Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Bendik Nilsen Brunvoll

Implementing Volume Rendering Optimizations for Real-Time Performance Directly on Microsoft's HoloLens 2

Master's thesis in Informatics
Supervisor: Gabriel Kiss
June 2022

NTNU
Norwegian University of
Science and Technology

Bendik Nilsen Brunvoll

Implementing Volume Rendering Optimizations for Real-Time Performance Directly on Microsoft's HoloLens 2

Master's thesis in Informatics
Supervisor: Gabriel Kiss
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Implementing Volume Rendering Optimizations for Real-Time Performance Directly on Microsoft's HoloLens 2

Bendik Nilsen Brunvoll

June 13, 2022

Preface

As I am writing this, there are approximately 24 hours left of my time as a student at NTNU. I would first of all like to thank my supervisor, Gabriel Kiss, and Frank Lindseth, who has acted as an unofficial co-supervisor. I would also like to thank my girlfriend, Tonje Toseth Haanshus, for supporting me through this whole process. I extend my thanks to Lachlan Deakin for his answers to my questions regarding his work, and Unity forum user bgolus for having helped many before me, and thus me indirectly, with their questions regarding shaders.

During my time at NTNU I have met many new friends, I have played on stage in my student organization Online's band, Output, and written for their magazine, and I have been part of a few projects through Hackerspace NTNU. My time as a student comes to an end, but a new chapter awaits as a graduate hardware engineer at Arm in Trondheim. I would like to thank everyone I have become friends with these last six years for making me grow both as a person and making my life as a student amazing.

Sammendrag

Det er ofte nødvendig å visualisere tredimensjonale anatomiske strukturer i medisinsk bildediagnostikk. En vanlig algoritme for dette er *volumgjengivelse*. 3D-strukturen blir vanligvis gjengitt på en vanlig 2D-skjerm. Dette begrenser dybdeføelsen. Utvidet virkelighet (AR) kan bli brukt for å øke dybdeføelsen ved å "sette ut" det digitale bildet i virkeligheten (sett gjennom en skjerm). Volumgjengivelse er en beregningsintensiv algoritme, og for applikasjoner i utvidet virkelighet blir disse beregningene vanligvis gjort av en ekstern PC.. Dette fører til en viss forsinkelse på grunn av kommunikasjon mellom AR-brillene og den eksterne maskinen, i tillegg til å gi en mer komplisert arkitektur på grunn av denne kommunikasjonen. Dette prosjektet har som mål om å implementere volumgjengivelse direkte på Microsoft sine AR-briller: HoloLens 2. For å oppnå en ytelse som er tilstrekkelig for samhandling med det gjengitte volumet blir flere optimaliseringer implementert og sammenlignet. Det beste volum-optimaliseringsparet blir testa på to ekkokardiologer og en anestesilege. Ytelsen er avhengig av både oppløsningen til volumet og hvor stor del av skjermen bildet dekker. Ytelsen til et ultralydvolum er tilstrekkelig for samhandling og kan brukes som basis for videre utvikling. Resultatene viser potensiale for en mer realistisk applikasjon med nødvendige samhandlingsmuligheter uten behov for kommunikasjon med en ekstern maskin.

Abstract

In medical imaging it is often necessary to visualize complex three-dimensional anatomical structures. A common algorithm for this use, especially when iso-surfacing and surface-based visualization are not available, is *volume rendering*. The 3D structure is commonly rendered to a standard two-dimensional display, limiting the depth perception of the image. Augmented reality can be used to alleviate this reduced depth perception by superimposing the digital image on the physical world. Volume rendering is a computationally intensive algorithm, and for augmented reality applications the rendering is often offloaded to a stronger, separate computer. Doing this introduces a certain delay from communication overhead and a more complex architecture. This project aims to implement an instantiation of volume rendering with a focus on ultrasound and computed tomography (CT) directly on Microsoft's augmented reality head-mounted device, HoloLens 2. In order to achieve a frame rate that allows for interaction with the rendered volume, several optimizations are implemented and compared. The best performing volume-optimization pair is also tested on two echocardiologists and an anesthesiologist. The resulting frame rate is dependent on the resolution of the volume and how much of the screen it covers; therefore rendering times for a set of common resolutions and distances are presented. The frame rate for the ultrasound volume was sufficient for interaction and can be used as a basis for further work. The results show potential for a more realistic application with necessary interaction options, and with a no communication overhead and less complex architecture than most existing solutions.

Contents

Figures

Tables

Code Listings

Chapter 1

Introduction

This project is a natural continuation of a preparatory project performed from August to December 2021. In that project, a simple volume renderer without optimizations or transfer functions was implemented, and a literature search was performed. The introduction and background will include many similarities to this project, but it is not necessary to read the report to understand this thesis [1].

1.1 Purpose

Medical imaging often involves visualizing anatomical structures from scans in 3D. This goes for several types of scans, both computed tomography (CT), magnetic-resonance imaging (MRI), and ultrasound. One such visualization algorithm is *Volume rendering* [2], which is a ray-tracing algorithm that can be used to visualize 3D data directly without the need of extracting iso-surfaces from pre-defined values. These 3D visualizations are typically rendered on standard 2D displays, which have the inherent attribute of lacking depth. This limits the depth perception of the user [3], meaning the mapping between 2D and 3D has to be done mentally by the user. While not a big problem for experts who look at, say, ultrasound images of specific structures regularly, for example surgeons and anesthesiologists also need these images in their work. Furthermore, the experts need to explain the anatomic details using these images. Augmented reality (AR) revolves around placing digital objects in the physical world, and has been shown to improve depth perception [4]. There are companies that work with increasing the depth perception of medical imaging using augmented reality; among them are HoloCare [5], MedApp [6], and Siemens [7]. There has also been conducted research projects regarding medical imaging in AR. Sauer et al. made a head-mounted device (HMD) that could be used to superimpose medical data on the patient [8]. Kutter et al. used Sauer et al.'s HMD as a basis for an HMD with a transparent screen and two cameras per eye, as well as several cameras in the room. Their goal was to retain quality without sacrificing performance [9]. Bichlmeier et al. rendered their visualizations on the skin of the patient with the aim of improving the "perception of 3D medical imaging data (mimesis) in context

to the patient's own anatomy" [10]. De Ridder et al. suggest that extended reality may be used to reduce "visual clutter" and allow "users to navigate the data abstractions in a 'natural' way" that lets them keep their focus [11].

Medical visualization can be quite heavy, especially when it comes to volume rendering. Both previous and contemporary research mostly uses a separate computer to perform the computationally intense rendering. This also goes for the implementations made by companies. Rendering on a separate machine is easier because it requires less intense optimization, but it introduces both a delay from communication overhead and a more complex architecture. Since volume rendering has been implemented, optimized, and tested on desktops before, this project focuses on implementing the rendering *directly* on Microsoft's augmented reality HMD HoloLens 2. The main beneficiaries of this project are those who need 3D visualization of medical data in their work and those who plan on implementing volume rendering in AR and wishes to have a more simple and portable architecture without the need of an advanced rendering station with a remote server. As a bonus, it is also less expensive, as the system requires less hardware. Indirectly, patients will also benefit from the research due to the intended increased quality of analyses.

The main purpose of this project is to visualize patient-specific ultrasound data and computed tomography scans and combine them with geometric models representing anatomical structures of interest. This visualization is done by using the volume rendering algorithm [2], computed with the HoloLens 2's hardware. It is important that the structures that are visualized are shown correctly and with enough quality to be recognized. As one of the main motivations behind using augmented reality, it is also important that the depth perception is actually improved. And most importantly, it is important that the application is usable. For this, the performance of the application should be good enough to allow interaction while also not being uncomfortable to use. According to Microsoft, the target frame rate is 60 frames per second (FPS), but that is not the focus of this project. This master project's research questions are therefore:

RQ1: How can volume rendering be implemented on an augmented reality device with a framerate of at least 15 FPS?

RQ2: Does HoloLens 2 improve 3D depth perception of the volume rendered content when compared to a standard display?

RQ3: Is the quality of the rendered content sufficient to perceive important anatomic structures?

1.2 Contribution

Most projects and products regarding volume rendering on augmented reality head-mounted devices perform the actual rendering on a separate machine. This project, however, shows potential for implementing the algorithm directly on Microsoft's HoloLens 2. The code base has been included in Chapter 4, and is part of

the contribution of this project. In addition to the code, the project has included profiling of different optimizations, comparing them. For smaller volumes more interaction can still be added, but the depth perception is reported to be better than on standard 2D displays, and the quality is sufficient to perceive important anatomic structures. This project can be used as a starting point to further develop a way to perform volume rendering on augmented reality devices with minimal setup and communication overhead.

1.3 Overview

Chapter 2 explores the history of volume rendering and background theory for discussing the results. Chapter 3 explains the methods of the project, and Chapter 4 contains details regarding the actual implementation. Chapter 5 presents the profiling results and key takeaways from the expert testing; Chapter 6 puts the results in context of the background theory, discusses the limitations and methods, and provides suggestions for further work. Chapter 7 concludes the project.

Chapter 2

Background

2.1 The principles of volume rendering

In 1988 Drebin, Carpenter, and Hanrahan proposed the volume rendering algorithm. One of its main advantages over previous visualization algorithms is that it minimizes computational artifacts such as aliasing and quantization. The main idea of their volume rendering algorithm is as follows, using CT data as an example:

The input CT data volume is converted into a set of *material percentage volumes*. These volumes will represent how much of a specific material is present at a given area in space. Each material is mapped to their own color and have their own opacity according to the desired classification technique. The color of these volumes is calculated by multiplying the assigned color with the material percentage. This can also be done for the opacity. These new color and opacity volumes can then be combined into a single *composite color and opacity volume*, which is the product of the color and opacity volumes.

The algorithm also generates a *density volume*, which is based on the assigned density value of each material. To detect the boundaries of each material, a 3D-gradient is calculated using the density volume. The outputs from this operation are the magnitude and x, y, and z directions of the gradients; all of these are represented by their own volumes. The *shaded color volume* combines the gradient volumes and the composite color and opacity volume to apply a lighting model to the data. Then, the volume will be transformed to fit in the viewport. This can then be used to produce the final output image [12]. Fig. 2.1 is a visualization of the algorithm, and is from the original paper.

A different way of performing volume rendering is by using ray casting [2], which is currently the most used method. Again using CT as an example: The original CT data is treated as a 3D array of *voxels* containing one CT value each. For simplicity, this example will follow Levoy's example. The array is therefore an $NxNxN$ cube. The image that will be produced consists of PxP pixels. For each pixel, a "ray" is shot through the scene and thus the cube as well. In Levoy's implementation, colors and opacities are accumulated by sampling the CT data at

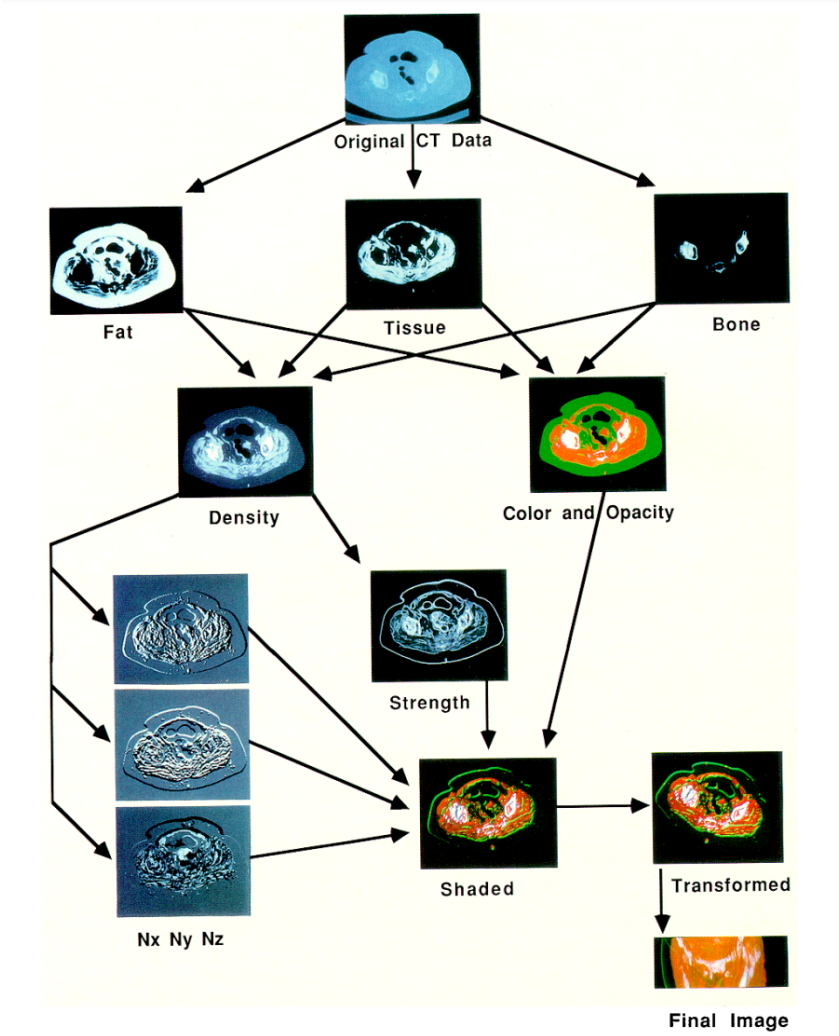


Fig. 2.1. The volume rendering algorithm visualized [12, Fig. 1]

equidistant sampling points along the ray with front-to-back blending. The colors and opacities of each voxel is found by trilinearly interpolating from the eight surrounding voxels' colors and opacities [2]. Fig. 2.2 from Levoy's paper shows the coordinate systems used in volume rendering: The resulting image with pixels (image space coordinates), the cube representing the 3D CT array (object space coordinates), and the scene (world space coordinates).

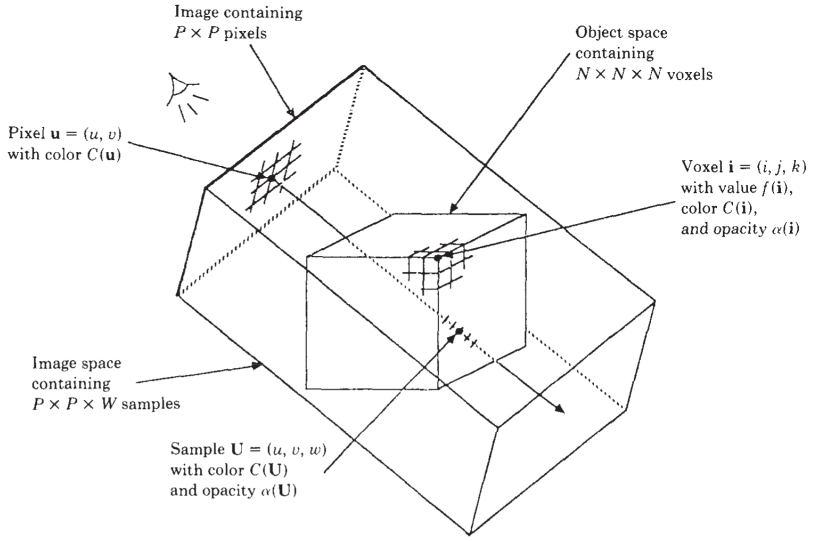


Fig. 2.2. "Coordinate systems used during volume rendering" [2, Fig. 2]

In the same paper Levoy also presented two optimizations that are still used today in different forms. The first optimization exploits the fact that volumes will often contain regions of empty space; Levoy called this "hierarchical spatial enumeration" [2], but it is often called empty space skipping or leaping nowadays. These regions will not contribute to the final image, and can therefore safely be skipped. Levoy's way of doing this was using an *octree*-implementation. This can be viewed as a pyramid hierarchy of volumes, with increasingly coarse granularity. Each level that is less detailed only signifies whether there is a non-empty voxel in this region, with 1 meaning there is something and 0 meaning the entire region is empty and can be skipped. If the sampled region is 1, the algorithm will go down a level and try again. If it is 0 it will simply skip to the next region. This way the algorithm can potentially save a lot of computations, and thus also a lot of time [2]. Fig. 2.3 comes from Levoy's paper and visualizes hierarchical spatial enumeration in a 2D grid. The distance between each sample point decreases as the level increases. The figure shows an example with initial sampling (the shaded cells), followed by increasingly larger leaps. When the ray enters a region that contains non-empty voxels, the algorithm gradually decreases the distance before it starts sampling the voxels.

The second optimization is "adaptive termination of ray tracing" [2], also known as "early ray termination". At some point further samples will not contrib-

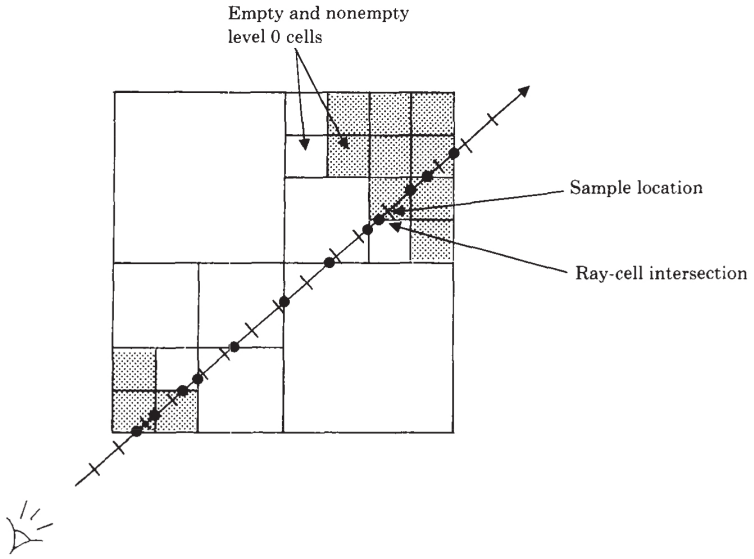


Fig. 2.3. "Ray tracing of hierarchical enumeration" [2, Fig. 4]

ute much to the final color, though at which point this is depends on the dataset. When the opacity reaches the set threshold, sampling is stopped and the color is determined. Since this value is dependent on the dataset, it is beneficial for the user to be able to set this threshold. A high threshold will lead to higher image quality, whereas a low threshold will lead to a better performance [2].

