DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Using Mobile AR Tracking Data to Generate 3D Geometry for Gameplay Interaction

Florian Rett

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

Using Mobile AR Tracking Data to Generate 3D Geometry for Gameplay Interaction

Verwendung von Mobile-AR Tracking Daten zur Generierung von 3D Geometrie für Spielmechaniken

Author:	Florian Rett
Supervisor:	Prof. Gudrun Klinker
Advisor:	Christian Eichhorn
Submission Date:	17.06.2019

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 17.06.2019 Florian Rett

Abstract

In recent years, Augmented Reality achieved some huge development progress and reached millions of consumers via their smartphones. Due to advances in mobile computing power, ARCore and ARKit are able to provide markerless motion tracking of the device's location and rotation through the real world. Thanks to those frameworks, many AR apps and games have been developed, allowing to overlay the real world with virtual content. However, the interaction between the virtual and the real world in such apps is quite limited at the moment. Virtual objects can be placed on even surfaces around the user, but there is no further understanding of other real objects present.

This thesis aims to develop a system that provides a deeper understanding of the surrounding real world. The presented system does not need any other input than the point cloud of visual features that is used by the AR frameworks to track the device's position. This point cloud is used to create 3D mesh geometry representing real-world objects. These meshes can then be used for additional interaction between the physical and the virtual world. A sample game was implemented to showcase the results of the proposed system using those meshes as core gameplay elements.

Kurzfassung

In den letzten Jahren wurden einige große Fortschritte in der Entwicklung von Augmented Reality erzielt und die Technologie erreichte Millionen von Nutzern auf deren Smartphones. Dank der stetig wachsenden Rechenleistung von mobilen Prozessoren sind ARCore und ARKit in der Lage, ohne die Hilfe von speziellen Markierungen die Position und Rotation des Geräts in der echten Welt zu verfolgen. Mit Hilfe dieser Frameworks wurde eine Vielzahl von AR Apps und Spielen entwickelt, in denen man die echte Welt mit virtuellen Inhalten überlagern kann. Allerdings ist aktuell die Interaktion zwischen der virtuellen und der echten Welt in diesen Apps noch sehr eingeschränkt. Virtuelle Objekte können auf geraden Oberflächen in der Nähe des Benutzers platziert werden, es findet allerdings kein Erkennen von weiteren realen Objekten statt.

Das Ziel dieser Arbeit ist es, ein System zu entwickeln, welches ein tieferes Verständnis der realen Umgebung ermöglicht. Das vorgestellte System benötigt als Input lediglich die Punktwolke aus visuellen Merkmalen, die auch von den AR Frameworks zum Tracking der Geräteposition verwendet wird. Diese Punktwolke wird verwendet, um 3D Mesh Geometrie zu erzeugen. Die so erzeugten Meshes können dann für weitere Interaktionen zwischen der physischen und der virtuellen Welt verwendet werden. Um die Ergebnisse des vorgestellten Systems zu präsentieren, wurde ein Beispiel-Spiel entwickelt, das diese Meshes als Kernelement der Spielmechaniken einsetzt.

Contents

1. Introduction

The concept of overlaying the real world with virtual content to transmit information in a more natural way than traditional media is all but new. Its origin can be dated back more than half a century ago to the first augmented reality display. Over the last decades, many science-fiction movies featured technologies like holographic displays, foreseeing a future where augmented reality is a natural part of every-day life. Today, there still is a huge discrepancy between what is possible with current AR technology and the visions brought to us by Hollywood, but the gap is closing.

In recent years huge breakthroughs regarding Augmented Reality technology have been achieved, the most important one probably being Google's ARCore [1] and Apple's ARKit [2], bringing markerless Augmented Reality capabilities to millions of mobile devices and giving developers the tools to easily create AR applications.

This thesis approaches the problem of enhancing the current capabilities of the mentioned AR frameworks by enabling new ways how the user can interact with the real world through an AR application. To achieve this, a reconstruction pipeline is proposed, that is able to detect arbitrary objects in the real world and generate collision for them, so they can then be used for user interactions in a game environment.

1.1. Motivation

While augmented reality technology has been around for some time now, it has experienced noticeable growth in recent years. The main reason for this is the steady increase in mobile computation power. This allows running systems, that were previously only working on stationary Desktop PCs, on much smaller and portable devices, which is a far more appealing application area for augmented reality. The first of such devices to become broadly accepted were AR glasses such as the Microsoft HoloLens, but due to the high price of multiple thousand dollars, they were not able to bring AR technology to the consumer market. The release of ARCore and ARKit changed this, as both platforms merely require a fairly modern smartphone as they are found in most households nowadays.

In addition to tracking the device's position in the real world without the use of markers, both AR frameworks offer a basic form of scene understanding. They detect horizontal and vertical planes around the user, which can then be used to place virtual content on real-world surfaces. However, the amount of interaction that is possible with this kind of scene understanding is very limited and results in little variety in the existing AR apps. Achieving a better scene understanding within the AR frameworks would allow developers more freedom when designing user interactions, which can be especially beneficial for video games using AR elements. This is an inevitable step in the further evolution of AR technology.

1.2. Goals of this thesis

This thesis presents an experimental approach to the described problem of creating a better scene understanding. The primary goal is to develop a system that uses the internal motion tracking data of the mobile AR frameworks and generates mesh geometry from them, which can then be used in a game environment. The system should be able to run on most consumer devices and therefore not rely on any additional input data such as a depth sensor. A pipeline for this kind of scene reconstruction is described, implemented and evaluated. A sample game is designed and implemented to show that the proposed system can be used in a game environment to enable gameplay mechanics that would not be possible with the default capabilities of the AR frameworks.

1.3. Thesis Outline

Chapter 2 - Related work is a more in-depth introduction to the research topic and provides the necessary background knowledge. In the beginning a general overview of the current state of Augmented Reality is given. The current use of AR technology in games is evaluated by describing and analyzing some popular AR games found in the mobile app stores. Also, an overview on surface reconstruction is given, presenting other solutions to a better scene understanding.

Chapter 3 - Reconstruction Pipeline describes the proposed calculation steps needed for the reconstruction, including descriptions of all the algorithms that were used.

Chapter 4 - Implementation focuses on the Android application that was developed to test the proposed reconstruction pipeline. Important systems of the implementation are described in more detail. There is also given a short description of the sample game that was designed to present the results.

Chapter 5 - Evaluation displays the results of the proposed reconstruction pipeline. At first, the used evaluation method is described, followed by the evaluated data and some discussions about the results. The observed limitations of the system are also stated.

