to the other. The convergence criterion is to have an average distance close to zero between the new and old cluster centers, meaning that cluster centers moved very little in the last iteration. We start the algorithm with a number of clusters found by dividing the total number of triangles with the maximum number of triangles in a meshlet. After convergence we check if the clusters fit into meshlets. If not, then we add one new cluster and repeat. By only adding one new cluster we minimize the total number of clusters at the cost of longer processing times.

The five methods just mentioned vary quite a bit in implementation complexity. With NVIDIA's algorithm arguably being the simplest to implement, as it just directly works on the index buffer. After this comes the greedy algorithm that uses a triangle and vertex adjacency structure in a sorted list instead of the index buffer, with the bounding sphere version adding a little complexity in terms of a triangle scoring function. Then we have Kapoulkine's which requires both a triangle and vertex adja-

gency structure, a k/d tree, and two scoring functions. Lastly, we have the k -medoids algorithm which not only requires a triangle adjacency structure, but also two iterative steps based on the breadth first algorithm, and to even be applicable it needs to be optimized and parallelized.

4. Experimental Setup

We compare the five different algorithms to see which one performs best, and why. Our hope is that this comparison allows us to distil more general principles for meshlet generation that transcend the specific hardware, and numbers used. To make sure that no bias is introduced into the experimental process we have set up a Vulkan visualization engine which visualizes all the objects from a new random point in space each frame. Our efforts to randomize the view point is to average out the effect of overdraw. By setting the random seed we make sure that all algorithms are tested with the same sequence of view points, we do this for a total of 100.000 frames and record different statistics for each method that will be presented below. The frames are rendered at a resolution of 1280x720 pixels. We perform the analysis on 99.999 of the 100.000 frames. The first frame shows a significantly higher render time, presumably because of some data transfer between the CPU and the GPU, which is not evident for the subsequent 99.999 frames. All experiments were run on a desktop with an Intel Core i9-9900k, 64GB of DDR4-2666 RAM, and one NVIDIA GeForce RTX 2080 Ti Turbo OC with 11GB of GDDR6 RAM. We report our results in average render time per frame in milliseconds, while also exploring different other metrics surrounding the meshlets that impact the render timer. We use five different models for our tests in this paper. The vertex and triangle count of each model can be seen in Figure 4. The Stanford Bunny, Happy Buddha and Asian Dragon are from The Stanford 3D Scanning Repository (<https://graphics.stanford.edu/data/3Dscanrep/>). The Seal Skull has been 3D scanned into a point cloud and digitally reconstructed as a triangle mesh (<https://www.morphosource.org/projects/000355763>). The topology optimized airplane wing [Aage et al. 2017; Aage et al. 2020] is the largest model in our comparisons. The last mesh has been created with PrusaSlicer (<https://www.prusa3d.com/>) using a model called Nobby (<https://www.prusaprinters.org/prints/35338-nobby-octopus-sculpt>). We use the same experimental set up when testing the different meshlet descriptors, using the best performing meshlet generation algorithm.

5. Results

We are interested in finding a good clustering algorithm for meshlet generation. To investigate this, we plot the render times of the different algorithms as a function of triangle count in Figure 5. We see a fairly linear trend. The solid lines show render times without meshlet culling, while the dashed lines include meshlet culling.

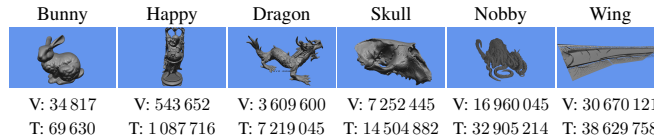


Figure 4. The six meshes used in our experiment and their numbers of vertices (V) and triangles (T).

Table 3. Render times.

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	0.15	0.28	0.87	1.70	4.21	4.68
Kapoulkine	0.13	0.22	0.79	1.33	4.09	4.10
greedy	0.13	0.23	0.76	1.36	4.11	3.74
bounding sphere	0.13	0.22	0.74	1.25	4.15	3.58
<i>k</i> -medoids	0.13	0.23	0.77	N/A	N/A	N/A

Figure 6 shows how many percent of the meshlets are culled on average, each frame. The actual render times can be seen in Table 3. Here it becomes evident that for the two smallest meshes there is not really any difference in performance between the best performing algorithms, but clearly, for the larger methods there is a difference in performance. Given the linear trend we also fit a regression line to each algorithm, and report the resulting slope in Table 4. The slopes are reported in nanoseconds per triangle, with and without culling, and we consider these slopes an overall measure of the performance of the different methods.. The *k*-medoids method is omitted in this table due to the few data points. The smaller the slope is the less an algorithm grows in render time as more triangles are rendered. The bounding sphere algorithm achieves the smallest slope, so extrapolating from our six meshes, it increases the least in render time as the number of triangles grow. Since the difference between the algorithms is evident both with and without culling of meshlets, it means that the clustering within the meshlets themselves also contribute to the difference in render times. When we compare the render times to the implementation complexity of the algorithm, we have NVIDIA's algorithm which is the simplest to implement, but this comes with a performance hit. On the other hand we have Kapoulkine's algorithm which achieved good render times but is rather complicated to implement. Right in the middle we have the greedy algorithm. This has the second smallest slope while also being quite simple to implement.

Each meshlet has a maximum number of vertices and a maximum number of primitives that it can contain. We find that all methods (except *k*-medoids) have a very high average vertex count. For each meshlet collection, we find the average vertex fill

Submitted to the *Journal of Computer Graphics Techniques* September 27, 2022

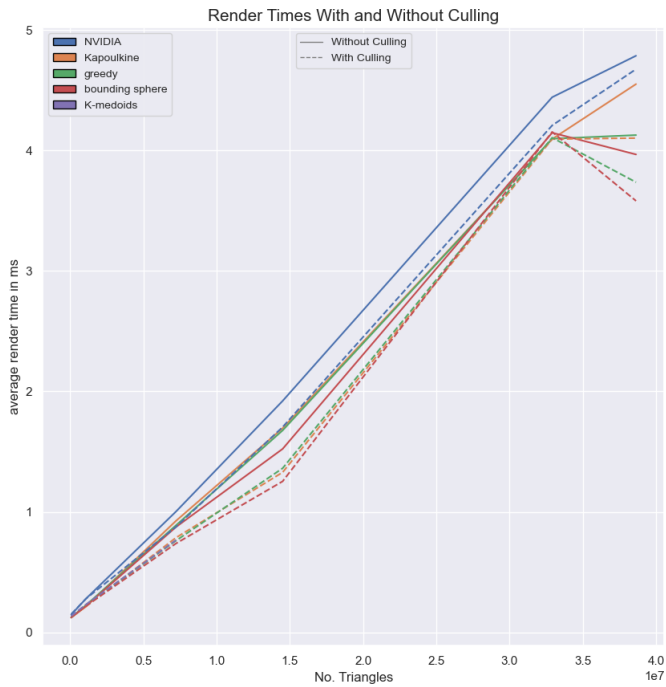


Figure 5. Average render time as a result of triangles based on the six meshes. Render times with meshlet culling are presented with a dashed line and render times without culling are presented with an opaque line.

Table 4. Slope of a linear regression fitted to the six mesh render times based on the four different algorithms with and without culling. The slope shows how much an algorithm increases in render time as more triangles are rendered. The time is given in nanoseconds.

Method	without culling	with culling
NVIDIA	0.1246	0.1207
Kapoulkine	0.1179	0.1114
greedy	0.1109	0.1057
bounding sphere	0.1091	0.1037

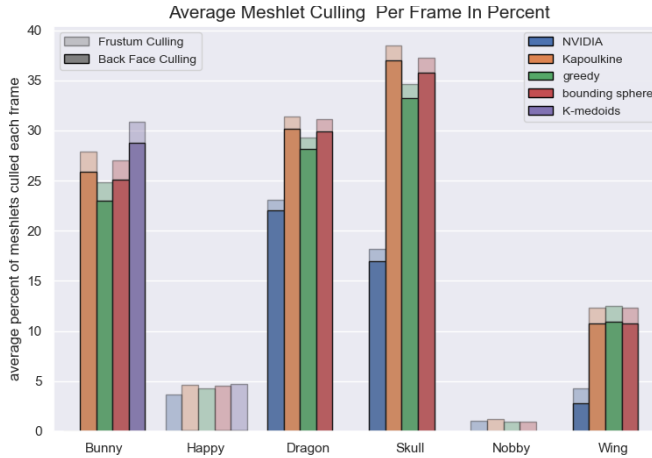


Figure 6. The average percent of meshlets that are culled for each frame when using the five different clustering algorithms. The culled meshlets are divided into two, the back face culled meshlets are represented by the fully opaque bars, while the frustum culled meshlets are represented by the semi-transparent bars.

(ratio of vertices in a meshlet to the maximum number it can hold). All other collections have an average above 0.99 (except for k -medoids with Bunny: 0.812, Happy: 0.770, Dragon: 0.811). With all algorithms achieving close to vertex-complete meshlets, i.e. meshlets that are filled with vertices to the limit, the vertex completeness does not help us explain the differences in render times.

To see why k -medoids generates meshlet collections with a lower average vertex completeness, we compare its distributions to the other algorithms in Figure 7. Since the nature of the k -medoids algorithm is to balance out the clusters we get a distribution of the number of vertices with two fat tails. This means that we will always be below capacity, and when we compare it to NVIDIA's, and especially Kapoulkine's, we see high peaks and only a tail to one side. Kapoulkine's algorithm performs better than both NVIDIA's and the k -medoids, and produces quite few meshlets when compared to the two. The numbers of meshlets produced by the different methods for the different meshes are listed in 6. Since the k -medoids algorithm is trying to distribute the triangles and not the vertices the distribution of the number of triangles show the same two tailed distribution. NVIDIA's and Kapoulkine's distributions are more interesting. Kapoulkine's has a peak at a high number of triangles, and a tail that

Submitted to the *Journal of Computer Graphics Techniques*

September 27, 2022

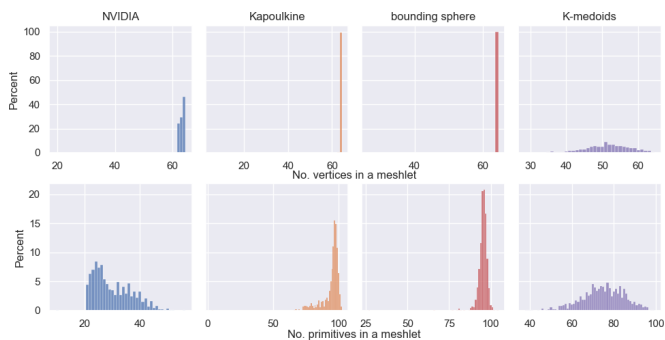


Figure 7. The distribution of the number of vertices and the number of triangles in each meshlet across four meshlet generation algorithms. The top row shows the vertices and bottom row is triangles. The meshlet collections are based on the Stanford Bunny mesh.

falls off towards smaller numbers, while NVIDIA's is opposite. This is most likely because of the index buffer, and how it does not promote locality as well as Kapoulkine's adjacency based method, resulting in less locality and more unique vertices. These results informed us that greedy strategies ensure more vertex- and triangle-complete meshlets.

Since vertex completeness did not help differentiate the algorithms, we instead inspect triangle completeness. Table 5 shows the average primitive fill (ratio of primitives to the maximum number of primitives). Unlike the vertex count, the primitive count varies quite a bit more across the different algorithms and meshes. If we compare this Table 3, we see a correlation between the methods that perform the best and their primitive fill being high (although not as simple as saying that the highest primitive fill yields the best render times). The primitive fill number also explains the variance in the meshlet collection sizes. If we look at NVIDIA's algorithm for instance, it produces more meshlets than the other algorithms. Since each meshlet holds fewer primitives, we need more meshlets to represent the meshes. The k -medoids algorithm does not achieve a high primitive fill for any of its three meshes. Since it fails to produce high vertex fill, it becomes even more difficult to achieve a high primitive fill. NVIDIA's algorithm has the lowest primitive fill, and also performs the worst, which indicates that it is difficult to build meshlets directly from the index buffer.

The NVIDIA and k -medoids algorithms both generate meshlet collections with a somewhat wide distribution of vertices and primitives (Figure 7). To investigate how this impacts the performance of meshlet collections, we sort the meshlets with respect to number of vertices and number of primitives. We only do this for the NVIDIA-

Table 5. The average primitive fill for meshlet collections.