As the years passed, computer hardware evolved. This evolution will be explained in further detail in Section 2.5. When Levoy presented his work, multicore processors were not available, and graphics processing units (GPU) were not as programmable [13]. As the texture mapping hardware was improved and GPUs became more general purpose, more optimizations were made possible. In 1994 Cabral, Cam, and Foran compared the operations of tomographic reconstruction and visualization, noting that they are similar in both a mathematical and an algorithmic sense. These two operations used to be handled by two different algorithms, but both algorithms can be converted to an algorithm that uses hardware accelerated texture mapping and summing buffers. By re-writing the algorithms to use these qualities, Cabral, Cam, and Foran were able to prove possibility for a "single hardware accelerated solution", achieving a speedup of about 100 [14]. As a side note, the classification of the different materials is nowadays done using *transfer functions*, which are mathematical functions that map a value from the volume to a color and an opacity. As texture mapping and texture memory technology improved, the transfer functions could be implemented as lookup-table textures.

Volume rendering using 3D-textures has also been optimized further. Krüger and Westermann addressed optimizing 3D texture volume rendering by implementing early ray termination and empty space skipping in 2003 on GPU [15].

These two techniques were already well-known optimizations for ray-casting, as seen in Levoy's work [2]. By terminating rays early, i.e. at a point where they will not contribute significantly enough anymore, and ignoring empty voxels, one can save valuable compute power that can be used for more useful computations. Krüger and Westermann split their *shader program*, i.e. a program that runs on a GPU, into three *passes*:

1. Finding the ray entry
2. Determining the ray direction
3. Ray traversal

Their way of implementing empty space skipping was to introduce a second 2D texture with 1/8 dimension of the original volume in each direction. In the R and G components of the texture, they would store the minimum and maximum values of that region. If the region is empty, it can be skipped [15].

Further research has been done by Li, Mueller, and Kaufman, who proposed partitioning the volume into smaller sub-volumes, and skipping those that only contained empty voxels according to their transfer function [16]. Also, they introduced a new algorithm for computing the intersection between the volume and the slicing plane, and an "orthogonal opacity map" which simplifies transforming from volume coordinates to opacity map coordinates. This was used to perform more efficient occlusion clipping, which they used as an alternative to early ray termination [16]. An occlusion technique that has been used in augmented reality is using the hands of a surgeon when they are in front of the holographic visualization, as done by Kutter et al. [9].

In 2020, Lachlan Deakin and Mark Knackstedt published their work on an empty space skipping technique which uses *Chebyshev distance maps* [17]. The Chebyshev distance, is the maximum difference between two points' components defined by Equation 2.1 [18, p. 324].

$$||x - y||_\infty = max\{|x_1 - y_1|, |x_2 - y_2|, ..., |x_n - y_n|\} \tag{2.1}$$

Deakin and Knackstedt's base algorithm follows Levoy's ray casting formulation, with rays intersecting a volume being sampled at n equidistant points. Using the transfer function, they could determine what regions contain only empty space. This information is used to create an *occupancy map*. This is implemented as a texture with dimensions equal to the original volume's dimensions divided by block size B. A value of 0 means that the region is occupied, and a value of 255 means that the region is empty. This occupancy map is then transformed in three stages; one per axis. These transformations scan the occupancy map (or the output from the previous transformation) in order to find the Chebyshev distance to the next occupied region. Deakin and Knackstedt also implemented *anisotropic* Chebyshev distance maps, which take ray direction into account. The previously mentioned distance maps would lead to extra samples of the distance maps when leaving occupied regions, but the anisotropic maps would allow the ray to jump greater distances in this case [17].

One of the motivations behind Deakin and Knackstedt's research was to optimize volume rendering for extended reality (XR), i.e. virtual and augmented reality. The reason behind this is that XR provides better depth perception than traditional screens. Their work was not tested on an XR device, however [17].

2.2 Augmented Reality

"Augmented reality is a system that enhances the real world by superimposing computer-generated information on top of it" [19]. This means that instead of being completely immersed in a virtual world, which is the case of virtual reality (VR), augmented reality (AR) places digital elements in the real world akin to holograms. There is (and has been) a lot of research and development going into AR, including medical imaging [5, 6], education [20, 21], collaboration[22, 23], and serious games [24].

Nowadays, augmented reality is available on several platforms; AR headsets, web-based applications, and mobile phones are all capable of running augmented reality. AR headsets are most interesting for this project's purposes, as they allow more or less hands-free interaction. For example, Microsoft HoloLens 2 is capable of running voice commands, and since it is a head-mounted device the user may look at it without occupying one hand as they may have needed with a phone. One drawback of AR headsets nowadays is the reduced computing power compared to desktop computers.

2.3 Volume rendering in augmented reality

HoloCare is a Norwegian company that started as a collaboration between Sopra Steria and Oslo University Hospital Intervention Centre. They make holographic medical applications for HoloLens with the goal of simplifying processes and improving the outcomes [5].

MedApp is a Polish company that specializes in developing holographic applications for the medical field, similar to HoloCare. The project Carnalife Holo visualizes anatomical structures from DICOM files in augmented reality. The visualized models can be interacted with using both voice commands and hand gestures [6].

Both of these companies perform the rendering on a separate computer, leading to a short latency when communicating between it and the HoloLens, as well as a more complicated architecture.

In 2008, Kutter et al. presented their work on using a "high quality hardware accelerated volume renderer" [9] in a medical AR framework. At the time their work was being done, real-time medical AR applications often sacrificed quality and complexity. In an effort to retain both quality and complexity, Kutter et al. used optimizations previously proposed by other researchers and combined them in a way that made sense for augmented reality.

This research was done before today's commercial AR headsets like HoloLens, so Kutter et al. used a custom setup. Their head-mounted device (HMD) included a transparent screen with two color cameras per eye. A "Dual Intel Xeon, 3.2Ghz, 1.8Gb Ram" and "Nvidia Geforce 8800 Ultra, 768Mb onboard memory" were used for computation [9]. In addition to this, they also placed cameras around in the room and on the HMD itself for tracking purposes. Fig. 2.4 shows results from Kutter et al.'s work. The left image shows "Focus and Context rendering with shaded volume rendering", and the right image shows direct volume rendering [9, Fig. 2]

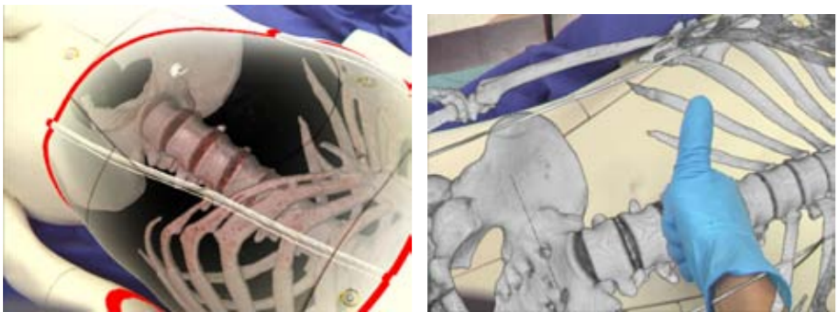


Fig. 2.4. Renders by Kutter et al.'s solution, taken from their original paper [9, Fig. 2]

In 2019 Anders Tasken, Erlend Barstad, and Sondre Tagestad worked on a project similar to this one as part of a university course. Their goal was to implement and evaluate volume rendering on the original HoloLens. In order to apply the volume rendering algorithm in Unity, they had to first convert the raw data to a 3D texture. Their volume renderer was ray-casting implemented as a shader program which would perform n intersection calculations per ray; similar to Levoy's base algorithm [2]. The color value was dependent on the alpha value in each intersection, calculated by accumulation.

The result of their project was that the data was visualized correctly, though it suffered from performance issues. The potential improvements they identified included rewriting the shader with an optimized algorithm and using the HoloLens 2 instead due to its superior hardware. They also suggested adding colors to make the model easier to understand [25]. Fig. 2.5 shows a volume rendered CT scan of a pig on a HoloLens 1.

In 2002, Sauer et al. developed an AR system that superimposed medical graphics, i.e. MR images, onto a "video view of the patient" [8]. The user would wear a custom head-mounted display that would act as the medium to place the graphics on the patient. Rendering was performed on a separate computer, and transmitted with a delay of around 0.1 seconds. To improve the application, they also added instrument tracking [8]. Sauer et al.'s system was the basis for both Kutter et al.'s [9] and Bichlmeier et al.'s [10] hardware. Fig. 2.6 shows the head-mounted device Sauer et al. developed [8, Fig. 1].

Bichlmeier et al. presented their "Hybrid In-Situ Visualization Method for Improving Multi-Sensory Depth Perception in Medical Augmented Reality" [10] in

Fig. 2.5. Visualized CT scan of a pig from Tasken, Barstad, and Tagestad's report [25, Fig. 10]

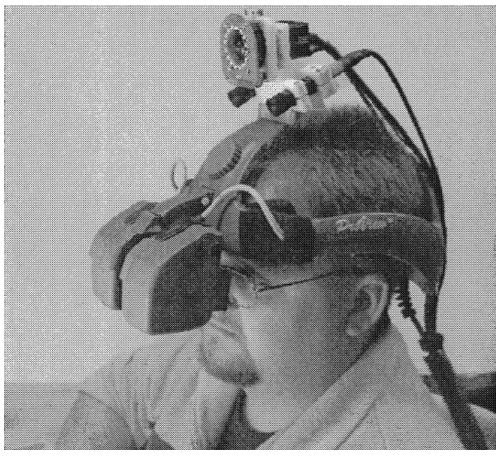


Fig. 2.6. The head-mounted device developed by Sauer et al., image taken from the original paper [8, Fig. 1]

2007. Their objective was to use AR for pre-op diagnoses, planning, and navigation. To accomplish this, they used the head-mounted display presented by Sauer et al., and identified three conditions their visualization would have to satisfy [10]:

1. The visualization must provide a non-restricted view of the region of interest
2. The data must be implemented in a way that makes distances within the scene be perceived intuitively
3. During operation, the surgical instruments must be integrated with the AR application to get intuitive visual feedback from interaction

Bichlmeier et al.'s solution was a technique to render the patient data on the skin of the patient themselves. The head-mounted device would send images to a computer that also got pre-op data and data from the tracked instruments. In addition to this, the computer also got position data from the head-mounted device and an array of infrared cameras hanging from the ceiling. The computer would then output "spatial registered 3D data" back to the device [10]. The concept is shown with a phantom showcase in Fig. 2.7 where they also include a tracked instrument. This image is from the original paper [10, Fig. 10 b].

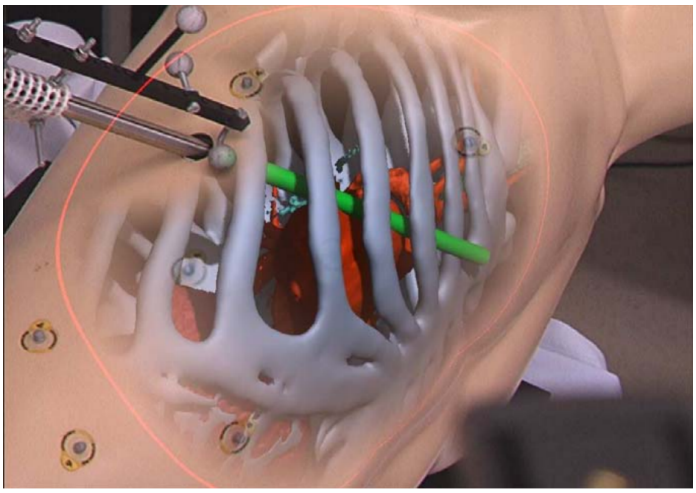


Fig. 2.7. Phantom showcase from Bichlmeier et al.'s original paper [10, Fig. 10 b]

2.4 Medical imaging

Medical imaging is the process of scanning and visualizing necessary medical data for analysis or medical intervention. This includes for example a fetus, a brain, or a heart. Three very common techniques for this are ultrasound, magnetic resonance imaging (MRI), and computed tomography (CT).

Ultrasound is a real-time method that is used to visualize a slice of the inside of a patient. This is quite different from MRI and CT, which are not real-time

and generate 3D-images. In principle, ultrasound is performed by the practitioner using a probe that includes both a transmitter and a receiver. The probe transmits a short ultrasonic pulse which goes through the body and is reflected by different parts of the insides. The probe receives these reflected waves, and sends them to the machine for processing. This is similar to what animals such as bats use to "see" [26, Ch. 17].

Ultrasound takes advantage of the different reflection coefficients and wave impedances of different materials to successfully differentiate between them [26, Ch. 17].

Magnetic resonance imaging uses principles of nuclear magnetism to scan the inside of a patient. Using the different *weighting*, i.e. the dominant tissue-specific MR parameter, MRI can be used to differentiate very well between different types of tissue. MRI will use either the density of protons, "the speed of recovery of the longitudinal nuclear magnetization following excitation" or "the tissue-specific fading of the MR signal" as weighting. Different configurations will be used for different application areas, as they will include different types of tissue. The data is then fed into a fast fourier transform to generate the final image [26, Ch. 23].

Computed tomography (CT) is a widely used non-invasive imaging method that uses x-rays to generate a 3D visualization of the scanned area. Since it is three-dimensional, the radiologist gains more spatial information than in a "regular" X-ray image. CT may be used for "patients with heavy trauma, fractures, and luxations" [26, Ch. 16], and is also an alternative to MRI if that is not a viable alternative for the patient. For example, MRI cannot be used if the patient is dehydrated. CT has also been used in other areas, such as forensic and archaeological applications [26, Ch. 16].

CT has evolved a lot over the years, but the general idea is that an X-ray source and an array of detectors move in a circle around the patient in such a way that they scan the area of interest. The data gathered from this process is then used to create different visualizations [26, Ch. 16]. For example, since different materials have different (known) absorbation/reflection abililites, it is possible to assign different colors to each materials in order to generate higher contrast.

2.5 Computer architecture

Computer hardware has come a long way since Levoy's initial algorithm was published. At that time multicore processors were not available [27], and GPUs did not support programmable shaders [13]. Fig. 2.8 shows the highest performing single cores per year compared to the VAX-11/780 [28, p. 3]. The graph shows a large increase in performance since the 80s due to Moore's law, i.e. that the amount of transistors per processor would double each year (amended to every two years in 1975), and Dennard scaling, i.e. power density being constant for a specified area of silicon when increasing the amount of transistors because of the reduction in transistor size. Dennard scaling ended in 2004, which led to the focus shifting towards multicore architectures. In recent years Moore's law has also

slowed down, which means nowadays it is beneficial to develop domain-specific architectures which excel at their specific task [28, pp. 4-5].

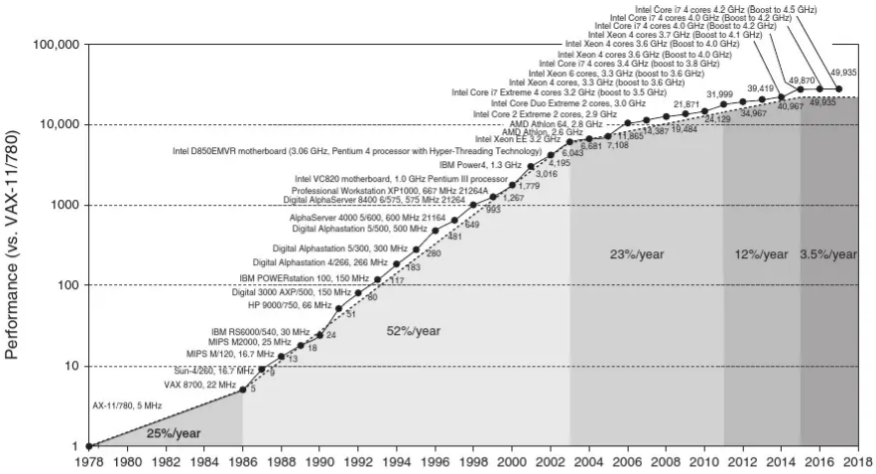


Fig. 2.8. "Growth in processor performance over 40 years" [28, p. 3]

Processors have used principles of parallelism for a long time. Michael J. Flynn defines four classes of parallel architectures, in a classification called Flynn's taxonomy [29]:

- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data stream (SIMD)
- Multiple instruction stream, single data stream (MISD)
- Multiple instruction stream, multiple data stream (MIMD)

A traditional single core processor is a SISD architecture; there is a single instruction stream, working on a single stream of data. Going to multicore, we now have several cores with their own instruction streams and their own data, i.e. MIMD. These cores work more or less independent of each other, but share the same main memory and some levels of cache; this is called a shared-memory system [30, Ch. 2].

A GPU is approximately a SIMD mixed with MIMD architecture, having lots of cores with many ALUs that perform SIMD operations. They are designed this way because their job is to convert a lot of data into pixels that can be represented on a screen. To do this, they use a *graphic pipeline* which involves many steps, many of which are programmable nowadays. These programs are called *shaders*, and can be applied to e.g. vertices (vertex shaders) and pixels/*fragments* (fragment shaders). [30, p. 32].

A GPU program is split into multiple *thread blocks* which are each assigned to their own multithreaded SIMD core. These cores have a number of *SIMD lanes*, which perform arithmetic and memory operations. Some GPUs have units that support half-precision operations, which take around half the time of a full 32-bit ploating point operation. The SIMD lanes are all connected to the same instruction

register, which contains a SIMD instruction. An interesting effect happens if there is a branch instruction in the core, and the threads take different directions. This is called *branch divergence*. The SIMD lanes in a core "cannot simultaneously execute different instructions" [28, p. 325], and the threads that have diverged will have to wait for their turn at the lanes. This does not mean that deep nested if-statements are inherently bad, but rather that when threads disagree frequently it hurts the performance [28, Ch. 4.4].

An important concept in computer graphics is *textures*. These are essentially images that can be used to give an illusion of depth and more complex geometry. For example, a simple gray cube can be turned into a wooden crate with a crate texture. Textures are stored in memory, and are accessed and mapped to the correct area of the models by a *texture mapping unit*. Compressing a texture reduces their size, thus decreasing the memory (and memory bandwidth) requirements [31]. A different way of looking at textures is as a 2D (or 3D in some cases) matrix. Matrices can be stored in multiple formats, where storing row-wise is very common. A less common way to store matrices is Morton-order, which essentially stores the elements in a z-like pattern. This way, every element that has indices that are close interpreted as Euclidean points are also close in physical space compare to if every row was stored after one another. This can be useful for memory accesses on GPUs [32], and by extension volume rendering [33].

Although processor performance has improved quite drastically, the speed of memory has not followed this trend. Fig. 2.9 shows the difference between processor and memory performance [28, p. 80]; a phenomenon known as the *memory wall* [34]. In order to alleviate the effects of the memory wall, memory has been organized into a *memory hierarchy* consisting of smaller, faster, and more expensive types of memory on the top and larger, slower, and cheaper at the bottom. The higher levels of the hierarchy are closer to the processor, while the levels further down are farther away. Fig. 2.10 visualizes this relation. An important concept regarding memory is *locality*. There are two kinds: *spatial locality* and *temporal locality*. When data has been requested from memory, it is likely that neighbouring data will be used; this is spatial locality. Temporal locality is that when data has been used, it is likely to be needed again. The memory hierarchy uses these two principles, putting recently used data and neighbouring data high up, and other data further down.

