Occlusion in Augmented Reality

Exploring the problem of rendering semi-occluded objects in augmented reality using a model-based approach.

Krzysztof Piotr Kuzma

Thesis submitted for the degree of
Master in Programming and System Architecture
60 credits

Department of Informatics
The Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Occlusion in Augmented Reality

Exploring the problem of rendering semi-occluded objects in augmented reality using a model-based approach.

Krzysztof Piotr Kuzma

Occlusion in Augmented Reality

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

For many decades researchers and manufacturers have tried to push the concept of augmented reality toward a comfortable, beneficial, and seamless experience. The concept of this technology promised high usability for fields such as medicine, architecture, and engineering, while keeping the experience smooth enough to avoid common negative effects such as nausea and dizziness. One of the most fundamental things towards realistic AR experience is occlusion by letting our virtual world and holograms be affected by the real world that surrounds us. In this Master's thesis I discover methods to apply occlusion in AR and the computational challenges that it brings. I focus on mesh-based reconstruction algorithms and consider generating meshes by getting insight into older algorithms like Marching Cubes and Poisson, as well as newly developed solutions. I answer a research question by creating our own mesh-based reconstruction pipeline: 'Can we develop efficient real-time, mesh-based method to occlude virtual objects with real objects in an AR scenario?'. I have concluded that developing such a solution is possible within a reasonable amount of time using Unity Engine and Marching Cubes, yielding good results in stereo-camera setup on modern hardware. The performance of the developed solution was measured using Unity Profiler and custom measurement tools. The findings indicate that the development of a mesh-based solution should be heavily based on GPU computational power and minimize data movement. In addition, AR scene management, the LoD system, memory pooling, and work time slicing are factors that affect performance and resource usage. Statistics show that handling and dispatching the world in larger sectors is significantly better for performance. The challenge of cached approach and marching cubes is its memory wastefulness and dealing with moving objects. This can be improved by developing solutions for smart section discarding, mesh optimization pass, or different mesh generation algorithms. From the research, we can conclude that AR technology has cultivated over the decades and has blossomed in recent years, but many areas, such as occlusion, require more research.

Contents

List of Figures

List of Tables

Preface

In recent years, user acceptance for VR and AR has grown, and more products have been released using that technology more efficiently. However, many aspects of AR are still problematic and could barely be researched or prototyped for the public. Occlusion is one of the most challenging problems that we need to solve to receive a highly usable and comfortable augmented reality experience. This can include different methods for representing depth, such as depth maps or meshes, and the generation of 3D models that have been studied for many decades. Early studies were mainly conducted on the CPU. With modern high-performance hardware, we can use graphics cards that are excellent at parallelizable tasks. Furthermore, we are able to use many of the existing tools; from sensors to game engines. In the past, the development of such systems would require a team of experienced developers with years of development. In this thesis, we approach the occlusion by reconstructing the environment as meshes to evaluate if the solution is efficient for real-time execution and what aspects should be considered while developing own occlusion systems for AR. This adds many aspects to consider based on mesh generation, world section management, LoD, processing queues, rendering, point-cloud translation, and other implementations related to performance.

Special thanks to Carsten Griwodz, my supervisor in this thesis, for the help and tons of interesting and inspiring conversations on the topics. I also want to thank all the passionate researchers, students, and developers who contributed in all the relevant areas of the thesis. All the small bricks contribute towards creation of the technologies we can witness in this master's thesis. Last but not least, I also want to thank all my awesome and loyal; friends and family for supporting me throughout the duration of this thesis.

Chapter 1

Introduction

1.1 What is Augmented Reality

As the world of information technology has cultivated rapidly in recent decades, ways to interact with technology and receive information naturally started to target as many human senses as possible. With the goal of immersing users in entirely different realities and environments, virtual reality was developed to mimic realistic sight, head movements, and body movements. Another direction, augmented reality is meant to project virtual-world on the real world to complement it and bring more utility into the real world with minimal delay. The field of AR and virtual reality has opened many possibilities for the market, not only for entertainment purposes but also for fields such as training, simulations, and visualization.

Augmented reality is one of the most promising technologies of the future. Over the last few decades, companies have attempted to implement and sell usable forms of AR and VR. Due to limitations in image and processing quality, users of AR and VR headsets faced severe discomfort [56]. Progress in sensor technology, increasing computing power, power efficiency, and technology to minimize circuit and radiator sizes allow us to make AR technology more immersive and convenient. However, the AR technology concept is full of potential for close-to-real-life experience, even with its many challenges; as it has a wide range of potential usages.

AR technology comes in many different forms. As the name suggests: we augment reality, and different forms of visual overlays can achieve it. In contrast to VR, AR combines reality and the virtual world; therefore, it must capture data from the real environment to add the augmentations. To explain occlusion in augmented reality, we can refer to Shah et al. paper 'Occlusion in Augmented Reality' [58]. As we speak of augmented reality, we can categorize it in three primary forms: video-based, optical-based,

and projector-based. Each of the forms has its own characteristics that affect how occlusion can be implemented and aligned, which I will explain in further part.

Both the hardware and software aspects of augmented reality still have much development ahead to fulfill our vision of a technology of superb usability. Since we are placing the virtual world on top of the real world, we need to solve the problem of synchronization between the virtual and the real world. Modern smartphones have sensors such as the gyroscope, accelerometer, camera, and GPS that allow developers to use them for AR and VR purposes. For AR, physical markers [54] can help us determine the location and placement of virtual objects. With the help of these, we are able to determine direction towards the ground and device rotation, which results in somewhat accurate and stable placement of holograms. However, as we progress towards AR development, we recognize that we need to gather even more information to make our AR more accurate and effective. One of the features that is significant for AR technology is the understanding of the surrounding world and the depth of view. Our human eyes help us to estimate depth and distance well enough for our personal needs. The physical properties of light and real objects result in visuals. In our world with Euclidean rules, objects and surfaces closer to us tend to **occlude** the objects that are behind them. These objects can become partially or entirely invisible to us until we remove all obstacles from our vision.

One of the most challenging parts in AR is rendering holograms occluded, as they would be just like the real objects. An hologram object closer to the user should occlude the object further away, regardless of whether that object is real or virtual. Occlusion is relevant in video- and optical-based augmented reality to significantly increase immersion and reflection.

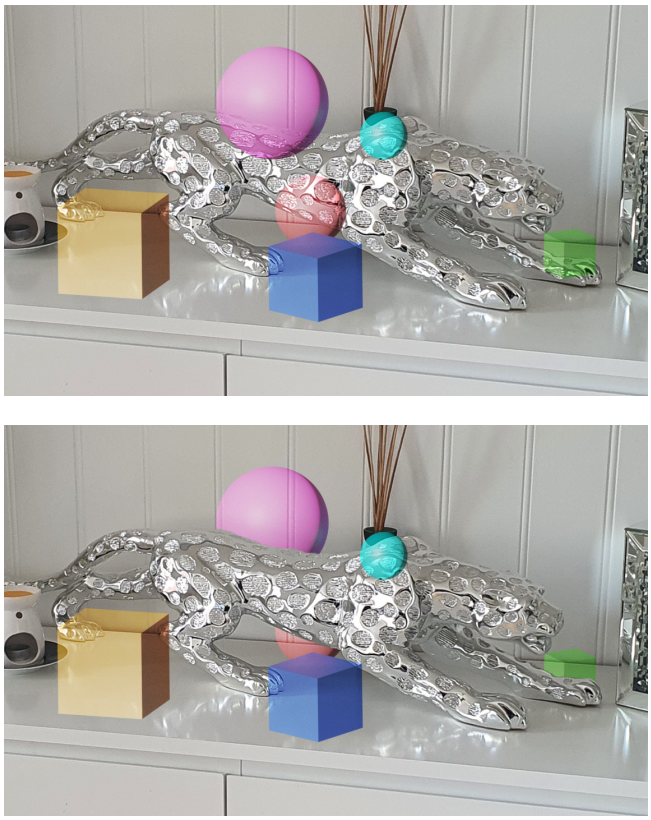

Figure 1.1: Illustration representing simple shape holograms placed around a tiger figure, with (bottom) and without (top) occlusion. Distance estimation, size, and positioning perception is significantly more confusing without occlusion.

Augmented Reality and Occlusion

Let us explore the following case: we are asked to develop training solutions for medicine students in augmented reality. With virtual tools, we are able to connect students and professors, no matter the distance. We can mix the usage of virtual tools and real equipment such as mannequins or real medical tools. If both our tools and our subjects are holograms, students will most probably not face many occlusion issues. If our subjects are real objects but the tools are virtual, we will face occlusion issues: we need to estimate which object should be on top of the other and what parts of holograms should be invisible. Without optical or video sensors, we cannot determine this. The problem can arise further when some of the scanned objects are transparent. Furthermore, we need to take into account the position and perspective of the eyes.

In this case, we face many problems to solve and consider. Some of these can be potentially hard tasks to determine due to the nature of the real world and make these data useful in a virtual environment. However, if

we do not implement occlusion features for medical use, the AR experience will suffer significantly. If we use tools such as syringes or endoscopes and the hidden parts are still visible, the experience will be very confusing for our sense of depth and can cause dizziness and lead to dangerous perception errors. The tools we use need to be efficient and comfortable enough to be used in real-world applications, not making the job harder or leading to hazards.

Since augmented reality combines the real world and the virtual world, providing immersive experiences is a challenging task that requires efficient data collection and processing methods. Computer understanding of the surrounding world is one of the most important challenges for robotics, autonomous cars, and VR/AR to increase their potential.

Even though technology is progressing quickly and computing power has increased significantly in recent decades, in many matters, we are still bound to hardware affordability and human productivity. Developing efficient solutions based on algorithms and sensors may still require high computational power. Connection over long distances can be an inefficient way to solve this, as wireless connection or long-distance computation latency can be visible to users. AI and deep learning could be one of the solutions that can save development time to address the problem of human-like perception of the environment. However, in this case, the efficiency of neural network latency and its computational stability can also be questionable.

Therefore, we need to find out if we are capable of gathering and using environmental data for occlusion purposes. What methods can we use, how efficient it is, what things we need to consider, and what trade-offs all possible solutions can imply.

As a starting point for this thesis, we should consider three types of AR [58]:

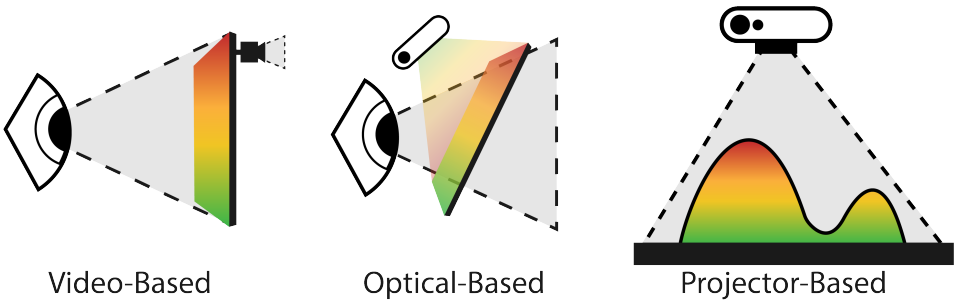

Figure 1.2: Three types of AR are visualized.

The three types of AR are the following:

1. Video-based uses camera and processing unit to add virtual world on top of real world and display it on screen.

Examples: Using camera and sensors of a smartphone to display holograms on the screen. VR headsets with cameras that display images from installed cameras. Pokemon Go or Invizimals games.

2. Optical-based adds a virtual world on top of the natural world by projections on glasses or transparent screens.

   Examples: Headsets that modify our entire vision to display holograms such as Project North Star and Microsoft HoloLens. Glasses that display interfaces on our vision like Google Glass.

3. Projector-based projects virtual world on top of the real world.

   Examples: Aquarium projections on floor, where fish and water interact dynamically with the user trough sensors. Clothing design applications that project an image on top of actual products for demonstration purposes.

To display holograms directly onto our view, we use video-based and optical-based methods, which have the most potential in occlusion. Therefore, we need to evaluate aspects of these AR display approaches:

Video-Based / Passthrough

+ Potentially easier to align the digital world with the virtual world. The user's vision is the same as the one the computer uses.

- Since image is recorded by the cameras, the latency and refresh rate are applied to non-AR elements, too.

- Fidelity is very dependent on the display device, circuit latency, and camera quality.

Optical-Based

+ Greater contact with the real world since holograms are projected only on top of the real view.

- It is harder to align the user view with the hologram view and the cameras/sensors responsible for depth.

- It is harder to achieve real black and transparency levels.

Taking into account the aspects of all three solutions, I concluded **optical-based approach will be main target** for this thesis. **Video-based approach will be secondary target** as both are capable of handling occlusion data in a similar way. The projector-based approach is not a good candidate for occlusion purposes, as the projected virtual reality is heavily dependent on the surface on which images are being displayed rather than being heavily dependent on the user/device viewport.

It is also important to mention two main approaches towards occlusion [58]:

1. **Depth-based approach** is converting the collected data to a depth mask that can later be injected into the Z buffer. The Z buffer is an image rendering step that contains depth data to render objects in the correct order and with correct occlusion. However, this method is highly image dependent. The cost is based on the number of pixels from the cameras that we use and the target rendering resolution. The depth mask on its own gives us a very poor and limited understanding of surfaces, and the data are limited to the current perspective of our cameras. This method tends to use multiple cameras or an IR depth sensor to create a depth mask.

2. **Model-based approach** is converting the collected data to 3D mesh data. This means that model generation is performed, which can be either limited to real-time data or cached for higher accuracy and temporal stability. Therefore, generated 3D models can be used with appropriate shaders to occlude holograms. Since we are generating models and using Vector4 (4-float structure for representing 4 dimensions) data from points of a point cloud, this process can be more costly than the depth-based approach. The point cloud is a collection of 3D point coordinates that represent the scanned real-world surfaces. The cost of the generation process is based on the target complexity of the meshes and the processing steps. Meshes generated in this method can be used for physics and other purposes easily. This method tends to use accurate distance sensors that result in a point cloud; however, it is also possible to use cameras.

Both of these approaches require very accurate synchronization and alignment with movement trackers and well-configured equipment. Any errors can lead to a change in the represented occlusion and user pawn. **In this thesis we will explore an approach towards model-based occlusion**, utilizing the limited resources we have. Note that these approaches can be combined in a hybrid approach. Since the model-based approach is expensive as the desired complexity grows and depth-based approach can be limiting; combining them together to solve a given use case could be considered.

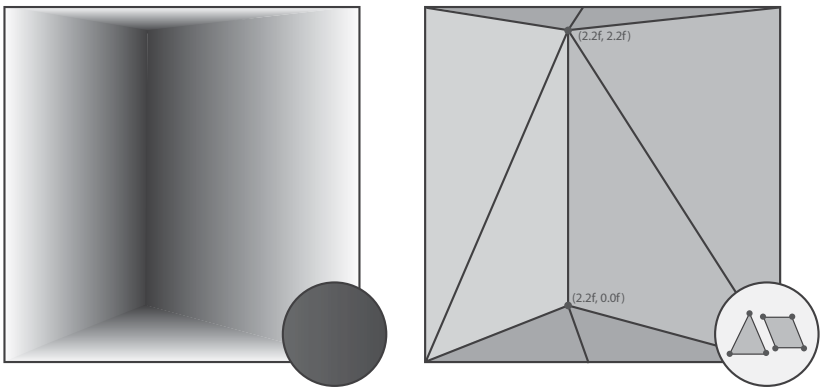

Figure 1.3: Illustration of two depth approaches. The illustration on the left shows the depth mask in pixels. The illustration on the right shows the room made of simple meshes that are used to render the data.

1.2 Research Question

In this thesis, we want to focus on the rendering, occlusion, and performance parts of AR. Model-based occlusion has a lot of potential not only for semi-occlusion purposes, but also for game logic. Our central research question is as follows:

Can we develop efficient real-time, model-based reconstruction method to occlude virtual objects with real object in an AR scenario?

1.3 Methodology

Since the objective of our research is to develop experimental software, there is no perfect definition of how it should be conducted, as it is difficult to choose the right methodology for problems in computer science. Computer science is a very wide range of areas that tend to be connected with all science fields that developed long before electric computers existed. However, the core origin of computer science can be disputed between fields like mathematics or engineering, and thus the usage of methods for computer science research. Knowledge, methods, and terms are hard to standardize because it requires common agreement between scientists and the entire computer industry. Eden et al. examined these philosophical approaches among computer science scientists [19]. He describes three different paradigms of computer science: *the rationalist paradigm, the technocratic paradigm and the scientific paradigm*. All of the mentioned paradigms can be applied to this research case, consider varied nature of AR: which depends on low-end engineering, a potential and more scientific approach that would consider the problem from Human-

Computer-Interaction field perspective.

The rationalist paradigm targets the part of computer science, that is, the branch of mathematics. All of the following problems and algorithms are connected to theoretical aspects, like compatibility or programming languages. Since we are working with concurrency and parallelity to some extent, it could be possible to use Hoare logic and concurrency proof to verify the safety and execution of the solution. However, solving these problems in a purely theoretical way will not directly help us in achieving a performant prototype in restricted amount of time, in a given environment with mathematics.

The scientific paradigm implies computer science as a branch of empirical sciences such as astronomy, economics, and geology. All these fields are chaotic in nature, and even when they may contain patterns, they can be highly unpredictable. An important example of computer science is artificial intelligence and deep learning. Choosing the right models for AI and training is a fundamental part, but even when we try to predict all possible scenarios, we can clearly find unwanted behavior. Object recognition AI tends to misinterpret objects, face-recognition AI with a restricted training set may fail to recognize the vast majority of different human appearances. Thus, even if the programs are the result of mathematics, the different layers of abstraction and interaction can put them on a par with the mental and cognitive processes. Furthermore, since we target GPU hardware with floating-point and chaotic IR real-time point scanning on top of parallelity, the results are highly unpredictable and mostly nondeterministic. Therefore, this paradigm can be somewhat important in this thesis, as we are working on a parallel task in which usability and visual perception depend on user acceptance.

The technocratic paradigm approaches computer science as a branch of engineering, as we design the architecture of systems, maintenance, and evolution. We do indeed use computers, as they are useful devices; they help us with utility. As the word "engineer" is delivered from Latin words ingeniare (’to create’) and ingenium (’clever’), the field of engineering establishes a clever link between scientific discoveries (like algorithms and methodologies in computer science) and practical real-life applications, such as the problems and issues we are trying to solve. As knowledge of software engineering is important for system design, optimization, and structure, it is the most important paradigm in this project.

In conclusion, I have chosen the technocratic paradigm as the main objective of this master thesis, as we wish to achieve tangible results tested in practice. Furthermore, it also implies that this master thesis will focus on designing and experimenting with a model-based occlusion system rather than explaining the mathematics behind the problem.

1.4 Approach

To develop solutions and evaluate the results, I need to specify the goals of this project. The specification of our goals and approach for this project is the following:

Functional Goals

- The solution must generate the mesh data to be used for occlusion.
- Data collection required for reconstruction must be performed live or in advance.

Non-functional Goals

- The solution should run in real-time (a single frame should not exceed 1000 ms)
- The solution should be efficient enough to reconstruct detail up to 1 cm.
- The solution should be efficient enough to run at least 30 frames per second (frame time less than 33.33 ms).
- The solution should minimize performance spikes.
- The solution should be easily adaptable by developers.
- The solution should work on x86 platforms as well as mobile ARM devices that can benefit from AR technology.

Note that these are optimistic assumptions. Completing these goals may have a positive impact on user experience when using the prototype system I developed. Developing a solution that is production-ready is not the goal, as this project serves as an exploration and experimentation attempt in AR-area.

Part I

The project

Chapter 2

Background and Preparations

2.1 Background and Preparations

2.1.1 Early Days of Augmented Reality and Computing

Human dreams of pursuing technology and solutions that bring us utility and entertainment by replacing or expanding our reality began in the 1950s [8]. **In 1955 Morthon Heilig** built and described his device named Sensorama, which was meant to be our "cinema of the future."

In 1966, Ivan Sutherland that we can call "the father of augmented reality", invented a head mounted display [8] that can be considered symbolic for augmented reality. His 1968 invention was called "The Sword of Democles", in which he used mechanical and ultrasonic sensors connected to the head and ceiling of the user to determine the position and rotation of the head [59]. The solution was far from comfortable for users and the sensors were limited in the prototype. His idea was adapted and improved in many years to come, leading to our modern products, such as Project North Star or Microsoft Hololens, building on the same idea using more compact technology and also proving more usability.

Back in the day, companies hired humans to perform mathematical calculations before digital computers were widely available to the public. Computers started to replace human workers in the 1960s and 1970s, as they started to simplify and automate many of the everyday tasks. Some of NASA's 'human computers' became programmers and testers using programming languages like Fortran [60].

Wellner paper from 1993 approaches the gap between the real and virtual world [80]. Digital tools provide us with a lot of refinement and help, as we can easily spell check, search, copy/remove/move, and read with text-to-speech technology. However, we are using paper with our extensive senses,

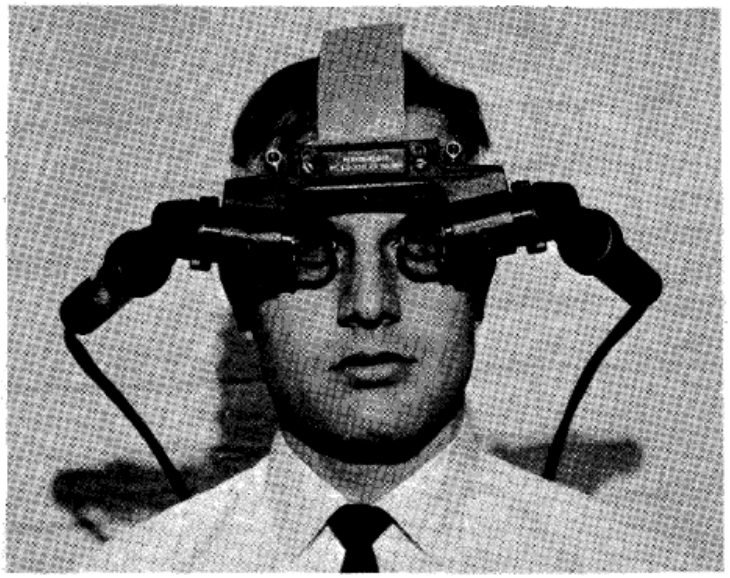

Figure 2.1: Sutherland's prototype shown in his 1968 paper. [59]

tactile skills, and a certain amount of physical material. Furthermore, in the modern world, we tend to move towards digitalization, with some possible trade-offs. This led to a projector-based solution in which users wrote on the surface of the table with pens and interacted with the fingers. It allowed the physical objects to mix with the paper and tools displayed by the projector. In the coming years, we could see the development of products such as interactive class whiteboards. Considering the complexity of such task; this field of AR can be considered as one of the least demanding ones in most common use cases. Having a virtual world through the projector can be a less demanding task compared to mesh generation, complex depth perception, and movement detection. However, the computing power mentioned above in these years would be sufficient even if the user experience potentially had much room for improvement compared to AR solutions of our times. Clearly, many ideas for new technologies have appeared over the years. Some of the ideas seemed good on paper as they provided solutions to our problems. In the end, even if products were released, they would have a tendency to be used and rejected because of essential issues with features and usage. It was either because of design flaws (ineffective in use, for instance) or the restrictions like computing power (what is possible to do and what we can afford to do) or because of the technological restrictions.

Consumer markets have been approached by VR and AR many times. **In 1995, Nintendo attempted to release a VR console named Virtual Boy** [14, 82]. The console was the first VR, stereoscopic vision console that was ever created. The console was released in Japan and North America, but it was never released in Europe or Australia, since Nintendo officially halted its production in 1996 due to poor sales. To this day, **Virtual Boy is the worst**

selling Nintendo console in the history. The console was equipped with a 20 MHz processor, 1 MB Dynamic Random Access Memory, and 1 KB cache. It was equipped with a 384x224 resolution scanner (not a screen) and a 50 Hz horizontal refresh rate. Furthermore, the console displays monochromatic images only, with 32 levels of red pixel intensity. Choices were made based on technological limits and costs. At that time, red LEDs were significantly cheaper to produce compared to green or blue InGaN LEDs. Red is on the lower end of the visible spectrum and is closer to the infrared. This means that producing red light requires more power than color, such as blue or violet, with higher wavelengths. The lower energy usage of the screen is beneficial for an affordable device powered by six AA batteries (or an AC adapter). It is also important to note that the device did not have advanced tracking hardware. In modern devices that use VR, AR, and even smartphones, gyroscopes and sensors track the movement, rotation, and speed of the devices. These data are then used to rotate view based on the devices/eyes position, making the experience much more immersive. Virtual Boy's view was static and game-dependent, and user movements did not affect the view. Not to forget that the games provided were 2D just like other Nintendo consoles at these times, trying to make it more interesting with approaching 3D view through stereoscopic vision.

A brave and innovative idea became an extremely poor product. Poor screen and technology cause headaches and nausea for users. Taking everything into account, the specifications of the console were meant for 2D games and were computationally much slower than the PS1 3D console released in 1994/1995 worldwide [13]. It is also very important to consider that immersive stereoscopic visions require two views, which almost doubles the computing requirements for rendering when compared to a single-screen display. In conclusion, it was very hard to make VR and AR tech usable in these times, as the technology we had was not enough to close the gap between the real world and the virtual world to cheat our human senses. Even if we had sufficient technology to close this gap and improve the VR experience to reduce the negative effect, it could hardly be affordable to customers. In addition, we can consider whether that unique experience was worth our health, time, and money. Sales of virtual Boy products can suggest the answer.

One of the later examples starts with the fundamentals that make AR usable as a technology, described by Drascic et al. [18]. The paper describes issues, a problem of AR depth perception that makes this technology actually usable. This paper is very relevant for both video-based and optical-based AR approaches, as both tend to deal with the issue of depth sensing. As expected, technology was developing in different directions. The article described the classification between the real world and the virtual world as follows; reality, augmented reality (AR), augmented virtuality (AV), and virtual reality (VR). The mentioned families of reality, mixed reality (MR), and VR combine different methods for display; direct view, stereo video, and stereo graphics. Drascic et al. studied the literature

and found that some researchers displayed virtual reality on top of the real one with a monitor, some with headsets with mirrors (like in Project North Star), while others used cameras to combine virtual world with real world through cameras. This shows that approaches to AR, AV, or VR implementations have developed broadly from the early beginnings. Video-based methods can actually allow us to use the headset both as AR and VR if developed correctly, whereas optical-based methods are not suited for displaying VR because of transparency of the screen. In later chapters of this thesis, I also describe the differences between such video-based and optical-based methods and what they mean for occlusion.

For correct depth perception, we need to use a calibrated stereo image, as depth is hard to perceive in stereoscopic projections, leading to misperception and errors that interfere with usability of an AR system. One of the interesting issues that appears in experiments is related to the mechanisms of human eyes. As we focus on real objects, close or distant, our eyes must adjust to the depth of the object we look at. Using optical-based systems, even if virtual objects are very far away, our eyes need to adjust to the location of our display surface located in front of our eyes. With the video-based approach, our eyes are adjusted to depth of display surface, which means that the issue is solved, but the focusing instinct in our eyes is entirely restricted.

More recent studies of AR have been extensively tested in the field of medicine [55]. The researchers wanted to use screens or headsets as an additional virtual overlay on top of the real world for tasks such as X-rays, scans, patient operations, or training. In Rolland et al. [56] compared video-based and optical-based approaches in the field of medicine. **The most fundamental issues they mention is in fact occlusion, fidelity of the real-world view, system latency and user acceptance.** Many of these factors could be improved with modern technology with significantly better tracking, synchronization, refresh rate, and computing power. There is no perfect solution in this case, possibly regardless of the field in which the technology is used. Optical-based (see-trough) is harder to synchronize, but world interaction is more direct; meanwhile, video-based approach results in easier unification and handling of the views, possibly for the loss in field of view, latency, and refresh rate of screen. As computing power and hardware accessibility increased, AR gradually gained traction. One of the other experiments we can mention is dynamic information exchange for fast decision making regarding airplane control, displayed on the screen for purposes such as easier identification of airplanes in poor weather conditions [55].