Method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	0.238	0.370	0.459	0.503	0.849	0.488
Kapoulkine	0.747	0.767	0.753	0.756	0.906	0.699
greedy	0.731	0.706	0.739	0.718	0.910	0.723
bounding sphere	0.756	0.744	0.759	0.755	0.911	0.751
k -medoids	0.600	0.568	0.597	N/A	N/A	N/A

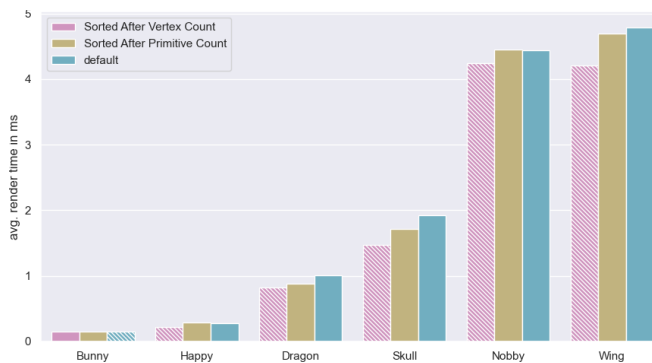


Figure 8. The average render times for the NVIDIA meshlet collections for each mesh as a result of sorting the meshlet list that is send to the GPU. The list is sorted based on number of vertices and primitives. The resulting render times are compared to sending the meshlet list as is. The hatched bar for each mesh show the best performing ordering.

based meshlet collections as the other algorithms generate more uniform meshlets. As seen in Figure 8, the order of the meshlets do play a role. The plot shows the render time when not culling any meshlets and using the NVIDIA descriptor B without index packing. We clearly see that sorting after vertex fill yields the best results. This is most likely due to the fact that vertices need to be loaded and transformed in the mesh shader, whereas primitives are represented by an index list that just requires loading in data. The reason why the render times are affected is that the GPU resources are used better. Meshlets are dispatched in groups to be processed in parallel, and if these groups are done processing at the same time, a new group can be dispatched without idle time. If the meshlets are of varying sizes, some will finish before others and will end up having to wait for the biggest meshlet to finish processing before a new group can be dispatched.

Since cullability increases performance of the meshlet collections, we find it in-

Submitted to the *Journal of Computer Graphics Techniques* September 27, 2022

Table 6. Number of meshlets.

method	Bunny	Happy	Dragon	Skull	Nobby	Wing
NVIDIA	2 321	23 321	124 797	229 043	307 774	628 686
Kapoulkine	740	11 246	76 127	152 261	288 230	438 582
greedy	756	12 231	77 538	160 436	286 956	424 325
bounding sphere	731	11 605	75 457	152 501	286 767	408 302
<i>k</i> -medoids	921	15 210	95 953	N/A	N/A	N/A

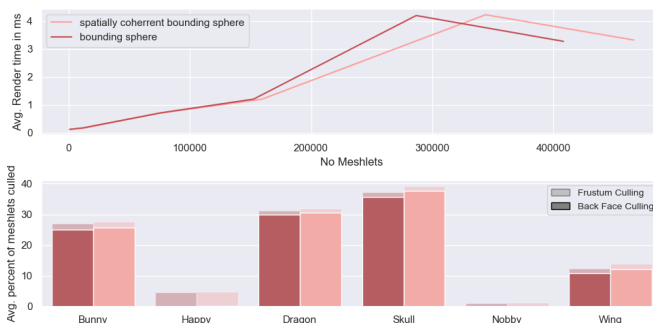


Figure 9. Comparison between the bounding sphere vertex meshlet collections with and without spatially coherent meshlets. The top plot shows the average render time, as a function of the size of the meshlet collections. The bottom plot shows the average percent of meshlets that are being culled per frame for each method.

interesting to explore the importance of the cullability of the meshlets. To test this we tweak our bounding sphere technique for generating meshlets. When a meshlet runs out of new triangles to add from its border, we finish the meshlet instead of going back to the vertex list to look for new candidates. This enforces spatially coherent meshlets. By doing this we create more compact meshlets, making them more likely to be frustum culled. This also reduces the chance of adding a triangle with a normal that deviates too much from the meshlet normal. The increased cullability comes at the cost of a larger meshlet collection. In Figure 9, we see that the more cullable spatially coherent meshlet collections are offset to the right of the normal meshlet collection because they contain more meshlets. For smaller meshes, the spatially coherent meshlet collections show better performance, despite having more meshlets. The increased number of meshlets seems to be offset by the larger amount of culling. The increased culling is however not sufficient to hide the larger loading and processing times for the big meshes. Here, the difference in render times between the two

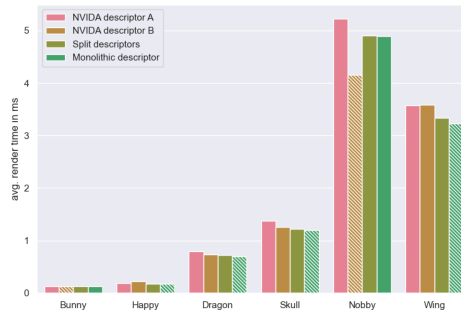


Figure 10. Performance comparison across the six meshes for the four different meshlet descriptors described in Section 2. For each mesh, the hatched bars highlight the descriptor with best performance.

meshlet collections is small.

Meshlet Descriptor Comparison We use our bounding sphere algorithm to test the four different meshlet descriptors described in Section 2. Results are in Figure 10. The type of descriptor that has the best performance varies from mesh to mesh. Only for Nobby we see a really big difference in render times. Here, the NVIDIA pack descriptor outperforms the other descriptors with as much as 1 ms. The Nobby model is a representation of a 3D print, because of this it consists of tubes. These tubes will have normals that point in all directions making it impossible to form meshlets with well defined normal cones, meaning that no or very little back face culling is taking place. Because of this, all visible meshlets are processed which gives an interesting insight into how much the meshlet culling affects performance. The high average render time for the NVIDIA descriptor A is most likely a result of overdraw, because the mesh has almost no cullable meshlets, and is divided into several chunks. NVIDIA descriptor B has some (680) meshlets that can be compressed.

6. Discussion

Most of our experiments show that vertex completeness is important. Exploring the meshlets generated from k -medoids show this the best. The distributions from Figure 7 and render times from Table 3 shows that one should prioritize vertex complete meshlets over balanced meshlets. Our investigation into spatially coherent meshlets show the same, albeit with a weaker signal. Spatially coherent meshlets results in better cullable meshlets at the cost of generating more meshlets. Generating more meshlets means having a bigger distribution of vertices and primitives. The differ-

ences here are small when compared to the k -medoids results because the portion of meshlets with lower vertex completeness is small, but for bigger meshes it starts to affect performance more. More vertex complete meshlets also means more uniform meshlets, and more uniform meshlets reduce render times. We saw this when sorting the NVIDIA meshlet collections in Figure 8.

Inspecting Table 3 in conjunction with Table 5 revealed the correlation between high primitive fill and better performance. It is interesting to explore the interaction between average primitive fill and vertex completeness by inspecting the k -medoids and the NVIDIA meshlet collections. For the Bunny mesh, we see an example where the average primitive fill on the NVIDIA meshlet collection is so low that the high average vertex fill cannot compensate for it. This demonstrates that one should not only optimize around one heuristic but take both into account. For the Happy mesh, the NVIDIA collection performs better than k -medoids, showing that vertex completeness is more important. For the Dragon mesh, the tables have turned and the k -medoids collection, with a better balance between the two, performs best. This interaction tells us that it is important to prioritize both vertex completeness and primitive fill. We also observe that it is hard to have a high primitive fill without having nearly vertex complete meshlets.

Striving for cullable meshlets is the third heuristic. Our experiments show that cullable meshlets can help balance out larger meshlet collections. Figure 9 exemplifies how meshlet collections slightly enlarged to increase cullability can indeed result in better performance. It does however not seem to affect performance as much as vertex completeness or maximizing the primitive fill.

The Skull and Nobby meshes produce some surprising results for some of the meshlet generation strategies. It is surprising that Kapoulkine's algorithm does not perform best on the Skull, as the data show more culling, and less meshlets. Perhaps the difference is that our method builds meshlets along the z-axis of the skull as opposed to from the middle and out, this could affect vertex loading, overdraw, and cache misses on the GPU.

Nobby shows that some meshes will be exceptions to the rule. It will be possible to find meshes where these heuristics and metrics break down. In fact, tuning one aspect of meshlet generation affects all the other aspects. The metrics, and indeed most of the factors we explore in this paper are highly correlated, and this can make it hard to isolate different aspects as they affect each other. Two collections of meshlets might differ in efficiency even if almost all meshlets are packed to capacity in both collections. Because of this, it becomes even more desirable to have an algorithm that is simple to implement. The greedy algorithm proves to be quite useful in practice as it achieves good render times across the meshes while also being simple to implement.

Lastly we conducted a small exploratory experiment which compared different ways of packing the meshlet descriptor data. Interestingly, we find that the mono-

lithic descriptor performs quite well. This is certainly interesting. The monolithic descriptor uses a simpler buffer setup, and by using one descriptor per shader stage, it becomes possible to add more meta data if desired.

7. Conclusion

We find, quite simply, that the more compact the meshlet collections become the better they perform. Meshlets have vertex and primitive limits, in this paper we used the suggested 64 vertices and 126 triangles. Since each triangle requires 3 vertices, the meshlets always hits the vertex limit before the triangle limit. In other words, it is absolutely paramount that a meshlet collection achieves a high average vertex fill. In this way, it becomes possible to also have a high average primitive fill. Because of this, we recommend the following strategy for optimizing meshlet generation: *make the meshlets vertex complete first and then maximize the primitive fill*. The combination of these two will create meshlets with large vertex reuse and locality, while also minimizing the total number of meshlets that are required to represent a mesh. Finally, we of course recommend to *strive for cullable meshlets*, but not at the cost of a too big increase in meshlet collection size. We found that performance rather quickly drops when the meshlet collections grow in size.

We also explored other properties of both the mesh shading pipeline and the meshlet collections. We found that high uniformity in the meshlet collections promotes even workload across processors on the GPU which yields better render times. Different meshlet descriptors do not have the biggest impact on render times, so working with monolithic meshlets could prove to be a good choice for scientific visualization where rendering is done on distributed systems. As an interesting topic for future work, descriptors that require less data unpacking in the mesh shader could yield improved render performance, and since dividing descriptors into two also did not affect performance too much, it could be interesting to explore whether new useful meta data could be added.

Acknowledgements

We would like to thank Rasmus Emil Christensen and Emil Toftegaard Gæde for the initial investigation into k -medoids based clustering of triangle meshes. This research was funded by Advokat Bent Thorbergs Fond (award no. 66.531).

References

- AAGE, N., ANDREASSEN, E., LAZAROV, B. S., AND SIGMUND, O. 2017. Giga-voxel computational morphogenesis for structural design. *Nature* 550, 7674, 84–86. URL: <https://doi.org/10.1038/nature23911>. 11

Submitted to the *Journal of Computer Graphics Techniques* September 27, 2022

- AAGE, N., SIGMUND, O., LAZAROV, B. B., AND ANDREASSEN, E., 2020. TopWingData. Dataset. URL: <https://doi.org/10.11583/dtu.12581615.v1.11>
- ARKIN, E. M., HELD, M., MITCHELL, J. S., AND SKIENA, S. S. 1996. Hamiltonian triangulations for fast rendering. *The Visual Computer* 12, 9, 429–444. URL: <https://doi.org/10.1007/BF01782475.3>
- BÆRENTZEN, A., AND ROTENBERG, E. 2021. Skeletonization via local separators. *ACM Transactions on Graphics* 40, 5, 187:1–187:18. URL: <https://doi.org/10.1145/3459233.10>
- CHOW, M. 1997. Optimized geometry compression for real-time rendering. In *Proceedings of Visualization '97*, 347–354. URL: <https://doi.org/10.1109/VISUAL.1997.663902.4>
- CIGOLLE, Z. H., DONOW, S., EVANGELAKOS, D., MARA, M., MCGUIRE, M., AND MEYER, Q. 2014. A survey of efficient representations for independent unit vectors. *Journal of Computer Graphics Techniques* 3, 2 (April), 1–30. URL: <http://jcggt.org/published/0003/02/01/.6>
- DALLY, W. J., KECKLER, S. W., AND KIRK, D. B. 2021. Evolution of the graphics processing unit (GPU). *IEEE Micro* 41, 6, 42–51. URL: <https://doi.org/10.1109/MM.2021.3113475.3>
- DEERING, M. F., AND NELSON, S. R. 1993. Leo: A system for cost effective 3d shaded graphics. In *SIGGRAPH '93*, ACM, 101–108. URL: <https://doi.org/10.1145/166117.166130.4>
- DEERING, M. 1995. Geometry compression. In *SIGGRAPH '95*, ACM, 13–20. URL: <https://doi.org/10.1145/218380.218391.3>
- DILLENOURT, M. B. 1996. Finding hamiltonian cycles in delaunay triangulations is np-complete. *Discrete Applied Mathematics* 64, 3, 207–217. URL: [https://doi.org/10.1016/0166-218X\(94\)00125-w.3](https://doi.org/10.1016/0166-218X(94)00125-w.3)
- ENGLERT, M. 2020. Using mesh shaders for continuous level-of-detail terrain rendering. In *ACM SIGGRAPH 2020 Talks*, ACM, 44:1–44:2. URL: <https://doi.org/10.1145/3388767.3407391.5>
- EVANS, F., SKIENA, S., AND VARSHNEY, A. 1996. Optimizing triangle strips for fast rendering. In *Proceedings of Seventh Annual IEEE Visualization '96*, 319–326. URL: <https://doi.org/10.1109/VISUAL.1996.568125.3>
- FORSYTH, T., 2006. Linear-speed vertex cache optimisation, September. Accessed: 2022-04-16. URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html.4
- HAINES, E. 2006. An introductory tour of interactive rendering. *IEEE Computer Graphics and Applications* 26, 1, 76–87. URL: <https://doi.org/10.1109/MCG.2006.9.3>
- HOPPE, H. 1999. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99*, ACM/Addison-Wesley, 269–276. URL: <https://doi.org/10.1145/311535.311565.3.4>