Caches act as a storage between the CPU's registers and the main memory which is often off-chip. These follow the principles of locality. There are several ways a cache may be implemented. A common way is "splitting" up the cache into a number of *sets*, where each *block* of data (a collection of neighbouring data in memory) can be placed anywhere in its assigned set. Which set this is is determined by Equation 2.2 [28, Appendix B].

$$BlockAddress \mod \#sets \tag{2.2}$$

When data is needed from memory and is not found in the cache, we call it a *cache miss*. This means the data has to be fetched from deeper in memory, which

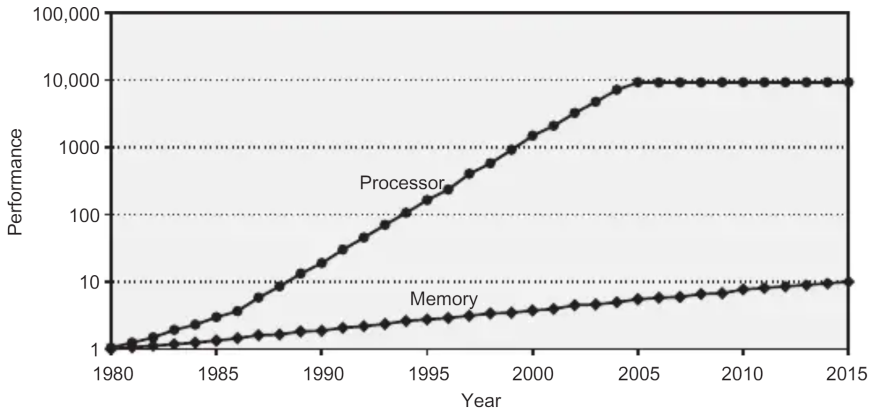


Fig. 2.9. "Starting with 1980 performance as a baseline, the gap in performance, measured as the difference in the time between processor memory requests (for a single processor or core) and the latency of a DRAM access, is plotted over time" [28, p. 80]

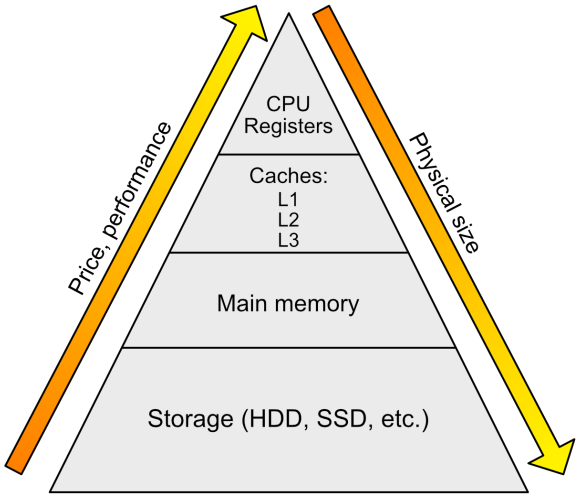


Fig. 2.10. A standard memory hierarchy for laptops and desktops visualized

can take a while depending on how deep the miss goes. This data will then be placed in the cache. If the corresponding set is full, a block of memory needs to be *evicted*. There are a few ways this can be done, but the most common one is *Least recently used* (LRU). The LRU policy dictates that the block that has gone the longest time without being used is to be evicted, following temporal locality [28, Appendix B].

As an example HoloLens 2, which is presented in Section 3.5.2, has a compute platform consisting of two four-core clusters; one more powerful, the other more low power. The more powerful one has a 512 KiB L1 cache and a 1 MiB L2 cache, while the other cluster has 512 KiB for both the L1 and the L2 cache. These caches also have varying amounts of sets. The clusters share a 2 MiB L3 cache [35]. Outside of this, the HoloLens 2 has a main memory of 4 GB [36].

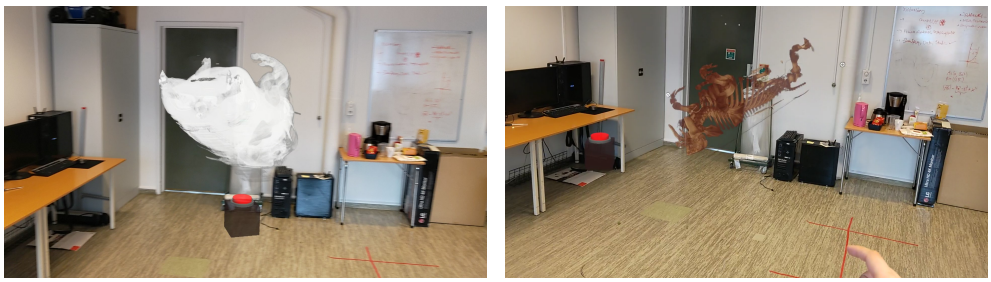
Chapter 3

Methods

As an extension of the preparatory project [1], which focused more on getting familiar with volume rendering and Unity's shaders, this research project focused more on implementing and testing advanced techniques for optimizing the performance on a device with limited computational power. The following sections will go further into detail regarding the different shader programs that were made, research strategy, data generation and analysis, and the test methodology.

3.1 Volume rendering shaders

During the master project, several programs have been implemented. The details regarding the actual implementation with code snippets is presented in Chapter 4, while this section will take a more theoretical look at the various approaches. The volume rendering was implemented as GPU shaders, storing the volume data as a 3D texture, and the transfer function as a 2D texture. Fig. 3.1 shows a CT scan of a pig rendered using the first unoptimized shader program next to the same pig rendered using the most optimized shader program (without diffuse shading). A demo video captured on HoloLens 2 can be found in Chapter 5.



(a) Pig with basic shader **(b)** Pig with Chebyshev shader

Fig. 3.1. A CT scan of a pig rendered with a basic shader program and an advanced optimized shader program, both rendered directly on a HoloLens 2

Every program starts in a similar way. A ray is shot through a volume. The ray

direction is the direction vector between the camera position and the volume's world coordinates. The ray's end is determined by the exit intersection between the ray and the volume's axis aligned bounding box (AABB), and the ray's origin is the position of a vertex in the image plane. The amount of equidistant samples n that will be taken along the ray [2] is determined by Equation 3.1 [17, Equation 2], where $\overrightarrow{max}$ is the maximum of the vector components, VolumeDims is the dimension of the volume texture, and q is a quality factor used to scale the amount of points if needed.

$$n = \overrightarrow{max}(VolumeDims) * |ray| * q \tag{3.1}$$

The ray step is defined by Equation 3.2 [17, Equation 3]:

$$step = \frac{direction(ray) * |ray|}{n-1} \tag{3.2}$$

The first program, Basic, simply samples along the ray and blends the volume data at each sample point as a grayscale color. The second program, Transfer, takes this a step further and applies a transfer function (similar to the classification techniques) to the volume data at each sample point, and blends that function's output. The blending is done according to Equations 3.3 and 3.4, where src is the transfer function's output and dst is the accumulated fragment color. r, g, b, and a are the red, green, blue, and alpha components of the colors:

$$dst_a = (1 - dst_a) * src_a + dst_a \tag{3.3}$$

$$dst_{rgb} = (1 - dst_a) * src_a * src_{rgb} + dst_{rgb} \tag{3.4}$$

The next program, ERT, adds early ray termination to the mix. This is a simple logic test that checks if the accumulated output opacity has reached a specified threshold value, and stops ray traversal if it has.

The occupancy map program is more complex. A new 3D texture is introduced, representing regions of the volume where every element will evaluate to 0 according to the transfer function. Every element of the new texture corresponds to a region of size B^3, meaning the texture, called an *occupancy map*, has dimensions equal to the volume's dimensions divided by B. Since the volume's dimensions may not always be divisible by B, a mapping equation is defined [17]: $position_{occ} = position_{vol} * \frac{Dims_{vol}}{B}$. The step of the occupancy map can be found in a similar way, switching the positions for steps.

For the ray sampling, the algorithm assumes the first element is non-empty. Volume sampling, transfer function, and color blending acts the same as before, but the algorithm will also check if the alpha value from the transfer function is zero. To make sure the algorithm does not assume every empty voxel implies that it should start skipping, it also keeps track of which region it currently is in. For each volume sample, the algorithm keeps track of its next iteration value i_min, the motivation for which will be explained shortly.

When the algorithm encounters an empty voxel *and* has entered a new region, it stops sampling the volume and starts sampling the occupancy map instead. This is done with no filtering; otherwise, the algorithm will be less accurate due to empty regions "bleeding" into non-empty regions (this is because the occupancy map is implemented with empty regions encoded as 255 and non-empty as 0). The algorithm checks if the region is empty or not to determine whether to skip or start sampling again. If the region is empty, the next i is found using Equation 3.5, which is similar to Deakin and Knackstedt's Δi but with a different step function [17, Eq. 8, 9]:

$$\Delta i = max(\overrightarrow{min}\left(\left\lceil \frac{f(\Delta u) + \lfloor u \rfloor - u}{\Delta u} \right\rceil\right), 1) \tag{3.5}$$

Where $f(x)$ is a step function where x $>$ 0 evaluates to 1, and 0 otherwise. u is the position in the volume mapped to the occupancy map's coordinate system, and Δu is the step of the volume also mapped to the occupancy map.

If the algorithm finds a non-empty region, it backtracks according to Equation 3.6 in order to not miss any non-empty voxels by accident, and starts sampling the volume as it did in the previous shader programs.

$$i = max(i - \lceil q \rceil, i_{min}) \tag{3.6}$$

i_{min} is the last iteration value where the volume was sampled, and is used as the boundary for backtracking to avoid infinite loops (as otherwise the ray would simply re-enter the empty region, skip, and then go back to where it started). q is the quality factor mentioned in Equation 3.1.

The final optimization program is *Chebyshev distance maps*, which in principle is quite similar to the *occupancy map*. The biggest difference here is how the next i is calculated. The occupancy map is transformed into a Chebyshev distance map, which tells the algorithm how far it is to the closest non-empty voxel; this can be used to determine the minimum amount of samples that can be skipped before potentially finding a non-empty voxel. If the distance is 0, the sample point is in a non-empty region, and otherwise the sample point is in an empty region. Chebyshev distance map sampling is done the same way as for occupancy maps, but the Δi is found differently, as shown in Equation 3.7,

$$\Delta i = max(\overrightarrow{min}\left(\left\lceil \frac{f(-\Delta u) + sign(\Delta u) * d + \lfloor u \rfloor - u}{\Delta u} \right\rceil\right), 1) \tag{3.7}$$

where $sign(x)$ evaluates to -1 if x is negative, and 1 otherwise. $f(x)$ is the same step function as in Equation 3.5, and d is the distance from the Chebyshev distance map. There has also been made a shader program that uses half precision for color blending.

Lastly, a shader that uses diffuse shading has been implemented. This one requires precomputed normal vectors which are acquired by calculating the gradients between each voxel in the volume, based on central differences. The direction of the normal vectors is stored in the volume's rgb components, moving the voxel

value to the alpha component. Diffuse shading also requires a light source. This is for simplicity defined as the camera, and the light direction is defined according to Equation 3.8,

$$l = direction(ray) + |\overrightarrow{min}(direction(ray))| \tag{3.8}$$

in order to make the light direction work as expected. For each volume sample, the r, g, and b values of src (the color to be blended) are defined according to Equation 3.9:

$$src_{rgb} = transfer(v_a) * (I_{ambient} + max(0, v_{rgb} \cdot l)) \tag{3.9}$$

where $transfer(v)$ is the transfer function, v is the volume data that has been sampled from the volume, $I_{ambient}$ is the ambient lighting, and l is the light direction. Normally, the diffuse lighting also uses light intensity, color, and the material's diffuse component, but these were assumed to be 1 for simplicity and as an optimization.

3.2 Strategy

Due to the nature of the research questions, this project followed the strategy "Design and Creation" [37, Ch. 8]. The *artifact* of the research is an *instantiation* of volume rendering on the HoloLens 2 and more importantly: an evaluation of optimizations with low memory overhead. Since RQ1 is looking for the frame rate of the program, it was beneficial to actually make an implementation and gather data through profiling. RQ2 and RQ3 also benefit from an application, as that implementation can be used in a demo. The demo will make it possible to get constructive feedback that can highlight both its strong parts and its shortcomings.

3.3 Data generation

In order to answer RQ1, quantitative date in the shape of frame rate and memory usage was gathered. In addition, volume size and resolution was also taken into account, along with shader configurations. Section 3.3.1 goes further into detail regarding the quantitative data. RQ2 and RQ3 benefited more from qualitative data from clinical experts. In order to answer this, a demo followed by an interview was arranged. Section 3.3.2 explains the demo and interview.

3.3.1 Test methodology

The most interesting parameter to measure for RQ1 is unsurprisingly the frame rate. It is also interesting to check the memory usage, as the HoloLens has less memory than most laptops nowadays. In order to check the frame rate of each shader program, tests had to be run for each volume-shader-parameter configuration. For example, the occupancy shader uses regions of size B; B is configurable,

and directly affects memory usage and frame rate. Each volume dataset has different dimensions with different memory requirements, and which may also affect the frame rate. Thus, every combination is interesting.

The frame rate information was gathered through MRTK's built-in profiler which displays frame rate, current memory usage, and peak memory usage. Microsoft has a profiling tool called PIX, which was considered. However the volume rendering program tended to crash whenever this was used, likely due to the high computational and memory requirements. The same argument stands for filming the testing, at least in the case of larger volumes. As such, the solution was to report the average frame rate through observation. The observed frame rates were written down in a spreadsheet to be analysed there.

Since the frame rate is dependent on the amount of pixels covered by the volume object, the tests were split into two parts: the initial rendering which happens 2.5 m away from the user, and dragging the volume to be at arm's length from the user (in the observer's case, approximately 60 cm). The observed average frame rate was noted down for each case. The scale and storage size of each volume was also written down.

Setting up highly reliable profiling tests for a HoloLens application can be challenging. Much of it is similar to traditional graphics benchmarking, in that one will move the camera around in a scene. However, in traditional applications and games, the camera can be fully controlled by a script. On the HoloLens the camera is "attached" to the user's head; since the frame rate is affected by the amount of the screen the volume object takes up, combined with involuntary head movements, each test's reliability is lowered. To compensate, the following actions have been made:

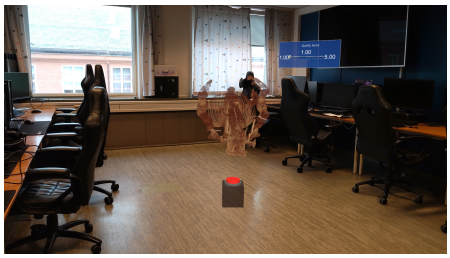
- The test is performed seated in a chair with a back
- The volume object is hard coded to initially spawn 2.5 m from the origin, which is defined as where the game started. As such, the chair should not move
- Each test will have five iterations to increase the overall reliability

Fig. 3.2 shows the two cases, i.e. near and far, for each volume.

3.3.2 Clinical experts

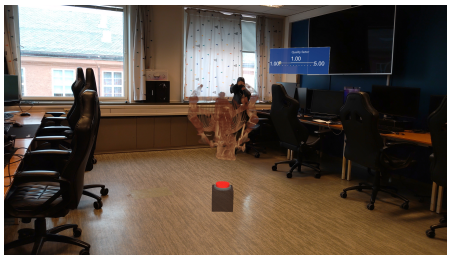
When the frame rate of the application was good enough for simple interaction and reduced eye strain, clinical experts from St. Olavs Hospital were contacted for a demo and interviews. The original idea was to have a demo showcasing a dynamic ultrasound volume render with and without shading, followed by a semi-structured interview focusing on the following themes:

- User experience (eye strain, good aspects, room for improvement etc.)
- Information about how they use 3D visualization in their usual work life
- Which use cases they thought may be the best fits for volume rendering on HoloLens

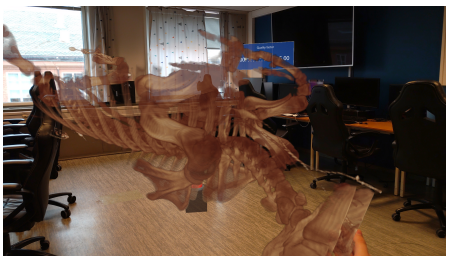


(a) The medium-sized pig CT dataset, rendered 2.5 m away from the viewer

(b) The medium-sized pig CT dataset, rendered an arm's length away from the viewer

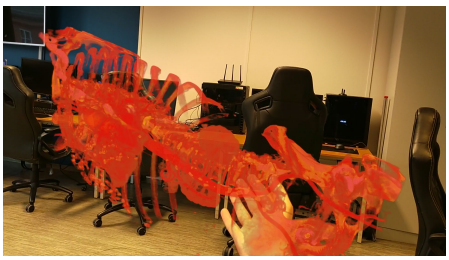


(c) The smaller-sized pig CT dataset, rendered 2.5 m away from the viewer

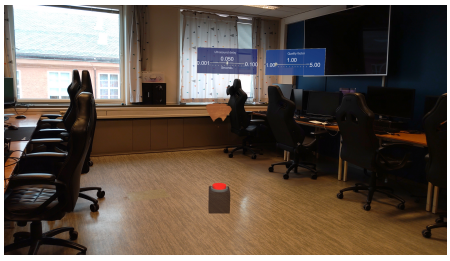


(d) The smaller-sized pig CT dataset, rendered an arm's length away from the viewer

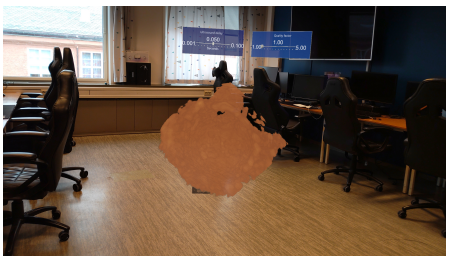
(e) The thorax CT dataset, rendered 2.5 m away from the viewer



(f) The thorax CT dataset, rendered an arm's length away from the viewer



(g) The ultrasound mitral valve dataset, rendered 2.5 m away from the viewer



(h) The ultrasound mitral valve dataset, rendered an arm's length away from the viewer

Fig. 3.2. The test cases shown on the HoloLens 2

However since they already talked about these themes during testing, the interview format was changed to an unstructured group interview. This way, the discussion flowed more naturally, facilitating more brainstorming. Quotes and insights were noted down in a text document. By including experts who are potential users for a mixed reality volume rendering application it was possible to get more ideas for further research. The experts that tested the application included two echocardiologists (experts) and an anesthesiologist with basic knowledge of ultrasound.