Technology and Computations

Although AR research began many decades ago, technological limitations and the complex nature of human perception were restricting its usability and development. The features granted by AR as we know today are

heavily graphical-based. To better illustrate this with market capabilities and affordability, consider the following example. One of the most significant achievements of product releases in the 1990s was the Sony PlayStation, which brought 3D and 2D graphics to the consumer market at relatively affordable prices and with ease of use. Simplicity of use and the price of consoles increased user accessibility to video games. According to the PS1 technical specification [13], the board was equipped with a 32-bit CPU running at a frequency of 33.87 MHz and with a 4KB instruction cache. The console was also equipped with 1 MB of VRAM (Video RAM) and 2 MB of RAM memory and **Extended Data Out** chips that provided lower latency than DRAM. Interestingly enough, this console was not equipped with **Floating Point Unit**, where in the modern world floating points are an essential part of graphic calculations. The lack of FPU leads to world and vertex coordinates being expressed as integers, total numbers. This fact led to the characteristic vertex snapping to the closest integer coordinate, instead of smooth vertex movements, as we know in modern float computer graphics. Animated models waved and changed shape unnaturally. This decision could be considered as poor, but affected prices and thus affected affordability that in any case led to the consoles big success. PS1 console was able to render resolutions between 256x224 and 640x480 (max colors) and calculated 360,000 polygons per second[12]. When we compare this computing power and memory capability with the potential needs that AR poses for rendering and occlusion,

A high-end computer setup at these times, equipped with a CPU and potentially a hardware accelerator (3dfx / Voodoo Graphics), was a less accessible option, not necessarily available to a wider public or getting close to the technological needs of AR. In the 1980s, consumer processors were suited to run at a clock frequency of around 1 MHz [28]. Modern CPUs are equipped with frequencies ranging between 3 and 5 GHz and ranging between 2 and 12 physical cores for laptops and workstation computers for home. In addition to that, both Intel and AMD work on optimization methods and mechanics that work either by default or can be implemented by developers for even faster computation. The technology like Intel Hyper-Threading or Intel Math Kernel Library that commit towards easier and more efficient parallel/concurrent computing. It clearly demonstrates that computing capabilities increased thousands of times between the 1990s and 2020s as architecture, hardware (by making smaller nanometer circuits), and software improved and are still improving. An increase in computational power for both the GPU and the CPU is necessary to close the gap between virtual-world accuracy and real-world accuracy.

2.1.2 Augmented Reality Today

Even if the technology had been developing for decades, the mentioned issues were leading to poor user acceptance that affected not only AR,

but the entire VR and MR field in general. Although the first prototype of the AR headset, The Sword of Democles, was researched in 1968 the VR and AR technology was insignificant on the consumer marked before 2012 when the Oculus project was successfully funded and other notable VR/AR products were developed; HTC Vive, Steam Index, Project Northstar, and HoloLens. This has begun a new era for the fields related to virtual reality that reach consumers and market users [1]. The VR and AR possibilities on the marked increased, as user interest and experience has grown following the improving quality of image, sensors, and computing power.

Today, AR and VR technology has finally gained significant traction; but few complex issues related to it remain unsolved. Many issues are close to impossible to solve, others require more computing power or technical compromises. We can conclude that making a real-time AR that is comfortable for users, especially occlusion, was impossible for many decades of IT. Even if the early AR-related project could start as early as in the 1960s, technology and computing power were far enough for our target needs; close to seamless experience that users would be comfortable with.

2.2 Choosing the Right Tools

To conduct the research, I needed to choose development tools. To simulate an AR environment, a 3D rendering API is required. The low-level part of the application needs to take care of tasks such as rendering, collision, and program logic. Such an application is called a game engine. The development of a game engine from scratch is very difficult, as it preferably requires a team of experienced developers to finish it in months or years. Game engines are programs that require real-time computation, as many of the calculations need to be done on a frame basis. A frame is a single image displayed on the screen as a result of rendering.

Virtual worlds are best experienced at framerates reasonable to the human eye. The more efficient data management, dispatching, and task parallelization, the more detail or FPS game can achieve. Game logic is often bound to the amount of FPS an engine is capable of producing. Another part of the logic like physics calculations may restrict its update to precisely N per real-time second, which potentially makes mechanics like slow motion easier to achieve.

Some of the data required for the logic of the game, netcode, and rendering can be prepared prior to execution to maximize performance. However, the performance-demanding nature of game engines motivates the use of high-performance languages or different optimization techniques. However, its use cases in the industry go far beyond the creation of games. Game

engines are also used to create 3D animations, simulations, training programs, and cinematography. One of the most recognized game engines on the market for 2022 are [4, 9]

- Unreal Engine
- Unity
- CryEngine
- GameMaker: Studio
- Godot
- Lumberyard
- Frostbite

Each of them has different financial models, use cases, tools, and community.

While working on an AR project with limited time and resources, I have decided to go for the engines with the most software/hardware support and a large community. The lack of support and developer tools can make the development time longer, potentially affecting project productivity. Developing VR and AR for Unity and Unreal Engine is a safe and beneficial option, and thus they were the main options to be considered while working on this thesis.

Unreal Engine

Unreal Engine is possibly one of the most popular game engines [9]. Unreal Engine was used in the creation of many popular and historic game titles, such as Unreal Tournament, Bioshock, and Mass Effect. It was also used in the production of 'The Mandalorian', making a gigantic breakthrough in cinematography by replacing green-screen technology with realistic environment and reflections directly visible during recording and rehearsals. The native language for Unreal development is the high-performance, well-established ahead-of-time language C++. It is also possible to use graph-based visual scripting methods for script programming. As a well-supported language, it supports most popular platforms and technologies such as AR and VR. It is a great pick for big projects focusing on photo-realism and performance.

Unity Engine

Unity Engine is one of the favorites in the indie game development scene, as it is one of the easiest game engines to learn [4]. Unity offers a

wide range of tools like camera-systems or pararellization systems. Some solutions like networking are premature when compared to Unreal or unofficial commercial solutions. However, the Unity developer community is significant, and libraries are often well documented and come with many examples thanks to the large community. Unity uses C# language comparable to Java syntax and C++ utilities. C# can be considered as one of the most popular and performant just-in-time languages with many optimization options, syntactic sugar and helpful tools, while being time-efficient. Even if the code is being compiled at run-time, Unity is not significantly worse performance-wise when compared to Unreal Engine. The fact that Unity is performant enough, time-efficient, supports cross-platform compute shaders, and is supported by most of the hardware that could be potentially used for testing makes it a very good candidate and the final pick for this project.

The current Unity versions implement different rendering pipelines with different goals [71]. Rendering pipelines are sequences of steps that the engine and graphics APIs need to take to render an entire frame; like drawing geometry shaders or calculating light and shadows. Over the years, Unity has used a rendering pipeline called the Standard Rendering Pipeline (SRP), but a single rendering solution did not meet the specific needs of all supported platforms. This led to the development of the Lightweight Rendering Pipeline (LWRP), which was later renamed the Universal Rendering Pipeline (URP). The goal of URP is to provide medium- or low-fidelity graphics that will run efficiently on both mobile phones, consoles, and computers. However, URP suffers from diminishing returns on performance gains when many lights are used in the default forward-rendering mode.

The contrast to this lightweight rendering pipeline is the High-Definition Rendering Pipeline (HDRP), which can compete with Unreal Engine in terms of high-fidelity graphics and performance. HDRP is much heavier to compute because of the different passes and buffers required to support the rendering of high-definition graphics or features like ray tracing. The impact in frames-per-second (FPS) when moving from URP to HDRP on high-performance devices is commonly significant. However, HDRP is much better suited for high amounts of lights and effects and thus will suffer less from diminishing returns as more lights and objects are added.

URP and HDRP are long-sighted solutions for Unity Engine in constant development. They are meant to be a replacement for SRP that is in maintenance mode. If none of the options mentioned is satisfactory as a fundamental for a given project, Unity also supports the implementation of a custom rendering pipeline through the Scriptable Rendering Pipeline API [72]. As we focus on VR/AR field, URP is the most optimal pick. Our goal is to render and occlude holograms with a good framerate. This means that features like photorealistic rendering and high-fidelity shaders are not needed. URP also makes the solution compatible with a larger number of

platforms, which is beneficial for AR use cases. This can include: headset wired to a computer, standalone headsets, or smartphones.

Programming Languages and Runtime

One of the aspects that has a direct impact on runtime performance and memory usage is programming language and how its environment compiles the code to the native machine code. In Pereira et al. 2017 [51] paper researchers measured the performance and use of resources of different programming languages used in the marked. Although the implementation methods might not be perfect for each language, the research paper shows data related to energy, time, and memory used while using various languages.

Language	Energy	Time	Memory
C	1.0	1.0	1.17
Rust	1.03	1.04	1.54
C++	1.34	1.56	1.34
Java	1.98	1.89	6.01
Pascal	2.14	3.02	1.0
C#	3.14	3.14	2.85
Go	3.23	2.83	1.05
Python	75.88	71.90	2.80

Table 2.1: The table shows the usage of resources from different programming languages. Value 1.0 is the reference metric for the most performant language in a given category. [51]

The result showed that C, Rust and C++ are on top of energy and time efficiency, while Pascal, Go, and C were the most memory efficient in conducted tests. Considering how C++ functions as an AoT (ahead of time) language, it is highly probable that Unreal will be more efficient than Unity and will promote better memory management. However, it is clear that C++ can be more time consuming to program, as there are more high-level languages.

The Pereira et al. places C# above average in the time list. Since this paper does not directly mention the C# runtime used, we assume that the less performant Mono (one of C# interpreter environments) runtime is also used by Unity Engine. It suggests that Unity programs can be slower when using Mono runtime than Unreal C++ code. C# is also slower than Java and significantly faster than Python. The results could potentially be improved by using a faster C# runtime than Mono; however, other runtimes are not supported by Unity Engine. Note that C# allows unsafe memory

operations in blocks **unsafe**, allowing memory operations comparable to C++. To deal with platform compatibility, Unity translates the code into C++ via IL2CPP, which will be mentioned in the next section.

At the moment of writing, Python is one of the most popular languages on the market, as many years before. However, none of the most popular game engines uses Python for their core systems. This state of facts can be caused by Python runtime performance, as CPU and GPU computing performance can directly result in a better relation of resources used to FPS and resolution produced.

Abstraction of Graphic APIs

Since both engines can be deployed to a large number of devices and architectures, they introduce a significant number of abstraction levels. Developer freedom is also an important factor when choosing an engine, as developers should be able to achieve all desired mechanics efficiently. AR technology can be used for mobile devices, AR/VR headsets, or other methods. While the Unreal Engine C++ code can be directly compiled into machine code, C# is a more complicated matter. The C# JIT (just in time) environment known as Mono is not available on other platforms than Windows, macOS, and Linux [73]. This problem required an alternative approach, which led to the development of IL2CPP (Intermediate Language to C++). IL2CPP translates the code into C++, which is compiled on any of the target platforms [68]. Since the code is optimized in the process and the code is compiled ahead of time, it is possible to notice performance improvements. Furthermore, it is important to note that each platform uses different graphic APIs. Graphics engines give a programmable interface to render 2D and 3D applications and give developers useful tools that can significantly affect rendering times, computing time, graphical fidelity, and/or shorten development times.

Some of the most known rendering APIs are Microsoft DirectX, Vulkan, Apple Metal, and OpenGL. DirectX is used as the main graphics API for Windows x86 operating systems and is not available on other platforms outside of Microsoft's ecosystem. On the other end of the spectrum, we have Metal utilized by macOS and iOS, and can be either an ARM (commonly used in mobile devices) or x86 system (commonly used for desktop and mobile computers). One of the most universal and modern solutions is Vulkan, which supports all major x86 and ARM operating systems. To achieve compatibility with these platforms, game engines such as Unity implement a layer of abstraction invisible to developers [67]. Then it translates features such as shaders, compute shaders, and draw commands to the target API without the developer's action required. It also needs to take all differences into account: half-precision floating points (called 'half'). It is a good option for applications such as mobile computing and deep learning [26] where it can result in a minor decrease in accuracy,

but saves noticeable amounts of memory and computing power. Single-precision (called 'float') and double-precision (called 'double') floating points can be more common on x86 platforms, where heat and size of components can be a smaller problem, and higher performance can be achieved. The compiler makes such usage of the 'half' type obsolete on some APIs/platforms (like DirectX), as half is converted to float. While it is challenging to determine all the possible details about algorithms behind these abstractions, some are possible to grasp through documentation or common sense.

Computation Shaders

Since mesh generation pipelines are resource-demanding operations, it is required to use as many available resources as we can. This is highly restricted to the hardware we own. However, most computers for use, such as games, simulations, or VR/AR, have multiple physical cores and a GPU with different types of computing cores in quantities of hundreds or thousands [28, 85]. Therefore, parallelism comes as an important element. Since it may be wise to use algorithms optimized for parallel computation (like marching cubes), a GPU is capable of much faster computations than CPU and RAM. Since GPUs are meant for high-complexity floating point computations common for use like display, computer graphics, games, physics/chemical/biology simulations, they are commonly equipped with very high efficiency VRAMs and thousands of shading cores meant for different purposes. The single shading core alone is significantly slower than the single CPU core; however, its high quantities make the difference.

As we want to send and process information on the GPU, we must choose a method to interact with the graphics card. First, we initially wanted to use CUDA or general-purpose computing APIs ported to C# from C++. This method would imply very good access to the CUDA materials. However, CUDA is restricted to Nvidia video cards, and using AMD/Nvidia compatible options may lead to loss of CUDA-related features. Another drawback is that these libraries may lead to issues when testing on all three popular x86 platforms like Windows, Mac OS X and Linux. In that case, performing reconstruction operations on phones or consoles might be complicated or even impossible, depending on the platform.

Due to the low platform and hardware compatibility, it was important to evaluate what tools and common graphics APIs the Unity libraries have to offer. Unity supports compute shaders [63] and translates them into the desired graphics API supported by the target platform. Compute shaders are used to calculate arbitrary data on the GPU, which are not necessarily related to graphics [46]. It will also improve the integration of the code into the Unity ecosystem. Furthermore, it helps us to reduce the amount of compatibility layers that would be needed to implement to support an

ever-changing list of platforms, APIs, and features. Still, it is very probable that such a solution may be less efficient than native code, as the generality of Unity abstraction layers might be slightly less efficient than the native implementations for each specific platform.

2.3 Design Development

2.3.1 Introduction

As we design and implement our system, we need to cover and evaluate various aspects of software and hardware. These aspects introduce sensor hardware, sensor data handling, mesh generation algorithms, data handling methods, and optimization techniques that could be applied.

This section is dedicated towards explanation and evaluation of the thesis project by developing real-time mesh generation systems.

2.3.2 Capturing the Reality

Before we can render the digital world on top of the real world, the fundamental requirement is the hardware used to collect data from the surrounding world. In the virtual world, we can quickly determine the camera's distance from an object in our view. Before we can occlude different objects based on factors such as geometry, distance, and order, we must collect data that are primarily suitable for the needs of the AR headset. We can use laser probing, IR probing, or a camera to estimate the depth of real world scenery and reconstruct it as a mesh or depth mask. The two approaches will be discussed in Section 3. This section will provide information on the weaknesses and strengths of different methods needed to collect various forms of data necessary to perform occlusion.

Environment reconstruction is essential in other fields of information technology, such as robotics. Boston Dynamics is known for cutting-edge research and development in the area of autonomous robots. Boston's robot "Spot" released in 2020 is an excellent example, as robotics also requires a perception of their surroundings. Spot can be used for a wide range of purposes, especially those that could put humans at high risk. Since the robot must operate under harsh conditions, it had to be supported by reliable technology that would allow constant operation. To solve this problem, the robot needs to perform a 3D mapping of the surroundings in real-time and a processing unit capable of performing all these tasks. The robot has been equipped with multiple sets of cameras, depth cameras, and IR sensors, especially at the front [6].

One crucial technology to consider for AR is the use of laser range probing, known as LiDAR. Light detection and probing are technologies that find good use in devices like 3D scanners, robotics, or autonomous cars (for recreation of the environment), at least to some extent. The data we receive are known as point clouds. Although this method can be very accurate in measurements, LiDAR has some challenges. We can inspect the presence of LiDAR for environmental reconstruction in the context of self-driving cars. Although LiDAR appeared to be used in autonomous car prototypes, companies such as Tesla opted out of this technology [3]. Light probing does not appear as a part of Spot's environment awareness either, but can be installed as an additional module. As for the augmented reality itself, Microsoft does not use LiDAR in HoloLens either. The accuracy of this technology proves to be very useful for environment mesh reconstruction of much higher quality. More accurate and higher-quality mesh generation can improve the fidelity of our occlusion. However, to do this, we need to convert our point cloud into meshes or depth data that our rendering engine can use.

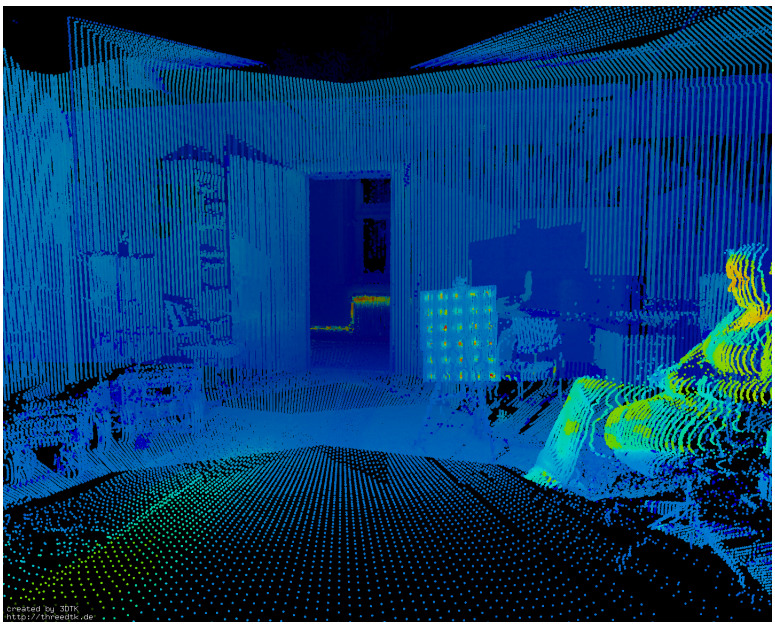

Figure 2.2: Illustration of the point cloud scan. To use point cloud for occlusion, it needs to be covered in a format suitable for 3D rendering engine: a mesh or depth map. [17]

1. The first point is the high price and maintenance costs of the sensors. A LiDAR setup can cost hundreds or thousands of dollars. Although this method might be affordable for prototyping, it can greatly increase production costs for the end-users. Thus, we need to ask the following; is it worth it? The answer is; not necessarily in all cases. Some research papers [35, 41] could suggest that it is possible to achieve a similar reconstruction quality with stereo cameras or depth cameras. Optical systems of two or more cameras are cheap, universal, and scalable. RGB-D cameras can also be a good option. We can balance camera quality to achieve the desired accuracy based on the processing power we can supply for reconstruction. It means that we can use Full HD cameras, or improve them to 4K, or even combine multiple views. In theory, such a method could allow developers to adjust the quality of the images. Limiting scan data can be challenging, but important, when improving performance. Point clouds appear to suffer from the effect of diminishing returns [84]. A higher probe rate only slightly improves the accuracy of the reconstruction, but the computational cost scales with the size of the data set. It is wise to reduce the probes used for model generation, as the precision granted by additional points may be insignificant to the computational cost [84]. This process may require an additional step in the processing of the point cloud, where the data are discarded. The computational cost depends on the type of mesh generation algorithm. Algorithms such as Marching Cubes translate the point data into its own weight system.

2. LiDAR and other optical methods are limited to distance measurements only. Cameras can be used for more AR features than this. We can use trackers or generate textures that would typically require dedicated cameras next to LiDAR.

3. In some cases, LiDAR can be ineffective in tracking and distinguishing moving objects. Furthermore, there is little literature on the subject of detecting moving objects from LiDAR scans [53]. This point is essential for AR and the mentioned autonomous cars. Our AR device (headset or phone) is in constant motion. That moment will make the LiDAR system very noisy and inefficient for occlusion. However, it is essential to note that LiDAR will be very accurate and effective in reconstructing the static environment. Correctly used LiDAR or IR can still result in outstanding accuracy, which is difficult to overcome by other methods. It can still shine in scenarios where the recognition of a camera environment would be inaccurate, such as Tesla incidents with white vehicles [81].

The mesh needs to be reconstructed by an algorithm that can translate the clouds into mesh or depth information. This operation can be expensive, inefficient, and subject to GPU optimization as it implies thousands or millions of points to compute.

IR and Cameras

Depth cameras would be a very convenient method to receive information. Depth information would be enough to allow efficient yet straightforward occlusion of objects. We can either use multiple cameras to generate a depth map or use the more accurate method by using depth cameras to do such a reconstruction.

Depth cameras emit infrared pulses into their environment. Then, infrared rays return to the camera, whereas other frequencies are filtered out. Newer IR camera technologies (such as Kinect v2) use Time-of-Flight technology that is slightly more accurate compared to pattern projection, recommended for mesh generation purposes [79]. A higher sending frequency results in a higher scan rate but in a shorter range. The lower sending frequency results in a scan frame rate, but in a higher range [43]. An AR headset must have a perception of the environment and follow user interaction. HoloLens, based on upgraded Kinect technology, uses one depth camera and two infrared illuminators to address this problem. Short-throw (high infrared frequency, pointed down) offers high accuracy and frequency tracking of users' hands and close objects. Long-throw (low-frequency) is used to map the environment. It also has multiple RGB cameras used for tracking. However, it is clear that we can receive IR interference from the elements of the environment, such as the Sun. IR cameras filter out other wavelengths, but usage of multiple headsets in the same area can cause interference. The paper by Li et al. [32] proposed solutions to reduce interference by using different IR wavelengths without hardware updates.

Spatial mapping (3D reconstruction) of our environment can be challenging. One of the main drawbacks of RGB-D cameras is their noise. As with other methods, 3D model generation can suffer from jitter and holes. Researchers at Keio University went several steps further and created an algorithm for mesh generation using data from multiple RGB-D cameras [35]. Their algorithm utilized the marching cube algorithm processed on GPU for high-quality mesh reconstruction of the environment. Nevertheless, some concerns arise when we need to use such model data for occlusion, as jitter and holes affect the quality negatively. This requires methods or whole reconstruction algorithms that support potential water proofing (closed meshes, without holes in topology). However, it can sufficiently decrease the surface quality, and thus the quality of our contours is crucial for occlusion. The inaccuracy of the IR can be even more visible to moving objects, such as the hands and fingers of the user, leading to errors in occlusion, as presented in the paper by Gugliermo et al. [25] that might need to be addressed.

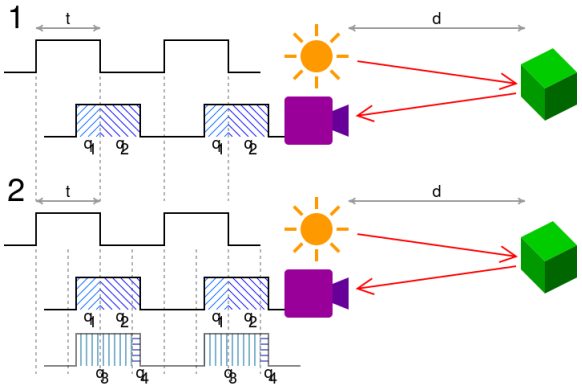

Figure 2.3: Illustration representing the working principles of time-of-flight cameras. ToF cameras are based on infrared pulses (from illuminators) of a given frequency, which are then picked up by the camera with a time delay. This information is used to estimate the distance. [10]

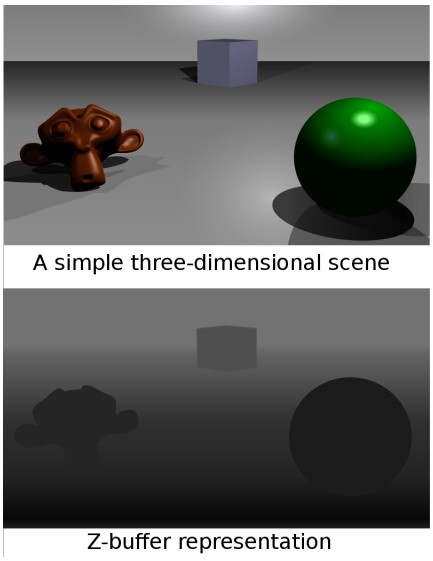

Figure 2.4: Illustration of the Z-buffer representation. [83]

Algorithms

As we choose the data collection method, we need to consider them with algorithms that will efficiently convert them to destination form and the functionality developers want on the side. As we map the surrounding world around the dataset, we need to add it to our AR application.

One of the relatively computationally cheap ways is to convert the data to an occlusion mask added to the Z buffer of the renderer. Depth map calculation algorithms tend to be somewhat simple and depend on the camera and target resolution.

An even more computationally demanding way to perform the reconstruction and gain more functionality is a model-based reconstruction of the environment. Mesh reconstruction implies a relatively complex calculation in a 3D space, where actions like forming new polygons need to be considered with neighboring polygons. This can require a search or lookup. When we can reconstruct our surroundings as a mesh, we may make immersive occlusion while allowing developers and programs to use our reconstruction for other means. In this way, we can more easily achieve the alignment of virtual objects with natural objects and many other features.

Along with the data collection method, we need to choose methods and algorithms to reconstruct our environment. The method based on depth information calculated with the cameras is known as the depth-based method for AR occlusion [58]. When using cameras only, algorithms need to estimate the depth of visible scenes to create a depth map. These cameras need to be at constant distance from each other, and their perspective should be according to our AR headset display. It is possible to achieve an acceptable effective depth map with just a single camera, using less common methods, such as depth estimation based on color shift model-based depth estimation [27] or camera movement-based methods [41]. Unfortunately, these methods may not be practical enough for large-scale AR or may require special hardware. Color-shift depth estimation requires a specialized multiple color-filtered apertures (MCA) camera and fails when different objects are of similar color. We could also try to implement more complex algorithms and approaches, such as the proposed 3D reconstruction of the occlusion boundary [31]. By creating a 3D reconstruction from outlined silhouettes, the researchers achieved a refined and accurate 2D estimation of the occluding objects used for rendering using two cameras.

Concluding: An array of at least two cameras (or a single depth camera) is an easier and more reliable method with fewer disadvantages that can be problematic for AR applications. When the cameras are in a rigid setup and are well configured, computing between two or more perspectives can be significantly easier. The depth mapping process is suitable for graphic card processing. Modern GPU libraries like OpenCV allow one to do it easily by computing stereo correspondence. The efficiency of other algorithms must be evaluated, as algorithms that are computationally more complex and demanding do not necessarily result in slightly better results usable for occlusion.

When the depth map is ready, it needs to blend into the Z buffer of our rendering engine. The Z-buffer contains the depth information of our scene, which is essential to display 3D polygons in the correct order and display a correctly occluded virtual world. Thus far, this method seems to be more lightweight and more straightforward than the model-based occlusion approach that we use in our thesis.

The second method implies 3D reconstruction and mapping of the real world. The scans of the natural world have to be converted into meshes that co-exist with the virtual objects. In this way, we do not need to deal with methods that require modifying the depth buffer, as the processing happens in its natural flow as the virtual cameras capture the surrounding world. This method can be much more challenging than the buffer-based one, as it implies many operations in a 3D space. On the other hand, it can be much more convenient when working with LiDAR or infrared cameras. Furthermore, it is even possible to achieve satisfactory reconstruction quality with a single monocular camera [41], which is already done by some mobile AR applications. The most significant benefit of model-based reconstruction is the functionality and possibilities that are inefficient to achieve with depth-based occlusion.

Using model-based occlusion will pose challenges to the surface quality of objects. This issue was presented in some occlusion papers [35] and Microsoft's presentation on HoloLens [43]. When we conduct surface reconstruction by methods like structured light in the IR domain, it can suffer from inaccuracies, causing jitter on the model's surface. Furthermore, missing or corrupted interpretation of the environment can also be a problem, and our algorithm needs to be efficient enough to do this reconstruction in real-time. Alex Kipman at the HoloLens presentation showed 3D mesh reconstruction as a virtual curtain over the real world, with no sense of what the objects are. The use of artificial intelligence was considered to build a better understanding of the surrounding environment, which can be useful in solving some AR challenges, including occlusion.