- JENSEN, M. B., JACOBSEN, E. I., FRISVAD, J. R., AND BÆRENTZEN, J. A. 2021. Tools for virtual reality visualization of highly detailed meshes. In *VisGap - The Gap between Visualization Research and Visualization Software*, The Eurographics Association. URL: <https://doi.org/10.2312/visgap.20211088>. 2, 5
- KAPOULKINE, A., 2017. meshoptimizer. Accessed: 2022-04-16. URL: <https://github.com/zeux/meshoptimizer>. 1, 2, 5, 9
- KARIS, B., STUBBE, R., AND WIHLIDAL, G. 2021. A deep dive into nanite virtualized geometry. In *Advances in Real-time Rendering in Games: Part I*, N. Tatarchuk, Ed., ACM SIGGRAPH 2021 Courses. Accessed: 2022-04-16. URL: <https://advances.realtimerendering.com/s2021/index.html>. 5
- KAUFMAN, L., AND ROUSSEEUW, P. J. 1990. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons. URL: <https://doi.org/10.1002/9780470316801>. 1, 10
- KERBL, B., KENZEL, M., IVANCHENKO, E., SCHMALSTIEG, D., AND STEINBERGER, M. 2018. Revisiting the vertex cache: Understanding and optimizing vertex processing on the modern gpu. *Proceedings of the ACM on Computer Graphics and Interactive Techniques I*, 2 (August), 29:1–29:16. URL: <https://doi.org/10.1145/3233302>. 4
- KUBISCH, C., 2018. Introduction to turing mesh shaders. NVIDIA Developer Technical Blog, September. URL: <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/>. 2, 4
- KUBISCH, C., 2018. Vulkan & OpenGL CAD mesh shader sample. Accessed: 2022-04-16. URL: https://github.com/nvpro-samples/gl_vk_meshlet_cadscene. 1, 2, 6, 7, 8
- KUBISCH, C., 2020. Using mesh shaders for professional graphics. NVIDIA Developer Technical Blog, December. URL: <https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/>. 2
- LEMPIAINEN, J., 2020. Meshlete: Chop 3D objects to meshlets. Accessed: 2022-04-16. URL: <https://github.com/JarkkoPFC/meshlete>. 5
- LIN, G., AND YU, T.-Y. 2006. An improved vertex caching scheme for 3D mesh rendering. *IEEE Transactions on Visualization and Computer Graphics* 12, 4, 640–648. URL: <https://doi.org/10.1109/TVCG.2006.59>. 4
- LINDHOLM, E., NICKOLLS, J., OBERMAN, S., AND MONTRYM, J. 2008. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro* 28, 2, 39–55. URL: <https://doi.org/10.1109/MM.2008.31>. 4
- NEFF, T., MUELLER, J. H., STEINBERGER, M., AND SCHMALSTIEG, D. 2022. Meshlets and how to shade them: A study on texture-space shading. *Computer Graphics Forum* 41, 2, 277–287. URL: <https://doi.org/10.1111/cgf.14474>. 5
- REBENITSCH, L., AND OWEN, C. 2016. Review on cybersickness in applications and visual displays. *Virtual Reality* 20, 2, 101–125. URL: <https://doi.org/10.1007/s10055-016-0285-9>. 2

Submitted to the *Journal of Computer Graphics Techniques* September 27, 2022

SANDER, P. V., NEHAB, D., AND BARCZAK, J. 2007. Fast triangle reordering for vertex locality and reduced overdraw. *ACM Transactions on Graphics* 26, 3, 89:1–89:10. URL: <https://doi.org/10.1145/1275808.1276489>. 4

SANTERRE, B., ABE, M., AND WATANABE, T. 2020. Improving GPU real-time wide terrain tessellation using the new mesh shader pipeline. In *Nicograph International (NicoInt 2020)*, 86–89. URL: <https://doi.org/10.1109/NicoInt50878.2020.00025>. 5

UNTERGUGGENBERGER, J., KERBL, B., PERNSTEINER, J., AND WIMMER, M. 2021. Conservative meshlet bounds for robust culling of skinned meshes. *Computer Graphics Forum* 40, 7, 57–69. URL: <https://doi.org/10.1111/cgf.14401>. 5

WALBOURN, C., 2014. DirectXMesh geometry processing library. Accessed: 2022-04-16. URL: <https://github.com/microsoft/DirectXMesh>. 5

WIHLIDAL, G., 2016. Optimizing the graphics pipeline with compute. Game Developer Conference 2016. Accessed: 2022-04-16. URL: <https://www.gdcvault.com/play/1023463/Optimizing-the-Graphics-Pipeline-With>. 5

Author Contact Information

Mark Bo Jensen	Jeppe Revall Frisvad	J. Andreas Barentzen
Technical University of Denmark	Technical University of Denmark	Technical University of Denmark
Richard Petersens Plads 324, 180	Richard Petersens Plads 324, 160	Richard Petersens Plads 324, 160
Lyngby, DK-2800, Denmark	Lyngby, DK-2800, Denmark	Lyngby, DK-2800, Denmark
mboje@dtu.dk	jerf@dtu.dk	janba@dtu.dk
	https://people.compute.dtu.dk/jerf/	https://people.compute.dtu.dk/janba/

M. B. Jensen et al., Efficient Rendering of Large-Scale Geometric Data using Meshlets, *Journal of Computer Graphics Techniques (JCGT)*, vol. ?, no. ?, 1–1, ????

Received: September 27, 2022
 Recommended: ???-?-?? Corresponding Editor: ??? ???
 Published: ???-?-?? Editor-in-Chief: Marc Olano

© ??? M. B. Jensen et al. (the Authors).
 The Authors provide this document (the Work) under the Creative Commons CC BY-ND 3.0 license available online at <http://creativecommons.org/licenses/by-nd/3.0/>. The Authors further grant permission for reuse of images and text from the first page of the Work, provided that the reuse is for the purpose of promoting and/or summarizing the Work in scholarly venues and that any reuse is accompanied by a scientific citation to the Work.

Submitted to the *Journal of Computer Graphics Techniques*

September 27, 2022



APPENDIX **A**

Different appendix

Bibliography

Bibliography

- Aage, Niels (2022). *Personal communications*.
- Aage, Niels, Erik Andreassen, Boyan S. Lazarov and Ole Sigmund (2017). “Giga-voxel computational morphogenesis for structural design”. In: *Nature* 550.7674, pp. 84–86. DOI: 10.1038/nature23911.
- Aage, Niels, Ole Sigmund, Boyan B. Lazarov and Erik Andreassen (2020). *TopWing-Data*. Dataset. Technical University of Denmark. DOI: 10.11583/dtu.12581615.v1.
- Adams, Dean C. and Erik Otárola-Castillo (2013). “Geomorph: An R package for the collection and analysis of geometric morphometric shape data”. In: *Methods in Ecology and Evolution* 4.4, pp. 393–399. DOI: <https://doi.org/10.1111/2041-210X.12035>.
- Adams, Dean C., F. James Rohlf and Dennis E. Slice (2004). “Geometric morphometrics: Ten years of progress following the ‘revolution’”. In: *Italian Journal of Zoology* 71.1, pp. 5–16. DOI: 10.1080/11250000409356545.
- Ahrens, James, Berk Geveci and Charles Law (2005). “ParaView: An end-user tool for large data visualization”. In: *The Visualization Handbook* 717–731. DOI: 10.1016/B978-012387582-2/50038-1.
- Akenine-Möller, Tomas, Eric Haines and Naty Hoffman (2018). *Real-Time Rendering, Fourth Edition*. 4th. USA: A. K. Peters, Ltd. ISBN: 0134997832.
- Åkesson, Karl-Petter and Kristian Simsarian (1999). “Reality Portals”. In: *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*. VRST '99. London, United Kingdom: Association for Computing Machinery, pp. 11–18. ISBN: 1581131410. DOI: 10.1145/323663.323665.
- Alabi, Oluwafemi S., Xunlei Wu, Jonathan M. Harter, Madhura Phadke, Lifford Pinto, Hannah Petersen, Steffen Bass, Michael Keifer, Sharon Zhong, Chris Healey and Russell M. Taylor II (2012). “Comparative visualization of ensembles using ensemble surface slicing”. In: *Visualization and Data Analysis 2012*. Ed. by Pak Chung Wong, David L. Kao, Ming C. Hao, Chaomei Chen, Robert Kosara, Mark A. Livingston, Jinah Park and Ian Roberts. Vol. 8294. International Society for Optics and Photonics. SPIE, 82940U. DOI: 10.1117/12.908288.
- Amenta, Nina, Marshall Bern and Manolis Kamvysselis (1998). “A new Voronoi-based surface reconstruction algorithm”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 415–421.

- Arkin, Esther M, Martin Held, Joseph SB Mitchell and Steven S Skiena (1996). “Hamiltonian triangulations for fast rendering”. In: *The Visual Computer* 12.9, pp. 429–444.
URL: <https://doi.org/10.1007/BF01782475>.
- Attene, M., S. Katz, M. Mortara, G. Patane, M. Spagnuolo and A. Tal (2006). “Mesh Segmentation - A Comparative Study”. In: *IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)*, pp. 7–7. DOI: 10.1109/SMI.2006.24.
- Attene, Marco, Francesco Robbiano, Michela Spagnuolo and Bianca Falcidieno (2007). “Semantic Annotation of 3D Surface Meshes Based on Feature Characterization”. In: *Semantic Multimedia*. Ed. by Bianca Falcidieno, Michela Spagnuolo, Yannis Avrithis, Ioannis Kompatsiaris and Paul Buitelaar. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 126–139. ISBN: 978-3-540-77051-0.
- Bacim, Felipe, Mahdi Nabyouni and Doug A. Bowman (2014). “Slice-n-Swipe: A free-hand gesture user interface for 3D point cloud annotation”. In: *2014 IEEE Symposium on 3D User Interfaces (3DUI)*, pp. 185–186. DOI: 10.1109/3DUI.2014.6798882.
- Bærentzen, Andreas and Eva Rotenberg (2021). “Skeletonization via Local Separators”. In: *ACM Transactions on Graphics* 40.5, 187:1–187:18. ISSN: 0730-0301.
URL: <https://doi.org/10.1145/3459233>.
- Bailey, Mike (2018). “Introduction to the Vulkan graphics API”. In: *SIGGRAPH Asia 2018 Courses*. ACM. DOI: 10.1145/3277644.3277800.
- Balakrishnan, Ravin and Gordon Kurtenbach (1999). “Exploring Bimanual Camera Control and Object Manipulation in 3D Graphics Interfaces”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '99. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, pp. 56–62. ISBN: 0201485591. DOI: 10.1145/302979.302991.
- Bastir, Markus, Daniel García-Martínez, Nicole Torres-Tamayo, Carlos A. Palancar, Francisco Javier Fernández-Pérez, Alberto Riesco-López, Pedro Osborne-Márquez, María Ávila and Pilar López-Gallo (2019). “Workflows in a virtual morphology lab: 3D scanning, measuring, and printing”. In: *Journal of Anthropological Sciences* 97, pp. 107–134. DOI: 10.4436/jass.97003.
- Batmaz, Anil Ufuk, Aunoy K Mutasim and Wolfgang Stuerzlinger (2020). “Precision vs. Power Grip: A Comparison of Pen Grip Styles for Selection in Virtual Reality”. In: *2020 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pp. 23–28. DOI: 10.1109/VRW50115.2020.00012.
- Battke, Henrik, Detlev Stalling and Hans-Christian Hege (1997). “Fast line integral convolution for arbitrary surfaces in 3D”. In: *Visualization and Mathematics*. Springer, pp. 181–195.