3.4 Data analysis

Quantitative analysis methods were used for the ratio data gathered from the profiling tests. The mean and standard deviation of each set of five tests were calculated. The mean is used as the "true" frame rate which is used for further analysis. The standard deviation acts as a measure of the mean's accuracy, and thus also as an indicator for the further analysis' quality and accuracy. In addition, the Pearson's product moment correlation coefficient was calculated to check for a relation between the frame rate and memory usage, volume size, and volume object scale. This way, it is possible to find less obvious relations between the different variables.

3.5 Tools

3.5.1 Unity

Unity is a game engine that is primarily used for making video games, but is also useful for mixed reality applications. In Unity the user is able to set up "scenes" with objects they can place in the world space. It is possible to run the application in Unity, and it has many useful plugins and libraries, such as MRTK. Unity also includes profiling tools, a user interface for development and design, standard templates, scripts, and objects, and much more.

The Mixed Reality Toolkit (MRTK) is a toolkit developed by Microsoft for mixed reality, i.e. AR and VR, development. The toolkit makes mixed reality development simpler by handling most of the AR/VR-specific functionality for the developer. This process is especially well tailored to developing on the HoloLens, as it is made by the same company. MRTK includes functionality such as hand tracking, spatial awareness, and eye tracking, among other things [38].

The simplest way of rendering for both eyes on HoloLens is simply rendering twice using two separate render passes. There are better ways, however. Unity's *Single-Pass Stereo Rendering* was used order to render for both eyes on the HoloLens. This method uses a single render pass, removing the CPU overhead of making two passes. In addition, the cache coherency between the draw calls will be much better, and the slowdown of using two render passes will not be as noticeable on the GPU. As a bonus, the energy efficiency also gets better [39]. The functions

in all capitalized letters seen in Chapter 4 are used to facilitate single-pass stereo rendering.

3.5.2 Microsoft HoloLens

Microsoft HoloLens is Microsoft's AR headset series. The original HoloLens came out in 2016, and was the "world's first fully untethered holographic computer" [40]. With the HoloLens, one can use holographic applications on a transparent screen, with the digital elements "placed" around in the room. The HoloLens comes with 2 GB RAM and Microsoft's custom *Holographic Processing Unit* (HPU), which is Microsoft's hardware accelerator for AR processing [40]. According to Chris Pietschmann, the first generation HoloLens' processor was a "Intel(R) Atom(TM) x5-Z8100P", which has four cores. HoloLens also uses an Intel GPU [41], though the exact details are unclear. Fig. 3.3 shows an image of the original HoloLens.

Fig. 3.3. Microsoft HoloLens 1 [42]

Microsoft HoloLens 2 is, as of 2022, Microsoft's newest released AR headset. It is the second generation of HoloLens, and includes upgrades to both processing and memory. The HoloLens 2 uses a *Qualcomm Snapdragon 850 Compute Platform* for computation, a new version of the HPU, a display consisting of 2K 3:2 light engines, and 4 GB of RAM compared to the first generation's 2 GB [36]. The Qualcomm Snapdragon 850 Compute Platform includes a CPU with eight cores and Qualcomm Adreno 630 GPU [43]. According to CPU-Monkey the performance of an Adreno 630 is around 737 GFLOPS (Giga floating-point operations per second), and according to ChipGuider the performance is around 712 GFLOPS [44, 45]. The reported performance of the GPU depends on which benchmarks have been used to profile it; CPU-Monkey does not take responsibility for the data on their website, which could mean that their numbers may not be very accurate. Fig. 3.4 shows the HoloLens 2.

Fig. 3.4. Microsoft HoloLens 2 [36], background removed

Chapter 4

Implementation

This chapter goes further into detail regarding how the program is structured and how it works. It will start by explaining the developer dashboard and the CPU logic around preprocessing, before further explaining the shaders. The full code can be found here:

- https://github.com/bednik/Master-Project

4.1 Developer dashboard

In order to make profiling simpler, and to make a more user-controllable application, a developer dashboard was made. Fig. 4.1 shows this dashboard in the game view of the Unity editor. Pushing the big red button starts the volume rendering. This was added early on, as changing configurations for a very slow render could be nauseating due to a low frame rate. It makes it possible to set configurations prior to volume rendering. The dashboard consists of six panels, each controlling the following:

- Which volume dataset to visualize
- Which transfer function to use
- Which shader should be used for rendering
- The early ray termination value
- The block size of the empty space skipping method
- Whether to use the original quality (amount of sample points hardcoded to 256) or Deakin's way of calculating amount of sample points (Equation 3.1)

In addition to the dashboard, three more panels were made. These panels control variables at runtime: Ambient light intensity, delay between each ultrasound image, and the quality factor. There is also a button here which when pushed stops volume rendering and sends the user back to the original dashboard. The different panels only appear when used with a data type or shader that uses their respective variables. Fig. 4.2 shows these panels in the game view of the Unity editor. Since the volume is an ultrasound set, the shader uses lighting and is advanced enough

to consider the quality factor, all three panels are loaded.

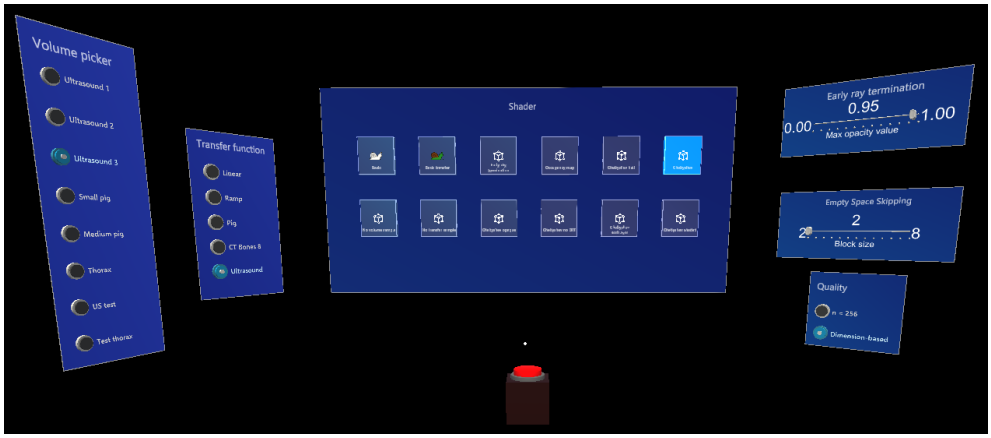


Fig. 4.1. The developer dashboard to be used for instantiation of a volume render

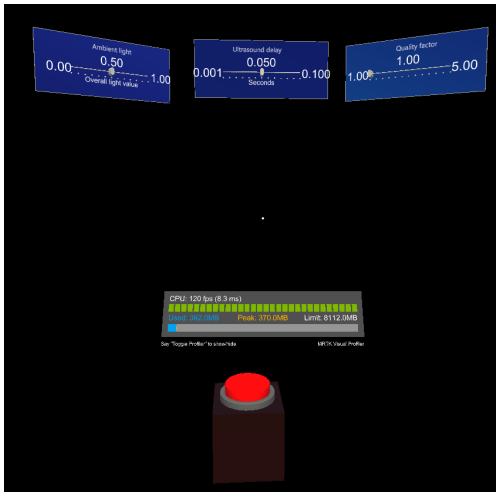


Fig. 4.2. The render controller panels for an ultrasound volume using the Chebyshev shaded shader

4.2 General infrastructure

4.2.1 Volume loading

In order to use the volumes in a shader in Unity, the volumes had to be converted into 3D textures. For simplicity, this was done "offline", i.e. using python scripts and a Unity editor tool. The ultrasound data used in this project was stored as H5 files. This script is a quite simple one, which uses the library *h5py* to load the cartesian volumes from the file and saves them as numpy array files. This

also handles the temporal aspect, saving every cartesian volume numbered in the correct order. The code is shown in Listing 4.1.

Code listing 4.1: H5 ultrasound file to numpy file converter

```
import h5py
import numpy as np
import sys
import os

input_folder = "../VolumeData/US/"
output_folder = "../Resources/VolumeRaw/US/"

f = h5py.File(input_folder + str(sys.argv[1]), 'r')
savelocation = output_folder + str(sys.argv[2])

# Fetch cartesian volumes
cartesian_volumes = f["CartesianVolumes"]

if not os.path.exists(savelocation):
    os.makedirs(savelocation)

for key in cartesian_volumes.keys():
    v = cartesian_volumes[str(key)]
    np.save(savelocation + "/" + str(key), v)
```

An editor script was made to turn this numpy array into a Unity Texture3D object. This script used the package *NumSharp*, which is possible to import using *NuGet for Unity*. Using this package, it is possible to flatten the numpy array. This flattened array is then parsed, and each element is placed in a normal C# byte array. This array is then set as the pixel data of the resulting Texture3D object, which is saved as an asset. The editor script will use every cartesian volume, storing them with a number at the end that makes loading them in the correct order simpler.

The CT data was stored as a RAW image, meaning it was already a simple data stream. Along with the data there were metadata files detailing the scale of the volume and its resolution among other things. The RAW file was parsed, and its values were added to a byte array and set as the pixel data of the resulting Texture3D object. However, the thorax data set consisted of 16-bit values. In order to make it fit within a byte texture, the data had to be scaled. This was done by reading two bytes, then concatenating them. Since the values were between -1024 and 3071, the script would add 1024 to make the value positive, then divide by 16. This process is shown in Listing 4.2. For the pig data, the process was similar to that of the ultrasound data as it already contained byte data.

Code listing 4.2: A snippet from the CT texture generator showing reading from a RAW file and byte concatenation

```
int textureSize = width * height * depth;

Texture3D density = new Texture3D(width, height, depth, TextureFormat.R8, false);

density.wrapMode = TextureWrapMode.Clamp;
density.filterMode = FilterMode.Bilinear;
```

```
density.anisoLevel = 0;

using (var stream = new FileStream(inputPath, FileMode.Open))
{
      var len = stream.Length;

      byte[] colors = new byte[textureSize];

      for (int i = 0; i < textureSize; i++)
      {
          byte lower = (byte)stream.ReadByte();
          byte higher = (byte)stream.ReadByte();
          short val = (short)(lower + (higher << 8));
          colors[i] = (byte)((val + 1024)/16);
      }

      density.SetPixelData(colors, 0);
      density.Apply();
}
```

Later in the project, the loaders were expanded to make compressed volumes using the compression format BC4. However, Unity did not have a native function for compressing 3D textures. Luckily, there was a way: transforming the 3D problem into a series of 2D problems. Since the raw data of a 3D texture is essentially a collection of 2D textures, which can be compressed, stacked on top of each other it was possible to compress each layer individually and then setting them as the texture data of a 3D texture on BC4 format (which is possible despite the lack of a compression function). It was not possible to do a simple *CopyTexture from 3D layer -> Compress -> CopyTexture back to layer position* from this experience. The process is shown in Listing 4.3. This follows directly after Listing 4.2. The code is inspired by a forum post on the Unity Forum by user ignarmezh [46].

Code listing 4.3: Code for compressing a 3D texture in Unity

```
// https://forum.unity.com/threads/texture3d-compression-issue.966494/
List<Texture2D> layers = new List<Texture2D>();
for (int z = 0; z < depth; z++)
{
    Texture2D t = new Texture2D(width, height, TextureFormat.R8, false);
    Graphics.CopyTexture(density, z, 0, 0, 0, width, height, t, 0, 0, 0, 0);
    EditorUtility.CompressTexture(t, TextureFormat.BC4, TextureCompressionQuality.
        Best);
    layers.Add(t);
}

List<byte> res = new List<byte>();
for (int z = 0; z < depth; z++)
{
    var tex2DData = layers[z].GetRawTextureData<byte>();

    for (int i = 0; i < tex2DData.Length; i++)
    {
        res.Add(tex2DData[i]);
    }
}

Texture3D storeTex = new Texture3D(width, height, depth, TextureFormat.BC4, false)
```

```
{
    wrapMode = TextureWrapMode.Clamp,
    filterMode = FilterMode.Bilinear,
    anisoLevel = 0
};
storeTex.SetPixelData(res.ToArray(), 0);
storeTex.Apply();
```

4.2.2 Volume builder and controller

The largest CPU script in this project is *Volume Builder*. This is controlled by the dashboard, and contains logic behind generating empty space skipping structures, transfer functions, local normals, choosing the correct shader, and setting the shader variables.

Whenever a new transfer function is chosen the builder creates a 2D texture with height 1 and width 256 that works as a color lookup-table. The values are made by linearly interpolating between a predefined set of points using MathNet, which is a package available through *NuGet for Unity*. These transfer functions are the following:

- Linear (Color and alpha = volume value)
- Ramp (Bias towards higher values)
- Pig (based on Slicer3D's built-in function for MR images)
- CT_BONES (Based on Slicer3D's built-in function for showing CT skeletons)
- Ultrasound (Made by Gabriel Kiss. Works well for showing the heart valves in the ultrasound dataset)

When the red button is pushed, the volume builder starts making the empty space skipping structures and normal map, depending on which shader was picked.

There is also a script that controls the volume rendering in real-time. Its responsibility lies in setting the ambient intensity and the quality factor if they are changed, and to update the ultrasound images. The latter is the largest task. Each time the ultrasound image changes, so does the empty space skipping structures and normal maps. These are calculated for each image.

4.3 Volume rendering

The largest focus of this project has of course been the volume rendering itself. The five methods described in this section will be presented in the order they were implemented, giving an idea of the development process. The very first program was a bare-bones application inspired by Tasken, Barstad, and Tagestad's [25] implementation, later extended with Deakin's [17] way of calculating amount of sample points and ray entry to make it more consistent for comparison. All subsequent shaders have used their predecessor as a base. Every shader, except basic, are extended to allow Unity's single pass stereo rendering, which is a fast way of rendering for both eyes.

All shaders share the same vertex shader, except for *Basic* which does not have the macro functions (the functions in all caps). Listing 4.4 shows the code for the vertex shader.

Code listing 4.4: The vertex shader common for all volume rendering shader programs

```
v2f vert(vertexData v)
{
        v2f o;

        UNITY_SETUP_INSTANCE_ID(v);
        UNITY_TRANSFER_INSTANCE_ID(v, o);
        UNITY_INITIALIZE_VERTEX_OUTPUT_STEREO(o);

        o.vertex = UnityObjectToClipPos(v.pos);
        o.uv = v.uv;
        o.t_0 = v.pos.xyz + 0.5;
        o.world = mul(unity_ObjectToWorld, v.pos).xyz;
        o.local = v.pos.xyz;
        return o;
}
```

In addition to the vertex shader, there are some other things that are common across all shaders. One of them is the function for calculating the ray exit that was used by Deakin [17]. This function is shown in Listing 4.5. The other is some setup code for determining ray direction, start, amount of sample points, and setting up necessary variables; this is shown in Listing 4.6. This is a mix of Deakin [17] and Tasken, Barstad, and Tagestad [25]. All specifically volume rendering calculations are done in the fragment shader.

Code listing 4.5: A function for determining the exit of a ray through a volume using a simplified AABB intersection used by Deakin

```
/*
        * Author: Lachlan Deakin
        * Date: Dec 9, 2021
        * File: shaders/volume_render.frag
        * Commit: 4b94c00
        * Type: Source code
        * Link: https://github.com/LDeakin/VkVolume/blob/master/shaders/
            volume_render.frag
*/
float3 ray_caster_get_back(float3 front_intersection, float3 dir) {
        // Use AABB ray-box intersection (simplified due to unit cube [0-1]) to get
             intersection with back
        float3 dir_inv = 1.0f / dir;
        float3 tMin = -front_intersection * dir_inv;
        float3 tMax = (1.0f - front_intersection) * dir_inv;
        float3 t1 = min(tMin, tMax);
        float3 t2 = max(tMin, tMax);
        float tNear = max(max(t1.x, t1.y), t1.z);
        float tFar = min(min(t2.x, t2.y), t2.z);

        // Return the back intersection
        return tFar * dir + front_intersection;
}
```

Code listing 4.6: Setup code common across all shaders

```
// Fragment kernel //
fixed4 frag(v2f vdata) : SV_Target
{
        UNITY_SETUP_INSTANCE_ID(vdata);
        UNITY_SETUP_STEREO_EYE_INDEX_POST_VERTEX(vdata);

        Ray ray;
        ray.origin = vdata.t_0;
        ray.dir = normalize(mul(unity_WorldToObject, vdata.world -
             _WorldSpaceCameraPos));
        float3 ray_exit = ray_caster_get_back(vdata.t_0, ray.dir);
        ray.length = length(vdata.t_0 - ray_exit);

        // Calculate amount of sample points and step length (with direction)
        int n = (_HighQuality == 1) ? int(ceil(float(max3(_VolumeDims)) * ray.
             length * _Quality)) : 256;
        float3 step_volume = ray.dir * ray.length / (float(n) - 1.0f);

        // This piece of code from Deakin makes performance smoother in some cases.
        // Deakin's words:
                // "This test fixes a performance regression if view is oriented
                     with edge/s of the volume
                // perhaps due to precision issues with the bounding box
                     intersection"
                // https://github.com/LDeakin/VkVolume/blob/master/shaders/
                     volume_render.frag
        float3 early_exit_test = ray.origin + step_volume;
        if (any(early_exit_test <= 0) || any(early_exit_test >= 1)) {
                return fixed4(0, 0, 0, 0);
        }

        float3 currentRayPos = ray.origin;
        half oneMinusAlpha = 1;
        fixed4 dst = fixed4(0, 0, 0, 0);

        float prev_alpha = 0;
```

4.3.1 Basic

As its name implies, this is a very basic method. As mentioned, *Basic* is inspired by Tasken, Barstad, and Tagestad [25]. The main difference from their work is that this shader does not allow slicing, and does not have a min or max threshold for the volume values to count. As mentioned, it also determined the amount of sample points and the ray entry and direction like Deakin [17]. This shader has also *not* facilitated for rendering for both eyes on the HoloLens.

The idea of *Basic* is that it loads a byte from the volume using the ray position as its coordinates, which is a 3D texture. This value is then blended with the current accumulated output color, which is returned when the ray exits the volume. The algorithm is shown in Listing 4.7. Fig. 4.3 shows an example of a volume texture, more specifically of a pig. Fig. 4.4 shows a slice of this dataset, and Fig. 4.5 shows a render of this pig dataset using the basic method.