Chapter 6 - Conclusion summarizes the thesis and the achieved results. Some ideas regarding further improvement of the proposed system and other future research are given.

2. Related Work

In this chapter, the current state of Augmented Reality, with a focus on mobile Augmented Reality, is described. Some examples of popular mobile AR games from both the Google Play Store and the Apple App Store are investigated regarding their use of AR technology and their limitations. Next, an overview of existing solutions and related work addressing these limitations is given.

2.1. Augmented Reality

Augmented Reality (AR) is most commonly described as the technology that overlays or combines the real world with digital information in some kind of interactive experience [3]. Furthermore, Augmented Reality is classified as a variation of Virtual Reality, with the main difference being that in Virtual Reality the user is completely immersed in a virtual environment, whereas in Augmented Reality the user still sees the real world around him, with virtual objects supplemented to it in order to enhance the reality as it is perceived by the user [4]. This distinction was further clarified by Milgram and Kishino by defining the "Virtuality Continuum" as a medium to classify different Mixed-Reality displays, with Augmented Reality being one subset of those. They use the term Augmented Reality to "refer to any case in which an otherwise real environment is 'augmented' by means of virtual (computer graphic) objects" [5].

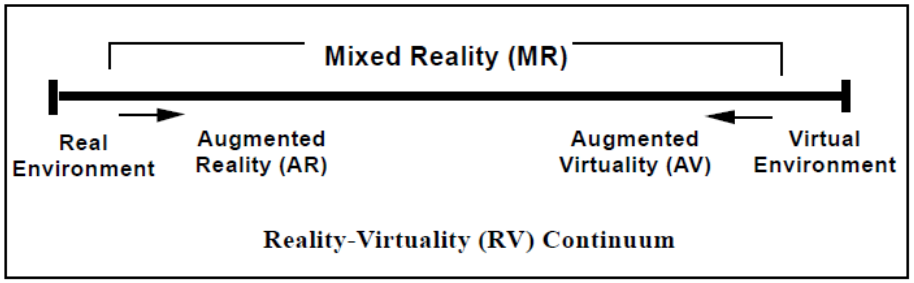


Figure 2.1.: The Virtuality Continuum by Milgram and Kishino [5] can be used to classify different Mixed-Reality displays. The scale describes the ratio of real and virtual objects of which the scene is composed with the left side being only real and the right side only virtual objects.

While the first concepts of Augmented Reality can be dated back to 1968 to the first application using an optical see-through head-mounted display to display simple wireframe

drawings on top of the real world, the term "Augmented Reality" was not used before 1992. In this year, Caudell and Mizell designed a see-through head-mounted display called HUDSET to assist in aircraft manufacturing tasks and described the technology to "augment" the user's visual field of view with additional information [6], therefore the term Augmented Reality was established.

2.1.1. Mobile Augmented Reality

The meaning of the term "Mobile AR" has evolved quite a bit over time. When it first came up, it referred to AR that could be experienced during locomotion, interpreting "mobile" as any kind of motion. Later, with computer hardware and displays becoming ever smaller and more portable, the meaning shifted towards Augmented Reality on a "mobile device" [7]. Nowadays the term is mostly used to specifically describe Augmented Reality Apps running on modern smartphones, as it will be the case in this thesis.

One of the greatest challenges when developing an AR application, especially for a mobile device with limited computational power and size, is the Tracking and Registration problem, which refers to the process of evaluating the device's current position and orientation to align the virtual content with the real world. Chatzopoulos et. al. categorize the existing tracking and registration solutions into sensor-based and vision-based approaches. Sensor-based methods use the device's internal sensors such as inertial or magnetic fields or ultrasonic and radio-wave measures to calculate the device's position and orientation. Vision-based methods estimate a pose from markers or feature points in a captured video [8].

The first implementations of Mobile AR Applications on a smartphone can be dated back to the year 2004 when M. Möhring et. al. presented a video see-through AR application that was able to detect markers and render 3D graphics on top of the camera image [9]. In 2009, Klein and Murray demonstrated the first Simultaneous Localization and Mapping (SLAM) system running on a consumer smartphone, which was able to track the phone's movement with 6 Degrees of Freedom in real-time [10].

Thanks to further increases of smartphone computational power and higher quality cameras being included, markerless Mobile Augmented Reality is nowadays available on millions of devices in a consumer-ready state. Today, Google and Apple are developing the main platforms for consumer Mobile AR. In 2017, Apple released ARKit [2], a Software Development Kit for developing AR applications on iOS devices. In the same year, Google released ARCore [1], an SDK with very similar capabilities targeting the development for mobile devices running Google's own Operating System Android.

ARKit and ARCore both use visual-inertial odometry (VIO) to track the device's motion through the real world, a technique that has its origin in the field of robotics and autonomous vehicles [11]. VIO approaches the task of motion tracking in unknown environments by combining the detection of distinct feature points in the captured camera image, also known as visual-odometry, with inertial measurements from internal sensors such as accelerometers and gyroscopes to achieve higher accuracy and consistency. During this process, an arbitrary coordinate system is created using the first registered camera location as the origin. This persistent coordinate system can then be used as a reference system for the positioning of the

augmented world.

Since their release both SDKs have received a number of updates, improving the overall stability and performance of the platform and adding more features. At the time of writing, ARCore and ARKit share a very similar feature set. Their core features are motion tracking, to allow markerless AR experiences, a basic scene understanding by detecting horizontal and vertical planes, which the user can do collision tests against to virtually interact with the real world, and the ability to share the mapped environment with other devices, therefore allowing multi-user AR experiences. They also both allow the detection and tracking of 2D images, to align virtual content with them, and light estimation of the real environment, to reflect the existing lighting conditions on virtual objects. In addition to those features, ARKit also offers object recognition and tracking of known 3D objects, whereas with ARCore it is possible to place virtual content on arbitrary angled surfaces, given enough feature points were detected on that surface [12, 13].