The following problems must be addressed with reconstruction algorithms, assuming that we do not use AI. As the program reconstructs the physical world in the form of a mesh, it needs to be done using a combination of several algorithms. As is typical in software development, it can be difficult to achieve efficient and effective algorithms simultaneously. One of the most popular reconstruction methods, valued for its flexibility and simplicity, is the marching cube algorithm. Marching cubes allow us to do the rebuilding, applying post-processing steps and corrections with the possibility for parallel and fast computing on the GPU.

As we are conduction reconstruction, deviations and missing information can cause holes in the model. Although this issue not only causes problems with occlusion, it can also affect the functionality in the application. Developers may want to use the mesh for applications, and the model can be used as a collision in Unity or Unreal Engine for mechanics, such as physics. To address the holes in our reconstruction, we may need a hole filling process (called waterproofing) [35, 84]. Hole-filling can be added as an additional step, done with additional post-processing on the 3D mesh; however, it can be a very expensive operation to execute. The process of waterproofing is directly related to how meshes are generated (more about

this in the chapter, 'Mesh Generation Algorithms'). Marching cubes do not "wrap" the surface meshes on top of the points, unlike algorithms like Poisson. However, the model itself is not without potential holes in the mesh.

The proposed Poisson algorithm presented in the research paper by Meerits et al. [35] can be efficient in dealing with both holes and jittery surfaces. Still, as suggested by the researchers, it tends to oversmooth the surface. The settings of such an algorithm should be evaluated and calibrated correctly, as the loss of detail caused by Poisson may be unacceptable when the mesh is used mainly for occlusion purposes. However, it proves to be a good option to consider for 3D reconstruction. Mesh generation algorithms will be explained further in section 3.1.2;

Investigating the articles and technology available today, we can observe that depth cameras or camera setups are among the most preferred methods for occlusion in augmented reality. Therefore, the data collected by the sensors can be calculated on a graphics card for better performance, as algorithms like marching cubes can easily benefit from the task being split between hundreds or thousands of shader cores. Good AR occlusion will require effective tracking of head movements, head location, effective world reconstruction, and eye-to-screen distance calibration. Depending on our data collection method, we will need to perform additional data processing steps, such as stabilizing the jitter (IR), data discarding or waterproofing, to ensure efficient processing and high quality of occlusion.

2.3.3 Tracking and Alignment

At all times, the position and rotation of the player must be very precisely aligned with the real world. This is a difficult task, as misalignment and lag will lead to symptoms such as dizziness and nussea [18].

Tracking methods, we can divide into the following groups, based on Rabbi et al. [54]:

1. **Sensor-based tracking:**
    (a) **Optical tracking:** Accurate tracking of controlled environments. Sensitive to noise and occlusion. Commonly used in VR/AR headsets like Hololens, HTC Vive Pro or Valve Index.
    (b) **Magnetic tracking:** Cheaper, less accurate than the optical method. Sensitive to magnetic disturbances.
    (c) **Acoustic tracking:** Slower at tracking, speed of sound waves. Affected by environment temperature and humidity.
    (d) **Inertial tracking:** Does not require many external references, making it usable in environments where other methods would

not be reliable. The method uses accelerometers and gyrocopes to track position. Commonly used for smartphones and as complementary sensors for VR/AR headsets.

2. **Vision-based tracking:**
   (a) **Model-based tracking:** Can be less robust than marker-based and has higher computational costs. Relatively cheap hardware. Markerless tracking. Potentially used by solutions such as HTC Cosmos.
   (b) **Marker-based tracking:** Cheaper to compute marker-less solutions. Not viable for long distances and large outdoor environments.
3. **Hybrid tracking:** Combination of multiple tracking methods. It can be very complicated and computationally expensive.

The following methods of mounting sensors and cameras:

1. **Inside-out tracking:** Tracking components are located on the headset and controllers.
2. **Outside-in tracking:** Tracking components are fixed to the environment and track the headset and controllers.

This topic could require additional extensive research that could be combined with occlusion sensors. Optical IR tracking of VR headsets such as HTC Vive Pro [5] is proving with high accuracy, inside-out tracking with accuracy up to a few millimeters in all experiments. The headset uses two or more IR towers and multiple optical sensors on the headset and controllers, which are used as reference points to calculate their position and rotation. However, these systems can be prone to interference [32, 54], ray occlusion, and may not be mobile enough for AR purposes. One of the cheaper and more reliable solutions is the inertial tracking that is present in devices such as smartphones. None of the solutions are perfect for all use cases. Inertial tracking combined with optical tracking or vision-based tracking method based on environment is one of the popular and good combinations. Well-developed inertial tracking is inexpensive and does not require a complicated setup in most situations. Optical-based setup using IR can be very accurate, more costly solution that can be used, assuming that we will not face interference with IR environment scanners. Vision-based tracking can be used for initial orientation synchronization or tracking without a complicated optical setup.

2.3.4 Storing the Data

Point clouds in the model-based approach need to be buffered and translated by the desired mesh generation algorithms. We can approach

this problem in two different ways:

Live Data Reconstruction
We only display the latest information to perform the reconstruction. The set of data and the amount of updates are adjusted to computing capabilities. Old information needs to be overewritten or disposed of.

+ Objects that came to vision temporally are removed from the occlusion data quickly.

- High temporal instability and holes in the accuracy of rendered model and accuracy in many cases, as the point-cloud is limited to the pool of rays produced in a very short period of time.

Cached Data Reconstruction
We can cache information collected to carry out the reconstruction. Data are collected over a whole period of time or for a specific period of time.

+ High accuracy and temporal stability as the mesh data is cached.

- Moving objects create permanent or long-term objects.

- It can be expensive to store N time of the sensor/model data, depending on the reconstruction algorithm and implementation.

Taking into account these aspects, the marching cubes algorithm is most suitable for cached data reconstruction, as it uses a stable array of weights. The weight values can be increased or reduced using rays. For high accuracy, live data reconstruction of marching cubes can be a very challenging task, as the gathered point cloud can be insufficient to reconstruct a model without significant holes in the model. The holes in the reconstruction will cause problems with the perception of occlusion and potential issues for the program logic that uses the mesh data.

This will be further evaluated in results.

2.3.5 Engine and Order of Execution

Each game engine is constructed of many subsystems that give it functionality, usability, and make our implementation easily extendable. In unity, the execution order of the core functionalities is as follows [66]:

1. Physics
   Physics updates are executed in fast time interval. This means that the physics update call can be executed multiple times per frame.

   (a) Execute FixedUpdate Functions
   (b) State Machine Update
   (c) Process Animation
2. Input Events
3. Game Logic
   Unity executes logic scripts developed by us.
   (a) Execute Update Functions
   (b) State Machine Update
   (c) Process Animation
4. Scene Rendering
   Unity dispatches rendering data to execute rendering. The single frame is being finished.
5. Gizmo and GUI Rendering
6. End Of Frame

While developing AR reconstruction logic, I conduct the dispatching via the 'Update' function of the 'Behavior' script class we can extend. It helps us to perform tasks per frame, which will help us potentially adjust the performance of AR systems to desired or available capacity. It also helps us to benefit from GPU before rendering the part for more efficient resource usage. By observing the delta time and mean time for single-sector processing, we can determine how much we should compute. However, updating a fast amount of data per frame should also suffice to make the timings stable. Since our tasks are demanding, they can take the majority of computing power. Therefore we either need to conduct frame smoothing or make the amount of computations relatively stable during each frame, and potentially adjust to CPU/GPU loads of other program features.

2.3.6 Objectives

To make the research results objective, transparent and useful to potential readers, I should define some points that were important to consider in this project.

Since we are working with computing and performance demanding tasks the resources will be our main reference point. Our main metrics in this system are: time elapsed (in milliseconds and ticks), frames-per-second, frame-time, RAM/VRAM usage with relation to cube size and points/eight updated per frame. These measurements can be put in tables or mentioned directly in text. Computer specifications should be noted and background processes should be minimized. To measure qualitative data,

we can use engine profiling tools that can give us access to a large part of this information [62]. Some specific cases, like code/dataset performance, may require development of simple, custom measuring tools.

The results of the program and the qualitative methodology give us a clear indication of whether the solution is more performant in terms of FPS and memory, but it cannot reflect the actual usability of the solution. The visual accuracy and fidelity of the algorithm are impossible to measure with mathematics, as they are qualified by developers and user perception. Anomalies and spikes in run-time are also hard to plot with a graph and thus need to be described and explained by analysis. Additional usability is something else to consider in the process, as such an AR framework can be used for many different purposes. Pictures of the solution can give the reader some room for interpretation.

The development of the AR-occlusion system is time consuming. Game engine tools and APIs help us to achieve the desired results faster than doing it from scratch in languages like C++. However, to cover all possible areas, it would require more time or multiple people. Furthermore, the right hardware equipment can be hard to get, and its software coordination with the virtual world is a big task to solve, and it was concluded that it is not a mandatory part of this thesis. Hardware objects will be replaced by virtual prototype counterparts that mimic the behavior of real hardware. The scope is mainly focusing on surrounding reconstruction and occlusion in real-time. Some of the problems that occur and are mentioned in this thesis will require more research to be effective solved.

The main goal is to test the model-based reconstruction usable for for AR occlusion. There are many algorithms and solutions to consider. The conclusion taken was to find a performant solution that is safe to be finished in the limited time. It is clear that we can go further with the solution, as the collected data may be useful not only for AR developers but for any computer science tasks that want to improve program performance via GPU computing.

The technology, libraries, APIs, and tools that will be used are under constant development. The features can change quickly over the coming years. This means that some parts of the thesis and linked web pages may become in some degree outdated over time. Unfortunately, when working with specific tools, this is unavoidable. The fundamental knowledge behind sensors, data-structures remains highly persistent over the decades. Therefore, for fresh and useful information, it is best to consider scientific articles, books and grasp some of the newest solutions.

Keeping these points in mind, we will create a working real-time prototype of AR reconstruction and occlusion. Approaches, observations, and explanations will be included in the Development section. Quantitative data and qualitative walk-trough are included in the results section of the

thesis. I have also included glossary where I mainly try to describe some of the significant terms with my own words, and list of acronyms at the end of the thesis for reading convenience.

Chapter 3

Development

3.1 Conducting the Computations

3.1.1 Hardware and Parallelization

Single-chip processing units have been shown to be the most efficient way to reap performance according to Moore's law; both for CPU and GPU technologies [57]. Both the CPU power and the number of cores have increased significantly in recent years. Manufacturers want to put more computing power on a single chip, avoiding the problems caused by power (and therefore thermal aspects), memory, and instruction level parallelism issues in chip architecture [28]. However, one of the biggest pitfalls of multicore programming is its complexity. To increase performance, we want to perform as many tasks as possible on all CPU cores. This leads to hazards related to memory access or race conditions that require synchronization or mutex (mutual exclusion). Since operating system scheduler can interrupt current thread at any time, we either trade security or speed. For instance, if data to be computed need to be used in incoming steps, we need to trade off speed to synchronize the jobs with a barrier. This is a clear trade-off but is essential to avoid race conditions. If we want to go deeper, we can evaluate interference at the level of atomic operations. However, considering all the possibilities is close to impossible to cover and automatically debug. Furthermore, when we develop operations on lower levels, they start to be quite architecture-specific. Operations like test-and-set, for instance, which is considered as a single atomic operation, are present in x86 but may not in ARM instruction set. Python is a very popular scripting language, which has disappointing support for multithreading compared to C# or C++ or newer languages like Rust and Go that approach currency differently than previously mentioned. Meanwhile, most of the languages have their specific uses, strong and weak sides, and concurrency still seems to remain one of the harder aspects of programming.

GPUs from Nvidia and AMD have clearly outpaced consumer GPUs in matters floating-point computations. Furthermore, modern GPUs tend to be equipped with high-speed memory and thousands of cores specialized in floating-point calculations used for shaders and any massive, independent computations. However, since GPUs tend to be excellent at parallelism and independent data, synchronization through barriers and atomic operations (leading to kernels being stopped by memory wait states) tend to lead to significant performance penalties [21]. In these tasks, the CPU still plays an essential role, as it reads the data from drives, loads them into RAM, and eventually sends them to the GPU to perform the computation. In the event of rendering, models or textures can be uploaded to the VRAM before desired frames are rendered. If the graphics are meant to be displayed on the screen, the merged data can be directly streamed through video output. Programs intended for the GPU are called **shaders**. Shaders intended for specific computations are often named **compute shaders** in graphic APIs that allow us to perform rendering and computations on the GPU.

Further details and comparisons between efficient GPU and single-threaded mesh generation algorithms can be found in the next section.

The way in which to conduct computations depends on the engine and the graphical API that we use. In our case, the compute shaders [63] are managed by Unity Engine. For programming of compute shaders, we use the HLSL styled language, and the approach is very similar to CUDA or OpenCL. How do we conduct such computations? Let us demonstrate a small part of marching cubes compute shader:

```
#pragma kernel CSMain

// We can import functions or values from other files.
#include <Assets/EclipseAR/ComputeShaders/ArrayUtility.cginc>
#include <Assets/EclipseAR/ComputeShaders/MarchingCubes.cginc>

// [...]
// ========= Input Data ========= //
// Size of marching cubes in XYZ dimensions.
int3 _matrix_size;
// Size of cubes, in meters.
float _cube_size;
// Border padding.
float _padding = 0;
// Input weights buffer.
StructuredBuffer<float> _voxels;
// Target tolerance
float _tolerance = 0.5f;
// Level of detail
int _detailReduction = 1;
//[...]
// ========== Output ========== //
// Output verts.
AppendStructuredBuffer<Triangle> _output_verts;
// [...]

[numthreads(8, 8, 8)]
void CSMain(int3 id : SV_DispatchThreadID)
{
        if (id.x > _matrix_size.x - _padding) return;
        if (id.y > _matrix_size.y - _padding) return;
        if (id.z > _matrix_size.z - _padding) return;

        float3 cubePosition = float3(id);
        int linearIndex = GetMatrixLinearIndex(id);

        CubeCorner cube[8];
        ReceiveCube(id, cube);
        ComputeCube(cubePosition, cube, linearIndex);
}
```

Figure 3.1: Part of the code of the GPU marching cubes compute shader.

In the code 3.1, the shader code starts with the *pragma* declarations common to shaders. The global variables are set before the compute-shader is dispatched. For the compute shader, the declaration of *#pragma kernel* is a requirement. The shader starts with the declared *CSMain* function. Very important to note is the attribute above the function *numthreads(8, 8, 8)*, which defines the number of threads in the group (8 * 8 * 8 in this case), described in three dimensions. According to that, each thread will receive an *int3 id*, which we later use to find the correct cube to be calculated by the thread. Each of these threads can be computed by different GPU cores. The more threads in a group, the fewer groups will be executed, which can improve performance. This means that *numthreads(1, 1, 1)* will lead to

the work of only one core at a time, without the benefits of parallelism. Note that threads that finished their work earlier than the other threads in their group may need to wait for the whole group/dispatch to finish and can be unusable by other processes. In this case, thread groups work like common concurrency barriers. At the end of each thread's work, the code adds triangle data to mesh AppendBuffer, which ensures atomically safe insert operation for the working cores.

```
public static void ApplyWeightsToBuffer(ComputeBuffer targetBuffer,
    Sector sector, int3 cubeCount,
    Vector3 offset = new Vector3())
{
    Vector4[] cloud = sector.updates.GetUpdates();
    if (cloud.Length == 0) return;

    ComputeShader pointsToWeights = ArManager.Singleton.pointsToWeights;
    pointsToWeights.SetInts("_matrix_size", cubeCount.x, cubeCount.y,
    ↪  cubeCount.z, 0);
    pointsToWeights.SetFloats("_origin", offset.x, offset.y, offset.z, 0);
    pointsToWeights.SetFloat("_cube_size", sector.cubeSize);

    ComputeBuffer cloudBuffer = new ComputeBuffer(cloud.Length,
    ↪  sizeof(float) * 4);
    cloudBuffer.SetData(cloud);

    int bufferSize = cloud.Length;

    pointsToWeights.SetBuffer(0, "_cloud", cloudBuffer);
    pointsToWeights.SetBuffer(0, "_weight", targetBuffer);

    pointsToWeights.Dispatch
    (
        0,
        bufferSize / 256,
        1,
        1
    );
    cloudBuffer.Dispose();
}
```

Figure 3.2: Part of the code of the AROperations file for dispatching the marching cube weight update by point cloud. This function sends rays to GPU for weight buffer update.

In the C# function 3.2, the points queued for update are sent to the allocated compute buffer. Since the single point is described by XYZ floating point values and Q describing the potential weight of the point, we use *sizeof(float) * 4*. The compute buffer reference is set and its values are changed to current needs, including the reference to compute buffer with fresh points. When all values and references are set as desired, the compute shader is dispatched using the function *Dispatch()*. This operation is blocked by default, as the code after dispatch can depend on

the data computed on GPU. The first parameter of the function specifies the kernel number, and the other three specify the amount of groups in XYZ dimensions. The rule of thumb is to divide by the number of threads declared in the compute shaders *numthreads*. In this scenario, the compute shader that we use declares *numthreads(256, 1, 1)*. Since we are working with a one-dimensional array of points, we use only the X dimension. This code could be further optimized to reduce the constant allocation of memory that garbage collectors need to handle, using valuable processor time. Furthermore, it is important to note that we cannot dispatch more than 65535 thread groups in total. In such a situation, the number of threads in a single group needs to be increased to cover the entire dataset. For a better understanding, consider the following examples and their explanations.

Example of thread groups: We want to process 10,000 points. Each point must be translated into marching cube weights by a single thread. Using *numthreads(1, 1, 1)* would make this operation significantly slower than CPU, as we calculate only one point at a time, as it is calculated by a thread group with only one thread. Furthermore, it would mean that we need to dispatch 10,000 threads, leading to an exception, as 65535 is the limit of groups. Using *numthreads(256, 1, 1)* results in 256 * 1 * 1 number of threads in a single thread group. This should result in 256 cores that calculate the weights for 256 points assigned to them in parallel. In the dispatch function, we need to calculate the total number of groups to cover the entire dataset, which is 10,000 / 256. Rounded to the higher positive integer results in dispatching of 40 thread groups with 256 threads each. Note that the last group to be executed contains only 16 points to evaluate, which means that the remaining 240 cores have no work to do and should remain idle. Therefore, it is wise to use guards with *return* as in the demonstrated compute shader core. This prevents the cores from running out of bounds. It helps to avoid potential problems related to the lookup of an invalid index when *int3 ID* can be outside the bounds of the array structure.

Example of thread ID usage: In marching cubes, the weights are represented in a 3D array. Each cube needs to be evaluated, which means that we must send a thread for each existing cube. Therefore, all cubes located in 3D can be numbered by the integer value *XYZ*. As the threads receive unique *int3 ID* within the given range, each becomes dedicated to a single cube they calculate. Some threads may exceed the 3D marching cube weight array size because of thread group size, but they will be stopped by guards that check if their ID does not exceed bounds of the array.

3.1.2 Mesh Generation Algorithms

We have a few different things to consider before we can generate a 3D model. One of them is that we use triangles or quads. We can approach the problem of model-based reconstruction in many ways. Reconstruction

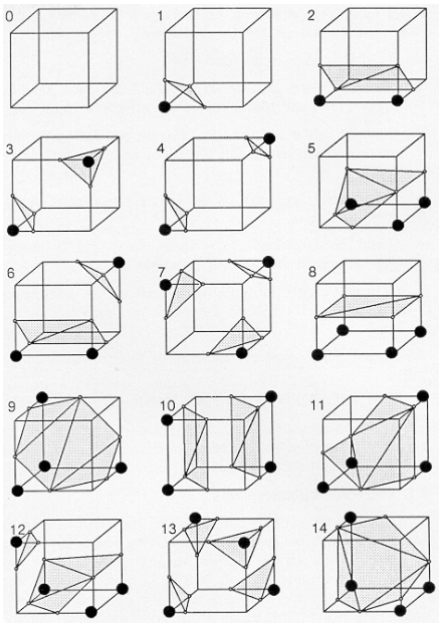

Figure 3.3: Marching cubes and all notable weight configurations[33]

algorithms and their properties and features, as well as complementing systems, are the key knowledge toward prototyping and designing new solutions for the future.

In this section, we cover some of the most significant algorithms that have been developed in the last decades. We will also look at papers that can be used for real-time reconstruction purposes and their results as a matter of quality and time required for computation, if available.

Marching Cubes

Marching Cubes algorithm is one of the most well-known mesh generation algorithms that was ever created and has been used in vast real-life appliances such as medical scans or voxel terrains of various complexity in 3D games. The algorithm was patented on June 5, 1985 and published in 1987 by William Lorensen and Harvey Cline. The patent has expired [15]. This section is based on the Nvidia article published by Geiss in GPU Gems 3 [22], website article 'Polygonising a scalar field' by Bourke [7], as well as the original paper published in 1987 by Lorensen et al. [33].

The detailed process of conducting marching cubes can be described in the following steps:

1. **Allocate weight structure and mesh structure.**

   Any 3D space is represented in weights. Weights can be described

as floating or integers, respectively. We allocate enough weights to cover the designated space volume. In this example, we assume that each cube covers a volume of 1 cubic centimeter. Each weight is one corner of a cube. The structure of the mesh depends on the number of busy cubes and the complexity of the combinations that occur. The memory aspect is discussed in another section. The weight table of Marching cubes can be reused and modified as many times as we need.

2. **Pick cube/s for which we want to conduct marching. Grab the 8 weights for the 8 corners of the cube/s.**

   Most of the weights in the table are shared between multiple cubes for computation: weight can be one corner for up to 8 cubes. Weights in far corners of a 3D array are contained in only one cube because array bounds. This 3D nature of marching cubes table ensures continuity in the model: all faces are always connected together, waterproof. It also means that these weights should not be changed whenever marching is in progress.

3. **Check if corner weight values are within surface level.**

   In marching cubes, we use a value called isosurface or surface level, which we use to define either if given weight above the value X should be considered as solid or as part of empty space. We generate a mesh only between empty and busy spaces. The functionality of the surface level depends on how well I manage the weight updates. Too low values may result in more model noise and lower temporal stability. Too high values can result in parts of the model missing and appearing as holes.

4. **Calculate combination number based on active/inactive weights, using bit shifting.**

   As we are finding that if a given weight is above or below surface level, we mark one of the corresponding 8 bits of an byte or integer as 0 (let us call it negative) or 1 (let us call it positive). If all corners are positive (combination number 255) or negative (combination number 0), no model will be generated and the cube marching process can be finished early.

5. **Lookup the generated combination number.**

   Since we are marking one of eight bites for each corner, we have 256 possible triangle combinations for each cube in total. To make the process easier, we use an array with vertex data for easy generation of triangles. Looking up the number we received in the marching cube table, we receive positions of vertices to generate the correct triangles for the specific cube.

6. **Interpolate vertex positions based on weights to smooth the model (optional).**

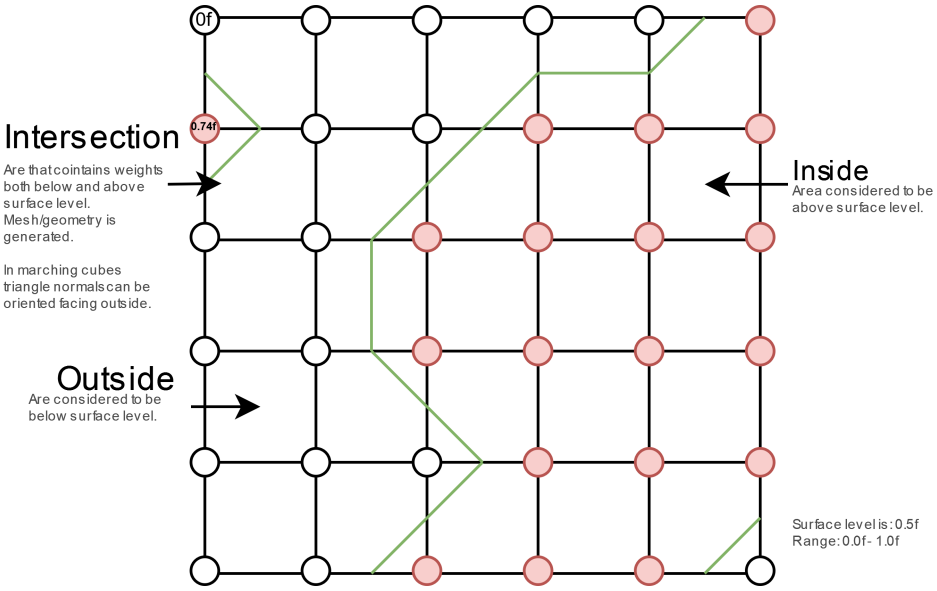

Figure 3.4: Representation of the marching squares algorithm (2D), a sibling algorithm of the marching cubes (3D).

As we generated the mesh for given cube it is almost ready to be added to the mesh we are generating. However, the weights can give us other useful information that can improve the quality of our mesh. Since weights can be floating points (or integers) and their range can be described in millions, these data can be used for interpolation. Interpolation of vertex positions can serve as an excellent way to smooth the model and reduce the "blocky" appearance of marching cubes.

7. **Insert triangles to the mesh data pool.**

   Three vertices form a triangle. These data can be inserted into append mesh array or fixed size array. In this step, we can also generate triangle-face normals based on vertex positions. This is relevant for us if we want to apply lighting effects, but less relevant for AR occlusion purposes.

As mentioned above, marching cubes use a 3D weight table. It makes it a preferably cached algorithm; we allocate the array once and all changing operations should be conducted on the same array. It helps to avoid constant allocation and release of memory, as well as extensive usage of **memset**, which can be time-consuming operations in real-time generation projects.

When marching cubes are used, it is common to use triangles, which is also the basic topology for meshes in the Unity Engine. The mesh topology is related to how a model is constructed in terms of the distribution and connection of the vertices and edges. When using triangles, all mesh data

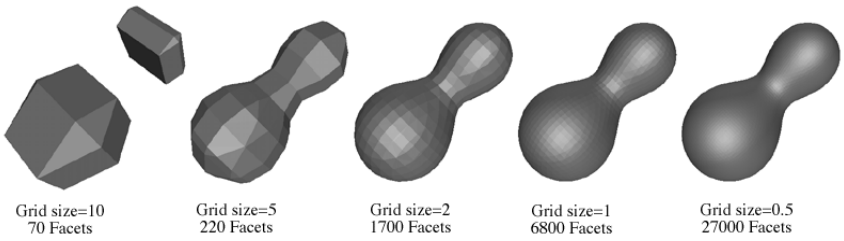

Figure 3.5: The surface quality of the Marching cubes depends on the size of the grid. [7]

are aligned to 3 vertices to represent all triangles in the model. The more optimal approach in cases where multiple triangles use the same vertices is index-based.

More efficient and complicated algorithms have appeared over the years. As Meerits et al. [35] and Zhang et al. [84] presented a more efficient way to handle large data sets. After all, a lot of memory and power can go wasted on unused cubes, and the model topology is bound to the cubic nature of the algorithm. Furthermore, very detailed marching cubes models can generate unnecessarily high-number triangles (including not needed back-faces) that need to be considered when using the mesh for occlusion and also take up computing power and memory. Marching Cubes, however, can still be considered a good candidate for this thesis because of their relatively low complexity and suitability for parallel computing.

Poisson

As the technology was progressing, many more or less competing mesh generation algorithms appeared in scientific papers. As with many things in computer science, a single problem can be approached in many different ways. One of the more well-known is the Poisson reconstruction algorithm released in 2006 that approaches the problem of surface reconstruction as a mathematical problem of the Poisson distribution. For this section we will use Kazhdan et al. papers: 'poisson surface reconstruction'(2006) [30] and complementary 'screened poisson surface reconstruction'(2013) [29].

The global and local fitting mentioned in the papers are important things to consider, as we can compute model data on a global or local scale. With a local scale, we can mean considering only a specific space or neighbors. The creation, connecting, and interpolation of triangles can occur on different scales in an algorithm for mesh generation. The global approach can result in safer and better mesh generation, but at the potential cost of computing time. Meanwhile, the local approach can be wiser, as we may only be interested in updating relevant sectors while meanwhile assuring their quality with minimized overhead caused by unnecessary re-computation.