Code listing 4.7: Volume rendering (Basic)

```
[loop]
for (int iter = 0; iter < n; iter++)
{
        float src = tex3Dlod(_Volume, float4(currentRayPos, 0));

        // Get the alpha directly from the texture, set the color by blending
        oneMinusAlpha = 1 - prev_alpha;
        dst.a += src;
        dst.rgb = mad(dst.rgb, oneMinusAlpha, src); // dst.rgb * (1 - prev_alpha) +
             src
        currentRayPos += step_volume;
        prev_alpha = src;
}

dst = saturate(dst);

return dst;
```

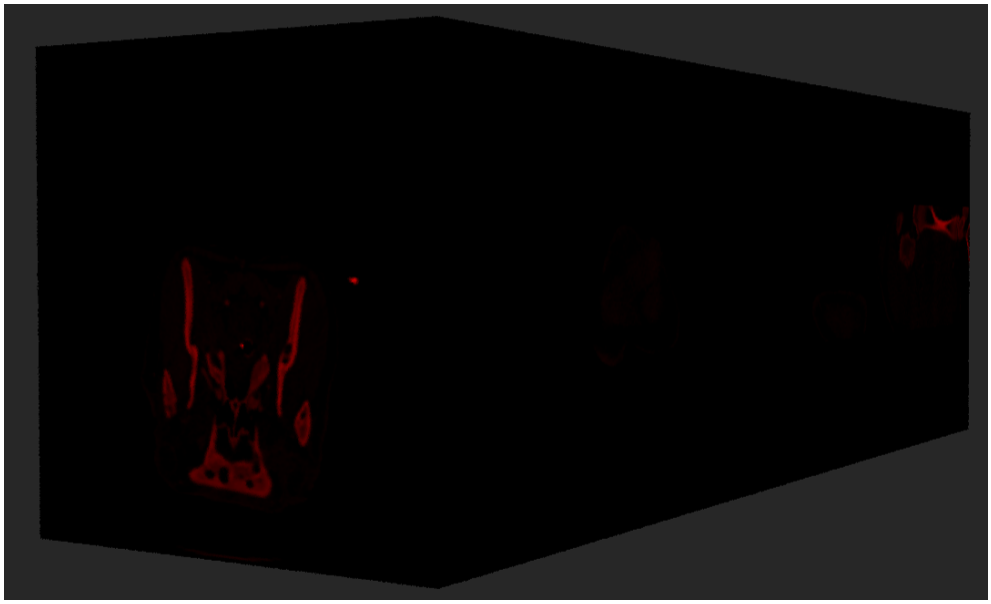


Fig. 4.3. A 3D texture containing CT data of a pig

4.3.2 Transfer

The method *Transfer* extends *Basic* with transfer functions. These functions take in the value loaded from the volume and determine a color and opacity from it. The function is a 1D texture implemented as a 2D texture with a height of 1, i.e. it has one row. The function is created on the CPU at runtime before starting volume rendering. This is done by linearly interpolating between a set of predefined points using Mathnet. Accessing the color is done by using the volume value and 0 as its UV coordinates. The color blending was also further simplified. This method can

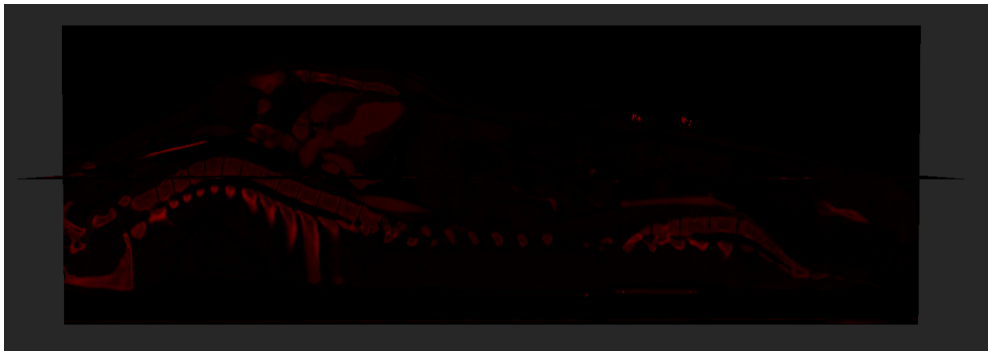


Fig. 4.4. A slice of a 3D texture containing CT data of a pig

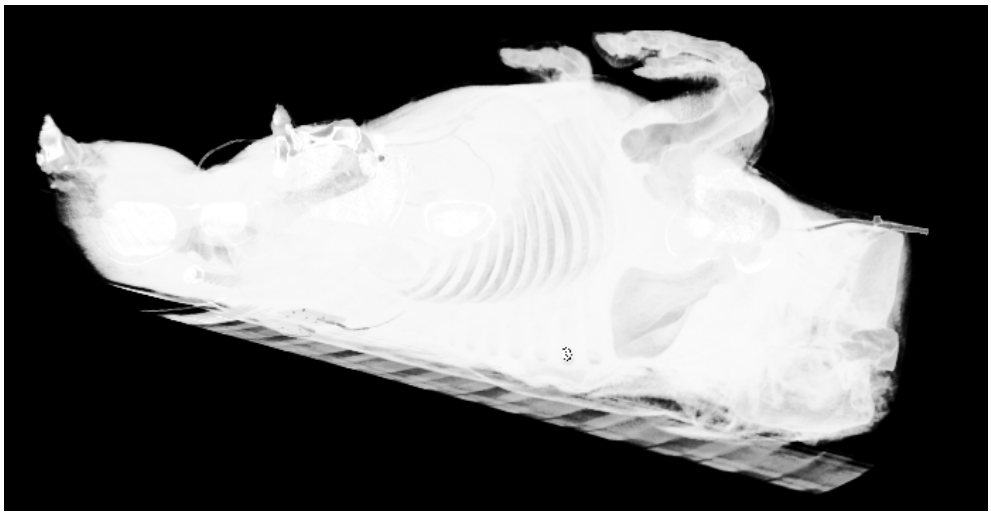


Fig. 4.5. A CT scan of a pig rendered using the Basic method

be seen as the "true" base of the subsequent methods. The updated algorithm is shown in Listing 4.8. Fig. 4.6 shows the transfer function used for the pig datasets. Fig. 4.7 shows the same pig dataset as in Fig. 4.5 rendered with a transfer function.

Code listing 4.8: Volume rendering (Transfer)

```
[loop]
for (int iter = 0; iter < n; iter++)
{
        // Sample the texture and set the value to 0 if it is outside the slice or
            not within the value thresholds
        float density = tex3Dlod(_Volume, float4(currentRayPos, 0));
        float4 src = tex2Dlod(_Transfer, float4(density, 0, 0, 0));

        oneMinusAlpha = 1 - dst.a;
        src.rgb *= src.a;
        dst = mad(src, oneMinusAlpha, dst);

        currentRayPos += step_volume;
}
```

Fig. 4.6. The transfer function texture used for pig datasets

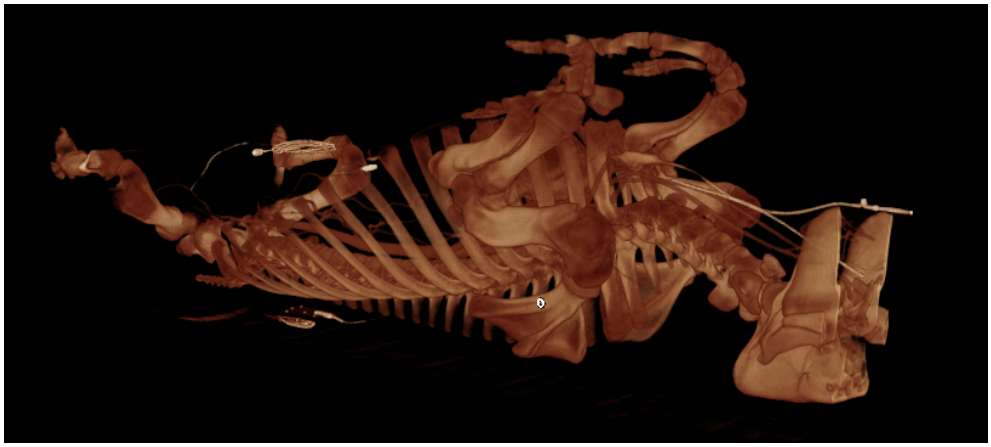


Fig. 4.7. The medium pig dataset rendered with a transfer function

4.3.3 Early ray termination

Early ray termination (ERT) is, as the name implies, the process of stopping the ray tracing before it exits the volume. The idea here is that at some point new colors will not affect the output color to a meaningful degree. This is implemented by setting the alpha value to 1 and breaking out of the loop when the color's alpha value has reached a specified value. Listing 4.9 shows how this was implemented,

and Fig. 4.8 shows the same pig as previously rendered with an ERT value of 0.95. Some artifacts are visible, but it is not so easy to notice.

Code listing 4.9: Early ray termination

```
[branch]
if (dst.a >= _ERT) {
        dst.a = 1;
        break;
}
```

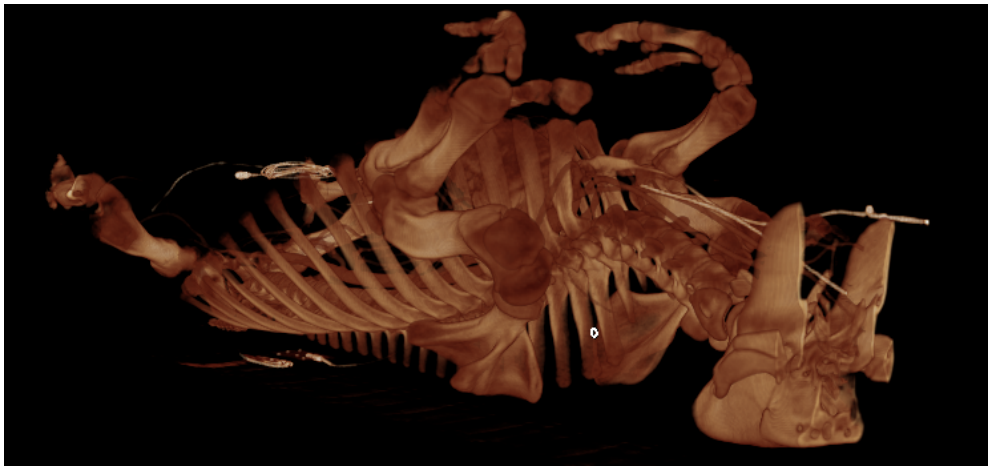


Fig. 4.8. The medium pig dataset rendered with early ray termination. The max value is set to 0.95

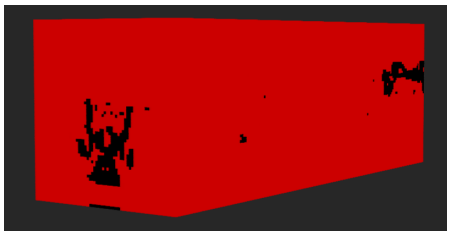
4.3.4 Occupancy map

As mentioned in Sec. 2.1, the second of Levoy's [2] was empty space skipping: the process of ignoring the parts of the volume that will not contribute to the final color. The *occupancy map* method follows Deakin's [17] formulation of the first step towards his Chebyshev distance maps. An occupancy map is a 3D texture that complements the volume texture. Each element in the occupancy map corresponds to a region in the volume of size B^3, where B is the extents of each region on each axis. The value of the region says whether there is an element in the volume which, when passed through the transfer function, would evaluate to an alpha value higher than 0. If the alpha is 0, the region's value will be 255. Otherwise, the value of the region will be 0. The occupancy map is generated by a compute shader, shown in Listing 4.10. As an example, Fig. 4.9 shows the occupancy map of the pig that has been used as an example previously with a block size of 4 and 8. Fig. 4.10 shows slices of the same occupancy maps. Occupancy maps with higher block sizes require less memory than those with lower block size and allows for larger jumps, with the risk of potentially performing more unnecessary volume samples.

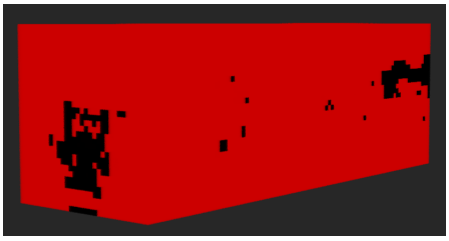
Code listing 4.10: Occupancy map generation

```
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    uint3 start = id * blockSize;
    uint3 end = start + blockSize;
    uint widthV, heightV, depthV, num_levels;
    volume.GetDimensions(0, widthV, heightV, depthV, num_levels);
    if (start.x >= widthV || start.y >= heightV || start.z >= depthV)
    {
        return;
    }

    bool empty = true;

    [loop]
    for (uint z = start.z; z < end.z; z++)
    {
        if (z >= depthV)
            break;
        [loop]
        for (uint y = start.y; y < end.y; y++)
        {
            if (y >= heightV)
                break;
            [loop]
            for (uint x = start.x; x < end.x; x++)
            {
                if (x >= widthV)
                    break;

                float elem = volume[uint3(x, y, z)];

                if (empty)
                {
                    float4 col = transferFunction.SampleLevel(
                        samplertransferFunction, float2(elem, 0.0), 0.0);
                    empty = col.a <= 0.0;
                }

                if (!empty)
                {
                    Result[id] = empty;
                    return;
                }

            }

        }
    }
    Result[id] = empty;
}
```

There is also some added logic and calculations to facilitate empty space skipping. Before the ray tracing, several constants are precomputed. These include mapping factors to convert between the coordinates of the volume and the occupancy maps, as well as variables that keep track of the skipping state. These are shown in Listing 4.11. The algorithm assumes that the first element is non-empty.



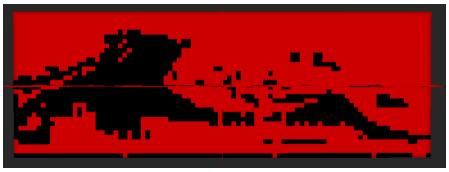
(a) Occupancy map of a CT scan of a pig, blocksize = 4



(b) Occupancy map of a CT scan of a pig, blocksize = 8

Fig. 4.9. Two occupancy maps of the same pig dataset highlighting the difference

(a) Occupancy map slice of a CT scan of a pig, blocksize = 4



(b) Occupancy map slice of a CT scan of a pig, blocksize = 8

Fig. 4.10. Two slices of occupancy maps of the same pig dataset highlighting the difference

Code listing 4.11: Occupancy map variable initialization

```
float3 volume_to_occupancy_u = _VolumeDims / _BlockSize;
float3 step_occupancy = step_volume * volume_to_occupancy_u;
float3 step_occupancy_inv = 1 / step_occupancy;
int i_min = 0;
int3 last_u_int = int3(0, 0, 0);
int i_reverse = -int(ceil(_Quality));
bool empty = false;
```

While tracing the ray, there are two possible states: Sampling and skipping. When sampling, the algorithm acts quite similar to the ERT shader. The main difference here is that it will also check if the alpha value from the transfer function is 0. If it is, the algorithm takes note of the ray's (floored) integer coordinates and switches state to skipping. In the sampling state the algorithm will also update the minimum iteration value to be the current iteration value so it does not enter an infinite loop.

While skipping, there is no color blending. Instead, the algorithm will load the value of the occupancy map without any filtering using the floored integer position in the occupancy map's coordinate system. Using that value, it determines whether to add the output of Deakin's Equation 9 [17, Eq. 9] to the iteration variable, or to backtrack a little bit and start sampling. The new ray position is evaluated as $i * \Delta t + t_{entry}$, where i is the iteration variable, Δt is the step size, and t_{entry} is the origin of the ray. Listing 4.12 shows the updated for-loop, and Listing 4.13 explains some of the used helper functions.

Code listing 4.12: Occupancy map skipping loop

```
[loop]
for (int i = 0; i < n; i) {
        float3 u = volume_to_occupancy_u * currentRayPos;
        int3 u_int = int3(floor(u));

        [branch]
        if (empty && any(u_int != last_u_int)) {
                //num_distance_samples++; // Test amount of samples
                float val = _OccupancyMap.Load(int4(u_int, 0));
                empty = val > 0;
                last_u_int = (empty) ? last_u_int : u_int;
                i = (empty) ? i + delta_i(delta_i3(step_occupancy, u,
                    step_occupancy_inv)) : int(max(i + i_reverse, i_min));
                currentRayPos = findSamplePoint(i, step_volume, ray.origin);
        } else {
                //num_volume_samples++; // Test amount of samples
                float density = tex3Dlod(_Volume, float4(currentRayPos, 0));
                float4 src = tex2Dlod(_Transfer, float4(density, 0, 0, 0));

                empty = src.a <= 0;
                last_u_int = (empty) ? last_u_int : u_int;

                oneMinusAlpha = 1 - dst.a;
                dst.a = mad(src.a, oneMinusAlpha, dst.a);
                dst.rgb = mad(src.rgb * src.a, oneMinusAlpha, dst.rgb);

                if (dst.a >= _ERT) {
                        dst.a = 1;
                        break;
                }
                i++;
                i_min = i;
                currentRayPos += step_volume;
        }
}
```

Code listing 4.13: Occupancy map helper functions

```
// Equation 4 (Deakin and Knackstead)
float3 findSamplePoint(int i, float3 delta_t, float3 t_entry) {
        return mad(i, delta_t, t_entry);
}

// Equation 8 (Deakin and Knackstead)
int3 delta_i3(float3 delta_u, float3 u, float3 delta_u_inv) {
        return ceil(((delta_u > 0) + floor(u) - u) * delta_u_inv);
}

// Equation 9 (Deakin and Knackstead)
int delta_i(int3 delta_i3) {
        return max(min3(delta_i3), 1);
}
```

4.3.5 Chebyshev distance maps

The final method that was implemented was Deakin's *Chebyshev distance maps* [17]. The general idea of this method is to extend the empty space skipping to allow dynamic jumps. The occupancy map is transformed into a map that explains for each region what the shortest distance to a non-empty region is. This is the maximum distance the ray may skip before potentially reaching interesting values. Once again, the map is created by a compute shader. This is done by three transformations implemented as their own kernels; one for each dimension.

The first transformation concerns the X-axis. It first iterates through each x value of every yz-plane, finding the distance between each non-empty element in that plane and stores it. Afterwards, it does the same thing in reverse to find the correct distance. The second and third transformations looks at the y and z axes, respectively. The Y transformation uses the output from the X transformation, investigating whether there is a closer non-empty region on the y-axis than on the x-axis. The Z transformation does the same thing with the output from the Y transformation as its input, searching along the z axis. Listing 4.14 shows the X transformation, and Listing 4.15 shows the Y transformation. The Z transformation can be made by switching the y's for the z's [17].