2.2. Current AR Games

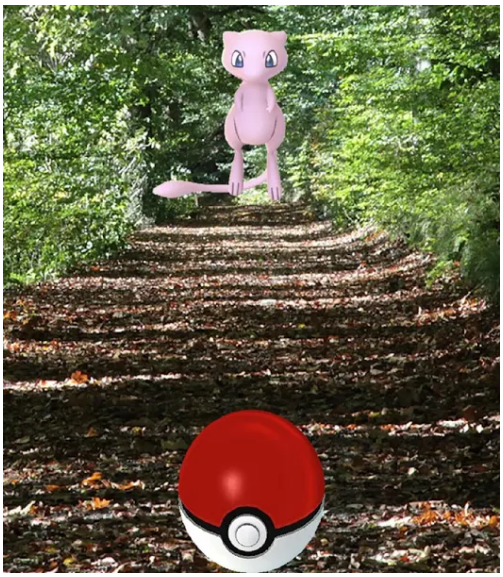
In this section, some examples of currently popular Augmented Reality games from both the Google Play Store and the Apple App Store are investigated regarding their use of AR technology to evaluate the current use of AR technology in games.

Pokémon GO by Niantic is by far the most popular and well-known AR game on the market and was also the first game for many people to experience AR technology in. In the GPS location-based game for iOS and Android, the player can walk around the real world to encounter wild pokemon and try to catch them. Later the caught Pokemon can be trained and used to fight against other players in so-called Gym battles. In addition to using the real world as the playing field, Pokémon GO also offers an AR mode while catching wild pokemon. In this mode the device's camera is combined with its orientation, so the pokemon appears to be floating in front of the user in the real world. It doesn't use features such motion tracking the device or plane detection to place the pokemon on real-world surfaces [14].

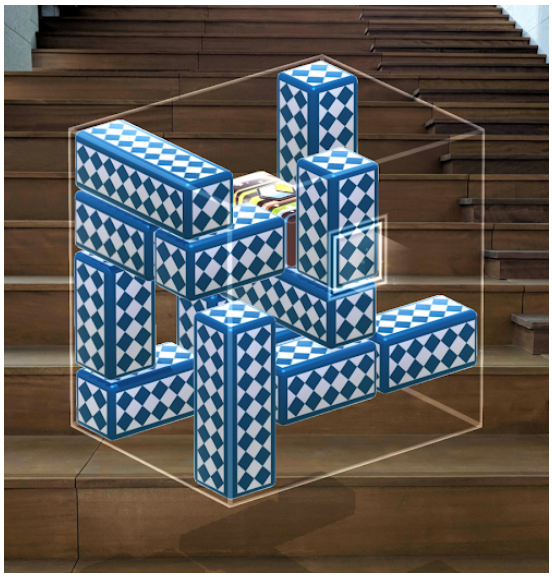
Brickscape by 5minlab is a 3D puzzle game where the player has to move blocks inside a box in order to free a block in the center. In normal mode, the box is displayed in front of a neutral background and can be freely rotated by the player. When AR is enabled, the game instead searches for planes to place the box on top of. The player can still rotate the box, but he can also physically move his phone to change the in-game viewing angle [15].

In A&E Television Networks' *Knightfall AR* the player finds himself in the 13th century during the siege of Acre. The player has to defeat waves of invading enemies by placing towers along the road or shooting at them with arrows and catapults. Knightfall AR uses ARCore's plane detection to place the playing field on a table in front of the player, similar to a board game. As in Brickscape, the player can look around by moving his phone. Additionally, the arrows and catapult shots are also aimed using the motion tracked device [16].

Similar to Knightfall, *AR Smash Tanks!* by Dumpling Design is played on a table top or the floor by detecting planes in the real world. In this multiplayer game, players fight battles with miniature tanks. The goal is to destroy the enemies' tanks with different kinds of weapons.



(a) Pokémon GO [14] while catching a wild pokemon.



(b) Brickscape [15] when AR mode is enabled.

Figure 2.2.: Screenshots from Pokémon GO and Brickscape.

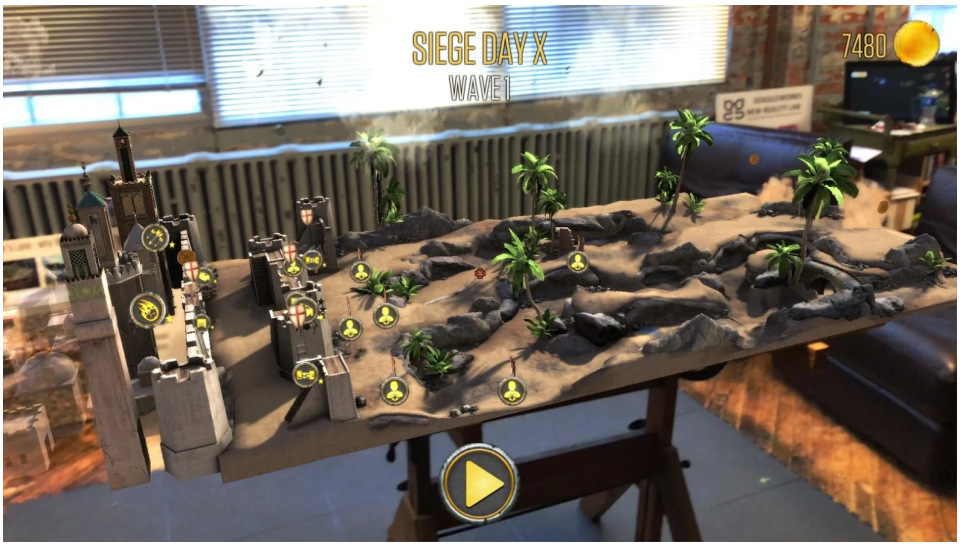


Figure 2.3.: Knightfall AR [16].

Like in the previous examples, looking around the virtual battlefield is done by moving the device around [17].

As it can be seen in the examples above, the scene understanding provided by ARCore and ARKit is suitable to develop simple AR games that place virtual content on the floor, on top of a table or on a wall. However, this does not allow for a very high degree of immersion while the player is engaging with such games, which has been shown to have quite a high impact on the user's enjoyment [18]. There are many competing definitions for the term immersion. Slater et al. describe immersion as a sense of "presence" a virtual environment creates for the user. This perception is mainly influenced by the technology used to deliver the virtual environment, i.e. the hardware the user is playing with [19]. Coomans and Timmermans, on the other hand, describe immersion as "a feeling of being deeply engaged where people enter a make-believe world as if it is real", hence the virtual environment needs to behave in a way that appears natural to the user [20].