The easiest approach to store and represent the data would be a regular 3D grid structure. However, it is clear that a 3D grid structure can be both detail-related, memory-related, and computationally impractical as the number of triangles grows. Therefore, the researchers used the **the adaptive octree** approach, where instead of a simple 3D of fixed detail, we can contain more 3D volumes in a tree-like hierarchy to increase accuracy on demand. In short: areas with no triangles or low quality will use a single 3D space node. Meanwhile, as the demands for accuracy increase, the data structure will scale up and split by adding children nodes to a certain limit to increase possible accuracy. The limit on how detailed our model can be is based on the depth of the octrees.

The pipeline for Poisson reconstruction can be described briefly as follows:

1. **Gather oriented points.** Poisson algorithm uses oriented point cloud points to determine the surface in incoming computations.
2. **Calculate indicator gradient and conduct indicator function.** To avoid generating an unbounded vector field of values, researchers encase the indicator function with a smoothing filter. The process helps determine the inside and outside of the scanned mesh to make a waterproof surface.
3. **Compute average spacing and implicit surface.** In these steps, we ensure that each sample from the previous step is proportional to its specific location on the surface. Therefore, we select an isovalue that is a weighted average based on the values at the sample positions. This process is rather complicated mathematically and can vary depending on the implementation.
4. **Triangulate mesh.** Mesh is triangulated based on data generated in the previous step. In this step, we need to consider points globally/locally, which requires octree searches.

The documentation of the CGAL library [52] contains the C++ implementation of the poisson algorithm. It is worth considering when trying to understand, experiment and implement Poisson and other reconstruction algorithms.

It can appear that octree is a clever data structure for mesh generation, since it reduces the waste of memory by giving us a lot of flexibility. However, it is a hierarchical data structure that uses linear search and is not suitable for GPU implementation. Medeira et al. [34] paper published in 2009 proposed a GPU octree using GPU streaming and pararellization methods for searches.

It is also important to note that such tree traversal, especially geometric search in octree, can be considered an extremely computationally wasteful operation in mesh generation, especially if it would be possible to minimize

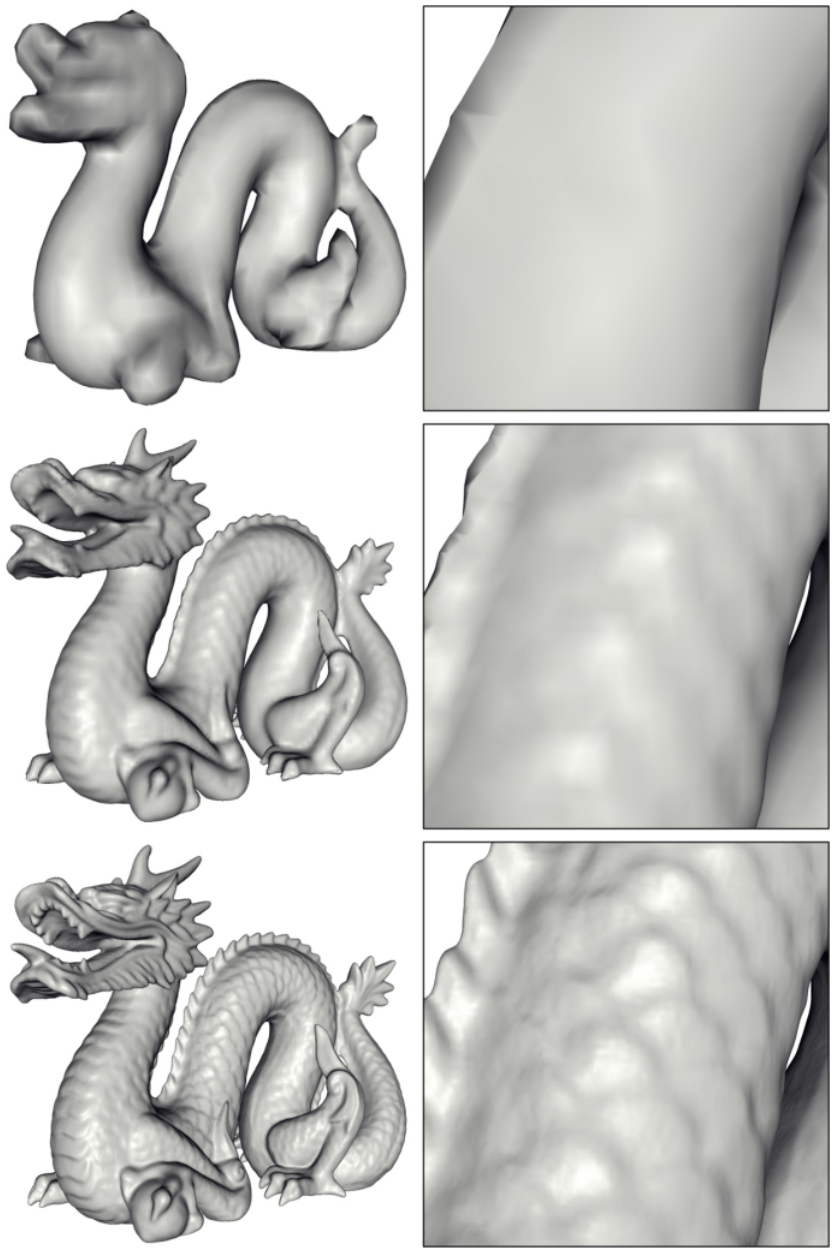

Figure 3.6: Effects of octree depth (6, 8, 10) on the generation of dragon model. [30]

that by methods like direct lookup. Hence, increasing the depth of the octrees of algorithms such as Poisson can lead to very long mesh generation times, as mentioned in Meerits et al. [35] (specific timing examples in the incoming sectors).

Considering aspects of Poisson and its adaptability, it can be considered as an algorithm worth mentioning for our case that requires high accuracy, memory efficiency, and adaptability to hardware computing power. However, the CPU Poisson demonstrated in the papers will not make real-time

Tree Depth	Time [sec.]	Peak Memory [MB]	# of Tris.
7	6	19	21,000
8	26	75	90,244
9	126	155	374,868
10	633	699	1,516,806

Table 3.1: Computation time of the generation of the dragon model mesh generation performed by Poisson with different depths of the octree tree. We can see that as the tree depth increases, the computation time (in seconds) increases significantly. The change in completion time between 7 and 8 is 4.333 times, while 9 and 10 are as high as 5.02. [30]

reconstruction for occlusion purposes possible because of the mentioned design choices. For comparison, Kazhdan et al. (2006) presented the following table (see Table **??**):

Method	Time [sec.]	Peak Memory [MB]	# of Tris.
Power Crust	380	2653	554,332
Robust Cocone	892	544	272,662
FastRBF	4919	796	1,798,154
MPU	28	260	925,240
Hoppe [et al.] 1992	70	330	950,562
VRIP	86	186	1,038,055
FFT	125	1684	910,320
Poisson	263	310	911,390

Table 3.2: Performance of Stanford Bunny mesh generation performed by different methods in Kazhdan et al. [30]

The run-time demonstrated in seconds concludes that the given CPU-based algorithms will not suffice. Bigger scenes can be even more complicated than mentioned Stanford Bunny, and any model needs to be transported from CPU to GPU and then rendered in each frame. Therefore, efficient, high-parallelity algorithms suitable for GPU computations are the only answer to the issue.

For further demonstration of Poisson features, we can consider CGAL, an open source library designed for geometry computations. In their user manual for Poisson surface reconstruction [52] authors present a case study demonstrating algorithm behavior in different use cases as well as libraries algorithms performance. One of the important things to consider in algorithms in poisson in its presented form is how memory and computation power scale with the dataset. This directly affects how well

this algorithm will perform memory-wise and in seconds of computation times. As we can see in 7.1 of the paper; 60 000 points were solved within 15 seconds; meanwhile, 1 800 000 points solved within 478 seconds. In addition to a 30x increase in both the data set and the computing time, the statistics show a 6.22% time increase as a scaling cost, suggesting a well-scaled time complexity. Note that it is clear that a higher number of points to consider will increase the computational time (more or less) in any mesh reconstruction algorithm, while it does not directly affect slightly the quality of the generated mesh. As we can see in Section 7.2 'Contouring' and Figure 61.14; with very low point set spacing, the number of points increases and the reconstruction error is low. However, the computing time cost is immense for such a small error reduction. Therefore, the best spot between accuracy and error is desired; between 0.2 and 1 average spacing, we can see a reasonable balance between reconstruction error and contouring duration. The interpretation of "reasonable" is hard to objectively define, as the result is a balance between computational power that we have, power that our accuracy requires to compute within a given time frame, and error. Even a large error like 0.76 mm for an average spacing of 2 can be insignificant for AR display and human eye to notice, meanwhile relieving a lot of CPU/GPU time and memory. Furthermore, as suggested in [7.3] and [7.4] due to the nature of the scaling and memory limitations, it is highly recommended to simplify the point cloud captured by the sensors.

The Zhou et al. [85] paper presents techniques that allowed reconstructing the previously mentioned bunny model in 190 ms on the CUDA GPU instead of 39 s when compared to the CPU. This results in a potential frame rate of 5.26 FPS, which can be considered as decent when compared to the CPU algorithms above, and maintaining high-quality surface reconstruction in reasonable time. Assuming that hardware improves with each year and we would be able to further develop an improved world/time slicing to ensure comfortable frame rates; it is an important option to consider for a potential project on larger scale.

Real-time Reconstruction Using Single-Camera

As technology develops, users want to use AR technology on devices that are less specialized for it: mobile devices. As mentioned before: using specialized equipment such as IR-sensors, LiDAR, or multiple cameras to perceive the depth are common tools for mesh reconstruction. However not all mobile devices are equipped with LiDAR or similar depth-sensing technology. However, considering that most modern smartphones have high-definition cameras, we can use them for depth sensing. Most flagship phones marked with this number are equipped with multiple cameras with different fields of view. These additional cameras provide different features for camera functions but, most of all, additional zoom levels. This

compensates for smartphones for the lack of movable camera lenses, but certain zoom levels still result in a decrease in resolution. This is caused by the fact that we zoom in on the pixel image instead of using a lens to manipulate the light, while the resolution stays the same. Theoretically, it is possible to use these cameras for depth sensing, but the different field of view, the small spacing, and the lack of standardization between the phones make it difficult.

Furthermore, since 3D printing technology has been growing in recent years, users may also be capable of building their own AR solutions by modifying existing AR displays or creating new AR display devices. Getting proper stereo cameras or creating our own multi-camera set with the correct spacing and calibrating it can pose a challenge.

As with the human eye, two optical sensing apparatuses make it easier to estimate the depth for real-time reconstruction. However, it would clearly be beneficial if we could conduct real-time reconstruction by using a single camera; something that is present in most smartphones and is easy to get from self-crafted AR equipment. In 2010, Newcombe et al. [41] released paper that proves that real-time reconstruction is possible using camera movement as a method of estimating depth. It is clear that depth cannot be accurately estimated from a single image. However, referring to the human eye; we are still able to estimate depth with one eye closed by moving our head. Algorithm in Newcombe et al. requires the same; it requires head movements from the users, which, on the other hand, comes naturally when using AR and VR. Even if the user were sitting in one place, the active area can be prepared by the users before AR activity occurs.

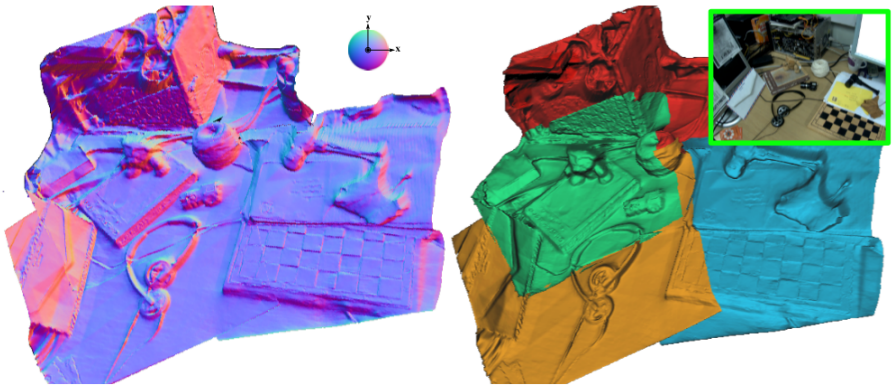

Figure 3.7: Demonstration of the results of the Newcombe et al. algorithm. We are presented with the normal map (left) and the surface built from four local reconstructions (right). As we can see, four local reconstructions generated in the process are merged into a single mesh. [41]

The pipeline can be briefly described as follows:

1. **Real-time camera pose estimation.** We create camera pose estimates

and a set of points of view. It is step necessary to create the 'base mesh'. Note that it is only a single part, a single view that will become part of the global model after completion. The 'cameras' work together to ensure good coverage of the world.

2. **An 'base surface' is generated, and 'base mesh' is polygonised based on zero level.** The algorithm generates a mesh according to the trajectory of the mesh. This mesh is later modified to wrap around the real objects in future steps.

3. **Selecting the cameras.** Cameras with visible surface overlap are chosen before the next step.

4. **Sampling and deforming.** As the reference and neighboring camera frames are sampled by the camera set, the base mesh is deformed to the correct shape. Due to this process, the mesh surface is guaranteed to be waterproof.

5. **Joining reconstructions into global mesh.** After the surface has been modified, the prepared reconstruction is merged into the global model. Unnecessary vertices are trimmed. The global model can be considered waterproof, since the vertices are joined together.

This relatively complicated pipeline can be observed in practice in the demonstration video[42]. The video also demonstrated the usage of the reconstructed model for occlusion purposes (in a static scene) with impressive results.

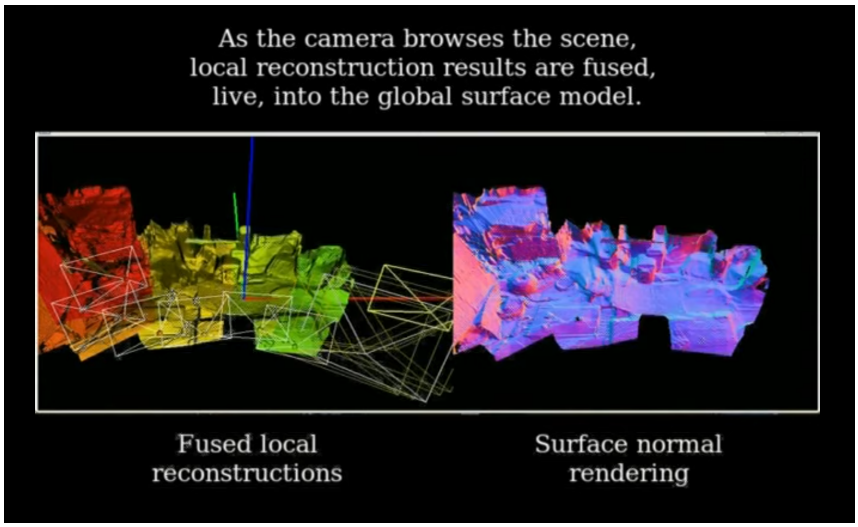

Figure 3.8: Demonstration of the Newcombe et al. algorithm in the video. The camera view (bright yellow frame to the left) is browsing the scene from left to right, generating view frames that after a certain time result in generated mesh patches. [42]

The paper states that the researchers used a 30Hz camera with 640x480 resolution and 80 grades horizontal field of view the lens. They utilized a

Xeon quad-core processor and two GPUs dedicated for variational optical flow computation and live rendering and storage of reconstructions. Sadly, the paper does not provide the exact hardware that was used in the development and testing, but we can conclude that we faced significant improvements in hardware compared to the one available in 2010 that could be already multiple years old. Without further testing, it is difficult to determine whether such a solution would run efficiently enough on mobile devices. However, it is highly probable that the process could be limited to only one GPU with the latest technology.

One of the other things to consider about this algorithm is to update the existing parts of a model. Also, camera sampling and mesh merging operations can be expensive. The addition of new parts to the mesh buffers is efficient; however, updating or removing existing parts can be difficult and expensive. This means that this algorithm would need a modification to be capable of updating existing sectors, but based on its speed to generate images, it will be highly inefficient at tracking and removing dynamics objects from the scene.

We can conclude that the ability to efficiently create high-fidelity, waterproof reconstructions with just a single camera is a significant leap for AR mechanics like occlusion and AR physics mechanics.

Multi RGB-D Camera Reconstruction

In Meerits et al. [35] published in 2019, a scientist developed the reconstruction algorithm with high mesh generation quality and temporal stability.

The process of Meerits et al. pipeline can be briefly described in the following way:

1. **Surface normal estimation.** Normals are calculated for each depth map with the gradient method, as required for the custom Moving Least Squares algorithm. The gradient method is comparable to the previously known principal component analysis method, but with much faster computation times due to time complexity and GPU parallelization potential. Researchers initially wanted to use mesh zippers to reduce rendering costs and obtain a waterproof mesh surface. Since this method does not work very well with GPU parallelization, a custom algorithm inspired by mesh zippering was implemented. At the end of this process, we receive normal and point data.

2. **Surface reconstruction and refinement.** Method generates mesh based on the generated point clouds and normal data. This part includes the steps of initial mesh generation, mesh erosion, and mesh merging.

3. **Final mesh generation.** At the end, all RGB-D camera meshes are merged into one single mesh ready to be rendered.

Taking into account the results, using OpenGL 4.5, an Intel Core i7-5930K 3.5 GHz CPU, 64 GB RAM, and Nvidia GTX 780, they were able to perform the process in **163 millisecond per frame by utilizing GPU**. They performed the same tasks, parallelizing on **CPU was leading to high computation time of 1.6 seconds**. In any of these cases, the results are far from satisfactory for AR experience in matters of time: 6 FPS for GPU and 0.625 FPS for CPU on 6 cores just for mesh reconstruction if the process was bound to FPS. If it could be an independent side process, they would still need to synchronize and share the GPU and CPU time; however, the updates for the whole perspective could happen even 6 times per second with less dependent FPS count of AR game engine.

Hardware and software are improving with each year. To evaluate the new hardware against the hardware used by the researchers in the article, **UserBenchmark** comparison tools for demonstration [76]. UserBenchmark is an online service that collects and analyzes millions of benchmarks of computer components available to users. The team is made up of scientists and engineers who provide a service that is meant to serve users (consumers) and refuse any sponsorship. The statistics shared by UserBenchmarks explore real-world performance rather than theoretical power or promises, making it a good source of information we need.

The results in which we are most interested are the following **Comparing to: Intel Core i7-5930K**

1. **Intel i7 8086K (used for this thesis):** 7% lower memory latency, 31% faster single-core speed, and 26% faster quad-core speed.
2. **Intel i7 12700K:** 5% higher memory latency, 87% faster single core speed and 90% faster quad core speed. It also has 12 physical cores, which is not taken into account.

Note that the data are based on global automated user benchmarks on different hardware. In addition, it is important to consider technological changes. Since Intel i7 12700K uses quite young DDR5 memory on the market, it might not come as a surprise that the memory higher latency of DDR5 may have an effect.

Comparing to: Nvidia GeForce GTX 780

1. **GTX 1080 (used for this thesis):** 142% better lighting performance, 147% better reflection handling, and 69% faster multi-rendering.
2. **RTX 3080:** 401% better lighting performance, 322% better reflection handling, 336% faster multi-rendering.

Assuming a best-case scenario where these numbers directly translate to the presented mesh generation algorithms, using the following GPUs could theoretically reduce the frame times between 25% and 75% which increases the potential amount of updates and AR application features that are possible to implement. Newer graphics cards also have slightly more video memory, which increases the potential mesh detail we are able to achieve. There is no doubt that GPU computational power increased the most and is still the most effective way to solve issues related to mesh reconstruction.

Meerits compared their mesh generation algorithm with **Turk and Levoy taking 48s** and **Marras et al. taking 9s**. It is a significant time difference caused by tree indexing where Meerits et al. uses spatial point lookup by line casting.

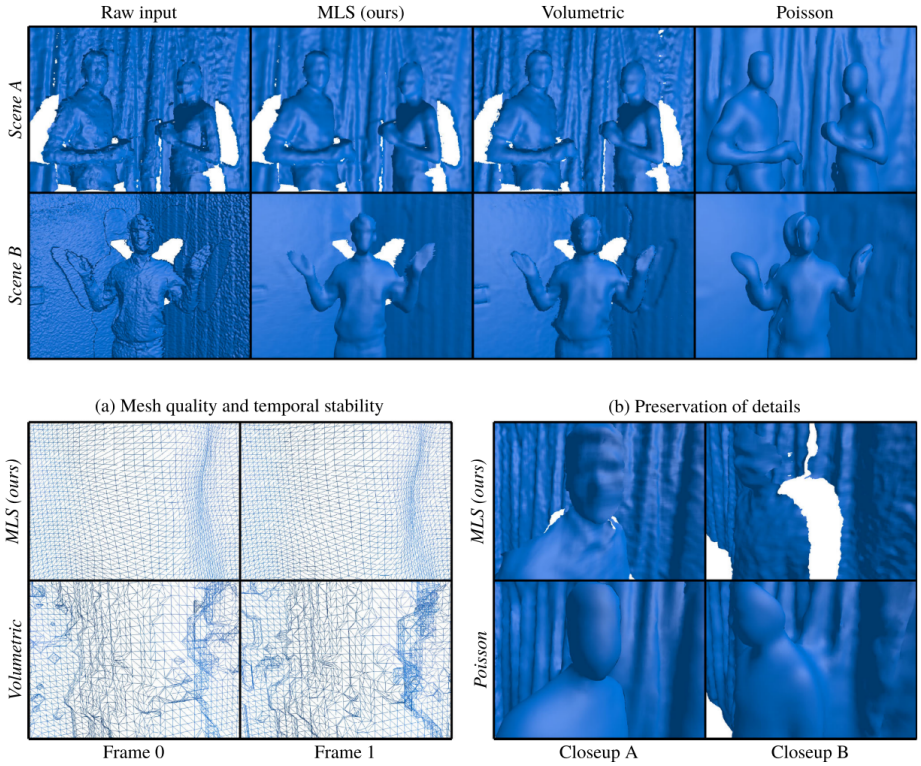

Figure 3.9: Comparison of the quality of MLS papers and other mesh generation techniques in Meerits et al. [35]

This significant difference was due to the time required for the movement of the data. The algorithm shows high potential for any required processing effects and reconstruction behavior: It is highly temporally stable and merges the meshes as a stage, resulting in an acceptable level of waterproofing. The solution is slightly better than implementations of pure Poisson and marching cubes and is one of the notable considerations when developing production-ready solutions for AR.

Concluding on the Mesh Generation Algorithms

The search for effective reconstruction algorithms to run in real time is a challenge. As we can see, over the years, researchers and large and small companies that invested in AR were able to push the technology to the edges where it will perform well enough for real-time reconstruction purposes.

It is clear that technology and solutions must be adjusted to the use cases. For instance: if our AR will be used for training in a static room, it can be a great idea to scan it with high-fidelity solutions like poisson with high number of octaves for detail preservation. However, it is visible that methods that poisson and competing CPU algorithms, especially those that use octree tend to be very slow and are not suited for GPU computation, as they use steps that may be hard to parallelize as they, for instance; imply linear data traversal. High-fidelity algorithms have tendency to take seconds to compute even on the best hardware, which makes them near to impossible to execute in real-time on per-frame basis. This also allows us to optimize the topology of the rooms to reduce the memory and performance footprint. Meerits et al. as one of the recent papers were able to achieve high-quality reconstruction in real-time, but it required the development of GPU-friendly algorithms inspired by slower algorithms that previously had to be executed mainly linearly or concurrently.

Occlusion for dynamic objects can be done through more lightweight approaches like those approached by Meerits et al. and even some of the older algorithms like marching cubes will do its job. If we can develop self-configuring optical systems, a depth-based approach or combination is an option. In this way, we follow the concept of ahead-of-time computations that slightly improve performance.

3.1.3 Memory

RAM, VRAM or Hybrid Approach for Computations

One of the main questions to consider when doing reconstruction and rendering is: Where and how the data should be located? The preferred location and form of the data depend on the problem, the context and the hardware. Based on the marching cube algorithm and the complexity of calculations, the most efficient and practical form of data storage are arrays with a fast lookup and a modification time of O(1). Since our algorithm is executed on GPU, all necessary data must be ready for constant modification and execution of the algorithm. Since the constant transportation of data to or from GPU is bottlenecked by CPU, memory, PCIe, and GPU bus; it is best performance-wise to hold the data on either RAM (when computed on CPU) or VRAM (when computed on GPU).

This means that hybrid computations will be inefficient, as the computed data (such as a mesh) still need to be sent to the GPU for rendering. The engine needs to wait until the data are copied to the GPU, increasing the frame time. Moreover, since the mesh data are generated in RAM and sent to VRAM, we need to reserve two times more memory for the same amount of data. One possibility to consider and test in the future could be if computing part of the data for low-detail sectors would be efficient to execute on multiple CPU cores, while the largest data sets are computed directly on GPU to avoid movement of data amounts that might increase frame time.

Computations on Low-power Devices

When we execute the marching cubes algorithm, we have a large and independent amount of data to "march". The generation of a cube is a relatively simple operation. Thus, as mentioned before; marching cube is an excellent algorithm in matter of pararellization on GPU excellent for high amount of required calculations. As technology progresses, both CPU and GPU technologies will improve; however, it is expected that GPUs will continue to be slightly ahead in matters of mass computation. However, it is important to note that it is cheaper to acquire high-capacity RAM than high-capacity VRAM with a GPU. In current times, it is common to see the production of GPUs with VRAM size from 4 to 12 GB for consumer use. This size did not increase significantly over the last generation of GPUs considering the Nvidia 3000, 2000, 1000, and 900 series. This can be a big disadvantage, as we need to use up limited VRAM/RAM memory and GPU/APU power to increase accuracy, while other features of our application also need to be executed. Memory usage on APU and ARM processors can be challenging to use for compute shader use because of the smaller shared memory and architecture designed for smaller device sizes, reduced heat, and reduced power consumption. This can lead to no performance gains or even reduced performance on ARM devices like mobile phones or x86 computers with an APU when compared to dedicated GPUs. However, due to a chip like Nvidia Tegra [44] we could increase the computing performance even on mobile devices due to higher performance graphics chips.

Marching Cubes and Memory

The marching cubes algorithm allows us to store data in a 3D dimensional weighting matrix. As a result, access to the data is computationally predictable: access to a 3D array happens in o(1) time. However, the o(1) notion only tells us that the operation does not scale with the data set.

In reality, the 3D structure behaves like a one-dimensional array since the

memory has one-dimensional addressing. To access a specific 3D index (x, y, z), it must be converted to a 1D array. In most languages, this operation will still be much cheaper than an operation such as hash-calculation conducted by the dictionaries required for lookup.

Consider that the player moves in an area of 8m x 2m x 8m.

To find the number of cubes, we use the following formula:

$$(x/cubesize) * (y/cubesize) * (z/cubesize)$$

Therefore, to determine the size of the marching cube array for this area, we use the following.

$$(x/cubesize + 1) * (y/cubesize + 1) * (z/cubesize + 1)$$

Note that Y is the height dimension in Unity, unlike other popular programs such as Unreal Engine, Blender, or Cinema 4D. In addition, note that we are adding one to each dimension size in the marching cube array. This is caused by the cubic nature of marching cubes; each cube has 8 corners. Adding one to each dimension is required, as it gives represented 7 corners on the cube.

And assuming that we want cube accuracy of 2cm:

$$(8/0.02 + 1) * (2/0.02 + 1) * (8/0.02 + 1) = \ 401 * 101 * 401 = \ 16,240,901$$

This means that our array will require 16,240,901 of the type used for weights; in this example, it is floating. Single-precision floating points are stored in 4 bytes of data.

$$16,240,901 * 4 \text{ bytes} = 64.96MB$$

This results in a total of 64.96 MB of data for the marching cube array only.

However, we also need to consider the mesh generated by the reconstruction. Each cube can require up to 9 vertices that are represented in 3D coordinates (float3). We might also need normals, which add an additional float3 for each cube.