- Bleisch, Susanne and Stephan Nebiker (2008). "Connected 2D and 3D visualizations for the interactive exploration of spatial information". In: *Proc. of 21th ISPRS Congress, Beijing, China*. 1999, pp. 1037–1042.
- Bolas, M.T. (Jan. 1994). "Human factors in the design of an immersive display". In: *IEEE Computer Graphics and Applications* 14.1, pp. 55–59. ISSN: 1558-1756. DOI: 10.1109/38.250920.
- Bolte, Benjamin, Frank Steinicke and Gerd Bruder (2011). "The jumper metaphor: an effective navigation technique for immersive display setups". In: *Proceedings of Virtual Reality International Conference*. Vol. 1. 2.
- Bookstein, Fred L. (1991). *Morphometric Tools for Landmark Data: Geometry and Biology*. Cambridge: Cambridge University Press. DOI: 10.1017/CB09780511573064.
- (1998). "A hundred years of morphometrics". In: *Acta Zoologica Academiae Scientiarum Hungaricae* 44, pp. 7–59.
- Bouaoud, Jebrane, Mohamed El Beheiry, Eve Jablon, Thomas Schouman, Chloé Bertolus, Arnaud Picard, Jean-Baptiste Masson and Roman H. Khonsari (2020). "DIVA, a 3D virtual reality platform, improves undergraduate craniofacial trauma education". In: *Journal of Stomatology, Oral and Maxillofacial Surgery*. To appear. DOI: <https://doi.org/10.1016/j.jormas.2020.09.009>.
- Bouguila, Laroussi, Evequoz Florian, Michèle Courant and Béat Hirsbrunner (2005). "Active walking interface for human-scale virtual environment". In: *11th International Conference on Human-Computer Interaction, HCII*. Vol. 5, pp. 22–27.
- Bowman, D.A., D. Koller and L.F. Hodges (1997). "Travel in immersive virtual environments: an evaluation of viewpoint motion control techniques". In: *Proceedings of IEEE 1997 Annual International Symposium on Virtual Reality*, pp. 45–52. DOI: 10.1109/VRAIS.1997.583043.
- Bowman, Doug A., Elizabeth T. Davis, Larry F. Hodges and Albert N. Badre (1999). "Maintaining Spatial Orientation during Travel in an Immersive Virtual Environment". In: *Presence* 8.6, pp. 618–631. DOI: 10.1162/105474699566521.
- Bowman, Doug A., Ernst Kruijff, Joseph J. LaViola and Ivan Poupyrev (2001). "An Introduction to 3-D User Interface Design". In: *Presence* 10.1, pp. 96–108. DOI: 10.1162/105474601750182342.
- Bowman, Doug A., Ryan P. McMahan and Eric D. Ragan (2012). "Questioning naturalism in 3D user interfaces". In: *Communications of the ACM* 55.9, pp. 78–88. DOI: <https://doi.org/10.1145/2330667.2330687>.
- Bozgeyikli, Evren, Andrew Rajj, Srinivas Katkoori and Rajiv Dubey (2016). "Point & Teleport Locomotion Technique for Virtual Reality". In: *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play*. CHI PLAY '16. Austin, Texas, USA: Association for Computing Machinery, pp. 205–216. ISBN: 9781450344562. DOI: 10.1145/2967934.2968105.

- Bozgeyikli, Evren, Andrew Raji, Srinivas Katkoori and Rajiv Dubey (2019). “Locomotion in virtual reality for room scale tracked areas”. In: *International Journal of Human-Computer Studies* 122, pp. 38–49. ISSN: 1071-5819. DOI: <https://doi.org/10.1016/j.ijhcs.2018.08.002>.
- Brombin, Chiara and Luigi Salmaso (2013). “A Brief Overview on Statistical Shape Analysis”. In: *Permutation Tests in Shape Analysis*. New York, NY: Springer, pp. 1–16. DOI: 10.1007/978-1-4614-8163-8_1.
- Brooks, F.P. (1999). “What’s real about virtual reality?” In: *IEEE Computer Graphics and Applications* 19.6, pp. 16–27. DOI: 10.1109/38.799723.
- Bruder, Gerd, Frank Steinicke and Klaus H. Hinrichs (2009). “Arch-Explore: A natural user interface for immersive architectural walkthroughs”. In: *2009 IEEE Symposium on 3D User Interfaces*, pp. 75–82. DOI: 10.1109/3DUI.2009.4811208.
- Bruder, Gerd, Frank Steinicke, Phil Wieland and Markus Lappe (2012). “Tuning Self-Motion Perception in Virtual Reality with Visual Illusions”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.7, pp. 1068–1078. DOI: 10.1109/TVCG.2011.274.
- Bryson, Steve (1996). “Virtual Reality in Scientific Visualization”. eng. In: *Communications of the Acm* 39.5, pp. 62–71. ISSN: 15577317, 00010782. DOI: 10.1145/229459.229467.
- Bulao, Jacquelyn (2021). *How Much Data Is Created Every Day in 2020?* TechJury Blog.
URL: <https://techjury.net/blog/how-much-data-is-created-every-day/>.
- Bunsch, Eryk and Robert Sitnik (2014). “Method for visualization and presentation of priceless old prints based on precise 3D scan”. In: *Measuring, Modeling, and Reproducing Material Appearance*. Vol. 9018. SPIE, 90180Q. DOI: 10.1117/12.2042635.
- Bunsch, Eryk, Robert Sitnik and Jakub Michonski (2011). “Art documentation quality in function of 3D scanning resolution and precision”. In: *Computer Vision and Image Analysis of Art II*. Vol. 7869. Proceedings of SPIE, p. 78690D. DOI: 10.1117/12.876647.
- Burgess, John (2020). “RTX on—The NVIDIA Turing GPU”. In: *IEEE Micro* 40.2, pp. 36–44. DOI: 10.1109/MM.2020.2971677.
- Busking, S, CP Botha, L Ferrarini, J Milles and FH Post (2011). “Image-based rendering of intersecting surfaces for dynamic comparative visualization”. English. In: *The Visual Computer: international journal of computer graphics* 27, pp. 347–363. ISSN: 0178-2789. DOI: 10.1007/s00371-010-0541-z.

- Buttussi, Fabio and Luca Chittaro (2021). “Locomotion in Place in Virtual Reality: A Comparative Evaluation of Joystick, Teleport, and Leaning”. In: *IEEE Transactions on Visualization and Computer Graphics* 27.1, pp. 125–136. DOI: 10.1109/TVCG.2019.2928304.
- Cabral, Brian and Leith Casey Leedom (1993). “Imaging vector fields using line integral convolution”. In: *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pp. 263–270.
- Cai, Siyi, Yu He, Haomin Cui, Xi Zhou, Dongsheng Zhou, Fu Wang and Ye Tian (2020). “Effectiveness of three-dimensional printed and virtual reality models in learning the morphology of craniovertebral junction deformities: a multicentre, randomised controlled study”. In: *BMJ open* 10.9. DOI: 10.1136/bmjopen-2020-036853.
- Cakmak, Tuncay and Holger Hager (2014). “Cyberith Virtualizer: A Locomotion Device for Virtual Reality”. In: *ACM SIGGRAPH 2014 Emerging Technologies*. SIGGRAPH '14. Vancouver, Canada: Association for Computing Machinery. ISBN: 9781450329613. DOI: 10.1145/2614066.2614105.
- Cao, Wenming, Zhiyue Yan, Zhiquan He and Zhihai He (2020). “A Comprehensive Survey on Geometric Deep Learning”. In: *IEEE Access* 8, pp. 35929–35949. DOI: 10.1109/ACCESS.2020.2975067.
- Chandler, Tom, Maxime Cordeil, Tobias Czauderna, Tim Dwyer, Jaroslaw Glowacki, Cagatay Goncu, Matthias Klapperstueck, Karsten Klein, Kim Marriott, Falk Schreiber and Elliot Wilson (2015). “Immersive analytics”. In: *Big Data Visual Analytics (BDVA 2015)*. IEEE, pp. 1–8. DOI: 10.1109/BDVA.2015.7314296.
- Chen, Michael, S Joy Mountford and Abigail Sellen (1988). “A study in interactive 3-D rotation using 2-D control devices”. In: *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pp. 121–129.
- Chow, M.M. (1997). “Optimized geometry compression for real-time rendering”. In: *Proceedings of Visualization '97*, pp. 347–354. URL: <https://doi.org/10.1109/VISUAL.1997.663902>.
- Christie, Marc, Patrick Olivier and Jean-Marie Normand (2008). “Camera control in computer graphics”. In: *Computer Graphics Forum*. Vol. 27. 8. Wiley Online Library, pp. 2197–2218.
- Christou, Chris G. and Poppy Aristidou (2017). “Steering Versus Teleport Locomotion for Head Mounted Displays”. In: *Augmented Reality, Virtual Reality, and Computer Graphics*. Ed. by Lucio Tommaso De Paolis, Patrick Bourdot and Antonio Mongelli. Cham: Springer International Publishing, pp. 431–446. ISBN: 978-3-319-60928-7.
- Cigolle, Zina H., Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire and Quirin Meyer (Apr. 2014). “A Survey of Efficient Representations for Inde-

- pendent Unit Vectors”. In: *Journal of Computer Graphics Techniques* 3.2, pp. 1–30. ISSN: 2331-7418.
URL: <http://jcgt.org/published/0003/02/01/>.
- Coffey, Dane, Chi-Lun Lin, Arthur G. Erdman and Daniel F. Keefe (2013). “Design by Dragging: An Interface for Creative Forward and Inverse Design with Simulation Ensembles”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.12, pp. 2783–2791. DOI: 10.1109/TVCG.2013.147.
- Collyer, M. L., D. J. Sekora and D. C. Adams (2015). “A method for analysis of phenotypic change for phenotypes described by high-dimensional data”. In: *Heredity* 115, pp. 357–365. DOI: <https://doi.org/10.1038/hdy.2014.75>.
- Cordeil, Maxime, Andrew Cunningham, Benjamin Bach, Christophe Hurter, Bruce H Thomas, Kim Marriott and Tim Dwyer (2019). “IATK: An immersive analytics toolkit”. In: *IEEE Conference on Virtual Reality and 3D User Interfaces (VR 2019)*, pp. 200–209. DOI: 10.1109/VR.2019.8797978.
- Cruz-Neira, Carolina, Daniel J Sandin, Thomas A DeFanti, Robert V Kenyon and John C Hart (1992). “The CAVE: audio visual experience automatic virtual environment”. In: *Communications of the ACM* 35.6, pp. 64–73.
- Cummings, James J. and Jeremy N. Bailenson (2016). “How Immersive Is Enough? A Meta-Analysis of the Effect of Immersive Technology on User Presence”. eng. In: *Media Psychology* 19.2, pp. 272–309. ISSN: 1532785x, 15213269. DOI: 10.1080/15213269.2015.1015740.
- Cutting, James E and Peter M Vishton (1995). “Perceiving layout and knowing distances: The integration, relative potency, and contextual use of different information about depth”. In: *Perception of space and motion*. Elsevier, pp. 69–117.
- Dally, William J., Stephen W. Keckler and David B. Kirk (2021). “Evolution of the graphics processing unit (GPU)”. In: *IEEE Micro* 41.6, pp. 42–51.
URL: <https://doi.org/10.1109/MM.2021.3113475>.
- Dam, A. van, A.S. Forsberg, D.H. Laidlaw, J.J. LaViola and R.M. Simpson (2000). “Immersive VR for scientific visualization: a progress report”. In: *IEEE Computer Graphics and Applications* 20.6, pp. 26–52. DOI: 10.1109/38.888006.
- Deering, Michael (1992). “High Resolution Virtual Reality”. In: *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH ’92*. New York, NY, USA: Association for Computing Machinery, pp. 195–202. ISBN: 0897914791. DOI: 10.1145/133994.134039.
- (1995). “Geometry Compression”. In: *SIGGRAPH ’95*. ACM, pp. 13–20. ISBN: 0897917014.
URL: <https://doi.org/10.1145/218380.218391>.
- Deering, Michael F. and Scott R. Nelson (1993). “Leo: A System for Cost Effective 3D Shaded Graphics”. In: *SIGGRAPH ’93*. Anaheim, CA: ACM, pp. 101–108. ISBN:

0897916018.
URL: <https://doi.org/10.1145/166117.166130>.
- Diepstraten, Joachim, Daniel Weiskopf and Thomas Ertl (2002). “Transparency in interactive technical illustrations”. In: *Computer Graphics Forum*. Vol. 21. 3. Wiley Online Library, pp. 317–325.
- (2003). “Interactive cutaway illustrations”. In: *Computer Graphics Forum*. Vol. 22. 3. Wiley Online Library, pp. 523–532.
- Dillencourt, Michael B. (1996). “Finding Hamiltonian cycles in Delaunay triangulations is NP-complete”. In: *Discrete Applied Mathematics* 64.3, pp. 207–217. ISSN: 0166-218X.
URL: [https://doi.org/10.1016/0166-218X\(94\)00125-W](https://doi.org/10.1016/0166-218X(94)00125-W).
- Doboš, Jozef and Anthony Steed (2012). “3D Diff: An Interactive Approach to Mesh Differencing and Conflict Resolution”. In: *SIGGRAPH Asia 2012 Technical Briefs*. SA '12. Singapore, Singapore: Association for Computing Machinery. ISBN: 9781450319157. DOI: 10.1145/2407746.2407766.
- Donalek, Ciro, S. G. Djorgovski, Alex Cioc, Anwell Wang, Jerry Zhang, Elizabeth Lawler, Stacy Yeh, Ashish Mahabal, Matthew Graham, Andrew Drake, Scott Davidoff, Jeffrey S. Norris and Giuseppe Longo (2014). “Immersive and collaborative data visualization using virtual reality platforms”. eng. In: *Proceedings - 2014 Ieee International Conference on Big Data, Ieee Big Data 2014*, pp. 7004282, 609–614. ISSN: 26391589. DOI: 10.1109/BigData.2014.7004282.
- Eiriksson, Eythor R., Jakob Wilm, David B. Pedersen and Henrik Aanæs (Apr. 2016). “Precision and Accuracy Parameters in Structured Light 3-D Scanning”. English. In: *International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences XL-5/W8*, pp. 7–15. DOI: 10.5194/isprs-archives-XL-5-W8-7-2016.
- El Beheiry, Mohamed, Sébastien Doutreligne, Clément Caporal, Cécilia Ostertag, Maxime Dahan and Jean Baptiste Masson (2019). “Virtual Reality: Beyond Visualization”. eng. In: *Journal of Molecular Biology* 431.7, pp. 1315–1321. ISSN: 10898638, 00222836. DOI: 10.1016/j.jmb.2019.01.033.
- El Beheiry, Mohamed, Charlotte Godard, Clément Caporal, Valentin Marcon, Cécilia Ostertag, Oumaima Sliti, Sébastien Doutreligne, Stéphane Fournier, Bassam Hajj, Maxime Dahan and Jean-Baptiste Masson (2020). “DIVA: Natural Navigation Inside 3D Images Using Virtual Reality”. In: *Journal of Molecular Biology* 432.16, pp. 4745–4749. ISSN: 0022-2836. DOI: <https://doi.org/10.1016/j.jmb.2020.05.026>.
- Elden, Magnus Kolåseter (2017). “Implementation and initial assessment of VR for scientific visualisation: Extending Unreal Engine 4 to visualise scientific data on the HTC Vive”. MA thesis.