The hardware currently used for mobile AR is very limited in its ability to deliver a high degree of immersion. In comparison to VR headsets, the field of view of a smartphone display is really small, there is no way to generate tactile feedback other than vibrating the whole phone, and the only way the user can interact with the game is via a 2-dimensional user interface. As changes in the hardware with the purpose to increase immersion cannot be easily made, the overall immersion of an AR game is mainly dependent on the software to appear as plausible as possible. One area where this could be improved is the physical interaction between virtual objects and the real world. These interactions need to be as accurate as possible, or the user will lose his immersion due to the virtual objects being clearly different from the world around them. Another area is rendering and especially the occlusion of virtual objects when they are positioned behind an object in the real world. Currently, the whole virtual world is just rendered on top of the camera image, making it always appear to be in front of the real world. Though when the phone is moved, due to the parallax effect the virtual object will move slower across the screen than the real object, therefore telling the user the visual sorting order of the objects is not correct. This discrepancy can also greatly reduce a user's immersion. Both of these goals can be achieved by a better scene understanding, which is able to detect and reconstruct more complex geometry than planes. A better scene understanding would also allow more freedom in the design of AR games, enabling more complex concepts of virtual objects interacting with the real world.

2.3. Surface Reconstruction

Surface reconstruction is usually referred to as the process of obtaining a 3-dimensional digital model of an object in the real world, which matches the surface of the real object as closely as possible. Such generated models can then be used in a number of different applications, such as video-games, movies, or other kinds of computer graphics applications. The field of robotics and autonomous vehicles is also using surface reconstruction techniques to create a map of the surrounding world to avoid collisions with real objects.

An important requirement for surface reconstruction is the ability to scan the real world and

hence gain digital information about it. This can be achieved by a huge variety of techniques such as laser scanning, photogrammetry, or structured light scanning [21]. Such scanners use various approaches to generate a point cloud of surface points along the scanned real world. These point clouds then need to be processed in order to be used in 3D applications.

In many established computer-aided-design applications, point cloud data can directly be imported and used as a digital representation of the scanned physical object. This approach, however, is computationally heavy and only suitable for a visual representation, but not for physical simulations, especially in the case of incomplete or missing data in the point cloud. A much more commonly used format in computer graphics applications is that of polygonal meshes. A Mesh consists of a set of points in 3d space (vertices). These vertices are connected by edges and thereby form polygonal shapes. Usually, triangles are used for these polygonal shapes, as they are the simplest geometric primitive. Therefore the process of converting a point cloud into a polygonal mesh is often called triangulation [22].

There is a huge variety of surface reconstruction algorithms, that differentiate in what kind of objects they are able to reconstruct, input requirements regarding the properties of the point cloud, and the imperfections in the point cloud data the algorithm is able to handle [21]. An important geometric structure that is worth pointing out because it is used by a good amount of those algorithms is the Delaunay triangulation and its dual structure, the Voronoi diagram [23].

2.3.1. Delaunay Triangulation Based Algorithms

The Delaunay triangulation for a set of points in two dimensions has the property, that the circumcircle of any triangle in the triangulation contains no other point of the sample set in its interior [24]. This property ensures that the Delaunay triangulation is unique for every point set, given that no four points are cocircular, i.e. that no circle can be found for which all four points lie on its perimeter. Even though it was originally only defined in two dimensions, the Delaunay triangulation can also be used for sample sets in higher dimensions. In 3D for example, it is in fact a tetrahedralization, with the condition that the circumsphere of each tetrahedron does not contain any other of the sample points. The uniqueness criteria is therefore extended to no five points being cospherical [25].

Some of the first surface reconstruction algorithms such as tangent plane estimation [27] or alpha shape [28] made use of the geometrical properties of the Delaunay triangulation as an auxiliary data structure for the reconstruction. The Crust algorithm by Amenta et al. was the first Delaunay triangulation based algorithm with a theoretical guarantee on the reconstruction result, given a high enough sampling density [29, 30, 26].

2.3.2. Live reconstruction

Most surface reconstruction algorithms aim to improve the quality of the reconstruction as much as possible, with the required computation time and hardware being less important. While this makes them unusable in real-time or near-real-time scenarios, there are some approaches towards live reconstruction of the environment [31].

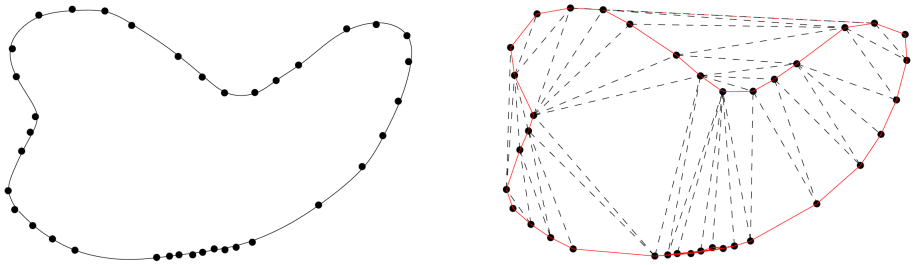


Figure 2.4.: Left: a sample curve. Right: The Delaunay Triangulation for the given sample curve [26]. Note that the Delaunay triangulation will always span the convex hull of all sample points.

The Microsoft Kinect enabled a lot of new developments in the field of surface reconstruction, as it made a depth sensor with decent quality available to everybody for a low price. As an example, KinectFusion uses the depth data provided by the sensor to track the Kinect's movement and reconstruct a 3D mesh representation of the physical scene in real-time. Despite the impressive results the system was able to achieve, it relies on the input of a depth sensor and is therefore not suitable to be used on consumer smartphones. KinectFusion is also optimized to utilize the parallel computation of a desktop GPU, which would be way too expensive for current smartphone hardware [32].

In 2010, Newcombe and Davison presented their system capable of live dense reconstruction with just a single moving camera. They used a parallel tracking and mapping system to calculate the camera's pose and extract visual feature points from the scene. Those feature points were used to generate dense depth maps which could then be merged into the actual scene reconstruction. In contrast to KinectFusion, this system uses only a single monoscopic camera, as it can now be found in every modern smartphone. However, the computations needed for the live reconstruction of the scene also required a high-end desktop computer with multiple dedicated graphics cards, making it impossible to run on mobile hardware [33].

3. Reconstruction Pipeline

In this chapter, a pipeline for surface reconstruction using mobile AR tracking data is proposed, including a description of the algorithms that were used for the implementation. It was observed in early testing, that the raw tracking data is not dense enough to allow a precise reconstruction of the real world. Therefore the proposed system is limited to the case of a tabletop scene with multiple separated objects on the table. Those objects can then be used as geometry in the final application. As an input, it takes the point clouds of visual features as they were perceived by the Mobile AR framework over multiple seconds. Those points first get processed to improve the sampling quality. The resulting point cloud is then clustered into multiple smaller point clouds that each represent a single object. Those clusters are each triangulated to obtain mesh geometry that can be used in a game environment.