Cube count calculation:

$$(x/cubesize) * (y/cubesize) * (z/cubesize) = cubecount$$
$$(8/0.02) * (2/0.02) * (8/0.02) = 400 * 100 * 400 = 16,000,000$$

Pessimistic mesh size:

$$\text{cube count} * \text{ max tris per cube} * 3 \text{ (triangle size)} * 7 \text{ (vert pos + normal)}$$
$$16,000,000 * 4 * 3 * 7 = 1,344,000,000 \text{ bytes} = 1,344 \text{ MB} = 1.34 \text{ GB}$$

As we can see, 1.34 GB of data can pose an issue, especially considering that it is for quite small area and, for fact, that VRAM tends to be smaller than computer RAM. However, removing normals that are mainly used for demonstration purposes removes 4 bytes per vertice, decreasing the size by 50% (because of possible 8-byte rounding). Furthermore, an entirely structured, non-appending approach that would provide dedicated memory for each triangle would result in doubled or even tripled memory usage. In the early stages of prototyping, especially with the structured approach, it was quite common to overfill the computers' memory on higher accuracy. Furthermore, it leads to paging, leading to extremely slower computation and a possible program crash. When calculating on GPU, we should reserve the maximal amount of data our algorithm will use. For predictable data structure where array memory resizes is not needed. The possible solution to reduce this significant mark is explained in the optimization section.

Constantly copying the entirety or even parts of these three data structures is not optimal if it can be avoided. Let us consider the theoretical speed of the following specifications:

```
Theoretical RAM transfer speeds (DDR4):
3200 MHz: 204.8 GBps

PCie theoretical transfer speeds (using the x8 slot):
PCIe 2: 32Gbps
PCIe 3: 63.01Gbps
PCIe 4: 125.44Gbps
```

Whenever we send the mentioned data to GPU and then back to CPU, then conduct other draw call operations, it will take more than 2 seconds. This delay is unacceptable when the player is interacting with the AR world, as we can expect as low as 0.2 frames per second if it is done on the main thread. Our goal is far beyond that where stable 30-60 FPS is a comfortable range. If the display hardware allows for it, a frame rate of 120 FPS or higher may make the experience much better for the users.

Considering these numbers and the nature of computer systems, it is clear that operations should be done mainly on CPU and RAM or GPU and VRAM. Achieving reconstruction performance requires reduced data transfer, memory resizing, and good distribution of work management. Parallel and concurrent computing will be essential, as it reduces the impact of the main engine thread loop completion time on multicore devices.

The engine loop computed on the CPU tends to be the main bottleneck that affects frame-rendering speed, as it computes most of the game logic and prepares to draw calls while the GPU may remain mainly idle.

Half Type for Weights

Accuracy of the weights is not extremely essential. Since byte is the minimal measure of memory, we could use byte instead. It would result in 75% less memory usage by using the marching cube compute buffer. Bytes would result in 256 interpolation states, so with 1 CM accuracy, it would result in 0.00390625 CM between each step, which could be acceptable for our use. Sadly, GPUs have memory padding, as they are mainly meant for high-quantities of parallel floating-point computing and similar computations. This means that we cannot use byte-sized data types for our weights. Thus, it would mean that we have two alternative options: bitwise operations and half-type operations. Bit-wise operation could benefit from finding the correct int type in the array and shifting/reading its bytes. This operation may result in an insignificant performance drop, as we need to calculate the correct indexing and perform bit operations. Half-type (half-precision floating point) is commonly used for low-performance devices where shader accuracy is not the main priority. Half uses two bytes instead of four and would reduce our buffer memory usage by 50%. However, according to the HLSL documentation, it is converted to floating in cases such as DirectX GPU devices [36]. This means that results will be obtained only when the project is built for platforms such as Android or iOS. Therefore, I decided to follow the standard floating point approach, as the memory buffer is less than a part of total memory usage.

3.1.4 Runtime Pipeline

Following the information collected and the potential for optimization, the pipeline to generate our mesh-coded solution can be described as follows.

1. **Gather points.**

   Point-cloud points are gathered in structures and queued for processing in the sector to which they correspond in the 3D space. Points can also be sorted into all neighboring sectors for improved consistency if they are close to the borders. This behavior can be replaced by implementing a border-syncing pass. Pool structures can be lists of dynamics sizes and are located in RAM. If the pool reaches specified limits, the oldest points are discarded to avoid memory overflow. In this step, we can also discard points we don't need.

2. **Sector update call: Translate the point cloud into marching cube weights**

Each frame-specific amount of world parts (sectors) need to be updated. Specific amounts of point-cloud data are queued from the array into point-cloud compute buffer, so they are sent over to VRAM. The compute buffer can be persistent and reused for each run. The compute shader runs a GPU thread for each point cloud point and translates it into a marching-cubes compute buffer (also VRAM). Weights are changed based on points proximity to the corner weight corresponds to. We try to translate as many points as possible to avoid clogging.

3. **Visual update: update meshes by marching.**

   Another step per frame that needs to be considered and performed is determining which sectors should be updated visually. By default, we update a set amount of sectors of different LoD per frame to make frame times more stable. The sectors are queued in FIFO, ordered by the integer number of the last update frame. This means that the oldest updated sectors in our vision are in queue for the marching cube update.

4. **Complementary passes.** As the mesh the generated, we can conduct optional complementary passes that would be useful for application developers (like sending physics shape to RAM), mesh quality improvements, or similar.

5. **Render into the depth pass.**

   It can be done as a simple shader that renders compute buffers. We render only sectors visible to us within a given range.

This pipeline design is the basis for the prototype mesh generation and occlusion system.

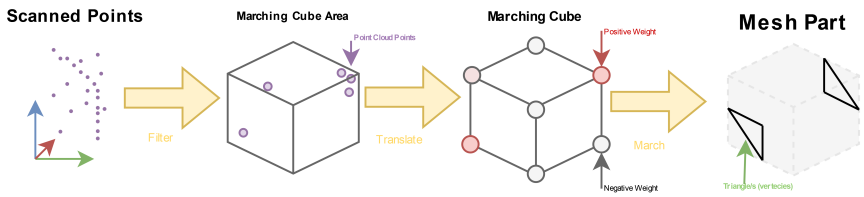

Figure 3.10: Visualization of the AR marching cube reconstruction process.

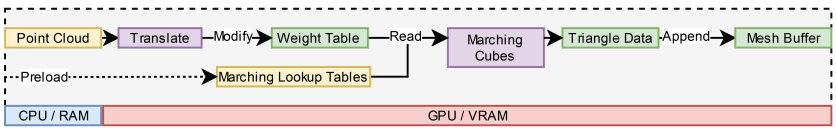

Figure 3.11: Diagram pipeline of AR marching cube reconstruction.

3.2 Optimization Methods

3.2.1 Reconstruction Sectors

Limit draw calls if dataset allows it. Send render calls in batches. Simplify shaders. Do not try to render empty data.

In the early stages of development, I have prototypes marching cubes on the CPU. Computing mesh data for an 8x8x4 meter room was taking several seconds with higher accuracy, like 5 CM per cube. It also means that the whole mesh had to be sent to GPU and was rendered by the camera even if some parts were not within cameras sight. When playing, our location can change; what if we want to walk through a whole building? Allocating hundreds of square meters of multi-level buildings is very close to unrealistic. To solve this problem, we should approach this problem similarly to the concept of "divide and conquer" in algorithm design [23]. Games or procedural shader programs load or/and generate objects based on camera location, where irrelevant parts/objects are unloaded or cut from rendering by occlusion culling [22]. the popular game Minecraft to generate, render, and store its large procedural world using the concept of chunks, where chunks are square segments of the world aligned in a 2D grid, loaded based on render distance [39]. We follow this approach by using cubic sectors of defined size aligned to a 3D grid. The single grid contains the weight and mesh and is used for the rendering process. In this way, we work on many memory buffers and are able to remove irrelevant data from processing and render queues.

In cases where sectors become extremely complex, we need to consider reducing the sector size or implementing sub-sectors. Subsectors can allow processing part of the high-accuracy sector to reduce the performance footprint. However, updating a subsector can be challenging with the preferred compute buffer types.

Each sector can be represented through Unity GameObject. GameObjects are more tangible data units that are easily accessible and editable in the Unity editor. However, instantiating, rendering, and pooling these objects will have a negative impact on performance. To make the solution easier to debug and check by other developers, this would require developing an additional editor tool. In our case, a more direct, but less tangible, approach in the editor is to directly queue camera visible sectors on the GPU.

These sectors need to be stored in a type of structure in which sector references are stored. The simplest option is to use a list, as the number of fragments may vary. However, if we assume that the sector size is 1 square meter; we need to store 1000 references to cover these 1000 cubic meters. When our camera is moving, we need to remove all references outside the

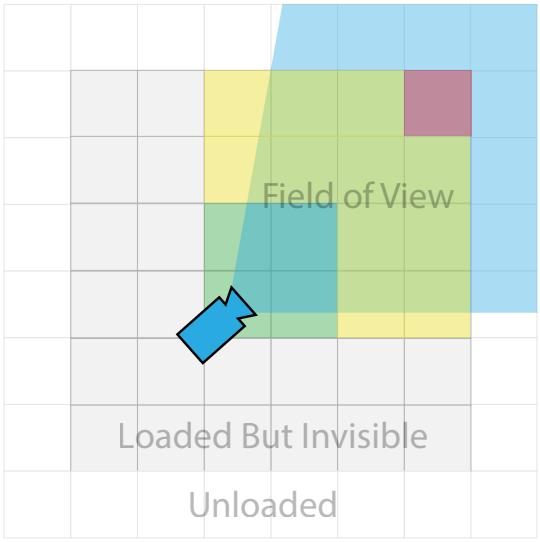

Figure 3.12: The illustration demonstrates the dynamic loading of the sectors. Green, yellow, and red triangles represent three levels of detail in reconstructed sectors that are within the frustum of the cameras. Gray is the active area of which sectors are loaded and considered for rendering and marching. The white area is out of range.

rendering distance. This can result in a high overall cost of o (n) for insert, delete, and find operations [40]. Therefore, the usage of dictionaries is a better option. Such a dictionary can use the int3 type as a key, where int3 contains three (XYZ) int values that will represent the sector index in three dimensions. Dictionary operations happen mostly in an expensive o(1) time; if a dictionary needs resize or insert, the additional operation may take o(n) time just like to list resize. Since the dictionary relies on hashing, the hashing of int3 will be more expensive than the hashing of a simple int, since each value of the class of structure must be included in the hash operation. However, the o(n) cost of single operations on a large sector set will make the dictionary more efficient to use in the given case.

Details about work distribution and frustum culling are explained in the following sections.

3.2.2 Timeslicing

Use the concept of divide and conquer. Segment the data for easier selection and batching. Adjust the amount of data computed to fit the target frame time.

The camera scripts need to filter out sectors that are unused from the ones used on each frame while camera movement is significant, as well as to allocate memory from the pool. Other significant operations that can occur per frame are point cloud translation and marching of marching cubes. The heaviest operation of these that needs to be computed over a constant

period of time is marching cubes for mesh generation, without counting the rendering operation at the end of each frame. Users of such AR systems can purchase, create, and own different hardware configurations that fit their needs and budget. Games and programs scale with the computational capabilities of a given device. As we will be working with different computing power, memory size, and memory speed, we need to adapt the amount of computation to allow it to finish within an acceptable frame time. Therefore, we need to manually or automatically specify the number of sectors computed per frame. Thus, we divide constantly changing scenes into update batches to adapt to available resources to hit desired frame rates. In the prototype program, computations occur via the FIFO queue. The queue is sorted by the time of the last update of each sector and is sorted using a sorted dictionary of active sectors to be updated in the frame.

3.2.3 Semi-Dynamic Memory and Pooling

Pool memory, avoid constant allocation and disposing. We can also group memory buffers for future use, such as dynamic memory, to save memory.

As we are programming different algorithms, we tend to allocate our memory on the stack and heap to perform the necessary computation jobs. However, when building these systems, the scale of our problem can change depending on our or users ever changing needs and context. In many cases, we often do not know the optimal scale of our data structure. It means that in many cases, we either need to discard some part of data or allocate a pessimistic amount of memory. Like mentioned in the planning chapter about memory, in the case of marching cubes, a 3D array of fixed size is unrealistic and close to impossible in real life settings. The testing computer used for the development operates on 8GB of VRAM, which would easily be overfilled with unused memory. Allocating gigabytes of unused data is wasteful memory management and will not lead to better performance.

Furthermore, shaders prefer continuous data structure. It means that if our data buffer is structured, then a cube at any 3D index contains unused triangles that GPU needs to render. It makes shader rendering much slower and makes memory management much harder. The empty space in the structured data buffer is unpredictable, but it is much easier to manage and update by index. Therefore, I decided to use the append buffer instead, where we insert triangle data continuously. It also allows us to conduct memory scaling and memory pooling.

Standard Compute Buffer

+ Cheap to update part of the model.

- Requires to allocate a pessimistic amount of memory. Wastefully.

- Terrible shader performance scaling.

- Slower data transfers.

Append Compute Buffer

+ Optimal shader performance scaling.

+ Possibility to scale memory to current needs.

+ Faster data transfers.

- Expensive to carry out updates.

As we conduct parallel marching cubes computations, triangles can be concurrently inserted into our buffer. The internal buffer count increases as the new triangles are inserted. If the buffer is already occupied, counter is set to 0 and the previous model is being overridden. The counter makes us aware of the total model size; thus, we avoid rendering empty or cluttering data. This method also does not require one to zero out the memory, since counter and overriding serve as our data constraints. As the prototype has developed, this step seems essential to achieve high model accuracy without overfilling the GPU or RAM memory.

One of the compute buffer types in Unity is Append Buffer, which allows us to add our triangles to the buffer in sequence, safely as an atomic operation. This means that the shader will not need to iterate through empty triangles, as the actual data are aligned in a sequence of a given length. This buffer allows method allows us to use a more dynamic model of memory. One of the old concepts used by languages like Java, C# and C++ is dynamic memory allocation of data structures like lists or dictionaries. Arrays are of fixed size. Lists and dictionaries scale with demands to reduce memory usage. If a list does not have enough space to insert an element, the size of that list doubles. This operation is expensive as we might not have enough empty space after the reserved memory of the lists to simply make it longer. Instead, we need to allocate the memory again and free the old one. This concept can be used in the prototype to significantly reduce memory usage. However, to make this method support sub-sectors without allocating multiple compute buffers, we need to constantly switch between two buffers: one for the completed sector, one for upcoming updates that are unfinished.

Other costly operations are constant allocation and reallocation. In games,

objects, such as particles, that would constantly be instantiated and disposed of are pooled. Instead, their structures are reinitialized. In some cases, pooling can have significant effects on performance, memory usage, and the garbage collection process. Therefore, we can pool the memory buffers of the marching cube weights and the mesh data.

The algorithm used in the prototype works in the following way: As a new mesh buffer is required, we allocate them with N bytes of memory, a minimal amount of buffer allowed. As the count of triangles increases, it is being cached. If the amount of buffer memory used exceeds M percent, a new buffer is being allocated. As we will have many mesh data sectors of different sizes, with the possibility of size shifting; we can benefit from memory pooling. Instead of constant reallocation, we can pool mesh buffers instead of disposing of them. As we can have the need for many sizes of mesh buffers, we create a dynamic list of pools. This practice can also be applied to marching cubes weight buffers, but the buffers need to be cleared by the compute shader or rewritten as they enter or leave the pool.

The drawbacks of this method are a minor performance (used for memory management and allocation) and the risk of overflow. When the number of triangles rapidly increases in a single processing, it is possible that the compute buffer will overflow. When marching cubes are conducted, we do not know how many triangles the current set-up will produce. Thus, we need to use the counter of the last computation. The default behavior for overflow is to overwrite from the beginning, which can result in artifacts in the model. This issue can be challenging to fix without performance impact, but its consequences can be seen as low and occasional. The next update of a given sector assigns a new larger compute buffer, which fixes the problem. It is an acceptable result.

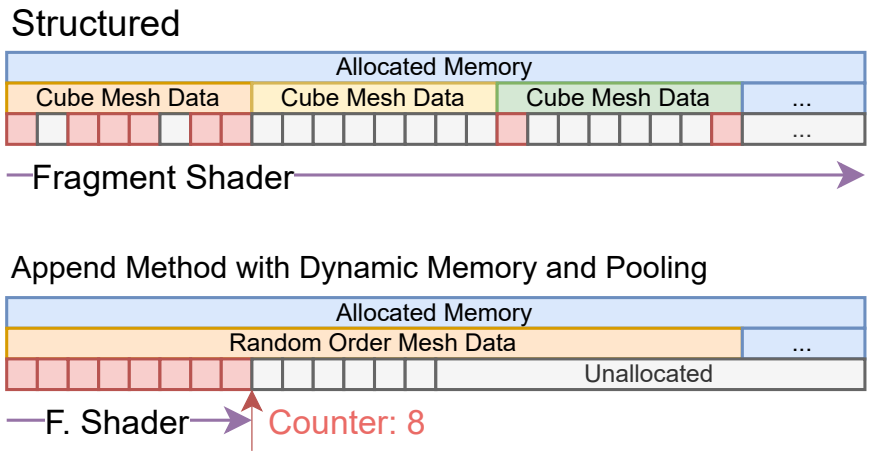

Figure 3.13: The illustration shows an example of the difference between memory approaches.

Unity Compute Shader API allows us to asynchronously fetch data from GPU. It can be used to store marching cubes data temporarily in RAM when the data are not needed at a given moment in VRAM (which tends to be smaller than RAM).

Other option: Append the Buffer as a Ring Buffer

Since we have a requirement for linear memory with preferably linear alignment of mesh data for the shader to render; the options are quite limited as we try to balance data predictability, memory, and rendering performance. Another approach that may make updates of subsectors easier is the ring buffer approach with append buffer. Instead of using multiple compute buffers, we can use a ring buffer and cache the data circularly. This helps us avoid problems where the unpredictable count of triangles of each sub-sector is not put in a predictable linear space. However, it has one drawback that we have to solve: data that are put as a border case, where a sub-sector transitions from buffer end to start. We have two solutions: these data need to be copied to a new buffer in linear order or all subsectors after a subsector in the transition need to be discarded and computed in the incoming update/s.

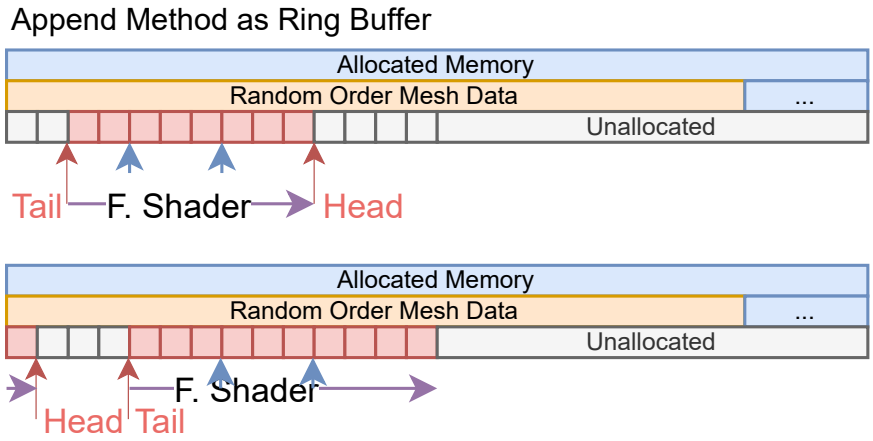

Figure 3.14: The illustration demonstrates the ring buffer approach.

Append with Round Buffer Approach

+ Optimal shader performance scaling.

+ Possibility to scale memory according to needs.

+ Fast data transfers.

+ Cheap to conduct updates.

- Border cases may require two drawing instructions or a specialized shader (when rendering) and a special buffer exchange approach (when more memory is needed).

- Subsector updates can only be performed in sequence and one at a time.

3.2.4 Frustum Culling and Occlusion Culling

Do not compute or render models that are not required at hand.

For optimization of rendering (and computing), we should consider three computing techniques to reduce wasteful rendering tasks [11]. Graphics APIs receive draw calls for group or single materials. Some of them can be out of camera line of sight, hidden behind other virtual objects, or their parts can be hidden from the cameras sight based on angle. These methods can give a significant performance boost when applied in specific cases, and some of them can also be applied to our sector-based computing approach.

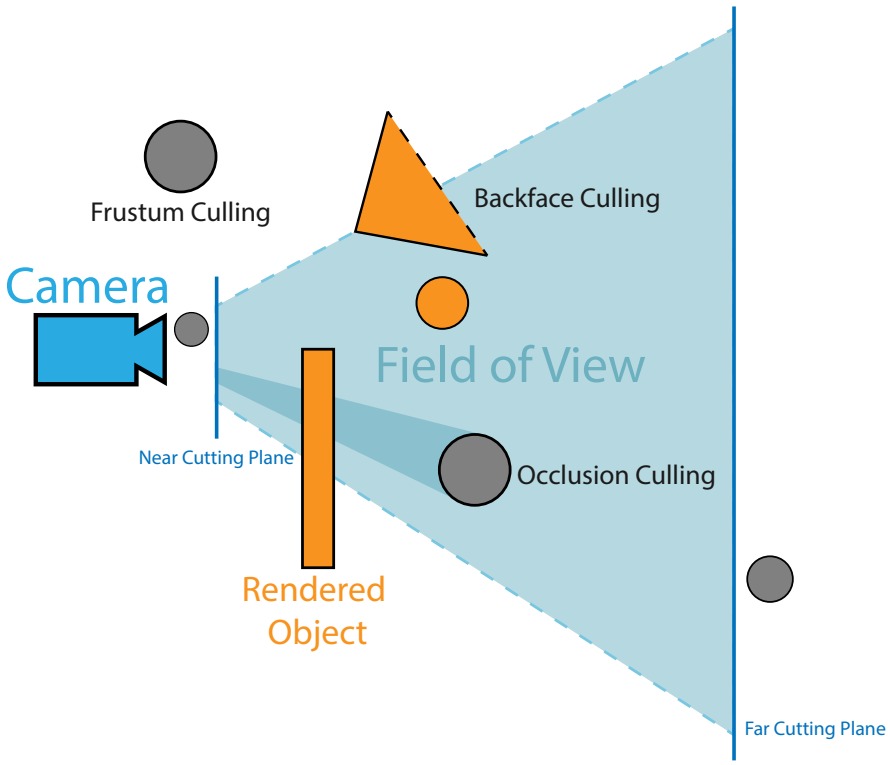

Figure 3.15: Representation of culling techniques.

Frustum Culling

One of the steps before computing and rendering an object is to determine whether it is in the desired viewing range. However, it means that all objects within the range of the cameras are added to the draw queue. It is wasteful, as objects on the side and behind the camera that are not within the camera viewing range (view frustum) are considered in the rendering process. One of the simplest solutions to this issue is frustration culling and frustration updates. We consider the camera orientation and calculate if objects are outside the camera frustum by angle and margin. Sectors outside of camera viewing points are not the main priority and can wait for updates. However, they still need to receive and queue point cloud points for future updates. The program iterates through all active sectors, updates them if they are within defined limits of the oldest ones, and renders them if they are within the camera frustum. Furthermore, sectors could be divided into even smaller subsectors; however, checking if each small subsector is within the frustum range can be more expensive than considering them in a draw call.

Occlusion Culling

Occlusion culling is difficult to apply without extensive testing and development in a context where the scene and models are constantly changing. Unity's occlusion culling system [70] pre-bakes its data in the editor, based on the parameters given by the developer. However, to let the object occlude any other object, it needs to be marked as **static** (immovable) and **occluder** (object that can hide other objects). Calculations behind occlusion culling make it hard and expensive to determine if two dynamic objects are occluding each other in real-time. However, if a dynamic object (occludee) is hidden behind the static occluder, we are sure that the pre-baked data give us enough information to determine that the dynamic object is behind the occluder with the current camera perspective.

Detecting and catching occlusion data is an expensive operation, but allows us to hide objects entirely hidden by other objects. Therefore, in a real-time reconstruction setting, this method will be highly inefficient. Therefore, large-scale occlusion culling cannot be applied in this case to improve performance and could be a subject of future research.

Back-Face Culling

Back face culling (also known as face-culling [47]) is optimization method that helps us avoid rendering faces oriented or hidden away from the camera perspective [2]. If a mesh face is oriented towards cameras looking directions, it means that it is a back-face invisible to the camera. Back-face

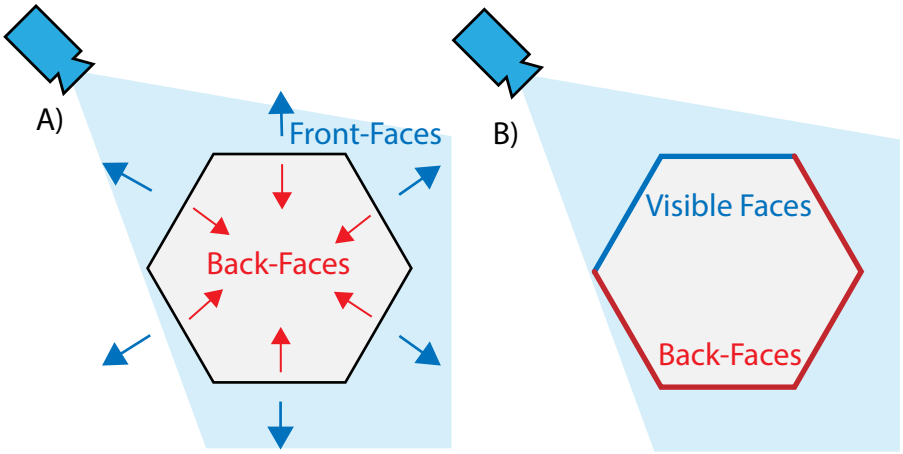

Figure 3.16: Two possible back-face interpretations. Illustration A) does not render inside faces of a model. Illustration B) does not render the back faces of the model.

culling 'cull' in Unity is enabled by default, however, in a more simple way. As the model has, outside and the inside, inside faces are back-faces that do not need to be rendered. While marching cubes are used, inward faces are not rendered. The other approach that considers the backs of the objects faces is harder to perform. To find if a given face is oriented away from camera perspective, we need to compare the camera angle with face angle. In our case, we may have millions of triangles to consider, which can quickly outrun any gains. We also should recall why this method for optimization can grant considerable gains: lightning and shadow computations. The rendering of lightning can be an expensive operation that can make back-face culling methods worth their cost. If our model is unlit, since we want to use it for occlusion, these operations can cause it to render slower than optimizing it. Constant back-face culling computation should be avoided. The cached approach, when the camera is not moving, could be considered; however, there is no easy straightforward approach in our case.

3.2.5 Draw Calls and Shaders

Limit draw call if possible. Send render calls in batches. Simplify shaders. Do not try to render empty data.

One of the most significant issues that can occur while using game engines is the bottleneck caused by draw calls and scene complexity. To render objects in Unity Engine we need to have GameObject with behaviors called Mesh Filter and Mesh Renderer or Skinned Mesh Renderer (used for rendering character meshes). These objects contain materials that are used as shaders. When frames are rendered, all necessary data must exist or be moved to the GPU. Furthermore, materials are dispatched for rendering

using material and scene data. The rendering pipeline can be slowed when each mesh in an object needs to be queued for rendering in separate draw calls [61]. In some cases, draw calls are resource intensive to the extent that computing a draw call takes less time than preparing it. Instead, it is more optimal to compute these data in shared batches, minimizing the number of draw calls and repetitive computations that are unnecessary. Unity suggests three optimization methods with the following priorities: SRP/static batching, GPU instancing, and dynamic batching. Let us begin with some examples to demonstrate possible problems.

GPU Instancing instead of rendering each object separately, the meshes are rendered in batches. This method is very effective when we need to render high counts of nearly identical objects. It is one of the most important methods for optimization of nature, such as drawing grass, trees, and stones, which can be the main reason for the bottleneck on the GPU side. In our case, each rendering segment will be different, but they all use the same shader instead of GameObjects with rendering components.

Draw Call Batching combines meshes to reduce draw calls. This can also be done manually using the custom editor tool using Mesh.CombineMeshes. Objects that never move in a scene can be combined into clusters for easy dispatch. This method should be balanced with culling techniques, as combined objects can be considered as one single object. Thus, a whole cluster needs to be visible or occluded at the same time. This method is sadly not as relevant as we are splitting the mesh into sectors to make computation and culling management easier.