- Englert, Matthias (2020). “Using Mesh Shaders for Continuous Level-of-Detail Terrain Rendering”. In: *ACM SIGGRAPH 2020 Talks*. Virtual Event, USA: ACM, 44:1–44:2. ISBN: 9781450379717.
URL: <https://doi.org/10.1145/3388767.3407391>.
- Evans, F., S. Skiena and A. Varshney (1996). “Optimizing triangle strips for fast rendering”. In: *Proceedings of Seventh Annual IEEE Visualization '96*, pp. 319–326.
URL: <https://doi.org/10.1109/VISUAL.1996.568125>.
- Fairchild, K.M., B.H. Lee, J. Loo, H. Ng and L. Serra (1993). “The heaven and earth virtual reality: Designing applications for novice users”. In: *Proceedings of IEEE Virtual Reality Annual International Symposium*, pp. 47–53. DOI: 10.1109/VRAIS.1993.380799.
- Fang, Jianbin, Ana Lucia Varbanescu and Henk Sips (2011). “A comprehensive performance comparison of CUDA and OpenCL”. In: *International Conference on Parallel Processing (ICPP 2011)*. IEEE, pp. 216–225. DOI: 10.1109/ICPP.2011.45.
- Foley, James D. (1987). “Interfaces for Advanced Computing”. In: *Scientific American* 257.4, pp. 126–135. ISSN: 00368733, 19467087.
URL: <http://www.jstor.org/stable/24979516> (visited on 11th July 2022).
- Forsyth, Tom (Sept. 2006). *Linear-speed vertex cache optimisation*. Accessed: 2022-04-16.
URL: https://tomforsyth1000.github.io/papers/fast_vert_cache_opt.html.
- Freitag, Sebastian, Dominik Rausch and Torsten Kuhlen (2014). “Reorientation in virtual environments using interactive portals”. In: *2014 IEEE Symposium on 3D User Interfaces (3DUI)*, pp. 119–122. DOI: 10.1109/3DUI.2014.6798852.
- Fruciano, Carmelo (June 2016). “Measurement error in geometric morphometrics”. In: *Development genes and evolution* 3, pp. 139–158. DOI: 10.1007/s00427-016-0537-4.
- Fruciano, Carmelo, Mélina A. Celik, Kaylene Butler, Tom Dooley, Vera Weisbecker and Matthew J. Phillips (2017). “Sharing is caring? Measurement error and the issues arising from combining 3D morphometric datasets”. In: *Ecology and Evolution* 7.17, pp. 7034–7046. DOI: <https://doi.org/10.1002/ece3.3256>.
- García-Hernández, Rubén Jesús and Dieter Kranzlmüller (2019). “NOMAD VR: Multiplatform virtual reality viewer for chemistry simulations”. In: *Computer Physics Communications* 237, pp. 230–237. ISSN: 0010-4655. DOI: <https://doi.org/10.1016/j.cpc.2018.11.013>.
- Gawrilowicz, Florian and J. Andreas Bærentzen (2019). “Optimal, Non-Rigid Alignment for Feature-Preserving Mesh Denoising”. English. In: *Proceedings of the International Conference on 3D Vision (3DV)*. IEEE, pp. 415–423. DOI: 10.1109/3DV.2019.00053.

- Giacomini, Giada, Dino Scaravelli, Anthony Herrel, Alessio Veneziano, Danilo Russo, Richard P. Brown and Carlo Meloro (2019). “3D Photogrammetry of Bat Skulls: Perspectives for Macro-evolutionary Analyses”. In: *Evolutionary Biology* 46.3, pp. 249–259. DOI: 10.1007/s11692-019-09478-6.
- Gleicher, Michael, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D. Hansen and Jonathan C. Roberts (Oct. 2011). “Visual Comparison for Information Visualization”. In: *Information Visualization* 10.4, pp. 289–309. ISSN: 1473-8716. DOI: 10.1177/1473871611416549.
- Gonizzi Barsanti, Sara, Giandomenico Caruso, LL Micoli, M Covarrubias Rodriguez, Gabriele Guidi et al. (2015). “3D visualization of cultural heritage artefacts with virtual reality devices”. In: *25th International CIPA Symposium 2015*. Vol. 40. 5W7. Copernicus Gesellschaft mbH, pp. 165–172.
- Gooch, Amy, Bruce Gooch, Peter Shirley and Elaine Cohen (1998). “A non-photorealistic lighting model for automatic technical illustration”. In: *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pp. 447–452.
- Goodall, Colin (1991). “Procrustes Methods in the Statistical Analysis of Shape”. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 53.2, pp. 285–339.
URL: <http://www.jstor.org/stable/2345744>.
- Gower, J. C. (1975). “Generalized Procrustes analysis”. In: *Psychometrika* 40, pp. 33–51. DOI: <https://doi.org/10.1007/BF02291478>.
- Guiard, Yves (1987). “Asymmetric Division of Labor in Human Skilled Bimanual Action”. In: *Journal of Motor Behavior* 19.4. PMID: 15136274, pp. 486–517. DOI: 10.1080/00222895.1987.10735426. eprint: <https://doi.org/10.1080/00222895.1987.10735426>.
- Haines, Eric (2006). “An introductory tour of interactive rendering”. In: *IEEE Computer Graphics and Applications* 26.1, pp. 76–87.
URL: <https://doi.org/10.1109/MCG.2006.9>.
- Harris, Alyssa, Kevin Nguyen, Preston Tunnell Wilson, Matthew Jackoski and Betsy Williams (2014). “Human Joystick: Wii-Leaning to Translate in Large Virtual Environments”. In: *Proceedings of the 13th ACM SIGGRAPH International Conference on Virtual-Reality Continuum and Its Applications in Industry*. VRCAI '14. Shenzhen, China: Association for Computing Machinery, pp. 231–234. ISBN: 9781450332545. DOI: 10.1145/2670473.2670512.
- Hasselgren, J., J. Munkberg, M. Salvi, A. Patney and A. Lefohn (2020). “Neural temporal adaptive sampling and denoising”. In: *Computer Graphics Forum* 39.2, pp. 147–155. DOI: 10.1111/cgf.13919.

- Hinckley, Ken, Joe Tullio, Randy Pausch, Dennis Proffitt and Neal Kassell (1997). "Usability analysis of 3D rotation techniques". In: *Proceedings of the 10th annual ACM symposium on User interface software and technology*, pp. 1–10.
- Holzinger, Andreas (2016). "Interactive machine learning for health informatics: when do we need the human-in-the-loop?" In: *Brain Informatics 3.2*, pp. 119–131.
- Hoppe, Hugues (1999). "Optimization of Mesh Locality for Transparent Vertex Caching". In: *SIGGRAPH '99*. ACM/Addison-Wesley, pp. 269–276. ISBN: 0201485605. URL: <https://doi.org/10.1145/311535.311565>.
- Husung, Malte and Eike Langbehn (2019). "Of Portals and Orbs: An Evaluation of Scene Transition Techniques for Virtual Reality". In: *Proceedings of Mensch Und Computer 2019*. MuC'19. Hamburg, Germany: Association for Computing Machinery, pp. 245–254. ISBN: 9781450371988. DOI: 10.1145/3340764.3340779.
- Hutchins, Edwin L., James D. Hollan and Donald A. Norman (1985). "Direct Manipulation Interfaces". In: *Human-Computer Interaction 1.4*, pp. 311–338. DOI: 10.1207/s15327051hci0104_2.
- Interrante, V., H. Fuchs and S.M. Pizer (1997). "Conveying the 3D shape of smoothly curving transparent surfaces via texture". In: *IEEE Transactions on Visualization and Computer Graphics 3.2*, pp. 98–117. DOI: 10.1109/2945.597794.
- Interrante, V., B. Ries and L. Anderson (2006). "Distance Perception in Immersive Virtual Environments, Revisited". In: *IEEE Virtual Reality Conference (VR 2006)*, pp. 3–10. DOI: 10.1109/VR.2006.52.
- Iwata, H. (1999). "Walking about virtual environments on an infinite floor". In: *Proceedings IEEE Virtual Reality (Cat. No. 99CB36316)*, pp. 286–293. DOI: 10.1109/VR.1999.756964.
- Jang, Susan, Jonathan M. Vitale, Robert W. Jyung and John B. Black (2017). "Direct manipulation is better than passive viewing for learning anatomy in a three-dimensional virtual reality environment". In: *Computers & Education 106*, pp. 150–165. DOI: <https://doi.org/10.1016/j.compedu.2016.12.009>.
- Jensen, Mark B., Egill I. Jacobsen, Jeppe Revall Frisvad and J. Andreas Bærentzen (2021). "Tools for Virtual Reality Visualization of Highly Detailed Meshes". In: *VisGap - The Gap between Visualization Research and Visualization Software*. Ed. by Christina Gillmann, Michael Krone, Guido Reina and Thomas Wischgoll. The Eurographics Association. DOI: 10.2312/visgap.20211088.
- Jerald, Jason (2015). *The VR book: Human-centered design for virtual reality*. Morgan & Claypool.
- Jiménez Fernández-Palacios, Belen, Daniele Morabito and Fabio Remondino (2017). "Access to complex reality-based 3D models using virtual reality solutions". In:

- Journal of Cultural Heritage* 23, pp. 40–48. ISSN: 1296-2074. DOI: <https://doi.org/10.1016/j.culher.2016.09.003>.
- Kabbash, Paul, William Buxton and Abigail Sellen (1994). “Two-Handed Input in a Compound Task”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '94. Boston, Massachusetts, USA: Association for Computing Machinery, pp. 417–423. ISBN: 0897916506. DOI: 10.1145/191666.191808.
- Kapoulkine, Arseny (2017). *meshoptimizer*. Accessed: 2022-04-16.
URL: <https://github.com/zeux/meshoptimizer>.
- Karis, Brian, Rune Stubbe and Graham Wihlidal (2021). “A Deep Dive into Nanite Virtualized Geometry”. In: *Advances in Real-time Rendering in Games: Part I*. Ed. by Natalya Tatarchuk. ACM SIGGRAPH 2021 Courses. Accessed: 2022-04-16.
URL: <https://advances.realtimerendering.com/s2021/index.html>.
- Kaufman, Leonard and Peter J. Rousseeuw (1990). *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons. ISBN: 978-0-47031680-1.
URL: <https://doi.org/10.1002/9780470316801>.
- Kazhdan, Michael, Matthew Bolitho and Hugues Hoppe (2006). “Poisson Surface Reconstruction”. In: *Eurographics Symposium on Geometry Processing*. SGP '06. Eurographics Association, pp. 61–70.
- Kazhdan, Michael and Hugues Hoppe (2013). “Screened Poisson Surface Reconstruction”. In: *ACM Transactions on Graphics* 32.3. DOI: 10.1145/2487228.2487237.
- Keefe, Daniel, Marcus Ewert, William Ribarsky and Remco Chang (2009). “Interactive Coordinated Multiple-View Visualization of Biomechanical Motion Data”. In: *IEEE Transactions on Visualization and Computer Graphics* 15.6, pp. 1383–1390. DOI: 10.1109/TVCG.2009.152.
- Kehrer, Johannes and Helwig Hauser (2013). “Visualization and Visual Analysis of Multifaceted Scientific Data: A Survey”. In: *IEEE Transactions on Visualization and Computer Graphics* 19.3, pp. 495–513. DOI: 10.1109/TVCG.2012.110.
- Kerbl, Bernhard, Michael Kenzel, Elena Ivanchenko, Dieter Schmalstieg and Markus Steinberger (Aug. 2018). “Revisiting The Vertex Cache: Understanding and Optimizing Vertex Processing on the Modern GPU”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2, 29:1–29:16.
URL: <https://doi.org/10.1145/3233302>.
- Khan, Sarah and Richard Chang (2013). “Anatomy of the vestibular system: a review”. In: *NeuroRehabilitation* 32.3, pp. 437–443.
- Kim, Kyungyoon, John V. Carlis and Daniel F. Keefe (2017). “Comparison techniques utilized in spatial 3D and 4D data visualizations: A survey and future directions”.