3.1. Aquiring the Point Cloud

The first step of the reconstruction is accessing the point cloud it uses as input. This point cloud is the set of 3D feature points as they are detected by the Visual-Inertial Odometry system of the used Mobile AR SDK (see 2.1.1). There are around 100-200 of those points detected per frame, depending on the exact scene in front of the camera. The points are scattered over the whole scene, making the point cloud way too sparse to be directly used as a representation of the real scene. Therefore it is necessary to process the input data in order to increase the quality and quantity of the points. Each point is described by a unique identifier, henceforth referenced as the *pointID*, the coordinates in 3D space (x, y, z) at which *location* the point was perceived by the VIO system, and a *confidence value* indicating the system's degree of certainty that the point coordinates are correct.

ARCore allows acquiring a copy of the latest point cloud each frame. Before the data for the reconstruction can actually be recorded, the table plane should be selected using the SDK's built-in plane detection. According to the tabletop scenario, this plane is used to define the playing area in which objects should be reconstructed. During a recording phase, the point cloud is acquired on a frame basis. Each point is checked wether it is positioned above the selected table plane or not to filter out points in the background that are not relevant for the reconstruction. The remaining points are added to an array and saved until the end of the recording phase to be then further processed.

3.2. Point Cloud Processing

After the data recording is finished, the sampled point cloud is processed to increase its quality and precision. As the scene was scanned for multiple seconds, the visual features were also detected multiple times. The points captured in different frames that are representing the same visual features also share the same point ID. Therefore it is possible to combine all samples with the same point ID into a single point with higher precision than the actual samples themselves. This processing needs to be done for every point ID independently, so it can be sped up by utilizing the multiple threads of modern smartphone processors. Therefore the array containing the recorded data is split into individual arrays for each point ID.

Algorithm 1: Combine multiple samples of one feature into a single one

```
input : Data: The points that should be processed
output: A single point with higher precision than the input points
foreach Point ∈ Data do
    if Point confidence < ConfidenceThreshold then
        Remove Point from Data
    end
end
avgLocation ← CalculateWeightedAverage()
while maximum distance of Point from avgLocation > DistanceThreshold do
    Remove Point from Data
    avgLocation ← CalculateWeightedAverage()
end
if Data.Length >= MinSamples then
    return avgLocation
end

Function CalculateWeightedAverage(Data):
    return (∑_Data confidence · location) / ∑_Data confidence
```

Algorithm 1 is run for each of those arrays. The algorithm takes the point data array as input and requires the 3 parameters *ConfidenceThreshold*, *MinSamples*, and *DistanceThreshold*. In a first step, all points with *confidence* value below the *ConfidenceThreshold* are removed, as too low confidence values can have negative impact on the results. Next, the weighted arithmetic mean of the point locations with their respective confidence values as weights is calculated according to equation 3.1. The reason for this is that the points with the same ID are not detected at the exact same location every frame due to inaccuracies in the VIO system.

$$avgLocation = \frac{\sum_{p \in Points} p.confidence \cdot p.location}{\sum_{p \in Points} p.confidence} \tag{3.1}$$

It was observed, however, that sometimes a feature point is detected at two completely

different locations, possibly multiple meters apart from each other. Calculating the weighted average of points with such a huge difference will result in a location somewhere in between the two measurements, that is clearly incorrect and does not represent the originally detected feature. Therefore the algorithm removes points from the calculation that are too far apart from the weighted average. This is done by first finding the point with the greatest distance to the beforehand calculated weighted average location. If this distance is smaller or equal than the *DistanceThreshold*, the algorithm returns the weighted average as new location for the current *PointID* and terminates. If the distance instead is greater than the *DistanceThreshold*, the point being too far away is removed and the weighted average is recalculated for the remaining points. These two steps of finding the point with the greatest distance and potentially removing it are repeated until all remaining points lie within the *DistanceThreshold* of the average. Finally, the *MinSamples* parameter is used to filter out points with too few samples to reduce the number of noisy points in the remaining reconstruction pipeline.

3.3. Clustering

For the next step, the point cloud that resulted from the processing algorithm needs to be clustered into smaller point clouds. The clustering of data is a well-known problem in computer science with a multitude of algorithms developed. In this thesis, the Density-Based-Spatial-Clustering-With-Noise algorithm [34], also known as DBSCAN, was implemented. DBSCAN was designed to detect clusters with arbitrary shapes where the data points have a high density and filters out noise points that do not fit well enough into any cluster. As the objects on the table in our scenario are clearly separated from each other, a density-based approach can reliably produce very good results. Also in contrast to other popular clustering algorithms such as k-means clustering, DBSCAN does not need to know the number of clusters beforehand in order to result in a correct clustering, which is important as the number of objects on the table should be flexible.

The DBSCAN algorithm is described in algorithm 2. It takes an array of points in 3D space as input, as well as the parameters ϵ and *MinPoints*. The ϵ parameter is used to define the so-called *Eps-neighbourhood* of a point P, i.e. the set with all points including P that lie within distance $<= \epsilon$ from P. The *MinPoints* parameter sets the minimum number of points that can form a cluster and is also important for the distinction of the different point classes used by the algorithm. Points are classified as either *core points*, *border points*, or *noise*. Core points are those points with at least *MinPoints* points in their Eps-neighbourhood. Border points have an Eps-neighbourhood that is smaller than *MinPoints*, but include at least one core point. As the name says, border points form the border of a cluster, as they are still close enough to the core to count as part of the cluster, but there are not enough other points to further extend the cluster. Noise is used for all points that are neither core nor border points, so either single points with no other points nearby or small clusters with fewer points than *MinPoints*, so they are not seen as a cluster by the algorithm.