SRP Batching should be usable in our context as we are using URP, a scriptable rendering pipeline. However, it is harder to implement directly as we are making the draw calls directly.

3.2.6 Level of Detail

Adjust complexity of generated model sector based on the distance from the camera.

Various factors affect the performance of the 3D application. Some of them are factors such as draw calls, shader complexity, and mesh complexity. We already mentioned that our parallel dispatch of marching cubes is scaling with the cubic complexity task and memory-wise. Reconstructing the environment with high accuracy produces a mesh with potentially the same cubic complexity.

As objects are farther away, our perception of detail diminishes in real life. When using computer screens or AR/VR headsets, this detail is limited by the pixel density. The density of pixels will increase as the computing power of the GPUs increases. At some point, we will reach the limits of our detail that are perceptible to the human eye. Similarly to FPS, it is hard to

determine what the upper limit for a common human truly is, but it is clear that in both cases a diminishing return occurs. The difference between 30, 60, 120, and 180 Hz displays is visible to users. However, the impact and notability are reduced as the FPS increases. This suggests that we want to find the sweet spot between detail and FPS to reduce waste [22].

When making games or simulations, game developers use various tricks to increase application performance by reducing detail without a significant impact on user experience. One of the relevant methods for our research is the level of detail.

The concept behind the LoD technique can be described in the following way: the farther away a 3D object is from the camera, the less complex the model is shown [69]. With pre-generated meshes, it often implies multiple versions of a single 3D model with different amounts of triangles. When a camera is close to the object, the original model can be viewed. For long distances, a model with a significantly reduced number of vertices can be used.

When reconstructing and rendering our surroundings, our data are categorized into sectors. For further optimization, each sector can be optimized. Provided that we can increase the computing performance by displaying less detailed versions of the sector at longer distances, we need approaches that help us adjust the computations and fidelity with the distance. Some of the possible solutions can be as follows:

1: Decimate
Simplify the model per-vertex as a post-processing step of model reconstruction.

The decimate is the operation of complexity reduction of a 3D model. Decimate can be commonly used by graphics designers to produce simplified models from their sculpts or for LoD variants of their 3D models.

Positives: Possibility of high detail preservation while reducing unnecessary model complexity that will affect program performance. Scales well to the designated goal model detail level. Works with any mesh reconstruction method.

Drawbacks: Computationally expensive, which may scale very badly. Decimate approach would need to be executed for each major mesh update. Requires a new mesh buffer.

2: Weight Matrix Division
Use only every N-th weight while reconstructing mesh via Marching Cubes. The whole matrix of marching cubes is constructed from weights.

Positives: The very cheap operation, as it is based on ' No memory cost. Easy to implement.

Drawbacks: Amount of cubes in dimensions must be in even and odd numbers. Reduction needs to follow this pattern. Mesh holes can occur very easily.

3: Weight Matrix Condensation
Method: Convert high-fidelity weight-map to a weight map of lower fidelity.

Positives: Lesser chance for mesh holes than in the previous method. We preserve the cached data.

Drawbacks: Unreliable weight quality upon conversion. Minor memory requirement increase.

4. Flush
Do not use a high-fidelity weight map, use new point data to generate a new weight map that is specifically made for a new cube size.

Positives: Reliable and predictable weight quality.

Drawbacks: Requires re-scanning of changed LoD areas. Possible memory requirement increases.

3.2.7 Impostor Objects

To reduce the rendering overhead caused by 3D objects, we can use a method commonly called 'impostor objects'. Impostor algorithms often generate one or more pictures of our 3D object with an alpha layer. These billboards rotate in the camera direction, and their texture can change based on the camera position. When an object is very far away, it can be culled in a way similar to the concept of LoD [20].

This technique gives developers and users the ability to improve performance even without affecting the perceived visual fidelity of the environment. Significant reduction in shader, light, and postprocessing effects on a 3D object can significantly improve program performance.

This technique for optimization could be very efficient in rendering complex holograms at longer distances. It will not interfere with occlusion meshes, as they behave like flat 3D objects (quads or planes) rotating towards the camera either by script or by shader. The question is whether this technique can be applied to occlusion meshes. The billboard nature of impostors does not grant the accuracy we might desire even on long distances, as well as taking pictures requires caching. Thus, this option would only be considerable for very long distances. More practical usage would be to directly generate a very simplified version of a mesh with very low number of vertices and simplified shaders, which may work well for model-based occlusion.

3.2.8 Other Methods to Consider

C#: Unsafe Code

Modern languages such as C#, Python, and Java help us with the feature that we know as managed memory. These languages allocate objects and variables mainly in the heap. Managed memory in C# like in many other imply usage of Garbage Collector. When an object or variable drops all existing references to it, it is detected by the GC and is disposed of. This helps developers to spare time related to memory management and potential issues that it can cause; just like segmentation time. However, as we know, stack memory is better for consistent allocations in terms of performance. Furthermore, the GC operation is not computationally free. It leads to overhead that we can cut by managing memory on our own hand, just like in C or C++ languages that can be significantly faster than C#, Python and Java. The unsafe code [38] of C# allows us to use features such as pointers, stack allocation, and unmanaged memory. It can improve performance of memory operations; however, it leads to memory-related issues related to memory like segmentation faults that cannot be detected by the common language runtime.

Unsafe code can be a good option to improve our time- and memory-demanding systems without using the C++ or C languages. However, it fits best for an eventual late optimization, as it requires a significant increase in work compared to potential performance gains. Therefore, we conclude that it is an important thing to consider in future work.

Engine: Data-Oriented Programming

Data-Oriented Technology Stack (DOTS) is one of the newer additions to the Unity Engine that is still in development. It is a different writing paradigm that uses multiples of unity libraries to achieve high logic-data separation for efficient multithreading and other potential optimization methods.

1. **ECS (Entity-Component-System):** Unity was designed mainly for an object-oriented approach using component-based design. Each GameObject contains components that define logic. Scripts are updated using different update functions and event calls. The high mutual dependencies of the components make them vulnerable to interference, making it harder to distribute across multiple cores automatically.

ECS approaches this problem with a data-oriented design. Data are highly aligned memorywise, as it is preferred to use blittable types, and are to a high degree independent. It makes the job perfect for distributing computational work across multiple cores, as each 'component' can theoretically be queued for computing on a different core [65, 75]. Following the data structuring of ECS can also make netcode development overall better, as the data are highly separated from the logic. Each in-game object (**entity**) contains **components** that contain the necessary data. Therefore, **systems** are designed to handle logic from all existing components

2. **Job System:** As the Unity Engine runs mainly on a single thread, the job system is a library that helps to introduce multithreading safely and simply without worrying about memory- and order-related hazards, such as race conditions[77]. Some of the Unity systems use the job system to improve performance. Note that this system relies on pararellity rather than concurrency.

3. **Burst Compiler:** Flexibility of the cross-platform language and JIT compilation such as C#, Java, or Python does not necessarily result in the highest possible performance on all possible target platforms. Burst is a compiler that translates Unity platform-independent code (C# and.NET) into native code for a given platform. It can greatly optimize the tasks of the job system, as it is a math and platform-aware compiler [77].

4. **Hybrid Renderer:** Is a rendering data collection system that sends the necessary data for rendering to the scriptable rendering pipeline used in the project [64]. The system is in active development, but has the potential to be slightly faster than GameObject rendering. Determining how fast this renderer is in various cases would require extensive testing. However, Unity developers tend to report an increase in rendering performance for the second version of the hybrid renderer.

DOTS results in a significant performance increase assuming that we have many components that systems can distribute across multiple cores for faster computation. Our current system does not contain a significant amount of entities that can benefit highly from multithreading. Developing our solution in a data-oriented paradigm can make it less efficient than it is currently.

ECS paradigm could yield positive results in the future potential branch of AR, where objects visible in the user's vision would be detected by deep learning AI. Meshes could be split based on interpreted objects and rendered/assumed as separate objects, or used for generation of colliders / trigger boxes for enhanced interaction capabilities. Since such AI operations tend to be heavy and somewhat brute-forced, it is questionable if such a solution would run on user computers anytime soon.

However, it is clear that some CPU computing tasks can be optimized through job system and burst compilers, but it can also be achieved with traditional concurrency methods that exist not only in C#, but also in many JiT or AoT languages such as Java, Go, Rust and C++.

Hybrid GameObject-ECS implementation is possible. Heavy physics-based games that involve the calculation of dozens, hundreds, or thousands of objects will greatly benefit from DOTS because of the multithread job distribution. If the developed AR solution is used in practice, developers can use DOTS to achieve higher performance for their game logic.

Part II

Conclusion

Chapter 4

Results

4.1 Evaluation

4.1.1 Testing Hardware

In the development and presentation of the testing results of the developed software, I have used the following computer hardware:

```
Operating System: Windows 10 Pro 64-bit
Processor: Intel(R) Core(TM) i7-8086K CPU
Motherboard: Asus Prime Z370-A, Rev X.0x
Memory: Corsair DDR4 3200 Mhz, 16 GBytes, dual channel
Drive: Samsung SSD 980 Pro (1TB), M.2 PCIe3

Graphics card: MSI NVIDIA GeForce GTX 1080
Display Memory: 16238 MB
Dedicated Memory: 8079 MB
Shared Memory: 8159 MB

Current Mode: 2560 x 1440 (32 bit) (144Hz)
Native Mode: 2560 x 1440(p) (59.951Hz)
```

4.1.2 Profiling and Quantitative Analysis: Approach

The main tool used for testing is the Unity Profiler and Profile Analyzer tools. It gives us the best insight into resource usage, and some of the solutions developed will be sampled by a self-developed profiling system developed to probe them correctly. We need to consider a few aspects while developing our own solution and using Unity Profiler:

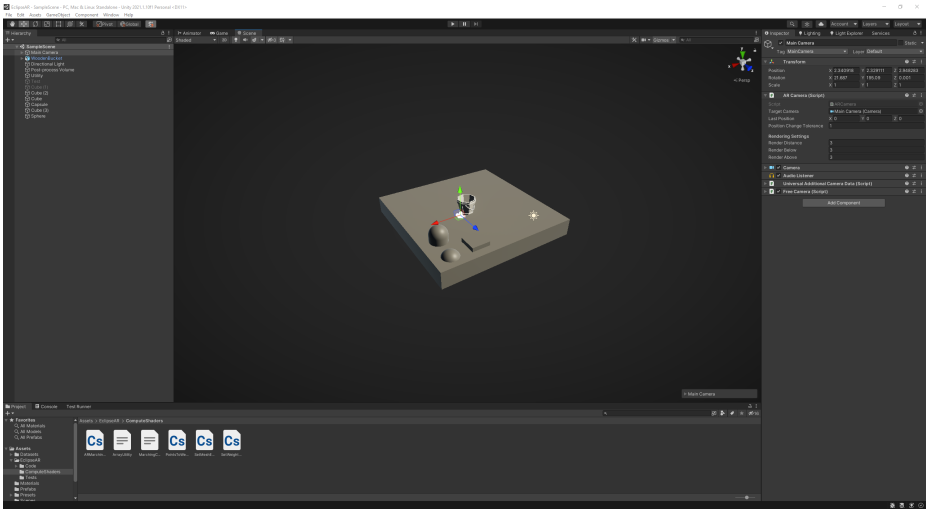

Figure 4.1: Unity project window of the solution showing the test scene. Hierarchy of game objects on the left.

1. **Warm up.** .NET Framework and C# is as mentioned before, a JiT environment that compiles code as the application runs through the mono interpreter. The code tends to be cached and ready at the start; however, this process can be unpredictable for us. Debugging should be performed a few seconds after starting up. Probed data must be discarded, as they can be unreliable due to the load caused by the initial initial drive, RAM, CPU, and GPU usage when the application instance is initialized. In custom profiler tests, the warm-up interval is set to 30 seconds.

2. **Reduce impact of testing environment.** With algorithm, data structure, Unity experience and Rider IDE we are capable of developing profiling code that minimizes its impact on the results. We apply similar rules to develop the test code. Consider the following examples:

   (a) **Allocate lists to target probe size.** Arrays in C# are of set size. The lists are of dynamic size. Dynamic resizing of the list is a costly operation that requires the allocation of a new memory chunk for a larger internal list array. As the number of probe data increases, this can lead to inaccuracies. All data structures should be static and allocated before tests are conducted.

   (b) **Avoid expensive operations like constant dictionary look-ups (especially on types with high hash calculation time), using event systems or comparing Unity objects to null if possible.** Even if some of these operations are relatively inexpensive, we can try to reduce their usage. Operations with high time complexity should be avoided.

   (c) **Comparing Unity objects to null is expensive.** C# uses managed memory by default. If an object loses all references to it, it is considered an object to be disposed of. The garbage

collector detects it, and the memory connected to the object is freed. Constant GC operations can be very expensive in themselves due to the automated nature of the process. Unity uses a modified managed memory approach. Unity objects can be destroyed, and when they are destroyed, all existing pointers to the object and all its subcomponents are invalidated (they become null). This leads to higher cost of compare operations, as the memory location reference object/s may not be existing anymore.

3. **Large samples, different times.** The more samples, the higher chance we can reduce false premises, reducing the impact of potential noise like background processing. Sampling of different processes, from multiple perspectives is wise, as it can affect the results. Tests should be avoided to be run together in parallel, for this we will use single comparison to a flag.

4. **Both single camera and dual camera performance should be considered.** Target devices can be different; video-based or optical-based. The number of views is an important factor for the final speed and the application. It is good to observe how it scales in our game engine and the rendering pipeline.

For research purposes, different parts of the developed solution will be tested using the profiling system. This includes fine- and coarse-grained operations. The tests will also include information and an explanation of the results and potential background mechanics. This can help us understand the complexity and costs behind the operations, which are essential parts for any person who desires to develop their own AR or simply wants to learn more about GPU compute shaders and performance for high-performance demands.

Different algorithms tend to run different amounts of times. A high probing rate is recommended for improved accuracy of the results; therefore, depending on the algorithm, we will collect between 1,000 and 1,000,000 measurements. Due to the slow trigger rate of some algorithms, some will require testing in multiple series. The measurements collected are described in ticks or milliseconds (ms). A tick is the lowest possible time unit that we can measure; **1 microsecond is adequate to 1,000,000 ticks.** According to my measurements the standard measurement cost is as follows:

Time for Start() and Stop() of the developed timer:

Average: 1.941 ticks, Mode: 2 ticks, Min: 0 ticks, Max: 23 ticks

Time for *Unity Debug.Log():*

Average: 1555 ticks, Mode: 1520 ticks, Min: 807 ticks, Max: 9637 ticks

The time cost for measurement **will not** be subtracted from the custom profiler tests. Profile analyzer tests exclude Unity Editor impacts. Calling Unity debug function is avoided to minimize performance impacts.

Due to limited access to AR headset hardware and correct scanners and head tracking sensors, a test must be carried out in **controlled environment, inside Unity Editor**. Real-world / virtual-world alignment is not part of this thesis and is a large and time-consuming task. The components required can be considered expensive and are not required for the scope of this investigation.

However, Unity Editor resting for AR performance and accuracy will still give us excellent results to evaluate very efficiently when compared to practice testing. With a good alignment solution and further calibration, the solution should be easy to apply in AR headset like Project North Star or smartphone devices, but it would require development of concurrent components specialized in reading point cloud stream from an IR/LiDAR sensor.

4.1.3 Occlusion Reconstruction Prototype

In this thesis, I have developed a prototype solution system to answer our research question and try to accomplish proposed objectives. GPU development versions were versioned on GitHub. The development of the occlusion solution was carried out in three stages.

1. Prototype of Marching Cubes on the CPU on the main thread. At this stage, reconstruction, even with an accuracy around 4 cm, is not running in real time. Sending the model from CPU to GPU results in an unnecessary penalty. Generating and adding mesh took several seconds for small volumes, and thus development was focused on the GPU-computed solution.
2. Reconstruction prototype on the GPU, with compute shaders and LoD features being tested on. It implied a less performant solution where the data were moving back to the CPU to be moved to the GPU again.
3. Working prototype of higher fidelity with a wide set of features that can serve as a proof-of-concept and a basis for future research and development.

The current solution includes, among others:

1. Prototype components for scanning the environment. In this case, colliders of the placed models on the scene imitate IR/LiDAR hardware.

2. Point cloud to marching cubes translation shader for sending point data directly to GPU for weight updates.

3. Interpolated marching cubes algorithm for translating weights to mesh.

4. Unlit test shader for prototype visualization purposes (used in screenshots). Unlit flat black shader for AR rendering purposes. In case of shadow support, lit shader must be used instead.

5. Saving of unused sectors in RAM for storage as an asynchronous operation. Sectors can be loaded from RAM into GPU again.

6. Sector system that allows rendering only to a specific range and within sight. LoD system can render reconstruction with various accuracy to allow users of the system to adjust its performance to the device and the desired FPS target.

7. Multiple AR camera support, in the case of multiple users or video sources.

8. A prototype of "melting" and ray penetration of a prototype raycast depth sensor.

The following figures demonstrate the setup of the main testing scene and the basic function.

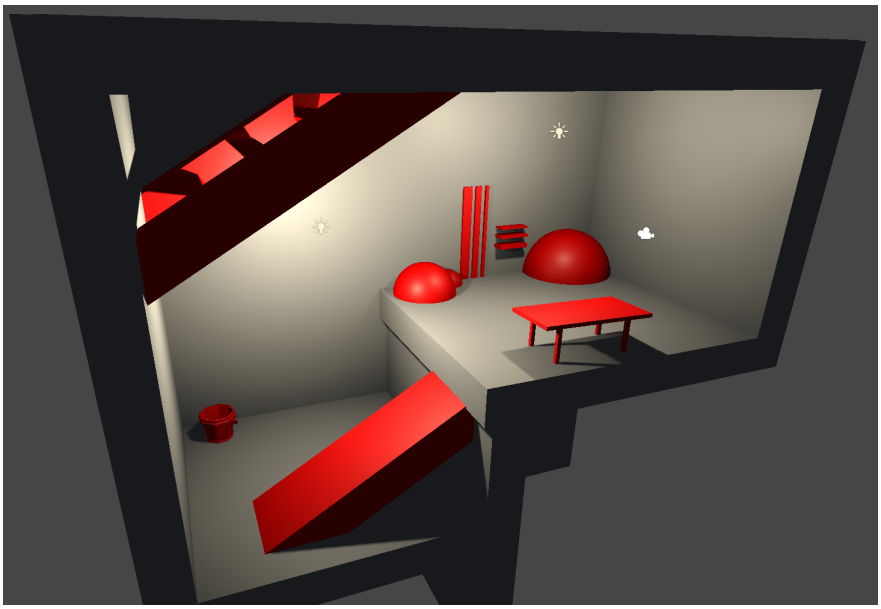

Figure 4.2: One of the prototype scenes in the Unity Editor. This test does not provide perfect coverage; however, it represents different surfaces of various thicknesses and angles to present differences in reconstruction speeds and accuracy. Objects of interest are colored red.

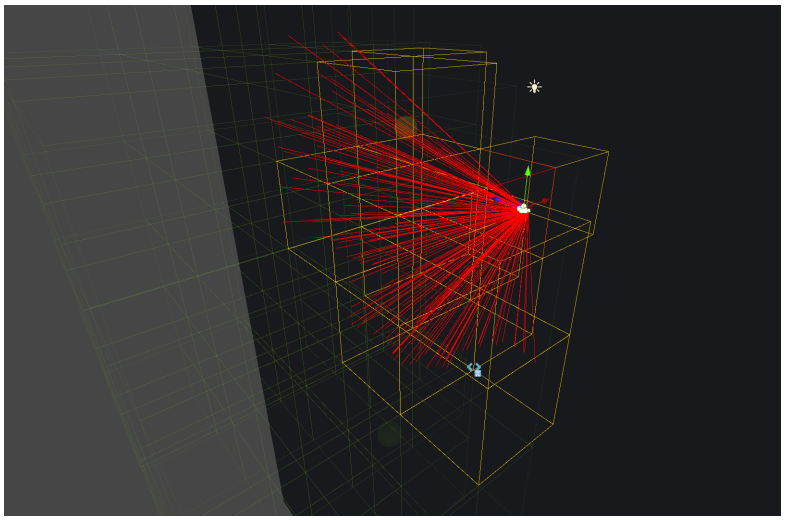

Figure 4.3: Camera scanning the environment with rays and returning point hit position imitating the behavior of scanning devices. Camera/s activate sectors of the world to be represented with different accuracy levels if desired. The highest quality sector is set to red, the middle quality is yellow, and the lowest quality is green.

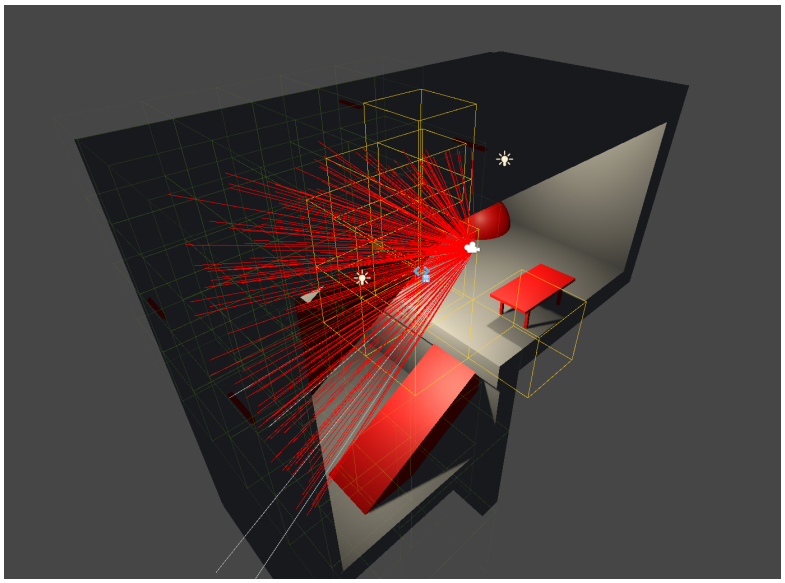

Figure 4.4: Camera scanning the environment with rays, additional perspective.

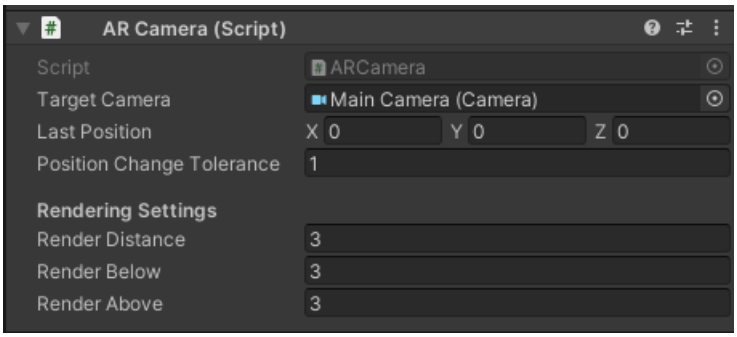

Figure 4.5: AR camera component is necessary to track camera position and load necessary sectors in cameras frustum.

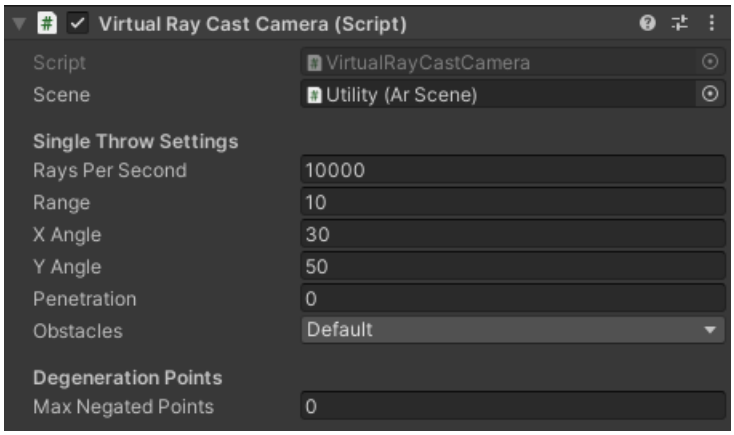

Figure 4.6: AR camera component for simple simulation of point-cloud collection by using pre-existing object models, colliders, and Unity raycasts.

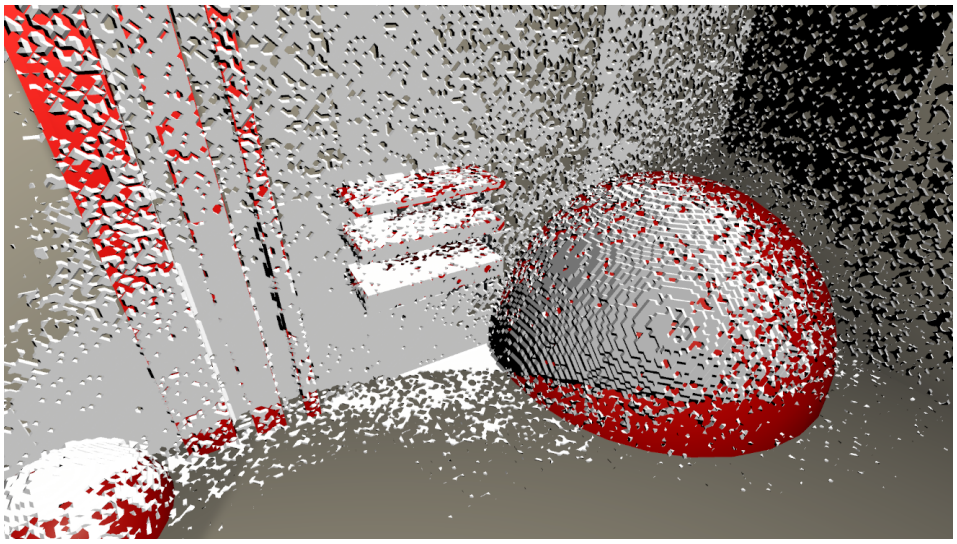

Figure 4.7: Reconstruction of the environment on the test scene. Some areas have less cover as the rays are not hitting them. 2 cm accuracy, 10,000 rays per second. Note that the triangles shown are not affected by the lightning system, but have colors assigned based on their angle for visualization purposes. Dark gray and red are testing environment objects that are scanned to generate a point cloud.

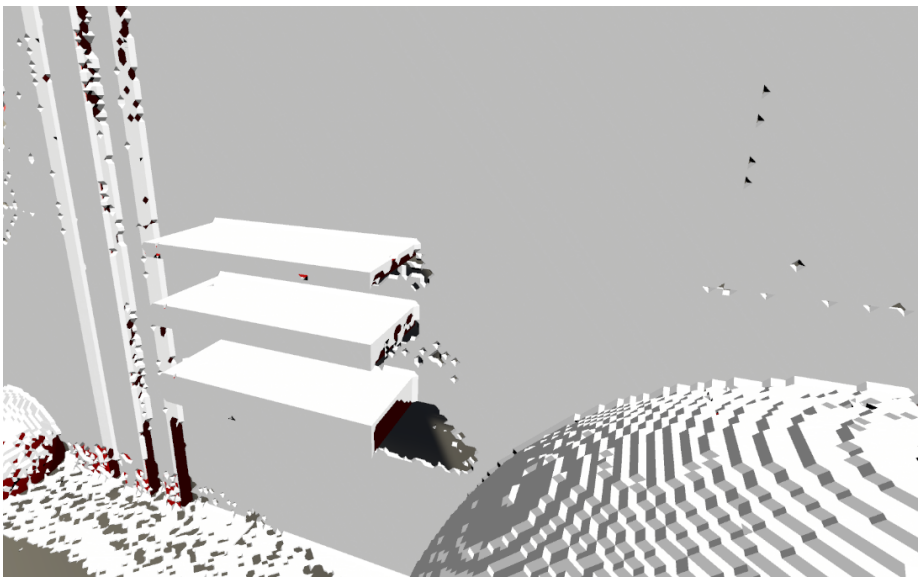

Figure 4.8: Reconstruction of the environment on the test scene. After moving the head, some areas not covered by rays need to be slowly covered. This is normal behavior since the rays only hit surfaces within the line of sight of the sensor. 2 cm accuracy, 10,000 rays per second.