- In: *Computers & Graphics* 67, pp. 138–147. ISSN: 0097-8493. DOI: <https://doi.org/10.1016/j.cag.2017.05.005>.
- Kim, Kyungyoon, Bret Jackson, Ioannis Karamouzas, Moses Adeagbo, Stephen J. Guy, Richard Graff and Daniel F. Keefe (2015). “Bema: A multimodal interface for expert experiential analysis of political assemblies at the Pnyx in ancient Greece”. In: *2015 IEEE Symposium on 3D User Interfaces (3DUI)*, pp. 19–26. DOI: 10.1109/3DUI.2015.7131720.
- Klingenberg, Christian Peter, Marta Barluenga and Axel Meyer (2002). “Shape analysis of symmetric structures: quantifying variation among individuals and asymmetry”. In: *Evolution* 56.10, pp. 1909–1920. DOI: 10.1111/j.0014-3820.2002.tb00117.x.
- Klingenberg, Christian Peter and Grant S. McIntyre (1998). “Geometric Morphometrics of Developmental Instability: Analyzing Patterns of Fluctuating Asymmetry with Procrustes Methods”. In: *Evolution* 52.5, pp. 1363–1375. URL: <http://www.jstor.org/stable/2411306>.
- Koch, Daniel, Tobias Hector, Joshua Barczak and Eric Werness (Mar. 2020). *Ray Tracing in Vulkan*. Khronos Blog. URL: <https://www.khronos.org/blog/ray-tracing-in-vulkan>.
- Koo, Bonsang, Raekyu Jung and Youngsu Yu (2021). “Automatic classification of wall and door BIM element subtypes using 3D geometric deep neural networks”. In: *Advanced Engineering Informatics* 47, p. 101200. ISSN: 1474-0346. DOI: <https://doi.org/10.1016/j.aei.2020.101200>.
- Koomey, Jonathan, Stephen Berard, Marla Sanchez and Henry Wong (2011). “Implications of Historical Trends in the Electrical Efficiency of Computing”. In: *IEEE Annals of the History of Computing* 33.3, pp. 46–54. DOI: 10.1109/MAHC.2010.28.
- Kreylos, Oliver, Gerald Bawden, Tony Bernardin, Magali I. Billen, Eric S. Cowgill, Ryan D. Gold, Bernd Hamann, Margarete Jadamec, Louise H. Kellogg, Oliver G. Staadt and Dawn Y. Sumner (2006). “Enabling scientific workflows in virtual reality”. In: *International Conference on Virtual Reality Continuum and Its Applications (VRCIA 2006)*, pp. 155–162. DOI: 10.1145/1128923.1128948.
- Kubisch, Christoph (Sept. 2018a). *Introduction to Turing Mesh Shaders*. NVIDIA Developer Technical Blog. URL: <https://developer.nvidia.com/blog/introduction-turing-mesh-shaders/>.
- (2018b). *Vulkan & OpenGL CAD Mesh Shader Sample*. Accessed: 2022-04-16. URL: https://github.com/nvpro-samples/gl_vk_meshlet_cadscene.
- (Dec. 2020). *Using Mesh Shaders for Professional Graphics*. URL: <https://developer.nvidia.com/blog/using-mesh-shaders-for-professional-graphics/>.

- Kunert, André, Alexander Kulik, Stephan Beck and Bernd Froehlich (2014). “Photoportals: Shared References in Space and Time”. In: *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work; Social Computing. CSCW '14*. Baltimore, Maryland, USA: Association for Computing Machinery, pp. 1388–1399. ISBN: 9781450325400. DOI: 10.1145/2531602.2531727.
- Landesberger, Tatiana von, Dennis Basgier and Meike Becker (2016). “Comparative Local Quality Assessment of 3D Medical Image Segmentations with Focus on Statistical Shape Model-Based Algorithms”. In: *IEEE Transactions on Visualization and Computer Graphics* 22.12, pp. 2537–2549. DOI: 10.1109/TVCG.2015.2501813.
- Laramee, R.S., B. Jobard and H. Hauser (2003). “Image space based visualization of unsteady flow on surfaces”. In: *IEEE Visualization, 2003. VIS 2003*. Pp. 131–138. DOI: 10.1109/VISUAL.2003.1250364.
- Laramee, Robert S., Helwig Hauser, Helmut Doleisch, Benjamin Vrolijk, Frits H. Post and Daniel Weiskopf (2004). “The State of the Art in Flow Visualization: Dense and Texture-Based Techniques”. In: *Computer Graphics Forum*. ISSN: 1467-8659. DOI: 10.1111/j.1467-8659.2004.00753.x.
- Lee, Chang Ha, Alan Liu and Thomas P Caudell (2009). “A study of locomotion paradigms for immersive medical simulation environments”. In: *The Visual Computer* 25.11, pp. 1009–1018.
- Lee, Kwan Min (2004). “Presence, explicated”. In: *Communication theory* 14.1, pp. 27–50.
- Leiserson, Charles E., Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez and Tao B. Schardl (2020). “There’s plenty of room at the Top: What will drive computer performance after Moore’s law?” In: *Science* 368.6495. DOI: 10.1126/science.aam9744.
- Lempiainen, Jarkko (2020). *Meshlete: Chop 3D objects to meshlets*. Accessed: 2022-04-16.
URL: <https://github.com/JarkkoPFC/meshlete>.
- Li, Wilmot, Maneesh Agrawala, Brian Curless and David Salesin (Aug. 2008). “Automated Generation of Interactive 3D Exploded View Diagrams”. In: *ACM Trans. Graph.* 27.3, pp. 1–7. ISSN: 0730-0301. DOI: 10.1145/1360612.1360700.
- Li, Wilmot, Lincoln Ritter, Maneesh Agrawala, Brian Curless and David Salesin (July 2007). “Interactive Cutaway Illustrations of Complex 3D Models”. In: *ACM Trans. Graph.* 26.3, 31–es. ISSN: 0730-0301. DOI: 10.1145/1276377.1276416.
- Li, Zhenxing, Maria Kiiveri, Jussi Rantala and Roope Raisamo (2021). “Evaluation of haptic virtual reality user interfaces for medical marking on 3D models”. In: *International Journal of Human-Computer Studies* 147. DOI: <https://doi.org/10.1016/j.ijhcs.2020.102561>.

- Liebelt, Joerg and Cordelia Schmid (2010). “Multi-view object class detection with a 3D geometric model”. In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 1688–1695. DOI: 10.1109/CVPR.2010.5539836.
- Lin, G. and T.P.-Y. Yu (2006). “An improved vertex caching scheme for 3D mesh rendering”. In: *IEEE Transactions on Visualization and Computer Graphics* 12.4, pp. 640–648.
URL: <https://doi.org/10.1109/TVCG.2006.59>.
- Lindholm, Erik, John Nickolls, Stuart Oberman and John Montrym (2008). “NVIDIA Tesla: A unified graphics and computing architecture”. In: *IEEE Micro* 28.2, pp. 39–55.
URL: <https://doi.org/10.1109/MM.2008.31>.
- Liu, Edward, Ignacio Llamas, Juan Cañada and Patrick Kelly (2019). “Cinematic rendering in UE4 with real-time ray tracing and denoising”. In: *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Ed. by Eric Haines and Tomas Akenine-Möller. Apress, pp. 289–319. DOI: 10.1007/978-1-4842-4427-2_19.
- Lombard, Matthew, Theresa B Ditton and Lisa Weinstein (2009). “Measuring presence: the temple presence inventory”. In: *Proceedings of the 12th annual international workshop on presence*, pp. 1–15.
- Lorensen, William E. and Harvey E. Cline (1987). “Marching Cubes: A High Resolution 3D Surface Construction Algorithm”. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '87. New York, NY, USA: Association for Computing Machinery, pp. 163–169. ISBN: 0897912276. DOI: 10.1145/37401.37422.
- Luebke, David, Martin Reddy, Jonathan D Cohen, Amitabh Varshney, Benjamin Watson and Robert Huebner (2003). *Level of detail for 3D graphics*. Morgan Kaufmann.
- Mackinlay, Jock D., Stuart K. Card and George G. Robertson (Sept. 1990). “Rapid Controlled Movement through a Virtual 3D Workspace”. In: *SIGGRAPH Comput. Graph.* 24.4, pp. 171–176. ISSN: 0097-8930. DOI: 10.1145/97880.97898.
- Manferdini, Anna Maria and Fabio Remondino (2010). “Reality-Based 3D Modeling, Segmentation and Web-Based Visualization”. In: *Digital Heritage*. Ed. by Marinou Ioannides, Dieter Fellner, Andreas Georgopoulos and Diofantos G. Hadjimitsis. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 110–124. ISBN: 978-3-642-16873-4.
- Mangiafico, Salvatore (2021). *rcompanion: Functions to Support Extension Education Program Evaluation*. R package version 2.4.6.
URL: <https://CRAN.R-project.org/package=rcompanion>.

- Marks, Stefan, Javier Estevez and Andy Connor (2016). “Towards the Holodeck: Fully Immersive Virtual Reality Visualisation of Scientific and Engineering Data”. eng. In: *None*. DOI: 10.1.1.929.8858.
- Martin, Ken, David DeMarle, Sankhesh Jhaveri and Utkarsh Ayachit (2016, updated 2018). *Taking ParaView into Virtual Reality*. Kitware Blog.
URL: <https://blog.kitware.com/taking-paraview-into-virtual-reality/>.
- McClanahan, Chris (2010). *History and evolution of GPU architecture: A Paper Survey*. Course Notes, College of Computing, Georgia Tech.
- McCullough, Morgan, Hong Xu, Joel Michelson, Matthew Jackoski, Wyatt Pease, William Cobb, William Kalescky, Joshua Ladd and Betsy Williams (2015). “Myo Arm: Swinging to Explore a VE”. In: *Proceedings of the ACM SIGGRAPH Symposium on Applied Perception*. SAP ’15. Tübingen, Germany: Association for Computing Machinery, pp. 107–113. ISBN: 9781450338127. DOI: 10.1145/2804408.2804416.
- McPherron, Shannon P., Tim Gernat and Jean-Jacques Hublin (2009). “Structured light scanning for high-resolution documentation of in situ archaeological finds”. In: *Journal of Archaeological Science* 36.1, pp. 19–24. ISSN: 0305-4403. DOI: <https://doi.org/10.1016/j.jas.2008.06.028>.
- Mendes, Daniel, Fabio Marco Caputo, Andrea Giachetti, Alfredo Ferreira and J Jorge (2019). “A survey on 3D virtual object manipulation: From the desktop to immersive virtual environments”. In: *Computer Graphics Forum* 38.1, pp. 21–45. DOI: <https://doi.org/10.1111/cgf.13390>.
- Merriam-Webster (2022). “Virtual reality.” *Merriam-Webster.com Dictionary*. Accessed 13 Jul. 2022.
URL: <https://www.merriam-webster.com/dictionary/virtual%20reality>.
- Messer, Dolores, Michelle S. Svendsen, Anders Galatius, Morten T. Olsen, Vedrana A. Dahl, Knut Conradsen and Anders B. Dahl (2021). “Measurement error using a SeeMaLab structured light 3D scanner against a Microscribe 3D digitizer”. In: *PeerJ* 9, e11804. DOI: 10.7717/peerj.11804.
- Miksch, Silvia and Helwig Hauser (2001). “Semantic depth of field”. In: *Proc. IEEE Symp. Information Visualization*. Citeseer, pp. 97–104.
- Mine, Mark R (1995). “Virtual environment interaction techniques”. In: *UNC Chapel Hill CS Dept.*
- Mine, Mark R., Frederick P. Brooks and Carlo H. Sequin (1997). “Moving Objects in Space: Exploiting Proprioception in Virtual-Environment Interaction”. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive*