The algorithm iterates over all input points. For a point P that has not been visited before, the *Eps-neighbourhood* is calculated. If P has more than *MinPoints* neighbours, a new cluster is

Algorithm 2: Density-Based Spatial Clustering with Noise

```
input : Data: Array containing the point cloud to be clustered
output: Clusters: Array of clusters
foreach Point ∈ Data do
    if Point not visited then
        Neighbours ← get all points in radius eps around Point
        if Neighbours.Length >= MinPoints then
            Cluster C ← ExpandCluster(C, P, Neighbours)
            Add C to Clusters
        else
            mark P Noise
            mark P visited
        end
    end
end
return Clusters

Function ExpandCluster(Cluster C, Point P, Point[] Neighbours):
    mark P visited
    Add P to C
    Seeds ← Neighbours
    foreach S ∈ Seeds do
        if S not visited then
            N ← get all points in radius eps around S
            if N.Length >= MinPoints then
                Add S to C                          /* S gets new core point */
                Add N to Seeds
            end
        end
        if S = Noise then
            mark S visited
            Add S to C                              /* S is now a border point */
        end
    end
    return C
```

generated with P as a core point, else P is classified as Noise. The cluster is then expanded with all of the neighbours of P. If a neighbour itself has enough neighbours to count as a core point, its neighbours are also added. This process propagates until all points of that cluster are found. If a point is found while expanding a cluster that was formerly classified as Noise, because of too few neighbours, it is now considered a border point of that cluster. The

algorithm continues creating and expanding new clusters until all points have been visited and are either part of a cluster or Noise.

3.4. Mesh Generation

The final step in the reconstruction pipeline is the triangulation of the point clusters to obtain mesh geometry. As the literature research regarding the topic of surface reconstruction (see 2.3) has shown, most triangulation algorithms are not suitable for the given problem. They either rely on additional information about the point cloud such as surface normals, which can not be obtained from the Mobile AR tracking data, or they take a very long computation time in order to improve the quality of the reconstruction results. Even those solutions achieving real-time performance rely on Desktop-PC hardware to do the heavy computational tasks, so they will most likely not work on a smartphone, especially not in real-time. However, a lot of the algorithms use the geometrical properties of the Delaunay triangulation as a base for the reconstruction. Such algorithms usually include one or more refinement steps to achieve better results and further improve the quality of the generated mesh, but due to the input data already being sparse and inaccurate, such a refinement is not really necessary as it probably would not be worth the extra computation time. For these reasons and to limit the complexity of the implementation, the Delaunay triangulation without any additional refinements was used to triangulate the previously computed clusters.

The Delaunay triangulation can be easily calculated with algorithm 3, which is widely known as Watson's algorithm [25]. As it was mentioned in 2.3.1, calculating the Delaunay triangulation for a set of points in 3 dimensions results in an array of tetrahedra. The only input the algorithm needs is the array of points to triangulate, which are the points of one of the previously computed clusters in our case. As a starting step, a so-called *super-tetrahedron* is added, that has to be large enough to contain all input points. The points are then inserted one by one, and a new valid Delaunay triangulation for the already inserted points is generated. Each time a new point is added, all existing tetrahedra are checked, if the new point is contained within their circumsphere and thereby violating the Delaunay property. If this is the case, that tetrahedron is no longer valid and needs to be removed. Removing those tetrahedra results in a polyhedral hole in the triangulation. The border faces of this hole, i.e. those triangles that are not shared by another of the invalid tetrahedra, are then used to create a new tetrahedron with the newly inserted point. After this process, the triangulation is valid again and the next point can be inserted.

After obtaining the array of tetrahedra satisfying the Delaunay condition, the actual triangles that are forming the mesh geometry need to be extracted. Each tetrahedron is made up by 4 triangle faces, but many of those triangles would lie inside the final mesh and are thus not needed for a solid mesh. The only triangles needed are the outer triangles spanning the point cloud. Due to the use of an initial super-tetrahedron, all such border triangles are part of a tetrahedron with the fourth point being a vertex of the super-tetrahedron. That means by iterating over all tetrahedra containing exactly one vertex of the super-tetrahedron and saving the triangle opposing that vertex, an array with the actual mesh triangles can be obtained.

Algorithm 3: Computing the 3-dimensional Delaunay triangulation

```
input : Data: Array containing the points of one cluster
output: Tetrahedra: Array of tetrahedra
Add super-tetrahedron to Tetrahedra
foreach Point ∈ Data do
    Tetrahedron[] invalidTetrahedra
    foreach T ∈ Tetrahedra do
        if Point is in circumsphere of T then
            Add T to invalidTetrahedra
        end
    end
    Triangle[] polyhedron
    foreach T ∈ invalidTetrahedra do
        foreach Triangle ∈ T do
            if Triangle is not shared by another tetrahedron in invalidTetrahedra then
                Add Triangle to polyhedron
            end
        end
    end
    Remove invalidTetrahedra from Tetrahedra foreach Triangle ∈ polyhedron do
        newTetrahedron ← Triangle + Point Add newTetrahedron to Tetrahedra
    end
end
```

In addition to vertices and triangles, meshes in computer graphics usually also make use of vertex normals for topics such as collision resolution or shading. Let v_1, v_2 and v_3 be the triangle vertices and v_4 the vertex of the super-tetrahedron, forming a Delaunay tetrahedron with the given triangle. Then the triangle's unsigned surface normal $\overrightarrow{n_u}$ can be calculated using equation 3.2.

$$\overrightarrow{n_u} = \frac{\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}}{|\overrightarrow{v_1v_2} \times \overrightarrow{v_1v_3}|} \tag{3.2}$$

$\overrightarrow{n} = \overrightarrow{n_u}$ only holds true, if v_1, v_2 and v_3 are ordered clockwise when looking down onto the triangle. As the ordering of the points is not known beforehand, the correct direction of the normal can be calculated with regard to the fourth tetrahedron vertex. According to the initial condition of the super-tetrahedron spanning all other points, v_4 always lies on the outer side of the triangle. Thereby $\overrightarrow{n}$ must roughly point in the same direction as $\overrightarrow{v_1v_4}$, i.e. the angle between the vectors must be smaller than 90°. The angle between two vectors can be computed using the dot product, with positive values indicating an angle that is smaller than 90°and negative values indicating an angle greater than 90°. A value of 0 would mean that the angle is exactly 90°. However, this is not possible, as v_4 needed to lie on the same

plane as the triangle for this to happen and hence the tetrahedron would not be valid in the first place. So the actual triangle normal can be calculated using equation 3.3.

$$\vec{n} = \overrightarrow{n_u} \cdot sign(\overrightarrow{n_u} \bullet \overrightarrow{v_1 v_4}) \tag{3.3}$$

It is important to note, however, that this only holds true for left-handed coordinate systems as they are usually used in computer graphics and was used in the implementation of this thesis. In a right-handed coordinate system, the result of the vector cross product is flipped and thereby also $\vec{n}$ is flipped. So in this case $\vec{n} = \overrightarrow{n_u}$ is correct, if v_1, v_2 and v_3 are ordered counter-clockwise, and $-\overrightarrow{n_u}$ is the correct normal if the vertices are ordered clockwise.