Code listing 4.14: Chebyshev generation, X transformation

```
[numthreads(1, 8, 8)]
void Trans1(uint3 id : SV_DispatchThreadID)
{
    int3 position = uint3(0, id.yz);
    uint width, height, depth;
    OutMap.GetDimensions(width, height, depth);

    if (id.y >= height || id.z >= depth)
        return;

    uint prev = uint(floor(InMap[position] * 255));
    OutMap[position] = ByteToFloat[uint2(prev, 0)];

    [loop]
    for (position.x = 1; uint(position.x) < width; position.x++)
    {
        uint val = min(prev + 1, uint(floor(InMap[position] * 255)));
        OutMap[position] = ByteToFloat[uint2(val, 0)];
        prev = val;
    }

    [loop]
    for (position.x = width - 2; position.x >= 0; position.x--)
    {
        uint val = min(prev + 1, uint(floor(OutMap[position] * 255)));
        OutMap[position] = ByteToFloat[uint2(val, 0)];
        prev = val;
    }
}
```

Code listing 4.15: Chebyshev generation, Y transformation

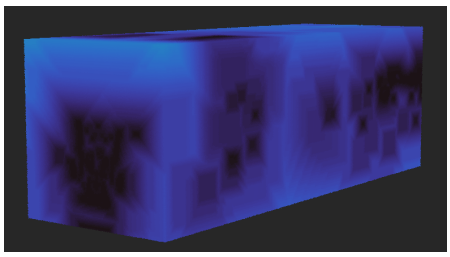
```
[numthreads(8, 1, 8)]
void Trans2(uint3 id : SV_DispatchThreadID)
{
    uint3 position = id;
    uint width, height, depth;
    OutMap.GetDimensions(width, height, depth);

    if (id.x >= width || id.z >= depth)
        return;

    [loop]
    for (position.y = 0; uint(position.y) < height; position.y++)
    {
        uint distance = floor(InMap[position] * 255);

        [loop]
        for (uint n = 1; n < distance; n++)
        {
            if (position.y >= n)
            {
                uint distance_n = floor(InMap[position - uint3(0, n, 0)] * 255);
                distance = min(distance, max(n, distance_n));
            }

            if ((position.y + n) < height && n < distance)
            {
                uint distance_n = floor(InMap[position + uint3(0, n, 0)] * 255);
                distance = min(distance, max(n, distance_n));
            }
        }
        OutMap[position] = ByteToFloat[uint2(distance, 0)];
    }
}
```

The transformations are adapted from Deakin's paper [17] to work with HLSL in Unity. While he used unnormalized values, this was not possible in this version of Unity and HLSL. To compensate a linear color lookup table, which essentially converts an integer between 0 and 255 to a float between 0 and 1, was made. This is the ByteToFloat texture that is referenced in Listings 4.14 and 4.15. Fig. 4.11 shows an example of a Chebyshev distance map using block sizes of 2 and 4. Fig. 4.12 shows the same slices in the normal view.

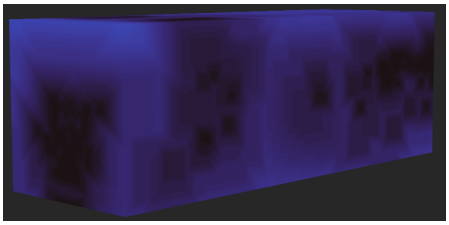
The ray sampling is similar to how *occupancy map* did it, but the delta_i function is different. Also, the value taken from the Chebyshev distance map is a distance rather than practically a boolean, but the logic for determining empty space is still the same, i.e. "empty = distance > 0". Listing 4.16 shows the new Δi:

Code listing 4.16: Chebyshev distance map function for determining next iteration value i

```
int3 delta_i3(float3 delta_u, float3 u, float3 delta_u_inv, half dist) {
        return int3(ceil(((-delta_u > 0) + mad(sign(delta_u), dist, floor(u)) - u)
            * delta_u_inv));
}
```



(a) Chebyshev distance map slice of a CT scan of a pig, blocksize = 2



(b) Chebyshev distance map slice of a CT scan of a pig, blocksize = 4

(c) Slice of a Chebyshev distance map slice of a CT scan of a pig, blocksize = 2

(d) Slice of a Chebyshev distance map slice of a CT scan of a pig, blocksize = 4

Fig. 4.11. Subfigures a and b show the full Chebyshev distance maps of a CT scan of a pig, while c and d show a slice. All are shown in the ramp view, with a brighter color implying a larger distance

(a) Slice of a Chebyshev distance map slice of a CT scan of a pig, blocksize = 2

(b) Slice of a Chebyshev distance map slice of a CT scan of a pig, blocksize = 4

Fig. 4.12. Slices of Chebyshev distance maps for a CT scan of a pig. More red means a larger distance

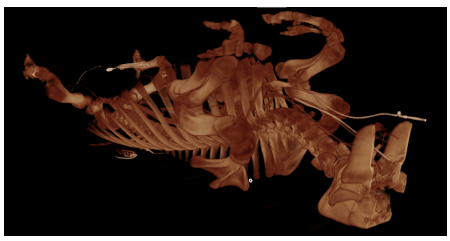
4.3.6 Diffuse shading

In order for diffuse shading to be possible, both a light source and surface normals are required. This can not be done the usual way, which relies on the mesh, as it would generate the wrong normals (The render is a transparent texture, not a mesh). To generate the normals, a dirrent route had to be taken. Deakin [17] needed the normals to use them for 2D transfer functions. This project is using an adapted version of Deakin's Algorithm 1 [17, Al. 1], storing the gradient directions in a four-channel 3D texture. The RGB values are the gradient direction, and the A value is the volume value. This is implemented as a compute shader, displayed in Listing 4.17.

Code listing 4.17: Normal map generation

```
[numthreads(8,8,1)]
void CSMain (uint3 id : SV_DispatchThreadID)
{
    float density = Volume[id];
    float3 normal = float3(0, 0, 0);

    int3 volumeDims;
    uint mip;
    Volume.GetDimensions(0, volumeDims.x, volumeDims.y, volumeDims.z, mip);

    int2 k = int2(1, -1);
    float3 gradientDir = 0.25 * (
      k.xyy * Volume[clamp(id + k.xyy, int3(0, 0, 0), volumeDims)] +
      k.yyx * Volume[clamp(id + k.yyx, int3(0, 0, 0), volumeDims)] +
      k.yxy * Volume[clamp(id + k.yxy, int3(0, 0, 0), volumeDims)] +
      k.xxx * Volume[clamp(id + k.xxx, int3(0, 0, 0), volumeDims)]
    );
    normal = normalize(gradientDir);

    Result[id] = float4(normal, density);
}
```

The light direction is defined as $rayDir + |min3(rayDir)|$, forcing the direction to be positive. The sampled color, src, is now calculated as $src.rgb * (I_{ambient} + max(0, volumeData.rgb \cdot lightDir))$, where $I_{ambient}$ is the ambient intensity, and volumeData.rgb is the local normal of the sample. As an optimization, both the ambient color and the diffuse constant are assumed to be 1. Fig. 4.13 shows the pig rendered with Chebyshev distance maps, both with and without shading. Fig. 4.14 shows the same pig with and without shading rendered on the HoloLens, along with a thorax dataset with and without shading.



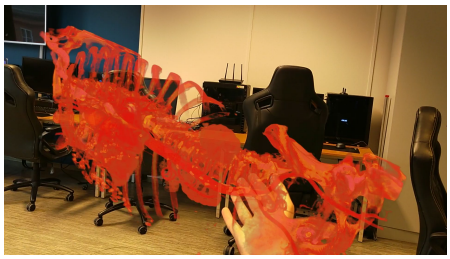
(a) Render without shading

(b) Render with shading

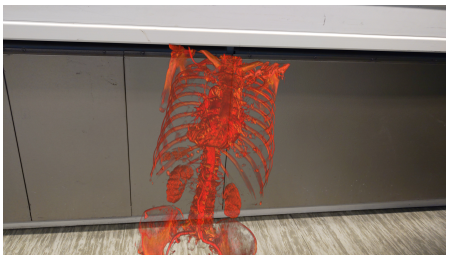
Fig. 4.13. Volume renders of a CT scan of a pig using Chebyshev distance maps (blockSize=4), both with and without diffuse shading

(a) Pig without shading

(b) Pig with shading



(c) Thorax without shading



(d) Thorax with shading

Fig. 4.14. The pig dataset and a thorax CT dataset rendered directly on a HoloLens 2 with and without diffuse shading

Chapter 5

Results

The technical results, i.e. performance, was gathered at the end of the project after the implementation work had ended. These results contain performance data against shaders and their configurations, as well as memory overhead, scale, and resolution. Analysis regarding memory, scale, and resolution was only performed for the best performing configuration of chebyshev distance maps due to time constraints. Other than that, they were tested five times per shader-configuration-volume combination, as explained in Section 3.3.1. The volumes that were used in the general profiling in a search for the best configurations are displayed in Table 5.1, along with their resolution, storage size, and scale. Note that volume US3 is a series of 13 ultrasound volumes with a storage size of 7.6 MB each. The volumes called "[volume] BC4" are compressed using the BC4 compression algorithm, thus the smaller storage size.

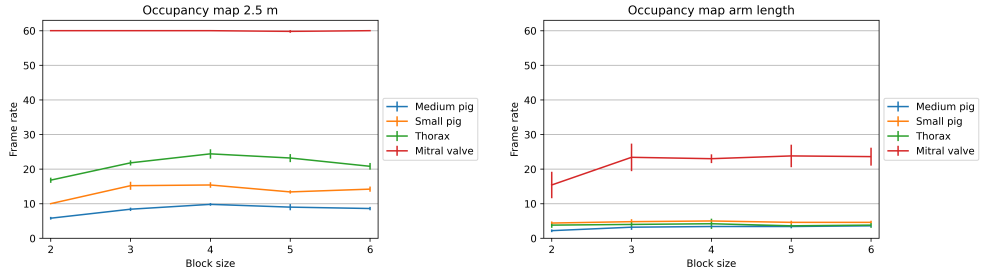
Table 5.1: Volumes used for profiling with data type, resolution (dimensions of the 3D matrix), storage size in megabytes (MB), and scale measured in meters

Volume	Type	Resolution	Storage [MB]	Scale [m]
Medium pig	CT	256 x 256 x 735	45.9	0.512 x 0.512 x 0.1469
Small pig	CT	154 x 154 x 441	10.0	0.512 x 0.512 x 0.1469
Thorax	CT	512 x 512 x 310	77.5	0.286 x 0.286 x 0.620
Thorax BC4	CT	512 x 512 x 310	38.8	0.286 x 0.286 x 0.620
Mitral valve	US	200 x 200 x 200	98.8	0.2 x 0.2 x 0.2
Mitral valve BC4	US	200 x 200 x 200	49.9	0.2 x 0.2 x 0.2

The qualitative results, i.e. the feedback from clinical experts, was gathered during the quantitative data generation period (performance profiling) when the frame rate turned out to be sufficient for a demo. The volume used for the demo was "Mitral valve".

5.1 Profiling results

The shaders that were tested with different configurations were the occupancy map and chebyshev distance map shaders. Each figure shows the performance of the occupancy map shader in two cases: with the volume being 2.5 m away from the viewer, and the volume being an arm's length away. Every test was performed with a quality factor of 1. The graphs use the standard deviance of the five tests performed for each shader-configuration-volume combination as a measure of reliability. Every test used an early ray termination value of 0.95. Fig. 5.1 shows these performance results for the occupancy map shader per block size on the uncompressed volumes.

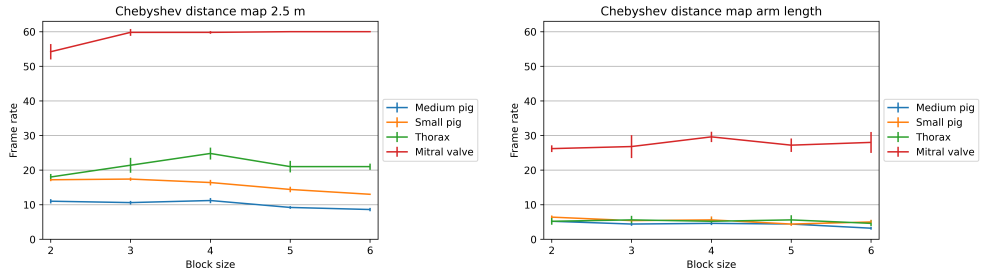


(a) Performance measured with the volume placed 2.5 m away from the viewer

(b) Performance measured with the volume placed arm's length away from the viewer

Fig. 5.1. Performance measured in frames per seconds of the occupancy map shader per block size with the volume placed 2.5 m away from the viewer vs arm's length from the viewer

Similarly, Fig. 5.2 shows the performance of the Chebyshev distance map shader with the same configurations as the occupancy map shader used in Fig 5.1.



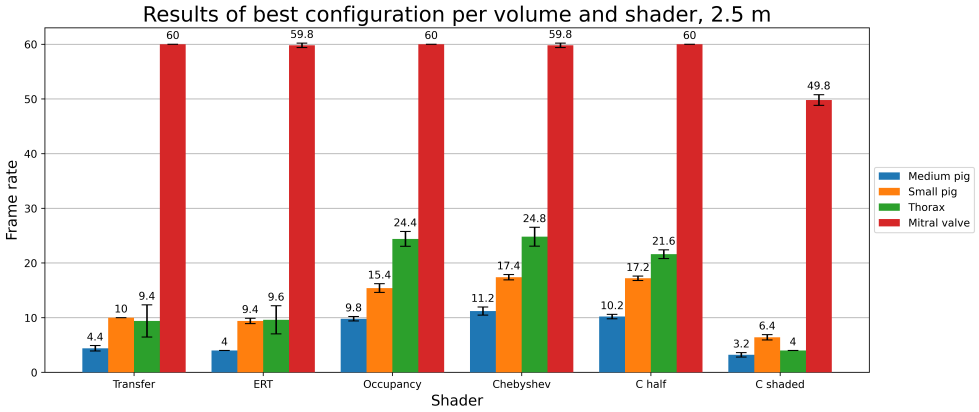
(a) Performance measured with the volume placed 2.5 m away from the viewer

(b) Performance measured with the volume placed arm's length away from the viewer

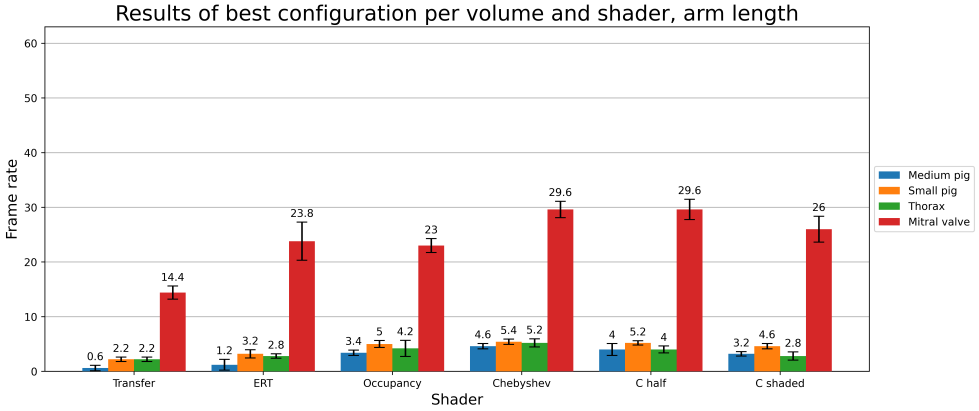
Fig. 5.2. Performance measured in frames per seconds of the Chebyshev distance map shader per block size with the volume placed 2.5 m away from the viewer vs arm's length from the viewer

Using the results from these tests, a seemingly optimal block size was determ-

ined for each volume: $B = 4$ in all cases except for "small pig", which preferred $B = 3$. The results of each "optimal" block size is visualized in Fig. 5.3, which shows each volume's best average performance grouped per shader. This is also shown for both cases: near and far. These graphs show all the shaders that were tested. The shaders "C half" and "C shaded" are the chebyshev shader using half precision for the color blending and the chebyshev shader with diffuse shading, respectively. The shaders are explained in further detail in Chapter 4.



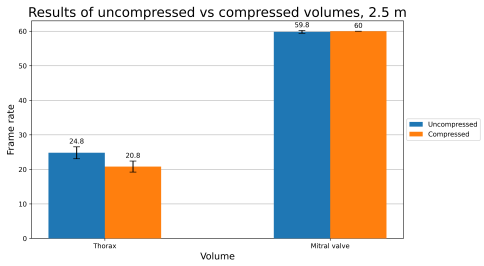
(a) Performance measured with the volume placed 2.5 m away from the viewer



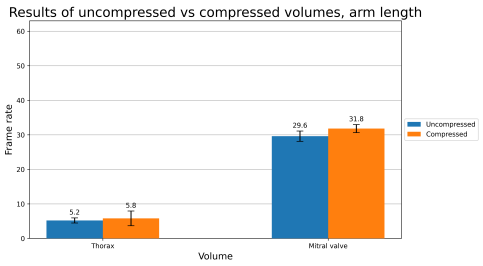
(b) Performance measured with the volume placed arm's length away from the viewer

Fig. 5.3. Performance measured in frames per seconds of the volumes grouped per shader with the volume placed 2.5 m away from the viewer vs arm's length from the viewer

As a final test, the chebyshev shader was tested with the BC4 compressed volumes, Mitral valve BC4 and Thorax BC4. Only the volume itself is compressed; the occupancy map and transfer function textures are the same as usual. The difference in performance between the uncompressed and compressed versions of the volumes is visualized in Fig. 5.4.



(a) Performance measured with the volume placed 2.5 m away from the viewer



(b) Performance measured with the volume placed arm's length away from the viewer

Fig. 5.4. Performance measured in frames per seconds of volumes grouped with their compressed counterpart with the volume placed 2.5 m away from the viewer vs arm's length from the viewer

Pearson's product moment correlation coefficient was calculated for the framerate and resolution, storage size, and scale for both the near and far cases. Table 5.2 shows the correlation factor between frame rate and the other variables. This was measured for the Chebyshev distance map shader with the best configurations. Resolution and scale were calculated by multiplying their respective elements, e.g. a resolution of 512 x 512 x 735 becomes 192 675 840.