- Techniques*. SIGGRAPH '97. USA: ACM Press/Addison-Wesley Publishing Co., pp. 19–26. ISBN: 0897918967. DOI: 10.1145/258734.258747.
- Mishra, Prerna and Urmila Shrawankar (2016). “Comparison Between Famous Game Engines and Eminent Games”. In: *International Journal of Interactive Multimedia and Artificial Intelligence* 4.1, pp. 69–77. DOI: 10.9781/ijimai.2016.4113.
- Mitteroecker, P. and P. Gunz (2009). “Advances in Geometric Morphometrics”. In: *Evolutionary Biology* 36, pp. 235–247. DOI: 10.1007/s11692-009-9055-x.
- Moran, A., V. Gadepally, M. Hubbell and J. Kepner (2015). “Improving Big Data visual analytics with interactive virtual reality”. In: *IEEE High Performance Extreme Computing Conference (HPEC 2015)*, pp. 1–6. DOI: 10.1109/HPEC.2015.7322473.
- Moreland, Kenneth (2009). “Diverging color maps for scientific visualization”. In: *International Symposium on Visual Computing*. Springer, pp. 92–103.
- Nam, Jung Who, Krista McCullough, Joshua Tveite, Maria Molina Espinosa, Charles H. Perry, Barry T. Wilson and Daniel F. Keefe (2019). “Worlds-in-Wedges: Combining Worlds-in-Miniature and Portals to Support Comparative Immersive Visualization of Forestry Data”. In: *2019 IEEE Conference on Virtual Reality and 3D User Interfaces (VR)*, pp. 747–755. DOI: 10.1109/VR.2019.8797871.
- Neff, T., J. H. Mueller, M. Steinberger and D. Schmalstieg (2022). “Meshlets and How to Shade Them: A Study on Texture-Space Shading”. In: *Computer Graphics Forum* 41.2, pp. 277–287.
URL: <https://doi.org/10.1111/cgf.14474>.
- Nehab, Diego, Joshua Barczak and Pedro V. Sander (2006). “Triangle order optimization for graphics hardware computation culling”. In: *Symposium on Interactive 3D Graphics and Games (I3D '06)*. ACM, pp. 207–211. DOI: 10.1145/1111411.1111448.
- North, Chris, Tim Dwyer, Bongshin Lee, Danyel Fisher, Petra Isenberg, George Robertson and Kori Inkpen (2009). “Understanding Multi-touch Manipulation for Surface Computing”. In: *Human-Computer Interaction – INTERACT 2009*. Ed. by Tom Gross, Jan Gulliksen, Paula Kotzé, Lars Oestreicher, Philippe Palanque, Raquel Oliveira Prates and Marco Winckler. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 236–249. ISBN: 978-3-642-03658-3.
- Pagendarm, Hans-Georg and Frits H Post (1995). *Comparative visualization: Approaches and examples*. Delft University of Technology Faculty of Technical Mathematics and Informatics.
- Parker, Steven G., James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison and Martin Stich (2010). “OptiX: A general purpose ray tracing engine”. In: *ACM Transactions on Graphics* 29.4. DOI: 10.1145/1778765.1778803.

- Parsons, Lawrence M (1995). "Inability to reason about an object's orientation using an axis and angle of rotation." In: *Journal of experimental psychology: Human perception and performance* 21.6, p. 1259.
- Paulsen, Rasmus R., Kristine Aavild Juhl, Thilde Marie Haspang, Thomas Hansen, Melanie Ganz and Gudmundur Einarsson (2019). "Multi-view Consensus CNN for 3D Facial Landmark Placement". In: *Computer Vision – ACCV 2018*. Ed. by C. V. Jawahar, Hongdong Li, Greg Mori and Konrad Schindler. Cham: Springer International Publishing, pp. 706–719. ISBN: 978-3-030-20887-5.
- Pausch, Randy, Tommy Burnette, Dan Brockway and Michael E. Weiblen (1995). "Navigation and Locomotion in Virtual Worlds via Flight into Hand-Held Miniatures". In: *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '95. New York, NY, USA: Association for Computing Machinery, pp. 399–400. ISBN: 0897917014. DOI: 10.1145/218380.218495.
- Perry, T.S. and J. Voelcker (1989). "Of mice and menus: designing the user-friendly interface". In: *IEEE Spectrum* 26.9, pp. 46–51. DOI: 10.1109/6.90184.
- Pham, Duc-Minh and Wolfgang Stuerzlinger (2019). "Is the Pen Mightier than the Controller? A Comparison of Input Devices for Selection in Virtual and Augmented Reality". In: *25th ACM Symposium on Virtual Reality Software and Technology*. VRST '19. New York, NY, USA: ACM. DOI: 10.1145/3359996.3364264.
- Phong, Bui Tuong (June 1975). "Illumination for Computer-generated Pictures". In: *Communications of the ACM* 18.6, pp. 311–317. DOI: 10.1145/360825.360839.
- Pohl, Henning, Klemen Liliija, Jess McIntosh and Kasper Hornbæk (2021). "Poros: Configurable Proxies for Distant Interactions in VR". In: *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. CHI '21. Yokohama, Japan: Association for Computing Machinery. ISBN: 9781450380966. DOI: 10.1145/3411764.3445685.
- Poupyrev, Ivan, Mark Billinghurst, Suzanne Weghorst and Tadao Ichikawa (1996). "The go-go interaction technique: non-linear mapping for direct manipulation in VR". In: *Proceedings of the 9th annual ACM symposium on User interface software and technology*, pp. 79–80.
- Preim, Bernhard and Steffen Oeltze (2008). "3D Visualization of Vasculature: An Overview". In: *Visualization in Medicine and Life Sciences*. Ed. by Lars Linsen, Hans Hagen and Bernd Hamann. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 39–59. ISBN: 978-3-540-72630-2.
- Prothero, Jerrold D and Donald E Parker (2003). "A Unified Approach to Presence and Motion Sickness". In: *Virtual and adaptive environments: Applications, implications, and human performance issues*, p. 47.

- R Core Team (2020). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria.
URL: <https://www.R-project.org/>.
- Razzaque, Sharif (2005). *Redirected walking*. The University of North Carolina at Chapel Hill.
- Reason, James T and Joseph John Brand (1975). *Motion sickness*. Academic press.
- Rebenitsch, Lisa and Charles Owen (2016). “Review on cybersickness in applications and visual displays”. In: *Virtual Reality* 20.2, pp. 101–125. DOI: 10.1007/s10055-016-0285-9.
- Recheis, Wolfgang, Gerhard W Weber, Katrin Schäfer, Rudolf Knapp, Horst Seidler and Dieter zur Nedden (1999). “Virtual reality and anthropology”. In: *European Journal of Radiology* 31.2, pp. 88–96. DOI: [https://doi.org/10.1016/S0720-048X\(99\)00089-3](https://doi.org/10.1016/S0720-048X(99)00089-3).
- Reddy, Junuthula Narasimha (2019). *Introduction to the finite element method*. McGraw-Hill Education.
- Riccio, Gary E. and Thomas A. Stoffregen (1991). “An ecological Theory of Motion Sickness and Postural Instability”. In: *Ecological Psychology* 3.3, pp. 195–240. DOI: 10.1207/s15326969eco0303_2. eprint: https://doi.org/10.1207/s15326969eco0303_2.
- Ripton, J. and L. Prasuethsut (Oct. 2015). *The VR race: What you need to know about Oculus Rift, HTC Vive and more*. techradar.
URL: <https://www.techradar.com/news/world-of-tech/future-tech/the-vr-race-who-s-closest-to-making-vr-a-reality-1266538>.
- Robinson, Chris and Claire E. Terhune (2017). “Error in geometric morphometric data collection: Combining data from multiple sources”. In: *American Journal of Physical Anthropology* 164.1, pp. 62–75. DOI: <https://doi.org/10.1002/ajpa.23257>.
- Rocha, Allan, Usman Alim, Julio Daniel Silva and Mario Costa Sousa (2017). “Decal-Maps: Real-Time Layering of Decals on Surfaces for Multivariate Visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 23.1, pp. 821–830. DOI: 10.1109/TVCG.2016.2598866.
- Rohlf, F. and Leslie Marcus (Apr. 1993). “A Revolution in Morphometrics”. In: *Trends in Ecology & Evolution* 8, pp. 129–132. DOI: 10.1016/0169-5347(93)90024-J.
- Sander, Pedro V., Diego Nehab and Joshua Barczak (July 2007). “Fast triangle reordering for vertex locality and reduced overdraw”. In: *ACM Transactions on Graphics* 26.3, 89:1–89:9. DOI: 10.1145/1276377.1276489.
- Santerre, Benjamin, Masaki Abe and Taichi Watanabe (2020). “Improving GPU Real-Time Wide Terrain Tessellation Using the New Mesh Shader Pipeline”. In: *Nico-*

- graph International (NicoInt 2020)*, pp. 86–89.
URL: <https://doi.org/10.1109/NicoInt50878.2020.00025>.
- Sanzharov, V. V., A. I. Gorbonosov, V. A. Frolov and A. G. Voloboy (2019). “Examination of the Nvidia RTX”. In: *International Conference on Computer Graphics and Vision (GraphiCon 2019)*. Vol. 2485, pp. 7–12. DOI: 10.30987/graphicon-2019-2-7-12.
- Schmidt, Johanna, Reinhold Preiner, Thomas Auzinger, Michael Wimmer, M. Eduard Gröller and Stefan Bruckner (2014). “YMCA — Your mesh comparison application”. In: *2014 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pp. 153–162. DOI: 10.1109/VAST.2014.7042491.
- Schnack, Alexander, Malcolm J. Wright and Judith L. Holdershaw (2021). “Does the locomotion technique matter in an immersive virtual store environment? – Comparing motion-tracked walking and instant teleportation”. In: *Journal of Retailing and Consumer Services* 58, p. 102266. ISSN: 0969-6989. DOI: <https://doi.org/10.1016/j.jretconser.2020.102266>.
- Schonberger, Johannes L. and Jan-Michael Frahm (June 2016). “Structure-From-Motion Revisited”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.
- Sellers, Graham and John Kessenich (2017). *Vulkan Programming Guide: The Official Guide to Learning Vulkan*. Addison-Wesley.
- Shearer, Brian M., Siobhán B. Cooke, Lauren B. Halenar, Samantha L. Reber, Jeannette E. Plummer, Eric Delson and Melissa Tallman (Nov. 2017). “Evaluating causes of error in landmark-based data collection using scanners”. In: *PLOS ONE* 12.11, pp. 1–37. DOI: 10.1371/journal.pone.0187452.
- Shoemake, Ken (1992). “ARCBALL: A user interface for specifying three-dimensional orientation using a mouse”. In: *Graphics interface*. Vol. 92, pp. 151–156.
- Sholts, Sabrina B., L. Flores, Phillip L. Walker and Sebastian K. T. S. Wärmländer (2011). “Comparison of coordinate measurement precision of different landmark types on human crania using a 3D laser scanner and a 3D digitiser: implications for applications of digital morphometrics”. In: *International Journal of Osteoarchaeology* 21.5, pp. 535–543. DOI: <https://doi.org/10.1002/oa.1156>.
- Sicat, Ronell, Jiabao Li, Junyoung Choi, Maxime Cordeil, Won Ki Jeong, Benjamin Bach and Hanspeter Pfister (2019). “DXR: A toolkit for building immersive data visualizations”. In: *IEEE Transactions on Visualization and Computer Graphics* 25.1, p. 8440858. DOI: 10.1109/TVCG.2018.2865152.
- Singh, Karan (2021). *Personal communications*.

- Slater, Mel, Martin Usoh and Anthony Steed (Jan. 1994). "Depth of Presence in Virtual Environments". In: *Presence: Teleoper. Virtual Environ.* 3.2, pp. 130–144. ISSN: 1054-7460. DOI: 10.1162/pres.1994.3.2.130.
- (Sept. 1995). "Taking Steps: The Influence of a Walking Technique on Presence in Virtual Reality". In: *ACM Trans. Comput.-Hum. Interact.* 2.3, pp. 201–219. ISSN: 1073-0516. DOI: 10.1145/210079.210084.
- Slater, Mel and Sylvia Wilbur (1997). "A framework for immersive virtual environments (FIVE): Speculations on the role of presence in virtual environments". eng. In: *Presence: Teleoperators and Virtual Environments* 6.6, pp. 603–616. ISSN: 15313263, 10547460. DOI: 10.1162/pres.1997.6.6.603.
- Slice, Dennis E. (2005). "Modern Morphometrics". In: *Modern Morphometrics in Physical Anthropology*. Boston, MA: Springer US, pp. 1–45. DOI: 10.1007/0-387-27614-9_1.
- Sloan, Peter-Pike J, William Martin, Amy Gooch and Bruce Gooch (2001). "The lit sphere: A model for capturing NPR shading from art". In: *Graphics interface*. Vol. 2001. Citeseer, pp. 143–150.
- Soldati, Marco, Mario Doulis and Andre Csillaghy (2007). "SphereViz - Data Exploration in a Virtual Reality Environment". In: *2007 11th International Conference Information Visualization (IV '07)*, pp. 680–683. DOI: 10.1109/IV.2007.105.
- Stalling, Detlev and Hans-Christian Hege (1997). "LIC on Surfaces". In: *Texture Synthesis with Line Integral Convolution*, pp. 51–64.
- Stauffert, Jan-Philipp, Florian Niebling and Marc Erich Latoschik (2020). "Latency and cybersickness: Impact, causes and measures. A review". In: *Frontiers in Virtual Reality* 1, p. 31. DOI: 10.3389/frvir.2020.582204.
- Stefani, Caroline, Adam Lacy-Hulbert and Thomas Skillman (2018). "ConfocalVR: Immersive Visualization for Confocal Microscopy". In: *Journal of Molecular Biology* 430.21, pp. 4028–4035. ISSN: 0022-2836. DOI: <https://doi.org/10.1016/j.jmb.2018.06.035>.
- Steinicke, Frank, Gerd Bruder, Klaus Hinrichs, Anthony Steed and Alexander L. Gerlach (2009). "Does a Gradual Transition to the Virtual World increase Presence?" In: *2009 IEEE Virtual Reality Conference*, pp. 203–210. DOI: 10.1109/VR.2009.4811024.
- STEUER, J (1992). "DEFINING VIRTUAL REALITY - DIMENSIONS DETERMINING TELEPRESENCE". eng. In: *Journal of Communication* 42.4, pp. 73–93. ISSN: 14602466, 00219916.
- Subtil, Nuno, Matthew Rusch and Ivan Fedorov (June 2019). *Tips and Tricks: Vulkan Dos and Don'ts*. NVIDIA Developer Blog. URL: <https://developer.nvidia.com/blog/vulkan-dos-donts/>.