After calculating the normal vectors for all triangles, they can be used to obtain approximate vertex normals. The direction of the normal for vertex v_i is the arithmetic mean of the normals of each triangle $T|v_i \in T$.

This reconstruction step results in a set of vertices, triangles and vertex normals, which can then be used in the implementation to generate mesh geometry at the locations of the scanned and reconstructed objects on the table. As ordinary mesh assets, these meshes can be used for any kind of gameplay interaction.

4. Implementation

This chapter describes the Android application that was implemented for developing and testing the proposed reconstruction pipeline. The application was implemented using Epic Games' Unreal Engine [35] in version 4.22, the latest at the time of writing. Unreal Engine offers native integration of ARCore and ARKit, including a unified AR Framework, making it easy to switch between the two SDKs. Furthermore, Unreal Engine supports building apps targeting mobile platforms, making the development process streamlined and allowing the developer to focus on the actual implementation instead of the building pipeline.

4.1. Application

The application is based on the *Handheld AR* template that is provided by Epic Games as a part of the engine. The template already handles the basic initialization for ARCore and ARKit so it can be built and tested on the device without further setup. The template provides some default functionality to demonstrate the AR SDK's capabilities. The user can tap on any surface that was detected and spawn simple virtual objects such as spheres or cones on top of them. There are some useful debug functions included, such as showing detected planes, the origin of the tracked AR world, or printing information about the current light estimation. The camera image is mirrored on the Skysphere behind the virtual content so the virtual objects appear to be part of the real world. However, most of these template functions were disabled, as they were not necessary for testing the proposed reconstruction system. Only the camera mirroring was not altered, as this is an essential part for every video-seethrough AR application, as they are found on mobile phones.

After launching the app, the currently detected visual feature points and the detected planes are rendered automatically. Every frame, this information is obtained from the AR SDK and the rendering data is updated accordingly. In this state, it can clearly be seen that the user has to move the phone around for more features to be detected. Before he can continue in the application, the user needs to select the plane he wants to use as table plane for the reconstruction pipeline. After selecting a plane by tapping on it, all other planes are hidden to remove visual clutter.

A simple main menu exists to control the application's basic functionality. Within this menu, the user can record the detected visual features as data for the reconstruction, write the recorded data to a file, and actually start the reconstruction pipeline after data has been recorded. There is also a button to start the sample game that was implemented to showcase the reconstruction results. The sample game is covered in depth in section 4.3.

While recording the data, the point cloud of feature points is obtained from ARCore every

frame. Those features which lie above the selected table plane are saved so they can later be processed. After the recording is finished, the complete recorded data can be saved as a .csv file on the phone, so in the case of any inconsistencies, the data can be transferred to a PC to track down potential bugs. For this purpose, it is also possible to start and test the application on a PC itself and load such a file with recorded data, as it can't be generated on the PC directly. After recording or importing the data, the actual reconstruction process can be started, as it is described in chapter 3.

4.1.1. Configuration Menu

Also accessable from the main menu, a configuration menu has been implemented. In this menu the user can adjust the parameters *ConfidenceThreshold*, *MinSamples*, *DistanceThreshold*, *Epsilon* and *MinPoints* that are used for the algorithms 1 and 2. The *ConfidenceThreshold* can be controlled by a slider to set any value between 0 and 1. The other parameters can be set directly via a text input field. *MinSamples* and *MinPoints* are then sanitized as integers, *DistanceThreshold* and *Epsilon* as floating point numbers.

As the recorded data is saved even after the reconstruction is finished, the reconstruction can easily be recomputed after making changes to the parameters, hence altering the results. By repeating this process we tested the parameters and extrapolated default values which provided good results in the testing environment. Those default values can be seen in table 4.1.

Table 4.1.: The extrapolated default values for the reconstruction parameters with ConfidenceThreshold c, MinSamples S_{min}, DistanceThreshold d, Epsilon ϵ, and MinPoints P_{min}.

c	S_{min}	d	ϵ	P_{min}
0.4	5	3.0 cm	3.0 cm	5

The *ConfidenceThreshold* should not be set higher than 0.4, except when there are a lot data points present, e.g. due to a long recording time. In a short recording, there are usually not many points with relatively high confidence, as the VIO system itself takes some time to gain enough confidence about the real world. Setting this parameter too high would produce significantly less sample points which would result in worse reconstruction results and might even affect the correct clustering of the points. This should also be kept in mind for *MinSamples*, as some actually useful samples might be discarded if it is set too high. Setting it too low, on the other hand, would create a lot of noise in the data, as points would be included that just appeared for a very short time and are therefore most likely not an important part of the scanned object.

The *DistanceThreshold* was not very easy to test and set to a good value. The error of a point being detected at 2 very distinct locations is not reproducible and only appears sparsely. However, it should be mainly dependent on the margin of error of the VIO system. If the system introduces a lot of locational drift due to inertial measurement errors, it should be

considered to set this parameter to a higher value. There were no noticeable different results observed with higher values than 3 cm, only with values smaller than 2 cm, where many more points were filtered out.

Epsilon is probably the parameter that needs to be adjusted the most, as it highly depends on the objects and the scene that is scanned. The objects' feature points need to be dense enough, else the clustering algorithm might split an object into 2 parts. On the other hand, the distance between the objects needs to be greater than *Epsilon*, otherwise those objects will be merged into a single one, that also spans the gap between the real objects. The given value of 3 cm achieved good results for a setup with few objects and enough space in between. However, on a more crowded table it often happened, that the features of multiple objects got merged into one big point cloud.

Finally, *MinPoints* needs to be set to at least 4 to avoid the generation of clusters with less than 4 points in them, as for such clusters no valid triangulation could be achieved. With the given value there are still some very small clusters of actual noise points created and then triangulated, but they can be easily filtered out in case they have a negative impact. Setting the parameter to a higher value, however, leads to missing information in the reconstruction for objects with a rather sparse sampling density, due to many points being discarded as noise.

4.1.2. Mesh Collision Testing

The collisions of the generated meshes are handled by Unreal Engine's internal physics engine. In order to test the collision, virtual balls can be thrown via another button in the main menu. The balls are spawned at the location of the camera and are thrown along the camera view direction. As they are affected by gravity, the balls fly in a ballistic curve away from the camera. When hitting the reconstructed geometry, the balls will bounce off according to their current velocity vector and the object's surface normal on the point of impact.