Table 5.2: Pearson's correlation coefficient between frame rate and resolution, max dimension, storage size (Total and for one image), and scale

Case	Resolution	Max dimension	Total storage	Single storage	Scale
2.5 m	-0.585	-0.928	0.578	-0.583	-0.937
Arm length	-0.677	-0.887	0.541	-0.670	-0.875

5.2 Expert tests

For the demo, which doubled as an unstructured interview, three themes were focused on; this was elaborated on in Section 3.3.2. These were:

- User experience (eye strain, good aspects, room for improvement etc.)
- Information about how they use 3D visualization in their usual work life
- Which use cases they thought may be the best fits for volume rendering on HoloLens

The participants in the interview and demo included two echocardiologists, one anesthesiologist, Anders Tasken [25] who worked on this project a few years ago, and my supervisor Gabriel Kiss. The key takeaways from the interview were the following, based on quotes and comments during the demo and afterwards:

- The improved depth perception is very welcome, especially for those less experienced with ultrasound

- Diffuse shading increases the depth perception
- The volume does not necessarily need to be very close to the user
- Facilitation for cooperation is critical
- Automatic orientation of the volume and the ability to instantly flip the volume would be welcome
- It would be helpful to see both sides of the volume simultaneously
- Being able to walk around the volume is "interesting"
- Possible use cases include planning where the precise location is critical, e.g. when repairing defects in the heart, showing the surgeon where to cut
- A possibility to fuse ultrasound images and CT images would be useful
- It would also be useful to add color Doppler to the ultrasound render

A demo video can be found here:

- **https://youtu.be/UwA2qusdhI0**

Chapter 6

Discussion

6.1 Profiling

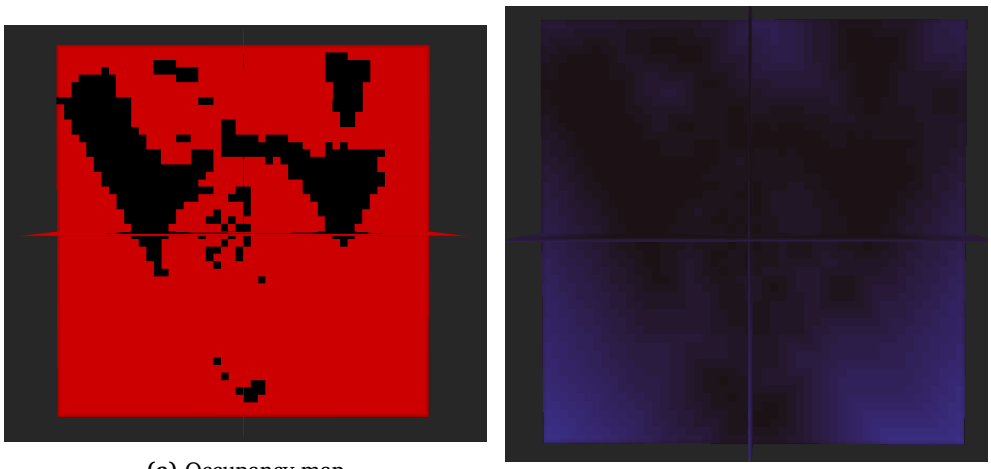
Table 5.1 shows the volumes used in the profiling. Of these there are two volumes of equal scale and different resolution, where the small pig is a downscaled version of the medium pig. Because they will cover the same amount of pixels and contain similar data, they can tell to which degree a larger texture will affect the performance. One obvious aspect of the algorithm that can be affected is the amount of sample points, where the maximum dimension of the volume is one of the factors as seen in Equation 3.1. There are also volumes of similar resolution to and smaller scale than the medium pig: the thorax volume. This can be used to gain insight regarding whether the scale, or rather the amount of pixels covered by the volume's bounding box, affects the performance. This is because the pixels that are not covered by the volume will not use the volume rendering shader. There are also a couple of compressed textures, Thorax BC4 and Mitral valve BC4, that were used to explore the possibility of optimized storage improving performance [31], or at least memory usage. The volume with the largest resolution, Thorax, has a lower resolution than Deakin and Knackstedt's smallest volume which had a resolution of 492x492x442. Also, they used a larger scale: $1\,\mathrm{m}^3$ placed $\sqrt{1}\,\mathrm{m}$ away from the camera [17].

Figures 5.1 and 5.2 contain the results from finding the best configurations of occupancy maps and Chebyshev distance maps for each volume, respectively. For occupancy maps, a higher block size will lead to fewer occupancy map samples and larger leaps when an empty region is entered. However, each region will likely contain more empty space as seen in Fig. 4.10 when comparing the two slices. This will then lead to more unnecessary volume samples. On the other hand a small block size leads to a large occupancy map, increasing the amount of occupancy map samples and decreasing the amount of skipped blocks per sample. This is most likely why most of the graphs seem to fit with a quadratic function; a small block size gives smaller skips, but a high block size can hide large amounts of empty space within "non-empty" blocks. A similar effect can be seen with the Chebyshev distance maps, as they use occupancy maps as their base. The added

bonus of Chebyshev distance maps is that they can skip longer with lower block sizes than an occupancy map. This is likely the reason why the performance of larger Chebyshev distance maps is better than the performance of larger occupancy maps. There are some differences, however. For example, the optimal configurations of Chebyshev distance maps seem to be more pronounced, especially for the thorax and the mitral valve. Also, specifically the mitral valve, an ultrasound dataset with a very different structure compared to the other datasets, seems to benefit from Chebyshev distance maps. The shape of the graphs are similar to Deakin and Knackstedt's results regarding block size [17].

Fig 5.3 shows the final results of the profiling using the average performance of the optimal configurations for each shader and volume. This also includes a shader using half precision for color blending and a shader that includes diffuse shading. Both of these use Chebyshev distance maps. Starting from the left, both the unoptimized shader using transfer functions and the ERT shader seem to perform similarly in the 2.5 m case. However, ERT is more dominant when the volume is closer to the camera and more pixels are covered, especially when we look at the ultrasound. This could mean that in general ERT does not give a huge boost to performance, but when the workload is increased the speedup from ERT is more prominent. One step to the right are the occupancy map results. For every volume except the ultrasound mitral valve there is a significant speedup in both the far and near cases. Those three volumes contain several larger continuous areas of empty space, thus lending themselves well to occupancy maps. The same goes for the mitral valve images, but the speedup does not seem to be as drastic. This may be because the dataset has a lower resolution, and therefore fewer sample points, than the other volumes. One will therefore see a smaller difference when skipping the few samples that are being done.

The three rightmost results are from three different Chebyshev shaders. The original, labelled "Chebyshev", shows a speedup for every volume. At this point the frame rate is around double the frame rate of the unoptimized shader. This optimization is also very significant for the mitral valve, considering how occupancy maps did not have a large impact. A likely reason is how the ultrasound volume is structured. While the other volumes have quite tight bounding boxes, the actual ultrasound data is cone-shaped. With Chebyshev distance maps, the shader can jump right into the area where non-empty data is located. The initial jump for ultrasound can be quite large if the ray comes from one of the volume's corners. In addition, the volume had a lot of empty space. Fig. 6.1 shows a slice of an occupancy map and a Chebyshev distance map, highlighting the cone shape and the amount of empty space. For the other volumes, the ray will often be close to non-empty regions, bringing out less of the benefit from Chebyshev distance maps. Deakin and Knackstedt made an *anisotropic* version of this skipping method, considering the direction of the ray. However, they concluded that the performance gain was not that great compared to the amount of preprocessing needed [17]. However, both the pig and the thorax can be compared to the snake volume they used, which is seems to have benefited from this method [17]. Another place



(a) Occupancy map

(b) Chebyshev distance map

Fig. 6.1. Slices of an occupancy map and a chebyshev distance map for a mitral valve ultrasound volume

where the performance could be better is when leaving a non-empty region and entering a very large area of empty space. Since the ray is still close to the volume, it will need a warm-up of sorts. Sampling the distance map will gradually give larger jumps, as long as the ray gets farther from non-empty regions. This is in a way similar to the octree implementations by both Levoy and Krüger and Westermann, where the empty space skipping would jump up and down levels based on the size of empty regions [2, 15].

An interesting observation regarding the half precision shader is how the performance is actually *worse* than with normal floating point precision. It is possible that there is some implicit casting that is being done, hurting the performance. In theory, if everything works as expected and intended, the performance should have been improved [28, p. 329]. The diffuse shading works as expected, on the other hand. Adding more multiplications, a max function and a dot product for every volume sample unsurprisingly lowered the performance. There are actually supposed to be more factors than in Equation 3.9 (A scalar diffuse factor, a scalar light intensity, and the light color), but since these are assumed to be 1, they have not been added to the code. This shader performs better than the unoptimized shader in the near case, but certainly worse than the normal Chebyshev shader. In addition, it requires precomputation of the normal map.

An optimization that was tested for Chebyshev distance maps was to not perform the color and lighting calculations when the opacity was zero, as those samples would not contribute to the final color. While this worked well on the laptop with a GTX 1660 Ti GPU used for development and simple debugging, it was detrimental to the performance on HoloLens 2. This is likely due to thread divergence [28, Ch. 4.4], as removing the if-statement improved the frame rate significantly. The HoloLens accepted the big branches of roughly equal size for

empty space skipping, likely because that can remove a lot of instructions and that they did not differ often. However, when an extra branch was added inside a branch, it became too much for the HoloLens' GPU. The threads within a group on a GPU like performing the same work, and when the branches become too deep and different too often there will be a lot of stalling [28, Ch. 4.4]. On a CPU, this would not be a problem, and many instructions could be skipped and a speedup could probably be seen (Though volume rendering on a CPU would still be considerably slower than on a GPU due to the high parallelism of GPUs).

Fig. 5.4 shows the results from testing the difference between using uncompressed and compressed textures. The compression method used was BC4, and the compressed volumes were the ultrasound mitral valve and the thorax. The expected outcome was that using compressed textures would not only lower memory usage, but at the same time lead to faster loads [31]. For the most part this held true, though the compressed thorax seems to have struggled a bit. It is unclear if the thorax results are completely valid, as something seemed off during testing. The mitral valve performed as expected, with compression leading to better performance. The compression tests are far from extensive enough, and the reflections regarding the thorax dataset may be painted by bias.

The final step that was performed in the profiling results was calculating the Pearson correlation coefficient between frame rate and resolution, storage size, and scale. Table 5.2 shows the results from this analysis. The coefficient is a number between -1 and 1, with -1 being a clear negative linear correlation and 1 being a clear positive linear correlation. From the tests performed for Chebyshev distance maps, there is a clear negative correlation between the scale and the frame rate. This is most likely related to how many pixels have to be evaluated by the volume rendering shader, as performance tended to decrease significantly when the volume was closer to the camera (thus covering a larger part of the viewport).

The resolution also seems to play a large part, though the correlation is not as strong linearly as with the scale. This may be because of a combination of early ray termination and empty space skipping reducing the true amount of sample points, but there is also another possibility. The resolution itself does not directly affect the amount of sample points; this is done by the max dimension, e.g. for dimensions (256, 256, 735) Equation 3.1 uses 735 as its factor. According to Table 5.2 there is a clear correlation between the max dimension and the frame rate, most likely due to the Equation 3.1. This does not necessarily mean the resolution is irrelevant, as a too large texture will not fit completely in the L2 cache, increasing load times; this follows from the memory hierarchy and the speed of memory accesses further down as seen in Section 2.5.

The storage size of each dataset was also considered. If one only looks at the total storage of each dataset it is possible to believe using more memory somehow makes the algorithm faster. The reason why the correlation coefficient is positive in this case is because the mitral valve dataset generally produced good results and consists of many smaller volumes. Looking at the storage of a single volume in each dataset paints a different picture. According to the Pearson correlation

coefficient of the storage size of each individual volume, there is a negative correlation between memory usage and frame rate. However, the storage size is more or less only affected by how many voxels there are in each volume. This is why the coefficient is almost identical to that of the resolution. Therefore, the actual storage size most likely follows the same arguments as for resolution.

6.2 Methods

The profiling method was highly susceptible to human error. Since the only part that was automatic in the tests was how frame rate was gathered by the profiler, everything else was tester-controlled. This does not affect the far cases of the profiling since that was done with no interaction, though it gives more uncertainty to the arm's length cases. These were performed with different rotations and what an "arm's length" was could vary based on how slouched or leaning forward the tester was. In order to alleviate this every test was performed five times, taking the average of these and noting their standard deviation to get a measure of quality. The ultrasound mitral valve seems to have been affected by human error, as its standard deviation tended to be a lot higher than other volumes. The thorax also seems to have been affected; at spawn it is rotated in an unnatural way, and could be rotated different ways that could cover the pixels differently.

In defense of this method, one can look at realism. In real life, the volume will probably be rotated and moved in different ways. The user may want to get a closer look sometimes, and they may want to get more of an overview. Using a more human-minded test method could therefore give a more realistic image of the performance. However, the two cases tested in this project can be seen almost as best and worst cases. The user will most likely rarely want to have the image right in front of their nose, nor do they want to have it so far away it is hard to see the details. Despite this slight unrealism, this was the simplest way to test without making a strong test framework. Since each test was performed several times it provides each data point with more accuracy, and the standard deviation is a measure of said accuracy. Despite the room for human error and slightly unrealistic test cases, the profiling gives a valid picture of the performance.

In order to provide even more accuracy, it would be possible to automatically do both rotation and translation of the volume, calculating an accumulated mean of the program's frame rate. After a few frames of each rotation and position, volume rendering could stop, the mean frame rate could be written to a text file, and a new test could then be run. This is partially inspired by Deakin and Knackstedt's test methodology [17], combined with the more simple method that was used in this project. The HoloLens would still sit on a tester's head to ensure that the volume is always in the viewport. By doing this, it would be easier to add more locations than the spawn location and the unscientific "arm's length" to the mix. It would also reduce the margin for human error, as all interaction would be controlled by the CPU.

Since the performance was consistently significantly below 30 frames per second,

there were bugs in the optimizations for a while, and that some core functionalities such as dynamic ultrasound and diffuse shading were missing, the demo was not decided on before very late in the project. Therefore, when it did happen it was not very carefully planned. Room for improvement includes voice recording, a specific set of test cases, and more helpful "training" regarding how to interact with a Hololens. These are things to consider when a more realistic application has been made; this project focused on the technical aspects of implementing and optimizing volume rendering on HoloLens 2.

6.3 Research questions

The research questions for this project were as follows:

RQ1: How can volume rendering be implemented on an augmented reality device with a framerate of at least 15 FPS?

RQ2: Does HoloLens 2 improve 3D depth perception of the volume rendered content when compared to a standard display?

RQ3: Is the quality of the renderer sufficient to perceive important anatomic structures?

For the ultrasound dataset it seems like simply adding early ray termination was sufficient for the requirements of RQ1. However, that frame rate is quite unstable, but this seems to stabilize with occupancy maps. For even higher performance, Chebyshev distance maps are viable. The other larger volumes did not reach 15 FPS in the near case, but some did reach it in the far case. It is also worth mentioning that empty space skipping increased the performance sufficiently for more seamless interaction, which means it may be almost usable in medium distances. Since they were much larger and contained a lot of detail, it could help the performance by introducing a slicing functionality that would almost act like empty space skipping. This will be elaborated on in the next section. The short answer to RQ1 is therefore:

RQ1: It is possible to achieve 15+ FPS on a HoloLens 2 if the volume is sufficiently small and by using both early ray termination and Chebyshev distance maps.

According to the experts that tested the program with the ultrasound mitral valve volume, using HoloLens for visualization definitely added better depth perception. They also liked being able to walk around the volume, and noted that the diffuse shading gave even better depth perception. The short answer to RQ2 is:

RQ2: HoloLens 2 improves the 3D depth perception when analyzing volume rendered content compared to a standard display.

The experts were able to identify the anatomic structure of the ultrasound volume, i.e. a mitral valve. A very important note is that this was done using a scan that clearly shows the valve opening and closing, with a well-designed transfer function for that case. The experts that tested the program were experienced echocardiologists, and were used to looking at these kinds of structures. More extensive research with pre-analyzed medical data is required, but from these tests the answer to RQ3 is:

RQ3: The quality of the renderer is sufficient to perceive important anatomic structures

6.4 Hardware comparisons

At the time Levoy presented the algorithm and its optimizations, rendering a skull consisting of 7 405 568 voxels (resulting in 14 million samples) took 1183 seconds, or 19.7 minutes. With the optimizations it took 105 seconds; 1 minute and 45 seconds. This was done on a Sun 4/280 [2], which had a processor with a clock frequency of 16.67 MHz capable of performing 1.6 *MFLOPS* (1 600 000 floating point operations per second). The system had 120 MB of memory [47]. This processor was a single core processor, as the first multicore processor was released in 2001 [27]. Compared to the Adreno 630's 700 GFLOPS (700 000 000 000 FLOPS) [44, 45], and considering the amount of floating point operations required for volume rendering, there is no doubt how the unoptimized algorithm can run much faster as a modern GPU program than on hardware from the 1980's. These are the effects of both Moore's law, Dennard scaling, development of specialized hardware, and all the other technological advancements the last 30 or so years. However, this floating point performance is still significantly lower than that of modern desktop (and laptop) GPUs. Deakin and Knackstedt used NVIDIA's GeForce GTX 1080 for their rendering, using a viewport of 1200^2 pixels [17]. This GPU has a floating point performance of 8.873 TFLOPS (Tera FLOPS) according to Tech Powerup [48], roughly 12.7 times as fast. These three are rather different architectures, and as mentioned earlier the reported performance is affected by the benchmarks that were used among other things, but the floating point performance should still give an indication of the expected performance difference. In addition, the viewport resolution, cache sizes, amount of pixels covered by the volume etc. also play a difference.

Compared to the HMD by Sauer in 2002 [8], the HoloLens 2 is very light. It has no cables going to a different machine, giving more mobility to the user. It does not require any extra cameras in the room, and the necessary cameras for the HoloLens are integrated in the device itself.

6.5 Limitations

There are, as always, a few limitations to the research. One such limitation is that the project only had one researcher. With a larger team it would be possible to implement more optimizations in parallel, as well as opening for someone to focus on more interactions that could be implemented. Combined with the small team size, the main bulk of the research was done over the course of one semester, and the pre-study was performed simultaneous to two other university courses. Getting access to Tasken, Barstad, and Tagestad's [25] application alleviated the time constraint by making it easier to start implementation.