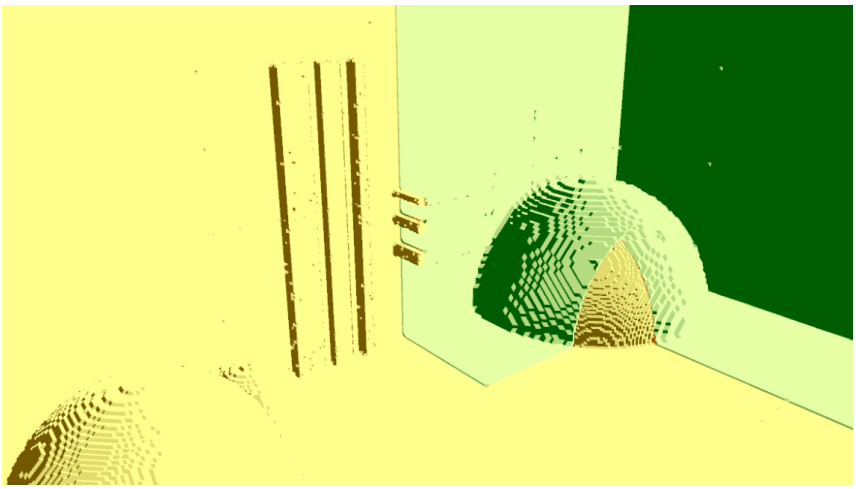

Figure 4.9: Mesh generated for occlusion, example with LoD levels enabled. Colored yellow area 0.2 cm cubes and green area 0.4 cm cubes.

4.1.4 Profiling and Analysis

First test was performed in the position demonstrated in Figures 4.10 and 4.11. For this test, I'm using the following settings as main reference:

1. **Sector size:** 2 square meters
2. **Cube size:** 0.02 meters (2 cm)
3. **Memory resize tolerance:** 80%
4. **Initial minimal mesh memory buffer size:** 0.4%
5. **Memory resize multiplier:** 2x
6. **Cache on RAM:** False
7. **Camera range (in sectors):** 3 forwards, 2 upwards, 1 downwards
8. **Camera count:** 2
9. **IR rays per second:** 10,000
10. **Rendering resolution (single camera):** 1920 x 1080

Profiling results show an increase in memory over time, indicating that the memory and grouping systems work as intended. When the GPU mesh memory for the sector is freshly allocated, it is allocated only to the specific percentage of pessimistic memory amount. The correct configuration is necessary to balance between avoiding vertex artifacts and wasting GPU memory. Constant exchange of the GPU mesh buffers might also lead to CPU/GPU penalty.

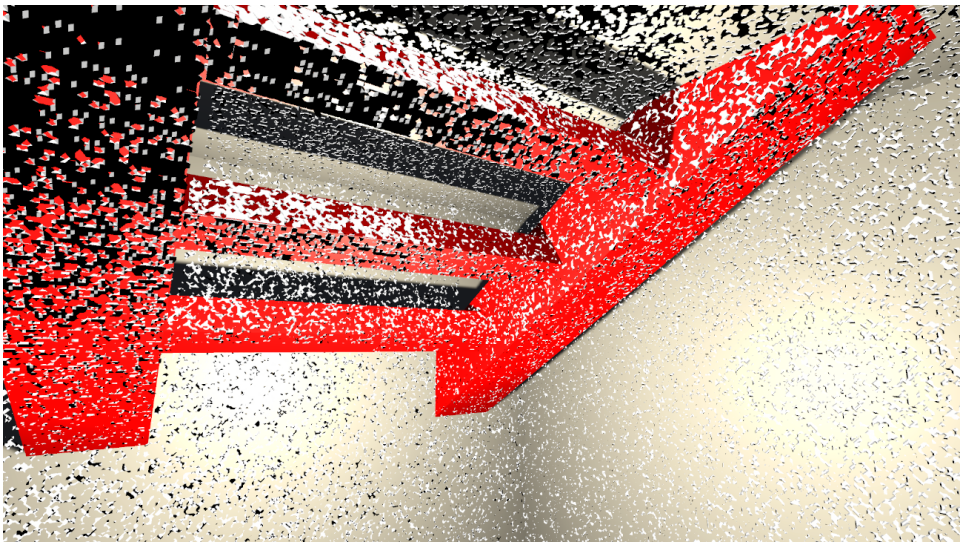

Figure 4.10: Results of the reconstruction of an uneven surface with an accuracy of 2 cm after 20 seconds. No head movements, two active rendering cameras. 10,000 rays per second.

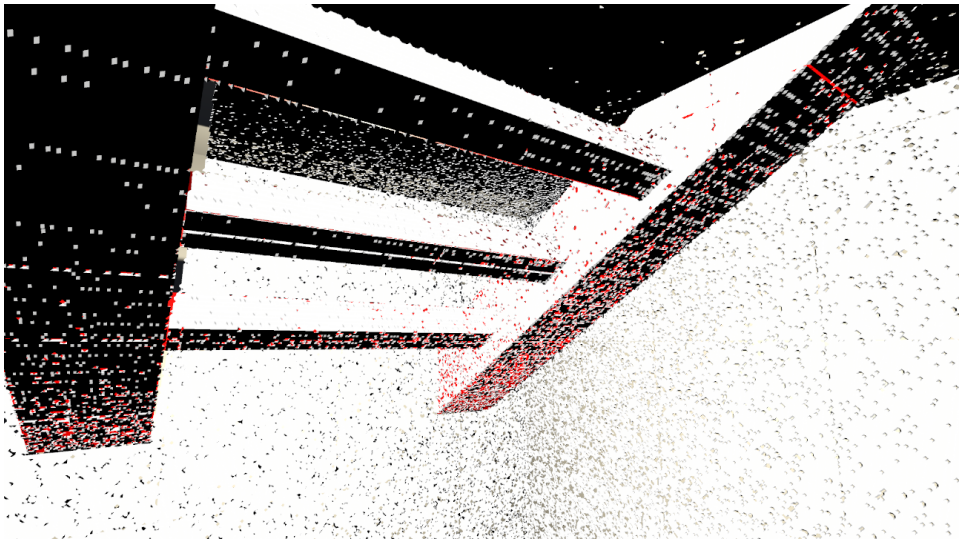

Figure 4.11: Reconstruction of an uneven surface with an accuracy of 2 cm after 120 seconds. No head movements, two active rendering cameras. 10,000 rays per second. Surface coverage is significantly better. Due to distance and ray spread, lack of head movements, covering of the whole area takes much more time.

Setting the initial memory reservation value too low results in artifacts (see 4.12). In such a scenario, some triangles are visibly corrupted. This is caused by the buffer overflow and the GPU starting to write vertex data from the beginning. There errors only occur when the mesh complexity increased significantly in a single update, beyond the 100% of buffer size

and thus the **mesh resize tolerance** threshold. In the common case where the triangle limit exceeds a specific value, the compute buffer is swapped with a larger one. When the buffer is resized, its new size is defined by **memory resize multiplier**. With current settings, the buffer size is doubled. If a pool of given size is existing and contains unused compute buffer; it is preferred over allocating a new one. Pool is discarding if there is too many buffers of given size.

By evaluating 600 frames after 120 seconds of the demonstrated camera setting (see 4.13) and the early 600 frames, we can better observe the system mechanics.

Considering the early 600 frames we can better observe how RAM/VRAM memory is warmed. Profiler shows a similar count of batches and growing/jumping amount of triangles that can be caused by how we generate and render our mesh. The beginning shows allocated 390.8 MB VRAM and 1.42 GB RAM usage. The frames at the end show allocated 600.0 MB VRAM and 2.14 GB RAM usage. This indicates an expected upward growth as we scan the area. The growth in frame time is not significant in the scope of 600 frames. The CPU timings during the duration are: 11.88 ms mean, 23.37 max, and 9.43 min.

The late 600 frames show stable RAM and VRAM memory. GC uses stable **235.9 MB** memory usage. A total of **2.18 GB** RAM memory is used. The profiler also indicates many GC.Allocations (based on the sector update count, even taking as much as **up to 0.02 ms each**) related to constant allocation on the heap. To minimize expensive GC operations, it is best to allocate dedicated static memory and perform a copy operation instead. Since C# classes are allocated in heap memory, their children are also allocated on the heap. However, C# also allows us to use the *unsafe* code [38]. In an unsafe context, the keyword *allocstack* [37] can be used to allocate referenced types such as arrays on the stack instead of the heap. Unsafe context and unmanaged memory is a great option for future optimization. It is worth noting that it requires significantly more development time and additional testing, as it can cause memory management issues such as memory leaks.

The highest noticeable timings (including editor) are at **21.42 ms** for the CPU and **18.39 ms** for the GPU, with the GPU being slower in some cases, suggesting CPU and GPU-get or dispatch bottlenecking. Most of the CPU time is related to **Semaphore.WaitForSignal()** wait states caused by *Gfx.GetComputeBufferData_Request()* operation of *ARScene.Update()*. The *ARScene.Update()* function takes **mean 8.75 ms to finish**, which is related to various CPU sector operations and GPU compute shader dispatches that are awaiting completion. The handling of Update() function of the ray component takes **mean time of 0.40 ms at 10000 rays per second**.

The rendering part of the URP takes between a mean time of 2.01 ms. The

profiler indicates **223 batches**, with **614.2 thousands triangles rendered**. The GPU memory used by the buffer is **0.81 GB**. The *ARScene.Update()* function takes up to **0.279 ms** of the frame time when the ray data need to be dispatched. The main camera takes **1.04 ms to render 84 draw calls**, while the secondary camera uses **0.908 ms to render 78 draw calls**, indicating that the setup is working correctly. The mean frame time is 13.49 ms and therefore **74 FPS** in the setup presented with two active full HD cameras. Min frame time is 9.97 ms and max frame time is 21.42 ms. This presents growth over the early frames, as model complexity builds up.

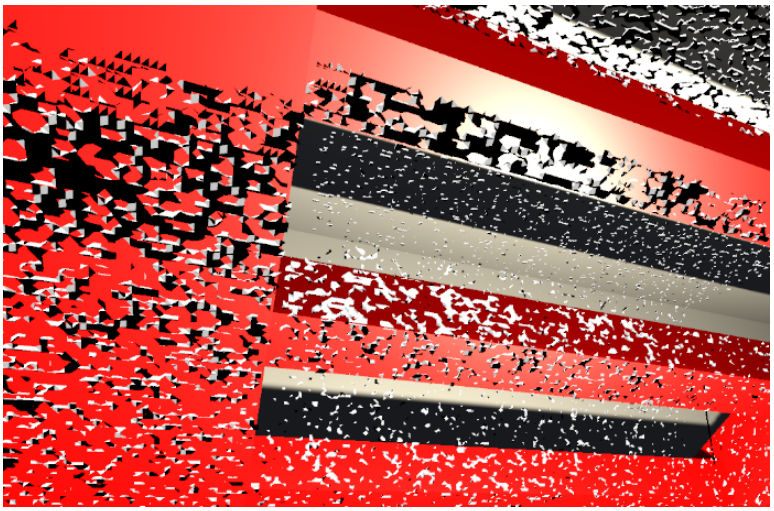

Figure 4.12: Graphical artifacts caused by compute buffer overflow within a single update when using too conservative memory-saving settings. **Left upper corner:** area of the corrupted triangles. **Right lower corner:** stretched corrupted triangle.

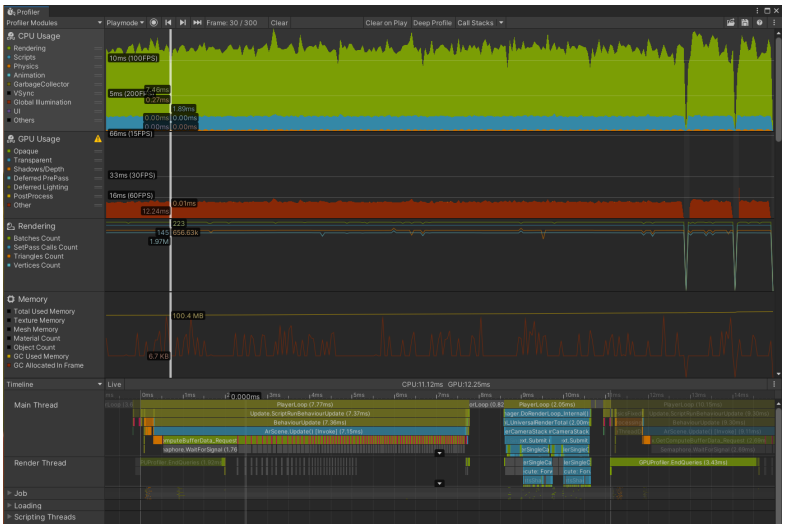

Figure 4.13: Profiling tool included in Unity Editor. Shows us the details such as various timings and resource usage. Note that profiling GPU usage can lead to additional overhead.

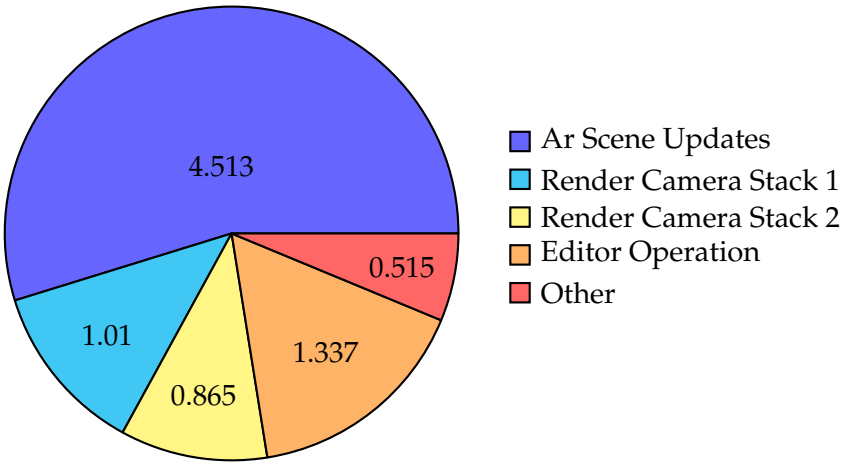

Figure 4.14: Pie chart representing CPU-time usage by important processes (ms). AR Scene Main Update is responsible for sector loading, camera visibility, and render dispatching. The Late Update of AR Scene updates weights and marches with a cost as low as 0.013 ms. 4 cubic meters.

4.1.5 Quantitative Analysis

Sector Size and Performance

Sector size is an important parameter to balance. It affects the size of the GPU dispatches and the number of draw calls. Lower sizes result in more draw calls, but smaller GPU dispatches and a better time distribution. Larger sizes result in fewer calls to the draw. This part proved to be difficult

to test reliably, as changes in sector size require potential adjustment in range and camera sector loading tolerance. Furthermore, the distance is defined by the number of sectors that are drawn forward by the camera. Note that the draw distance data had to be adjusted accordingly to increase the reliability of the results. From the profiling data shown in Table 4.1 we can conclude that a larger sector size is better suited to cover larger volumes, as it has the best coverage-to-cost relationship. However, increasing the size can lead to bottleneck caused by GPU computation limits, thread count limit, and rendering costs. Therefore, these settings must be balanced in the AR play area and the desired LoD settings for longer distances.

The farthest we can go is defined by the memory limit and computational power we have at hand. With $8m^3$ and a short reconstruction and rendering range, we are able to achieve high but very unstable frame rates at 1 cm cube size, 0.4% initial memory reservation, and 4 sectors in total. The profiler reports values around 6.51 ms for the CPU and 6.34 ms for the GPU most of the frames. Occasional spikes note increase of GPU time to values as high as 46.46 ms and 44.28 ms for CPU. It is caused by hitting memory limits; 8.54 GB of memory are allocated to buffers only. This means that the allocated memory for the weight buffers of the meshes and marching cubes weight buffers exceeds 8GB of dedicated GPU memory, leading to the usage of slower shared video memory (virtual memory allocated to RAM, which requires PCIe data movement [16]). In this case, the shared video memory is 8GB, half of the total system RAM.

Size	Sectors	CPU Time	GPU Time	RAM (Total)	VRAM (Buffers)
1 m^3	5260	424.13 ms	415.06 ms	5.75 GB	4.03 GB
2 m^3	631	59.89 ms	59.15 ms	4.82 GB	4.02 GB
4 m^3	78	15.19 ms	14.85 ms	5.73 GB	4.89 GB
8 m^3	10	8.17 ms	8.05 ms	4.84 GB	4.08 GB

Table 4.1: Profiling data of different sector sizes. Due to view frustum, sectors are set to a setting that will yield very similar area coverage. Momentary data collected from the 1000 frame approximate after the beginning of the application. As the sector size increases, the higher the GPU timings we can observe. The initial memory allocation was set to 0.8% for all cases. Ray buffer sizes are not adjusted to chunk size in this chart, which with growth of Sector structure allocation results in visible RAM growth. Fluctuations in memory are caused by varying ray-area coverage in a given sector-size setting.

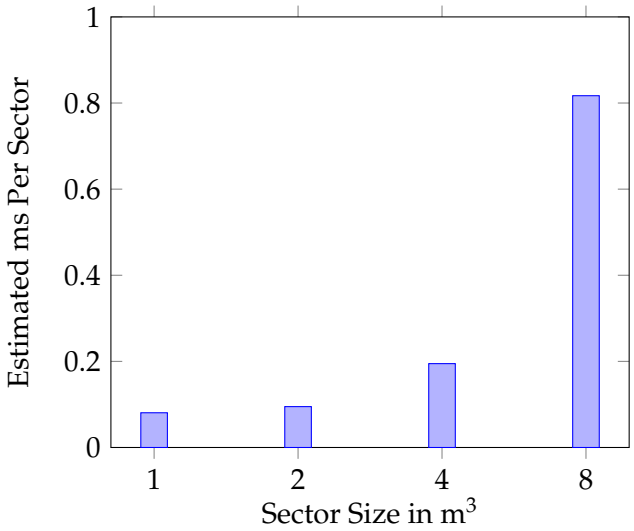

Figure 4.15: Approximate cost of sector computation based on Table 4.1. $8m^3$ sector can fit 8 of $4m^3$ sectors inside, therefore optimistic expectation is 8x reduction of costs for each size. However we can observe significant diminishing returns as $4m^3$ sector is 23.84% of the $8m^3$ sector cost. Which means that we receive less than 5x of the computation cost. This is even more magnified when using chunk size of $1m^3$ and $2m^3$. In this case $1m^3$ section is almost as expensive as single $2m^3$ sector.

Cube Size and Resources

Another important part to evaluate is the performance and resource usage based on the cube size. As we can see in Table 4.2, the VRAM usage increases by 6-7 times. Variation in memory is related to frame times, incoming rays, mesh generation, and dynamic mesh allocation. At 1 cm, the dedicated VRAM is entirely used up, which results in large RAM reservations for shared video memory. This results in undesired frame-time spikes up to 107.82 ms. To benefit from smaller cube sizes, a LoD or shorter loading range needs to be used. The following results present a significant drawback of marching cubes. Lower cube size increase the complexity of the model. However, increasing complexity does not guarantee the best relationship between accuracy and memory usage. For instance: some flat surfaces might not require thousands or millions of triangles and could be represented by only a few triangles and look the same.

Ray Count And Performance

Modern point-cloud scanning devices can reach speeds of millions of points generated per second. Profiling the ray-cast component will help

Cube Size	Mean Frame Time	Max Frame Time	RAM	VRAM
1 cm	16.27 ms	107.82 ms	18.32 GB	17.57 GB
2 cm	12.46 ms	20.93 ms	3.88 GB	3.16 GB
4 cm	10.79 ms	20.84 ms	1.23 GB	461.3 MB
8 cm	9.86 ms	20.70 ms	0.86 GB	75 MB

Table 4.2: Performance and usage of resources of different cube sizes. 60 active sections, 4 m^3 section size, 0.4% initial minimal memory buffer size. 600 frames sample using the Profile Analyzer tool. Memory picked from 1000th frame form beginning.

us observe the cost of ray-cast operations and potential cost of sending rays to GPU. As we can observe in Table 4.3, performance scales nearly linearly with the number of rays. However, subtracting the update times from the frame times we can conclude that the *Update()* operation and the ray-cast physics operations are the main part affecting the performance significantly. For 1 million rays per second, the profiler reports 37.13 ms, which is used for conducting the ray-cast operation part of the update function, which takes 99.82 ms in total. The performance cost of processing more points with a GPU is not significant when compared to ray-casting costs. With an improved setup and a LiDAR/IR scanner, managing points in separate threads can prove to be significantly better performance.

Ray-Casts Per Second	Update Time	Frame Time
1.000	0.022 ms	11.70 ms
10.000	0.149 ms	12.64 ms
100.000	1.38 ms	16.93 ms
1.000.000	99.82 ms	118.72 ms

Table 4.3: Raycast profile data for prototype components per second and how it affects performance. Each sector has its own point queue; in this test, its capacity is equal to the cast count of the rays. The point-cloud RAM buffer is also set to 1 million points; it is used for GPU dispatch. This ensures optimal performance to keep up with the generated point cloud.

Quantitative Performance Tests

In addition to profiler measurements, more quantitative measurements have been collected in some parts of the systems, as previously suggested. The count of measurements depends on how often given functions are used during runtime.

Measured Functionality	Samples	Average (ticks)	Min (t.)	Max (t.)
Apply Weights (1024 thr.)	1000	2497.69	1624	3441
March Cubes (512 thr.)	1000	62.969	22	643
Get Mesh Buffer	100	139.68	31	2987
Pool Mesh Buffer	100	34.4	4	160
Clear Mesh Buffer (128 thr.)	100	18.79	5	38
Pool Weights Buffer	100	91.41	9	2834
Get Weights Buffer	100	55.69	1	324
Clear Weights Buffer	100	42.25	5	75
Cached Sect. RAM to GPU	100	631.56	3	9346
Verify Sector (caching on)	10000	109.494	3	333318
Camera Cal. Sect. (moving)	100	40452.59	19642	161902
Draw Call Sector	10000	87.20	70	2362
Draw Call Get Count	10000	1149	412	95749

Table 4.4: Measurements of different parts of the system in ticks. Per execution of the operation. Movement of the camera in the scene. RAM caching enabled.

According to the results, getting the count from the compute buffer leads to a cost as high as 1149 ticks, with a maximal measured time of 95,649 ticks. Although it is still below 1 ms, it is a high value for the operation that receives the count of triangles in the append buffer. It may be one of the most significant impacts on time costs per active sector seen in Figure 4.15. Counting triangles manually is an efficient alternative to consider. Storing the counters in a shared buffer could make retrieval more efficient as a single operation. Operations like: applying point cloud, evaluating sectors for update or unloading, calculating camera sectors, and recovering sections from RAM back to GPU are quite costly to execute. Some of the function timings are related to the size and count of sectors. The mentioned parts of the code could be further evaluated for additional late optimization possibilities.

Measured Functionality	Samples	Average (ticks)	Min (t.)	Max (t.)
Apply Weights (1024 threads)	1000	2521.164	2219	4627
March Cubes (512 threads)	1000	61.56	46	235
Verify Sector (caching off, static)	10000	4.0886	3	24
Camera Calculate Sectors (static)	100	78470.4	62180	99789
Draw Call Sector	10000	79	64	249
Draw Call Get Count	10000	1441.7813	412	36457

Table 4.5: Comparison of static perspective timings for static perspective in Figure 4.8.

Measured Functionality	Samples	Average (ticks)	Min (t.)	Max (t.)
Apply Weights (1024 threads)	1000	2466.052	1576	3671
March Cubes (512 threads)	1000	60.364	23	177
Verify Sector (caching off, static)	10000	4.1248	3	22
Camera Calculate Sectors (static)	100	99855.56	78080	125054
Draw Call Sector	10000	81.1228	66	532
Draw Call Get Count	10000	1235.52	412	36561

Table 4.6: Comparison of static perspective timings for static perspective in Figure 4.10. We see some variation in the functions, which depends on the camera position and the number of sections loaded.

4.1.6 Additional Findings

Compute Buffer Clearing Methods

Pooling weight buffers requires clearing its contents. To receive a clear buffer, we can do one of the following: set data, dispatch clearing compute-shader, or discard data and allocate new buffer. In Table 4.7 we can see that allocation is slightly more expensive than *Dipose* and shader-based clearing. This confirms that pooling and clearing with a compute shader should improve performance.

Operation	Average Ticks	Min Ticks	Max Ticks
Allocate	70	55	48201
Dispose	5	3	1853
Clear Shader (1024 threads)	4	3	3737

Table 4.7: Different compute buffer timings relevant for clearing. 10,000 samples.

Thread Count and Performance

Thread group count defines the number of GPU threads that can be dispatched in a single group. As we can observe in Table 4.8, thread count does not have a significant impact on the timings of a simple shader. However, with this simple clear compute shader, the number of data that we can clear depends on the number of threads (up to 1024) in a single group and the group count limit (up to 65535). Too low numbers may limit performance due to occupancy; meanwhile, a high number of threads can cause performance issues if we reach GPU resource limits. Therefore, the thread count per group should be placed between the minimal recommended (like 32 threads for Nvidia) and 1024 (max)

depending on the target hardware and size of the dataset. A total count of 512, 256 or 128 should be a reasonable starting point to avoid issues. Note that the total number of threads should be a multiple of 32.

Operation	Average Ticks	Min Ticks	Max Ticks
1024 (max)	5	3	1796
512	3	3	1717
256	6	3	1774
128	5	3	1843

Table 4.8: Clear data compute-shader performance using different thread count per group. 1,000 samples. Operation on 6144 floats.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

Optical-based and video-based AR technologies are still in significant development. As we discovered while conducting research for this thesis, computing power and sensor technologies had to improve significantly to allow for viable use of VR and AR. However, we still have a long way to go. The occlusion and management of the scene is one of the topics hardly mentioned in the scientific papers. Evaluating the research question with the knowledge collected and the data collected from our prototype:

Can we develop efficient real-time, mesh-based reconstruction method to occlude virtual objects with real object in an AR scenario?

We can answer the following: Yes, it is possible. It is possible using Unity Engine, but it requires efficient resource management and heavy optimization such as world sector management and pararellization using GPU. It is not efficient to use the CPU for computation of the high-accuracy model or Z-buffer operations. Modern GPUs are very efficient at parallelizable tasks, and we can continue to observe significant improvements over the years. Furthermore, since rendering the data happens on the GPU, being instructed by CPU; it is most efficient to keep related thata on GPU if possible to reduce necessary PCIe movement and memory waste. We observed that updating specific parts of the model can be challenging, as it would require structuring of the triangle buffers, which then makes rendering inefficient. Since model-based occlusion is resource-intensive, it is possible to mix it with a depth-based approach. One of the issues of scene management worth mentioning is smart detection of which sections and subsections to update and which section data are outdated and should be removed. At some point, ray updates will not yield any significant model reconstruction improvements, and queuing them for update begins to be a waste of CPU/GPU time and their memory. To avoid

ghosts and errors, an approach of removing data about a heavily changed area could yield a better occlusion experience for heavily cached mesh generation approaches. The literature on AR based on optical and video is quite limited, and the field would benefit greatly from further research.

Furthermore, we have achieved most of the objectives of the prototype and have successfully collected useful data that will help develop AR occlusion solutions in the game engines.

- *Solution runs in real-time.*
- *Solution is capable of reconstructing details even below 1 cm.*
- *Solution can run at frame rates above 60 FPS using both cameras, with accuracy as high as 2 or 1cm in Unity Editor.*
- *We did not register significant performance spikes.*
- *The solution should be an easily adaptable fundamentals for developers that wish to develop own mesh-based occlusion solutions.*
- *Solution uses Unity compute shaders, so it should work on ARM devices and consoles.*

5.2 Possible Improvements and Future Work

There are many ways to improve the accuracy and performance of our prototype in the future. Unmanaged memory management, concurrent point handling, better reconstruction algorithms, and improved sector management algorithms are great examples. Using CUDA and other API features for async and batched GPU dispatching could improve performance at the cost of hardware compatibility. When using the buffered model approach and model generation, we need to discard and regenerate some parts of the view as the surroundings change. This feature should be improved and requires additional research to improve performance and avoid ghost objects. Additionally, sub-section computation can be implemented for even smoother computation and work distribution. The concept has great potential with the ring-buffer approach using mesh compute buffers, especially with large sector sizes.