- Sutherland, Ivan E. (1965). "The Ultimate Display". In: *Proceedings of the IFIP Congress*, pp. 506–508.
- (1968). "A Head-Mounted Three Dimensional Display". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I. AFIPS '68* (Fall, part I). San Francisco, California: Association for Computing Machinery, pp. 757–764. ISBN: 9781450378994. DOI: 10.1145/1476589.1476686.
- Tang, Ying, Huamin Qu, Yingcai Wu and Hong Zhou (2006). "Natural Textures for Weather Data Visualization". In: *Tenth International Conference on Information Visualisation (IV'06)*, pp. 741–750. DOI: 10.1109/IV.2006.77.
- Teather, Robert, Andriy Pavlovych, Wolfgang Stuerzlinger and I. MacKenzie (Jan. 2009). "Effects of tracking technology, latency, and spatial jitter on object movement". In: *Symposium on 3D User Interfaces (3DUI 2009)*. IEEE, pp. 43–50. DOI: 10.1109/3DUI.2009.4811204.
- Ten Berge, Jos M. F. (1977). "Orthogonal procrustes rotation for two or more matrices". In: *Psychometrika* 42, pp. 267–276. DOI: 10.1007/BF02294053.
- Thomas, J. J. and K. A. Cook (2006). "A visual analytics agenda". In: *IEEE Computer Graphics and Applications* 26.1, pp. 10–13. DOI: 10.1109/MCG.2006.5.
- Toledo, Thiago and Waldemar Celes (2011). "Visualizing 3D Flow of Black-Oil Reservoir Models on Arbitrary Surfaces Using Projected 2D Line Integral Convolution". In: *2011 24th SIBGRAPI Conference on Graphics, Patterns and Images*, pp. 133–140. DOI: 10.1109/SIBGRAPI.2011.45.
- Treisman, Michel (1977). "Motion Sickness: An Evolutionary Hypothesis". In: *Science* 197.4302, pp. 493–495. DOI: 10.1126/science.301659. eprint: <https://www.science.org/doi/pdf/10.1126/science.301659>.
- Ulinski, Amy, Catherine Zanbaka, Zachary Wartell, Paula Goolkasian and Larry F. Hodges (2007). "Two Handed Selection Techniques for Volumetric Data". In: *2007 IEEE Symposium on 3D User Interfaces*. DOI: 10.1109/3DUI.2007.340782.
- Unity (2017). *2017.3.0f3 Release Notes (diff since 2017.3.0f2 at the end)*". Accessed 19 Oct. 2022.
URL: <https://unity3d.com/unity/whats-new/unity-2017.3.0>.
- Unity Graphics Team (2020). *Personal communications*.
- Unterguggenberger, J., B. Kerbl, J. Pernsteiner and M. Wimmer (2021). "Conservative Meshlet Bounds for Robust Culling of Skinned Meshes". In: *Computer Graphics Forum* 40.7, pp. 57–69.
URL: <https://doi.org/10.1111/cgf.14401>.
- Usher, Will, Pavol Klacansky, Frederick Federer, Peer-Timo Bremer, Aaron Knoll, Jeff Yarch, Alessandra Angelucci and Valerio Pascucci (2018). "A Virtual Reality Vi-

- sualization Tool for Neuron Tracing”. In: *IEEE Transactions on Visualization and Computer Graphics* 24.1, pp. 994–1003. DOI: 10.1109/TVCG.2017.2744079.
- Usoh, Martin, Kevin Arthur, Mary C. Whitton, Rui Bastos, Anthony Steed, Mel Slater and Frederick P. Brooks (1999). “Walking Walking-in-Place Flying, in Virtual Environments”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '99. USA: ACM Press/Addison-Wesley Publishing Co., pp. 359–364. ISBN: 0201485605. DOI: 10.1145/311535.311589.
- Vasylevska, Khrystyna, Hannes Kaufmann and Vasylevska Khrystyna (2014). “Influence of vertical navigation metaphors on presence”. In: *Challenging Presence- Proceedings of 15th International Conference on Presence (ISPR 2014)*, pp. 205–212.
- Vorderer, Peter, Werner Wirth, Feliz Ribeiro Gouveia, Frank Biocca, Timo Saari, Lutz Jäncke, Saskia Böcking, Holger Schramm, Andre Gysbers, Tilo Hartmann et al. (2004). “Mec spatial presence questionnaire”. In: *Retrieved Sept 18*, p. 2015.
- Wagner, Jorge, Wolfgang Stuerzlinger and Luciana Nedel (2021). “The effect of exploration mode and frame of reference in immersive analytics”. In: *IEEE Transactions on Visualization and Computer Graphics*. To appear. DOI: 10.1109/TVCG.2021.3060666.
- Walbourn, Chuck (2014). *DirectXMesh geometry processing library*. Accessed: 2022-04-16.
URL: <https://github.com/microsoft/DirectXMesh>.
- Wald, Ingo and Steven G. Parker (2019). “RTX Accelerated Ray Tracing with OptiX”. In: *ACM SIGGRAPH 2019 Courses*.
URL: <https://sites.google.com/view/rtx-acc-ray-tracing-with-optix>.
- Waltenberger, Lukas, Katharina Rebay-Salisbury and Philipp Mitteroecker (2021). “Three-dimensional surface scanning methods in osteology: A topographical and geometric morphometric comparison”. In: *American Journal of Physical Anthropology* 174.4, pp. 846–858. DOI: <https://doi.org/10.1002/ajpa.24204>.
- Wang, Yimin, Qi Li, Lijuan Liu, Zhi Zhou, Zongcai Ruan, Lingsheng Kong, Yaoyao Li, Yun Wang, Ning Zhong, Renjie Chai et al. (2019). “TeraVR empowers precise reconstruction of complete 3-D neuronal morphology in the whole brain”. In: *Nature communications* 10.1, pp. 1–9.
- Ware, Colin, Kevin Arthur and Kellogg S. Booth (1993). “Fish Tank Virtual Reality”. In: *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*. CHI '93. Amsterdam, The Netherlands: Association for Computing Machinery, pp. 37–42. ISBN: 0897915755. DOI: 10.1145/169059.169066.

- Ware, Colin and Daniel Fleet (1997). "Context sensitive flying interface". In: *Proceedings of the 1997 Symposium on Interactive 3D graphics*, 127–ff.
- Watson, B. and D. Luebke (2005). "The ultimate display: where will all the pixels come from?" In: *Computer* 38.8, pp. 54–61. DOI: 10.1109/MC.2005.274.
- Weber, Gerhard W (2015). "Virtual anthropology". In: *American journal of physical anthropology* 156, pp. 22–42.
- Weigle, C. and R.M. Taylor (2005). "Visualizing intersecting surfaces with nested-surface techniques". In: *VIS 05. IEEE Visualization, 2005*. Pp. 503–510. DOI: 10.1109/VISUAL.2005.1532835.
- Weyrich, Tim, Mark Pauly, Richard Keiser, Simon Heinzle, Sascha Scandella and Markus H Gross (2004). "Post-processing of Scanned 3D Surface Data." In: *PBG*, pp. 85–94.
- Wihlidal, Graham (2016). *Optimizing the Graphics Pipeline with Compute*. Game Developer Conference 2016. Accessed: 2022-04-16.
URL: <https://www.gdcvault.com/play/1023463/Optimizing-the-Graphics-Pipeline-With>.
- Wijk, Jarke J. van (2002). "Image Based Flow Visualization". In: *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH '02*. San Antonio, Texas: Association for Computing Machinery, pp. 745–754. ISBN: 1581135211. DOI: 10.1145/566570.566646.
- Wikipedia (2021). *Blender (Software) - Suzanne*.
URL: [https://en.wikipedia.org/wiki/Blender_\(software\)#Suzanne](https://en.wikipedia.org/wiki/Blender_(software)#Suzanne).
- Wiley, D. F., N. Amenta, D. A. Alcantara, D. Ghosh, Y. J. Kil, E. Delson, W. Harcourt-Smith, F. J. Rohlf, K. St. John and B. Hamann (2005). "Evolutionary morphing". In: *Visualization Conference (VIS 05)*. IEEE, pp. 431–432. DOI: <https://doi.org/10.1109/VISUAL.2005.1532826>.
- Wilson, Preston Tunnell, William Kalescky, Ansel MacLaughlin and Betsy Williams (2016). "VR Locomotion: Walking Walking in Place Arm Swinging". In: *Proceedings of the 15th ACM SIGGRAPH Conference on Virtual-Reality Continuum and Its Applications in Industry - Volume 1. VRCAI '16*. Zhuhai, China: Association for Computing Machinery, pp. 243–249. ISBN: 9781450346924. DOI: 10.1145/3013971.3014010.
- Wirth, Werner, Tilo Hartmann, Saskia Böcking, Peter Vorderer, Christoph Klimmt, Holger Schramm, Timo Saari, Jari Laarni, Niklas Ravaja, Feliz Ribeiro Gouveia, Frank Biocca, Ana Sacau, Lutz Jäncke, Thomas Baumgartner and Petra Jäncke (2007). "A process model of the formation of spatial presence experiences". eng. In: *Media Psychology* 9.3, pp. 493–525. ISSN: 1532785x, 15213269. DOI: 10.1080/15213260701283079.

- Wolfartsberger, Josef (2019). “Analyzing the potential of Virtual Reality for engineering design review”. In: *Automation in Construction* 104, pp. 27–37. DOI: <https://doi.org/10.1016/j.autcon.2019.03.018>.
- Wu, Enhua and Youquan Liu (2008). “Emerging technology about GPGPU”. In: *IEEE Asia Pacific Conference on Circuits and Systems (APCCAS 2008)*, pp. 618–622. DOI: 10.1109/APCCAS.2008.4746099.
- Yang, Yonggao, J.X. Chen and M. Beheshti (2005). “Nonlinear perspective projections and magic lenses: 3D view deformation”. In: *IEEE Computer Graphics and Applications* 25.1, pp. 76–84. DOI: 10.1109/MCG.2005.29.
- Yi, Li, Vladimir G. Kim, Duygu Ceylan, I-Chao Shen, Mengyan Yan, Hao Su, Cewu Lu, Qixing Huang, Alla Sheffer and Leonidas Guibas (Nov. 2016). “A Scalable Active Framework for Region Annotation in 3D Shape Collections”. In: *ACM Trans. Graph.* 35.6. ISSN: 0730-0301. DOI: 10.1145/2980179.2980238.
- Yimam, Seid Muhie, Chris Biemann, Ljiljana Majnaric, Šefket Šabanović and Andreas Holzinger (2015). “Interactive and Iterative Annotation for Biomedical Entity Recognition”. In: *Brain Informatics and Health*. Ed. by Yike Guo, Karl Friston, Faisal Aldo, Sean Hill and Hanchuan Peng. Cham: Springer International Publishing, pp. 347–357. ISBN: 978-3-319-23344-4.
- Zeller, Cyril, Randy Fernando, Matthias Wloka and Mark Harris (2004). “Programming Graphics Hardware”. In: *Eurographics 2004 - Tutorial*. DOI: 10.2312/egt.20041034.
- Zhai, Shumin (Nov. 1998). “User Performance in Relation to 3D Input Device Design”. In: *SIGGRAPH Comput. Graph.* 32.4, pp. 50–54. ISSN: 0097-8930. DOI: 10.1145/307710.307728.
- Zhao, Jiayan, Jan Oliver Wallgrün, Peter C. LaFemina, Jim Normandeau and Alexander Klippel (2019). “Harnessing the power of immersive virtual reality-visualization and analysis of 3D earth science data sets”. In: *Geo-spatial Information Science* 22.4, pp. 237–250. DOI: 10.1080/10095020.2019.1621544.
- Zhao, Jingbo, Robert S. Allison, Margarita Vinnikov and Sion Jennings (2017). “Estimating the motion-to-photon latency in head mounted displays”. In: *IEEE Virtual Reality (VR 2017)*, pp. 313–314. DOI: 10.1109/VR.2017.7892302.
- Zhou, Qian-Yi, Jaesik Park and Vladlen Koltun (2018). *Open3D: A Modern Library for 3D Data Processing*. arXiv:1801.09847 [cs.CV]. arXiv: 1801.09847.