A very different limitation comes from the hardware. Microsoft's HoloLens 2's processing power and memory is comparable to that of a smart phone [36], not taking the holographic processing unit into account. While they have come a long way, it is still demanding to run ray-tracing algorithms. While not a limitation on the project as a whole, considering the comparison of optimizations, it still provides an extra challenge for reaching user friendly frame rates. This limitation, however, is part of the motivation itself and does not hinder the research in a very meaningful way except making it hard to perform profiling in the worst cases. Unity was used in order to more easily make the application run on HoloLens. With MRTK and Microsoft's other tools, "everything" was good to go on the mixed reality part. However, that does constrain the program to run the way Unity wants. This means compute shaders are a bit more complicated to work around, it is not possible to compress a 3D texture with a native function, and memory layout is not as simple to define. That does not mean that it is impossible, but it may lead to unexpected hacky methods at times. HLSL, at least in Unity, does not support unnormalized texture values, so even with an R8 (byte values) texture the values will be floats in the range 0-1. Because of this, a few conversions were made between float and byte by multiplying by 255 and flooring when loading from the input texture, and loading from a byte texture with values 0-255 (which are interpreted as floats between 0 and 1) from a lookup-table texture as seen in Listings 4.14 and 4.15. **This means that the Chebyshev distance map implementation may be incomplete, and the results could potentially be better**. The resulting rendered volume does not seem to be affected by this, however.

6.6 Future work

There are several directions further research can go. The technical aspects will be discussed first, followed by more of an application focus.

One optimization that has not been tested in this project is using Morton order [32, 33], also known as Z-order, matrices. When a matrix is stored as a Morton-order matrix, each element will be close to each other in both index and in physical position. It should be possible to rearrange the texture elements in the correct order, but there may be difficulties regarding filtering between elements. Locality is a very central theme in optimization [28, Appendix B]. When shooting a

ray through the volume, spatial locality is not necessarily leveraged as much as it should be with conventional storage. Deakin and Knackstedt also introduced their anisotropic Chebyshev distance maps. Although they concluded that the speedup was not impressive, it could still be interesting to see how it performs on HoloLens 2. A third optimization that can be considered is Li, Mueller, and Kaufmann's occlusion clipping to see how that holds up against the simpler early ray termination [16].

A very slow optimization that may have a large impact is simply waiting for better hardware. With luck an augmented reality headset with some hardware accelerated ray tracing pops up, making volume rendering a lot easier in AR. Until then, further research will have to look at optimizations.

In order to get more data on optimizations, more varied volumes could be generated. In the case of very large volumes, it would be interesting to explore the possibility of only storing the non-empty regions on the GPU to lower the memory requirements. This has been done in other research projects [17]. There should also be much more rigorous testing, preferably controlled by the CPU. The exact framework and format is up for debate, but the suggestion from Section 6.2 is a good starting point. Part of the testing should also focus more on compression. For example, more compression formats could be explored, the effect of compressing the empty space skipping structure could be interesting, and compressing the diffuse shading texture (volume with gradients) is also beneficial due to the memory usage.

In could be beneficial to add slicing or region of interest functionality to the application. This way it is possible to "remove" the parts of the volume that are not interesting to the user, while also doubling as a supplement to empty space skipping. The calculated amount of sample points will be the same, but the program will skip to the first sample point that is inside the accepted area. Since the amount of pixels covered by the volume negatively affects the performance, removing chunks of uninteresting data should be helpful.

Further work also includes working on the feedback given by the practitioners. For example, it is very important to facilitate for cooperation. Since they always work in teams, it is key to be able to show their coworkers what they are explaining. Therefore, implementing collaborative features, e.g. being able to see the same things on different HoloLenses, is an important step. In order to place the volume at the same location, an image/QR-code or similar can be used as an anchor. The echocardiologists also wanted color Doppler mode to be added to the volume. This could be an extra transfer function or need more data, but it should be possible to add without too much of an impact. Sometimes, the experts use CT volumes fused with ultrasound. This could prove more difficult, but if one can remove the parts of the CT volume that are not as important, it should increase the CT performance, thus making it possible. An interesting challenge is to show both sides of the volume simultaneously, as this could require an extra volume render. There may be tricks that can be used to achieve this effect, but that requires more research. A much simpler interaction technique that may alleviate some of the

need for two renders simultaneously is adding a way of flipping the volume by the push of a button or with a voice command. This would be a simple transformation, with minimal performance drops. More future work regarding use cases and needs of practitioners can be found by conducting a survey with this goal.

Chapter 7

Conclusion

An instantiation of volume rendering on an augmented reality device has been made. Several optimizations have been implemented and evaluated. These optimizations are transfer functions as a lookup table, early ray termination, and empty space skipping with occupancy maps and Chebyshev distance maps. For lower resolutions, the frame rate reached around 30-60 FPS depending on the distance between viewer an volume, whereas higher resolutions reached around 5-20 depending on the distance. The quality of the renders has been evaluated both with and without diffuse shading. The evaluation of the quality was done together with two echocardiologists and an anesthesiologist, using an ultrasound scan of a mitral valve as a test. According to these experts, there is a clear advantage regarding depth perception on the HoloLens, and they were able to identify the anatomical structures.

Further research can include further optimizations, but also more information about the needs of practitioners should be researched. The performance for small textures show a strong possibility of implementing volume rendering directly on head-mounted augmented reality devices. For larger textures, more optimization is needed to make the user experience more seamless, but it shows promise.

Bibliography

[1] B. N. Brunvoll, *Volume rendering on a mixed reality device - preparatory project*, (Unpublished - internal document), Dec. 2021.

[2] M. Levoy, 'Efficient ray tracing of volume data,' *ACM Transactions on Graphics (TOG)*, vol. 9, no. 3, pp. 245–261, 1990.

[3] C. Moro, C. Phelps, P. Redmond and Z. Stromberga, 'Hololens and mobile augmented reality in medical and health science education: A randomised controlled trial,' *British Journal of Educational Technology*, vol. 52, no. 2, pp. 680–694, 2021.

[4] J. A. Jones, J. E. Swan, G. Singh, E. Kolstad and S. R. Ellis, 'The effects of virtual reality, augmented reality, and motion parallax on egocentric depth perception,' in *Proceedings of the 5th symposium on Applied perception in graphics and visualization*, 2008, pp. 9–14.

[5] H. AS. (2021). 'Holocare,' [Online]. Available: `https://holocare.com/` (visited on 25/10/2021).

[6] M. S.A. (2021). 'Carnalife holo,' [Online]. Available: `https://medapp.pl/en/carnalife-holo/` (visited on 25/10/2021).

[7] Siemens, *Cinematic rendering*, `https://www.siemens-healthineers.com/medical-imaging-it/syngo-carbon-products/cinematic-rendering`, (Accessed on 06/11/2022).

[8] F. Sauer, A. Khamene, B. Bascle, S. Vogt and G. Rubino, 'Augmented-reality visualization in imri operating room: System description and preclinical testing,' in *Medical Imaging 2002: Visualization, Image-Guided Procedures, and Display*, International Society for Optics and Photonics, vol. 4681, 2002, pp. 446–454.

[9] O. Kutter, A. Aichert, C. Bichlmeier, J. Traub, S. Heining, B. Ockert, E. Euler and N. Navab, 'Real-time volume rendering for high quality visualization in augmented reality,' in *International Workshop on Augmented environments for Medical Imaging including Augmented Reality in Computer-aided Surgery (AMI-ARCS 2008), New York, USA*, Citeseer, 2008, pp. 104–113.

[10] C. Bichlmeier, F. Wimmer, S. M. Heining and N. Navab, 'Contextual anatomic mimesis hybrid in-situ visualization method for improving multi-sensory depth perception in medical augmented reality,' in *2007 6th IEEE and ACM international symposium on mixed and augmented reality*, IEEE, 2007, pp. 129–138.

[11] M. de Ridder, Y. Jung, R. Huang, J. Kim and D. D. Feng, 'Exploration of virtual and augmented reality for visual analytics and 3d volume rendering of functional magnetic resonance imaging (fmri) data,' in *2015 Big Data Visual Analytics (BDVA)*, IEEE, 2015, pp. 1–8.

[12] R. A. Drebin, L. Carpenter and P. Hanrahan, 'Volume rendering,' *ACM Siggraph Computer Graphics*, vol. 22, no. 4, pp. 65–74, 1988.

[13] I. Buck, *The evolution of gpus for general purpose computing*, `https://www.nvidia.com/content/GTC-2010/pdfs/2275_GTC2010.pdf`, (Accessed on 10/05/2022), Sep. 2010.

[14] B. Cabral, N. Cam and J. Foran, 'Accelerated volume rendering and tomographic reconstruction using texture mapping hardware,' in *Proceedings of the 1994 symposium on Volume visualization*, 1994, pp. 91–98.

[15] J. Kruger and R. Westermann, 'Acceleration techniques for gpu-based volume rendering,' in *IEEE Visualization, 2003. VIS 2003.*, IEEE, 2003, pp. 287–292.

[16] W. Li, K. Mueller and A. Kaufman, 'Empty space skipping and occlusion clipping for texture-based volume rendering,' in *IEEE Visualization, 2003. VIS 2003.*, IEEE, 2003, pp. 317–324.

[17] L. J. Deakin and M. A. Knackstedt, 'Efficient ray casting of volumetric images using distance maps for empty space skipping,' *Computational Visual Media*, vol. 6, no. 1, pp. 53–63, 2020.

[18] M. M. Deza and E. Deza, 'Encyclopedia of distances,' in *Encyclopedia of distances*, Springer, 2009, pp. 1–583.

[19] 'Augmented reality,' in *Encyclopedia of Multimedia*, B. Furht, Ed. Boston, MA: Springer US, 2006, pp. 29–31, ISBN: 978-0-387-30038-2. DOI: `10.1007/0-387-30038-4_10`. [Online]. Available: `https://doi.org/10.1007/0-387-30038-4_10`.

[20] H.-K. Wu, S. W.-Y. Lee, H.-Y. Chang and J.-C. Liang, 'Current status, opportunities and challenges of augmented reality in education,' *Computers & education*, vol. 62, pp. 41–49, 2013.

[21] K. Lee, 'Augmented reality in education and training,' *TechTrends*, vol. 56, no. 2, pp. 13–21, 2012.

[22] Z. Szalavári, D. Schmalstieg, A. Fuhrmann and M. Gervautz, ' studierstube": An environment for collaboration in augmented reality,' *Virtual Reality*, vol. 3, no. 1, pp. 37–48, 1998.

[23] S. Lukosch, M. Billinghurst, L. Alem and K. Kiyokawa, 'Collaboration in augmented reality,' *Computer Supported Cooperative Work (CSCW)*, vol. 24, no. 6, pp. 515–525, 2015.

[24] R. Woll, T. Damerau, K. Wrasse and R. Stark, 'Augmented reality in a serious game for manual assembly processes,' in *2011 IEEE International Symposium on Mixed and Augmented Reality-Arts, Media, and Humanities*, IEEE, 2011, pp. 37–39.

[25] A. Tasken, E. Barstad and S. Tagestad, *Final report - tfe4503*, Final delivery in the course TFE4503 at NTNU in Trondheim, 2019.

[26] R. Kramme, K.-P. Hoffmann, R. S. Pozos and T. M. Buzug, *Computed Tomography*, 1st ed. 2011. Berlin, Heidelberg : Springer Berlin Heidelberg : pp. 311–342, ISBN: 3-540-74657-9.

[27] IBM, *Power 4: The first multi-core, 1ghz processor*, `https://www.ibm.com/ibm/history/ibm100/us/en/icons/power4/`, (Accessed on 04/21/2022), Aug. 2011.

[28] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach; 6th ed.* Cambridge, MA: Morgan Kaufmann, 2019.

[29] M. J. Flynn, 'Very high-speed computing systems,' *Proceedings of the IEEE*, vol. 54, no. 12, pp. 1901–1909, 1966.

[30] P. S. Pacheco, *An Introduction to Parallel Programming*. Burlington, MA: Morgan Kaufmann, 2019.

[31] G. Knittel, A. Schilling, A. Kugler and W. Straßer, 'Hardware for superior texture performance,' *Computers & Graphics*, vol. 20, no. 4, pp. 475–481, 1996.

[32] A. E. Nocentino and P. J. Rhodes, 'Optimizing memory access on gpus using morton order indexing,' in *Proceedings of the 48th Annual Southeast Regional Conference*, 2010, pp. 1–4.

[33] E. W. Bethel and M. Howison, 'Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning,' *The International Journal of High Performance Computing Applications*, vol. 26, no. 4, pp. 399–412, 2012.

[34] W. A. Wulf and S. A. McKee, 'Hitting the memory wall: Implications of the obvious,' *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, Mar. 1995, ISSN: 0163-5964. DOI: `10.1145/216585.216588`. [Online]. Available: `https://doi.org/10.1145/216585.216588`.

[35] WikiChip, *Snapdragon 850*, `https://en.wikichip.org/wiki/qualcomm/snapdragon_800/850`, (Accessed on 06/13/2022), Oct. 2019.

[36] Microsoft. (2021). 'Hololens 2,' [Online]. Available: `https://www.microsoft.com/en-us/hololens/hardware` (visited on 05/11/2021).

[37] B. J. Oates, *Researching Information Systems and Computing*. London: SAGE Publications, 2006, ISBN: 9781412902243.

[38] Microsoft. (2021). 'What is the mixed reality toolkit,' [Online]. Available: https://docs.microsoft.com/en-us/windows/mixed-reality/mrtk-unity (visited on 29/11/2021).

[39] Unity, *Single-pass stereo rendering for hololens*, https://docs.unity3d.com/Manual/SinglePassStereoRenderingHoloLens.html, (Accessed on 06/10/2022), Sep. 2017.

[40] Microsoft. (2021). 'Hololens (1st gen) hardware,' [Online]. Available: https://docs.microsoft.com/en-us/hololens/hololens1-hardware (visited on 05/11/2021).

[41] C. Pietschmann. (2016). 'Detailed hololens hardware specs,' [Online]. Available: https://build5nines.com/microsoft-hololens-2-hardware-specs-vs-hololens-1/ (visited on 05/11/2021).

[42] W. Commons. (2016). 'File:ramahololens.jpg,' [Online]. Available: https://commons.wikimedia.org/wiki/File:Ramahololens.jpg (visited on 05/11/2021).

[43] Qualcomm. (2021). 'Snapdragon 850 mobile compute platform,' [Online]. Available: https://www.qualcomm.com/products/snapdragon-850-mobile-compute-platform (visited on 05/11/2021).

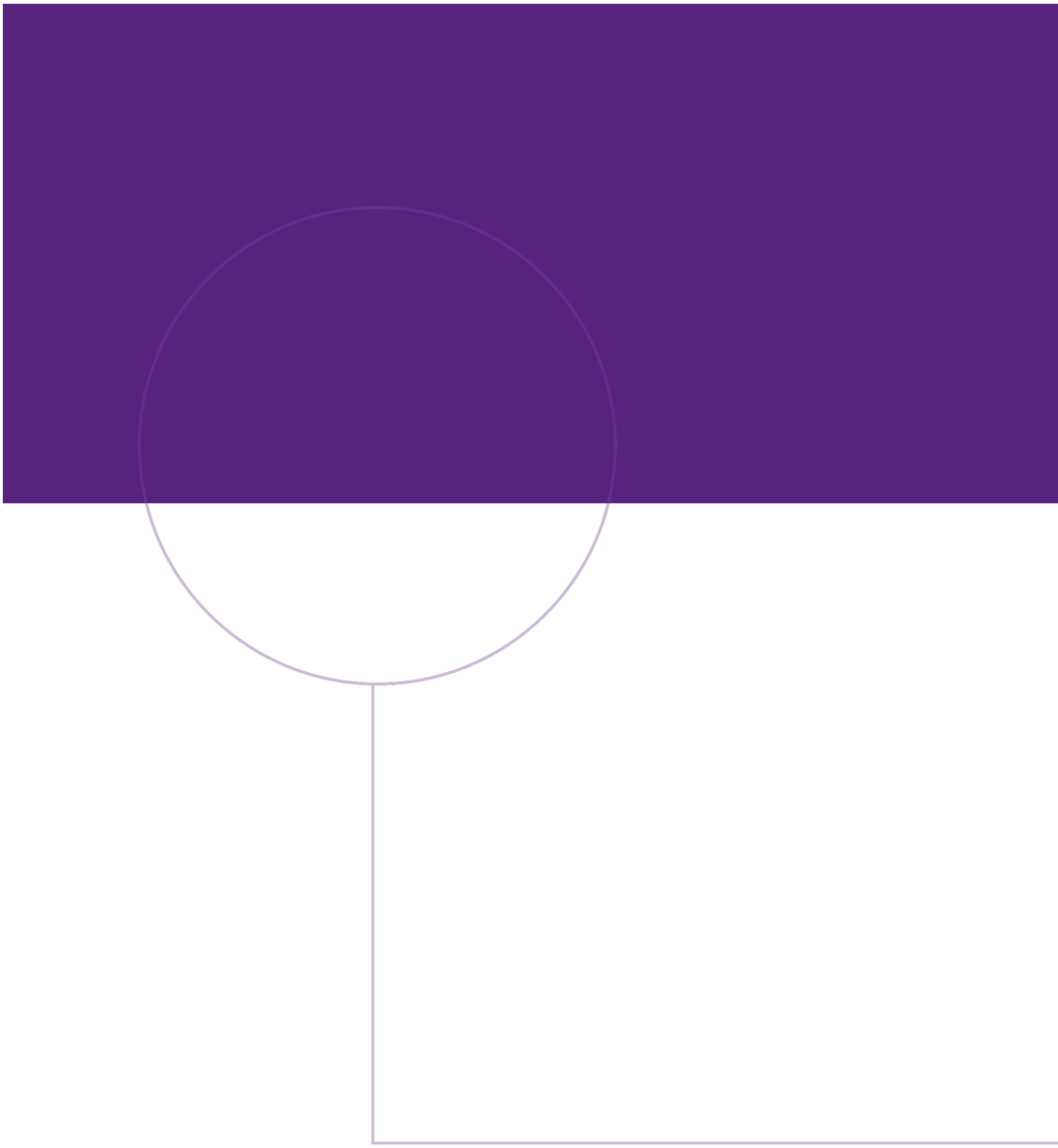
[44] CPU-Monkey, *Qualcomm adreno 630*, https://www.cpu-monkey.com/en/igpu-qualcomm_adreno_630-171, (Accessed on 06/10/2022).

[45] Chipguider, *Adreno 630 gpu @ 650 mhz specs*, https://chipguider.com/?gpu=adreno-630-650-mhz, (Accessed on 06/10/2022).

[46] ingnarmezh, *Texture3d compression issue*, https://forum.unity.com/threads/texture3d-compression-issue.966494/, (Accessed on 05/31/2022), Sep. 2020.

[47] J. W. Birdsall, *The sun hardware reference*, http://www.sunhelp.org/faq/sunref1.html, (Accessed on 04/21/2022), Nov. 1995.

[48] TechPowerup, *Nvidia geforce gtx 1080 specs*, https://www.techpowerup.com/gpu-specs/geforce-gtx-1080.c2839, (Accessed on 06/10/2022).



NTNU

Norwegian University of
Science and Technology