Like in many fields of computer vision and algorithms in general; there are a lot of settings to tweak to find the best spots between fidelity and performance. This is a very time-consuming and repetitive task for a single human. Therefore, to find a better solution for the desired range and fidelity setting, AI and deep learning could be used to adjust the other attributes to maximize performance.

There are no significant papers available that cover handing of the AR/VR scene, but it remains an significant part when dividing the scene into sectors for efficient processing and easier management. Furthermore, there is potential in areas such as Nvidia tensor cores and AI [45], to further accelerate mesh generation and handle data loading and discarding in a potentially more efficient way using the latest technology.

It should also be noted that most hardware capable of handling accurate mesh-based reconstruction is performance-oriented. We cannot achieve this efficiency with mobile devices and their ARM processors. Like high-fidelity VR; if we want high refresh rates and high-definition images, we need to use an x86 computer with a highly efficient graphics card to be able to render high-quality graphics for two eyes. This requires a cable connection or hardware that is carried on the user's back, which slightly affects the usability. Currently, we cannot efficiently fit high-performance hardware into AR/VR headsets. Therefore, we need to discover ways to perform these computations remotely with minimal latency (potentially using network and 5G wireless communication) and continue developing more energy- and space-efficient hardware. Wireless input and image transmission allow us to use higher performance hardware that is less mobile, opening new possibilities not only for AR/VR but also for fields such as telemedicine and telesurgery in which we need a latency lower than 1 ms [50]. Wireless headsets would also greatly benefit from efficient wireless power transmission. However, efficient and safe wireless power transmission technologies need to mature [78].

5.2.1 Accuracy Improvements

The point cloud translation compute shader assumes a loss of accuracy of up to 50% of the cube accuracy used. This means that a precision of 1 cm can result in a deviation of up to 0.5 cm. This deviation is reduced by configuring and improving the interpolation step of the algorithm. On the basis of the test, we can observe the steps. It is related to the concurrent writing and atomicity of the reading, writing operations that can lead to problems, as each point needs to take the value of the weight and interpolate it towards the calculated value based on points proximity to the weight position in the 3D space. Furthermore, interpolation can result in slower mesh weight modification; more rays need to hit a given area to augment the model closer to the desired form. This can lead to slower perceived model generation, but to a more accurate and visually smoother model. This limitation is related to the nature of Marching Cubes and might be easier to deal with using other approaches for mesh reconstruction.

Even if Marching Cubes is an old algorithm, we still find uses for it, and Grosso et al. [24] recently published in 2021 is one of the most potential methods to significantly improve the quality of marching cubes. It is an answer to some of the problems of the original algorithm. The dual-

marching cubes algorithm presented by the researchers was performed via CUDA, resulting in significant improved surface quality with great timings between 3 ms for smaller sets (MRI/CT scans, 384 x 512 x 80) and 92 ms (fMRI scan, 512 x 512 x 1047) for the largest sets presented. The algorithm results in nearly interactive computation times while still providing excellent quality. In addition, it uses pattern simplification, which can potentially improve the size and complexity of the topology. However, even such timings can be problematic for real-time mesh generation, as the algorithm can be visibly slower than the Marching Cubes; occlusion mesh updates would be slower, but the quality could be higher. It is worth noting that if we want to achieve frame rates of 120 FPS, our total frame time cannot exceed 8.3 or 16.6 ms for the 60 FPS target. Operations as high as 3 ms or 92 ms add a significant amount of time to the frame time, assuming that the frame needs to wait for the operation to finish. However, it is clearly possible to partially update a sector and divide the computational time between multiple frames. It is also possible to experiment with separate GPU dispatches for mesh generation. However, it is still probable that the dual marching cubes algorithm will be objectively worse in matters of resource usage and surface quality when compared to Newcombe et al. and Meerits et al. approaches. The algorithm has issues with generation of topologically correct surface, unlike standard marching cubes, due to specific unsafe cube configurations. However, this drawback may be insignificant for AR occlusion or even collision purposes. The dual marching cube algorithm is a solution definitely worthy of consideration as an improvement of marching cubes for our purposes and many others.

5.2.2 Scriptable AR Reconstruction Pipeline

As mentioned in previous chapters, our AR reconstruction and rendering solution needs to take multiple steps to produce a sector ready to render. As this solution could be improved in the future, it is a good idea to make each step of AR reconstruction replaceable or extendable. It would give developers the ability to extend and test features for reconstruction and rendering to their own needs, without wasting performance by still benefiting from sector-management options and LoD. Some of the features worth mentioning can be shown:

1. Area prediction based on neighboring points.
2. Simplify the mesh to generate better LoDs.
3. Cached calculation for occlusion and backface culling.
4. Generation of impostors for distant objects.
5. Post-processing for physics.

These operations can be considered as somewhat heavy and thus can be implemented as a post-processing step conducted on a desired basis. Note that the implementation of some of them would require further research.

5.2.3 Caching and Ghosts

Caching the model through the weights of the marching cubes poses a big problem when occluding moving objects. It works well for static objects; temporal stability can be improved, as well as model accuracy, no matter how dense the ray coverage is. Moving objects are harder to distinguish, as there is no way to verify well enough if objects. This means that the space that is no longer occupied or has changed will have ghosts that remain there. There are a couple of possible approaches to solving this problem; however, it is hard to find a solution without drawbacks.

Interpolate Ray Trajectory and Generate Negative Points
As an object has moved, rays will be able to travel through space. This value of weights between can be reduced towards 0 to slowly 'melt' meshes that are no longer in that space. Unfortunately, this method can affect the corners of other objects in a negative way and make the mesh more temporally unstable according to my tests. As rays can pass near the corners, the negative points can both slow down runtime on the CPU side visibly and destabilize these corners.

History
Saving N amount of weight history is another option to consider. However, each frame can cost significant amounts of memory. If, in the worst case, a square meter with an accuracy of 1 cubic centimeter is used, a single marching cube array can contain 1.030.301 weights and take around 4 MB of memory. Using and recomputing these histories can take a significant amount of processing power and memory.

Taking into account previously mentioned reconstruction algorithms like Meerits et al. and speed of GPU memory, we are able to use the local mesh and global mesh approach. In that case, the obsolete sectors can be discarded. We can compute and store certain meshes locally and then merge them into a global mesh buffer. However, in the worst case; this method can use **twice as much memory**.

Vanish Over Time
We could reduce all weights over time or reduce weights that were not updated for longer periods of time. This method might be quite effective in many cases as long as our head is moving and the rays hit all possible points from time to time. However, it would require one to save the last update timestamp for weights or weight groups.

Dynamic Instead of Cached
We may clear out the memory as mentioned in caching section, however, we lose the benefits of stability, as it requires us to clear or reallocate memory for each update.
Change Detection Algorithms
It is possible to create a dedicated algorithm that decides which sectors of the world will be discarded. This can be done as an algorithm-based system or as a deep learning AI system. Furthermore, since AR hardware tends to scan surroundings with short and long passes, the short pass can be handled by a high-accuracy, non-cached mesh generation algorithm to cover. This requires further research in the field.

Custom Mesh Generation Algorithms
Creating and implementing custom algorithms that are less prone to ghost objects, with different approaches to model generation and caching. Mentioned Meerits et al.[35] paper is one of the best examples of such an implementation. Another option is to approach the problem in a hybrid way, combining mesh-based approaches with depth-based approaches. This would require additional research.

5.2.4 Afterword

There is much work to do in the field of AR occlusion. However, we can conclude that AR will improve occlusion, tracking, and alignment as technology improves, and we possibly already are at a point where optical-based AR experience is sufficient for some work and training.

Per Aspera Ad Astra

Glossary

Ahead of Time Referring to code compilation; compiling code before the program is run. AoT languages require longer compilation, but can result in higher performance. AoT language examples: Rust, C, and C++.

Compute Shader A Compute Shader is a shader mainly used to compute any data on the GPU [46].

Concurrency Running computations concurrently, sometimes in parallel, with potential thread waiting states when mutual exclusion is needed.

Delta Time In game engines: the interval in seconds from the last drawn frame to the current frame [66].

Edge A connection between two vertices.

Face A closed set of edges, in which a **triangle** face has three edges, and a face **quad** has four edges.

Fixed Delta Time The interval in seconds at which physics and other fixed frame rate updates are performed. In Unity it happens in fixed intervals, n times per second. In Unreal Engine this is an optional feature known as 'Physics Sub-Stepping'. It can be activated to make physics calculations more accurate and stable for a cost of performance [66].

Frame In rendering context: single rendered picture of the graphic program. The total number of frames rendered by the graphics card per second is known as FPS.

Frame Time Total time (often noted in microseconds) required to render a single frame. Frames per second are calculated in the following way: 1000 ms / frame time (ms).

Frustum Culling Frustrum Culling is a feature that disables the rendering of objects outside the viewing area of the cameras. [70].

Just in Time Referring to code compilation; compiling code while the program is running. Compilation of JiT languages at runtime can speed up prototyping and testing. JiT language examples: Java, Python and C#.

Lit Shader Lit shaders are the most common shaders. They have full shading and lightning support, making them more expensive to render when compared to Unlit Shaders. [74].

Mesh A polygon mesh is a collection of vertices, edges, and faces that defines the shape of a polyhedral object. Part of the 3D model.

Model 3D model or a mesh, is a representation of a object or objects in three dimensions. Models are created from many elements, such as vertices, triangles, and edges. It can also contain textures and material data.

Occlusion Culling Occlusion Culling is a feature that disables the rendering of objects when the camera does not currently see them because they are obscured (occluded) by other objects. Occlusion culling is different from fault culling. Frustum culling only disables the renderers for objects that are outside the camera viewing area but does not disable anything hidden from view by overdrawing. Note that when using occlusion culling, you will still benefit from frustum culling [70].

Pararellity Running computations in parallel, avoiding interference.

Pipeline In programming; refers to the sequence of processing steps that processing units such as CPU or GPU need to undertake to complete the designated task. Rendering pipelines like URP and HDRP in Unity are examples of graphical pipelines. The occlusion process, as it consists of multiple sequential steps, can also be called a pipeline..

Point Cloud A collection of 3D point coordinates that represent the scanned real-world surfaces. Look 2.2 for a demonstration..

Quad A closed set of four edges. One of two forms for representing faces in a 3D model.

Rendering The process of generating 2D or 3D images in the execution of a computer program.

Rendering Pipeline The Rendering Pipeline is the sequence of steps that the graphical engine (or a graphics API) takes when rendering objects [48].

Shader A Shader is a program designed to run on some stage of a graphics processor. Shaders provide the code for certain programmable stages of the rendering pipeline. They can also be used in a slightly more limited form for general GPU computation [49].

Spatial Mapping Technique of mapping the real environment, for example, for AR purposes..

Topology In computer graphics, term topology relates to the structure flow of a 3D model. For example: The model can be modeled with triangles or quads, which all lead to different topologies. Topology can also refer to how the vertices and edges are distributed in the model, which can matter in terms of performance and 3D animation..

Triangle Triangle, or tris in short. A closed set of three edges. One of two forms for representing faces in a 3D model.

Unlit Shader Unlit shaders are more lightweight material shaders in rendering engines. These shaders can be rendered as Lit Shader; however, it does not compute shading and light data, making its appearance unaffected by them. [74].

Vertex (Computer Graphics) A position (usually in 3D space) along with other information such as color, normal vector, and texture coordinates.

Viewing Frustum Is area of the world visible to the camera. Look 3.15 "field of view", between the near cutting plane and the far cutting plane..

Viewport A framed area on a display screen for viewing information.

Acronyms

AI Artificial Intelligence.

AoT Ahead of Time.

API Application Programming Interface.

APU Accelerated Processing Unit.

AR Augmented Reality.

AV Augmented Virtuality.

CPU Central Processing Unit.

FPS Frames Per Second.

GC Garbage Collector.

GPU Graphical Processing Unit.

HCI Human-Computer Interaction.

HDRP High-Definition Render Pipeline.

HLSL (Microsoft) High Level Shading Language.

IR Infrared.

JiT Just in Time.

LIDAR Light Detection and Ranging.

LoD Level of Detail.

OS Operating System.

RAM Random Access Memory.

SRP Standard Render Pipeline.

URP Universal Render Pipeline.

VR Virtual Reality.

VRAM Video Random Access Memory.

Bibliography

[1] Adriana Paíno Ambrosio and M. Isabel Rodríguez Fidalgo. "Past, present and future of Virtual Reality: Analysis of its technological variables and definitions." In: *Culture and History Digital Journal* 9.1 (2020). ISSN: 2253797X. DOI: 10.3989/CHDJ.2020.010.

[2] Ask a Game Dev. *Game Optimization Tricks (part 2): Backface Culling*. URL: https://askagamedev.tumblr.com/post/92638684416/game-optimization-tricks-part-2-backface-culling.

[3] AutoPilot Review. *Elon Musk on Cameras vs LiDAR for Self Driving and Autonomous Cars*. 2019. URL: https://youtu.be/HM23sjhtk4Q.

[4] Andrzej Barczak and Hubert Woźniak. "Comparative Study on Game Engines." In: *Studia Informatica* 2.23 (2020), pp. 5–24. ISSN: 1731-2264. DOI: 10.34739/si.2019.23.01.

[5] Peter Bauer, Werner Lienhart, and Samuel Jost. "Accuracy investigation of the pose determination of a vr system." In: *Sensors* 21.5 (2021), pp. 1–17. ISSN: 14248220. DOI: 10.3390/s21051622.

[6] Boston Dynamics. *Spot specifications*. 2021. URL: https://support.bostondynamics.com/s/article/Robot-specifications.

[7] Paul Bourke. *Polygonising a scalar field*. 2016. URL: http://paulbourke.net/geometry/polygonise/.

[8] Julie Carmigniani et al. "Augmented reality technologies, systems and applications." In: *Multimedia Tools and Applications* 51.1 (2011), pp. 341–377. ISSN: 13807501. DOI: 10.1007/s11042-010-0660-6.

[9] Eleftheria Christopoulou and Stelios Xinogalos. "Overview and Comparative Analysis of Game Engines for Desktop and Mobile Devices." In: *International Journal of Serious Games* 4.4 (2017), pp. 21–36. DOI: 10.17083/ijsg.v4i4.194.

[10] Cmglee. *Time of flight camera principle*. 2017. URL: https://commons.wikimedia.org/wiki/File:Time_of_flight_camera_principle.svg.

[11] Daniel Cohen-Or et al. "A survey of visibility for walkthrough applications." In: *IEEE Transactions on Visualization and Computer Graphics* 9.3 (2003), pp. 412–431. ISSN: 10772626. DOI: 10.1109/TVCG.2003.1207447.

[12] ConsoleDatabase. *Sony PlayStation/PSOne*. URL: https://www.consoledatabase.com/consoleinfo/sonyplaystation/.

[13] Rodrigo Copetti. *Playstation Architecture - A practical analysis by Rodrigo Coppeti*. 2019. URL: https://www.copetti.org/writings/consoles/playstation/.

[14] Rodrigo Copetti. *Virtual Boy Architecture*. 2021.

[15] Marching Cubes and Isosurface Ext. *Marching Cubes*. 2010. URL: https://daac.hpc.mil/gettingStarted/Marching_Cubes.html.

[16] Goran Damnjanović. *What is shared GPU memory?* 2021. URL: https://levvvel.com/what-is-shared-gpu-memory/.

[17] Dorit Borrmann, Hassan Afzal, and Jacobs University Bremen gGmbH. *Robotic 3D Scan Repository*. URL: http://kos.informatik.uni-osnabrueck.de/3Dscans/.

[18] David Drascic and Paul Milgram. "Perceptual Issues in Augmented Reality." In: *Stereoscopic Displays and Virtual Reality Systems III* 2653.December 2013 (1996), pp. 123–134. DOI: 10.1117/12.237425.

[19] Amnon H. Eden. "Three paradigms of computer science." In: *Minds and Machines* 17.2 (2007), pp. 135–167. ISSN: 09246495. DOI: 10.1007/s11023-007-9060-8.

[20] Ellie Harisova. *Inside Game Development: Using Impostors*. 2020. URL: https://80.lv/articles/inside-game-development-using-impostors.

[21] Marwa Elteir, Heshan Lin, and Wu Chun Feng. "Performance characterization and optimization of atomic operations on AMD GPUs." In: *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (2011), pp. 234–243. ISSN: 15525244. DOI: 10.1109/CLUSTER.2011.34.

[22] Ryan (NVIDIA Corporation) Geiss. "Generating complex procedural terrains using the gpu." In: *GPU Gems* (2007), pp. 7–37. URL: http://www.cse.chalmers.se/edu/year/2013/course/TDA361/Advanced%20Computer%20Graphics/Generating_Complex_Procedural_Terrains_Using_t.pdf.

[23] Michael T Goodrich and Roberto Tamassia. *Algorithm Design And Applications*. 2015.

[24] Roberto Grosso and Daniel Zint. "A parallel dual marching cubes approach to quad only surface reconstruction." In: *Visual Computer* 38.4 (2022), pp. 1301–1316. ISSN: 01782789. DOI: 10.1007/s00371-021-02139-w. URL: https://doi.org/10.1007/s00371-021-02139-w.

[25] Simona Gugliermo. "Occlusion handling in Augmented Reality context." In: (2019). URL: http://www.diva-portal.org/smash/record.jsf?pid=diva2%5C%3A1361930&dswid=7954.

[26] Nhut Minh Ho and Weng Fai Wong. "Exploiting half precision arithmetic in Nvidia GPUs." In: *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017* (2017). DOI: 10.1109/HPEC.2017.8091072.

[27] Y.-S. Ho et al. "Advances in Multimedia Information Processing - PCM 2015: 16th Pacific-Rim Conference on Multimedia Gwangju, South Korea, September 16-18, 2015 Proceedings, Part II." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9315.September 2016 (2015). ISSN: 16113349. DOI: 10.1007/978-3-319-24078-7.

[28] Sanders Kandrot and Jason Edward. *CUDA by Example: An Introduction to General Purpose Graphical Processing Unit*. 2010. ISBN: 9780131387683.

[29] Michael Kazhdan and Hugues Hoppe. "Screened poisson surface reconstruction." In: *ACM Transactions on Graphics* 32.3 (2013), pp. 1–13. ISSN: 07300301. DOI: 10.1145/2487228.2487237.

[30] T. Kimura et al. "Bilateral simultaneous epididymal leiomyoma: a case report." In: *Hinyokika kiyo. Acta urologica Japonica* 44.12 (1998), pp. 901–903. ISSN: 00181994.

[31] V. Lepetit and M. O. Berger. "Handling occlusion in augmented reality systems: A semi-automatic method." In: *Proceedings - IEEE and ACM International Symposium on Augmented Reality, ISAR 2000* February 2000 (2000), pp. 137–146. DOI: 10.1109/ISAR.2000.880937.

[32] Lianhua Li et al. "Multi-camera interference cancellation of time-of-flight (TOF) cameras." In: *Proceedings - International Conference on Image Processing, ICIP* 2015-Decem.September (2015), pp. 556–560. ISSN: 15224880. DOI: 10.1109/ICIP.2015.7350860.

[33] William E. Lorensen and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1987* 21.4 (1987), pp. 163–169. DOI: 10.1145/37401.37422.

[34] Daniel Madeira, Esteban Clua, and Thomas Lewiner. "GPU octrees and optimized search." In: *Proceedings of the 8th Brazilian Symposium on Games and Digital Entertainment* (2009), pp. 73–76.

[35] Siim Meerits, Vincent Nozick, and Hideo Saito. "Real-time scene reconstruction and triangle mesh generation using multiple RGB-D cameras." In: *Journal of Real-Time Image Processing* 16.6 (2019), pp. 2247–2259. ISSN: 18618219. DOI: 10.1007/s11554-017-0736-x.

[36] Microsoft. *HLSL - Scalar Types*. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/direct3dhlsl/dx-graphics-hlsl-scalar.

[37] Microsoft. *unsafe (C# Reference)*. 2022. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/unsafe.

[38] Microsoft. *Unsafe Code*. 2022. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/language-specification/unsafe-code#229-stack-allocation.

[39] Minecraft Wiki. *Chunk*. URL: https://minecraft.fandom.com/wiki/Chunk.

[40] Net-informations. "C# Dictionary Versus List Lookup Time." In: (). URL: http://net-informations.com/faq/general/dictionary-list.htm.

[41] Richard A. Newcombe and Andrew J. Davison. "Live dense reconstruction with a single moving camera." In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (2010), pp. 1498–1505. ISSN: 10636919. DOI: 10.1109/CVPR.2010.5539794.

[42] Richard A. Newcombe and Andrew J. Davison. *Live dense reconstruction with a single moving camera*. 2010. DOI: 10.1109/CVPR.2010.5539794. URL: https://youtu.be/CZiSK7OMANw.

[43] Next Reality. *How The HoloLens 2 Works, Explained By Microsoft's Alex Kipman - YouTube*. 2019. URL: https://www.youtube.com/watch?v=S0fEh4UdtT8&feature=emb_rel_end.

[44] Nvidia. "NVIDIA ® Tegra ® X1 NVIDIA'S New Mobile Superchip." In: *Nvidia White Papers* (2015), pp. 1–41.

[45] Nvidia. "NVIDIA A100 Tensor Core GPU." In: *White Paper* (2020), pp. 20–21. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[46] OpenGL. *Compute Shader*. URL: https://www.khronos.org/opengl/wiki/Compute_Shader.

[47] OpenGL. *Face Culling - OpenGL Wiki*. URL: https://www.khronos.org/opengl/wiki/Face_Culling.

[48] OpenGL. *Rendering Pipeline*. URL: https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview.

[49] OpenGL. *Shader*. URL: https://www.khronos.org/opengl/wiki/Shader.

[50] Imtiaz Parvez et al. "A survey on low latency towards 5G: RAN, core network and caching solutions." In: *IEEE Communications Surveys and Tutorials* 20.4 (2018), pp. 3098–3130. ISSN: 1553877X. DOI: 10.1109/COMST.2018.2841349. arXiv: 1708.02562.

[51] Rui Pereira et al. "Energy efficiency across programming languages: How do energy, time, and memory relate?" In: *SLE 2017 - Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering, co-located with SPLASH 2017* (2017), pp. 256–267. DOI: 10.1145/3136014.3136031.

[52] Gaël Guennebaud Pierre Alliez, Laurent Saboret. *CGAL 5.4 - Poisson Surface Reconstruction*. 2022. URL: https://doc.cgal.org/latest/Poisson_surface_reconstruction_3/index.html.

[53] Gheorghii Postica, Andrea Romanoni, and Matteo Matteucci. "Robust moving objects detection in lidar data exploiting visual cues." In: *IEEE International Conference on Intelligent Robots and Systems* 2016-November (2016), pp. 1093–1098. ISSN: 21530866. DOI: 10.1109/IROS.2016.7759185. arXiv: 1609.09267.

[54] Ihsan Rabbi and Sehat Ullah. "A Survey on Augmented Reality Challenges and Tracking." In: *Acta Graphica* 24.1-2 (2013), pp. 29–46. ISSN: 0353-4707. URL: http://hrcak.srce.hr/file/150828.

[55] Jennifer Preece Helen Sharp Yvonne Rogers. "6.2.17 Augmented and Mixed Reality." In: *Interaction Design, Beyond Human-Computer-Interaction, 4th Edition*. 2015.

[56] Jannick P. Rolland and Henry Fuchs. "Optical Versus Video See-Through Head-Mounted Displays in Medical Visualization." In: *Presence: Teleoperators and Virtual Environments* 9.3 (2000), pp. 287–309. ISSN: 10547460. DOI: 10.1162/105474600566808.

[57] Nadathur Satish, Narayanan Sundaram, and Kurt Keutzer. "Optimizing the use of GPU memory in applications with large data sets." In: *16th International Conference on High Performance Computing, HiPC 2009 - Proceedings* December (2009), pp. 408–418. DOI: 10.1109/HIPC.2009.5433185.

[58] Manisah Mohd Shah, Haslina Arshad, and Riza Sulaiman. "Occlusion in augmented reality." In: *Proceedings - ICIDT 2012, 8th International Conference on Information Science and Digital Content Technology* 2 (2012), pp. 372–378.

[59] SUTHERLAND IE. "Head-Mounted Three Dimensional Display." In: 33.pt 1 (1968), pp. 757–764. DOI: 10.1145/1476589.1476686.

[60] Clive Thompson. *The Gendered History of Human Computers*. 2019. URL: https://www.smithsonianmag.com/science-nature/history-human-computers-180972202/.

[61] Unity. *Optimizing draw calls*. URL: https://docs.unity3d.com/2022.1/Documentation/Manual/optimizing-draw-calls.html.

[62] Unity. *Profiler*. 2022. URL: https://docs.unity3d.com/Manual/Profiler.html.

[63] Unity Docs. *Compute Shaders*. URL: https://docs.unity3d.com/Manual/class-ComputeShader.html.

[64] Unity Docs. "DOTS Hybrid Renderer." In: (). URL: https://docs.unity3d.com/Packages/com.unity.rendering.hybrid@0.4/manual/index.html.

[65] Unity Docs. *Entities*. URL: https://docs.unity3d.com/Packages/com.unity.entities@0.16/manual/index.html.

[66] Unity Docs. *ExecutionOrder*. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html.

[67] Unity Docs. "Graphics API Support." URL: https://docs.unity3d.com/Manual/GraphicsAPIs.html.

[68] Unity Docs. *IL2CPP*. URL: https://docs.unity3d.com/Manual/IL2CPP.html.

[69] Unity Docs. "Level of Detail (LOD) for meshes." URL: https://docs.unity3d.com/Manual/LevelOfDetail.html.

[70] Unity Docs. *Occlusion Culling*. URL: https://docs.unity3d.com/Manual/OcclusionCulling.html.

[71] Unity Docs. *Render Pipelines Introduction*. URL: https://docs.unity3d.com/Manual/render-pipelines-overview.html.

[72] Unity Docs. *Scriptable Render Pipeline introduction*. URL: https://docs.unity3d.com/Manual/scriptable-render-pipeline-introduction.html.

[73] Unity Docs. *Scripting Restrictions*. URL: https://docs.unity3d.com/Manual/ScriptingRestrictions.html.

[74] Unity Docs. *Shading models in Universal Render Pipeline*. URL: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@14.0/manual/shading-model.html.

[75] Unity Learn. *Entity-Component-System*. URL: https://learn.unity.com/tutorial/entity-component-system#5c7f8528edbc2a002053b67b.

[76] User Benchmark. *User Benchmark*. URL: https://www.userbenchmark.com/page/about.

[77] Ajay Venkat. *Unity Job System and Burst Compiler: Getting Started*. 2020. URL: https://www.raywenderlich.com/7880445-unity-job-system-and-burst-compiler-getting-started#toc-anchor-004.

[78] Yan Wang et al. "A view of research on wireless power transmission." In: *Journal of Physics: Conference Series* 1074.1 (2018). ISSN: 17426596. DOI: 10.1088/1742-6596/1074/1/012140.

[79] Oliver Wasenmüller and Didier Stricker. "Comparison of kinect v1 and v2 depth images in terms of accuracy and precision." In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10117 LNCS (2017), pp. 34–45. ISSN: 16113349. DOI: 10.1007/978-3-319-54427-4_3.

[80] Pierre Wellner. "Interaction with Paper on the Digital Desk." In: *Cacm* 36.7 (1993), pp. 87–96.

[81] Danny Yadron and Dan Tynan. *No Title*. 2016. URL: https://www.theguardian.com/technology/2016/jun/30/tesla-autopilot-death-self-driving-car-elon-musk.

[82] Matt Zachara and José P. Zagal. "Challenges for success in stereo gaming: A virtual boy case study." In: *ACM International Conference Proceeding Series* January 2009 (2009), pp. 99–106. DOI: 10.1145/1690388.1690406.

[83] Zeus. *Z buffer*. 2009. URL: https://commons.wikimedia.org/wiki/File:Z_buffer.svg.

[84] L. Y. Zhang, R. R. Zhou, and L. S. Zhou. "Model reconstruction from cloud data." In: *Journal of Materials Processing Technology* 138.1-3 (2003), pp. 494–498. ISSN: 09240136. DOI: 10.1016/S0924-0136(03)00127-4.

[85] Kun Zhou et al. "Highly parallel surface reconstruction." In: *Microsoft Research Asia* (2008), p. 10. URL: https://www.microsoft.com/en-us/research/publication/highly-parallel-surface-reconstruction/.