4.2. System

Unreal Engine offers two different ways of programming game logic: C++ as a traditional programming language and the *Blueprints* Visual Scripting system. With Blueprints gameplay elements can be implemented using a node-based interface. Blueprints scripts are able to access most of the engine's gameplay scripting API. They are compiled and executed in a special virtual machine, allowing faster iteration times than traditional code at the cost of computation speed. This makes Blueprints the perfect choice for prototyping and light game logic. Our application was developed in a hybrid approach of Blueprints and native code. Blueprints were used for high-level application logic and all User Interface elements. The data recording and reconstruction algorithms were completely implemented in C++, along with all file logging functionality. This was done due to the higher performance C++ code can offer in comparison to Blueprints, along with the fact that Unreal Engine's asynchronous and multi-threaded computation is only accessible via the C++ API.

4.2.1. Multi-Threaded Computation

Usually in a game engine, all code is executed inside the main game thread to assure correct sequential execution of all game logic. This means that executing a function that takes longer than the time of one frame to complete will halt the remaining game logic and rendering until the function is finished. This can cause anything from small hitches up to the game actually appearing to freeze, which is not desirable as it impacts the user experience negatively. The solution to this problem is to run such complex functions on a different thread than the game thread, so game logic and rendering can continue as usual. This also needs to be done for the various calculation steps of the reconstruction pipeline. As the evaluation in chapter 5 shows, the individual processing steps can sometimes take multiple seconds to complete. Even the lowest measured times are a multitude larger than one 60th of a second which is the optimal frame time for a fluent gaming experience [36]. Therefore the whole reconstruction functionality should be run in a separated thread. This also allows to utilize multiple threads and make use of a multi-threaded system to speed up the reconstruction.

In Unreal Engine, such asynchronous code can be written via the use of *Async Tasks* [37]. To utilize them, a child class of an AsyncTask needs to be created. In this child class, the logic to be run asynchronously and the input parameters needed to initialize the class can be defined. During runtime, an arbitrary amount of these tasks can be instantiated. A new thread is created for each of those tasks and the engine internally handles assigning the necessary computing power to them. Once the computation of a task is done, there is a callback to the game thread, so its results can then be used.

In our application, all steps of the reconstruction are computed in Async Tasks so they do not block the game thread, but only the triangulation step creates multiple tasks to run in multiple threads in parallel. As it was mentioned in chapter 3, the point cloud processing could as well make use of a multi-threaded implementation. However, it does not take very long compared to the other steps anyway, so the small performance gains would not be worth the overcomplication induced by ensuring the completion of every single task before the next step can be started. Such an insurance is not necessary for the triangulation step as there is no need to wait for the other tasks to finish. Instead, each generated mesh can just be spawned once its task is done.

4.2.2. Procedural Mesh Generation

The established workflow for getting a mesh asset into the engine is via the use of special file types such as *.fbx*. Game Artists create the needed meshes in a 3D modeling program such as *Blender* or *Maya* and export a *.fbx* file from there. These files can be imported into the game engine to later use the mesh in the game. However, to create and load meshes during runtime, once the application is built and deployed to the device, a more dynamic solution needs to be used. For this purpose, Unreal Engine offers another way of creating meshes at runtime in the form of the *Procedural Mesh Component* [38]. As the name implies, this component can be used to create a procedural mesh. The created mesh is then asynchronously processed by the engine to generate collision information for it.

To generate a mesh at runtime with this component it requires an array of vertex locations and an array of triangles. The triangle array's length needs to be a multiple of 3, as every 3 consecutive entries in the array refer to the vertex indices that define a triangle. The component can also take more optional information in the form of vertex normals, texture coordinates, vertex colors, and tangents. All those optional information is supplied for each vertex, so their respective array's lengths need to match the number of vertices, else it would be completely discarded and default values are used instead. In our implementation, only the vertex normals (see section 3.4) were supplied. As the mesh's purpose is to provide physical interaction and not to be rendered, it is not necessary to use either vertex colors or texture coordinates for its generation. The vertex tangents would only be used for some special shaders such as bump-mapping shaders, so they were also omitted.

4.2.3. Occlusion

To create the illusion of virtual objects being part of the real world, the camera image is rendered on the Skysphere as a background to the virtual scene. By this approach, however, virtual objects are always rendered on top of the real objects, even if they are actually positioned behind them. As already mentioned in section 2.2, this can have a negative impact on the user's immersion and introduces confusion due to missing occlusion. The generated meshes can be used to mitigate this problem by correctly occluding virtual objects behind the real ones. To achieve this, a custom occlusion material was implemented. Figure 4.1 shows this material inside Unreal Engine's node-based material editor. The material accesses the camera image that is also used as the scene background and displays it at the screen position of the generated mesh. To be indistinguishable from the background image, the material is set to not react to the scene lighting.

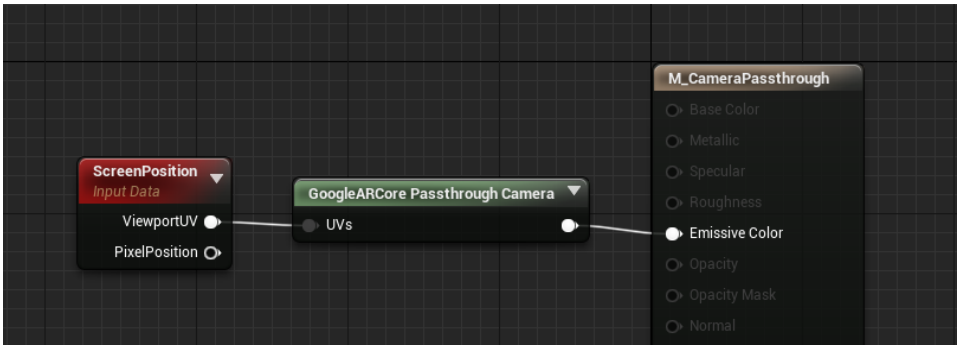


Figure 4.1.: The occlusion material inside Unreal Engine's Material Editor.

Applying this material to a mesh in the AR scene makes it look invisible, as it is rendered the exact same way as the background. In contrast to the background, however, the mesh still has a location in the scene and is rendered on top of other virtual objects that are positioned behind it, thereby occluding them with the camera image of the real world. Given that the reconstructed mesh exactly matches the topology of the real object, the use of this material

can create the illusion of the real object occluding virtual objects. An example of the result when using this material can be seen in figure 4.2.

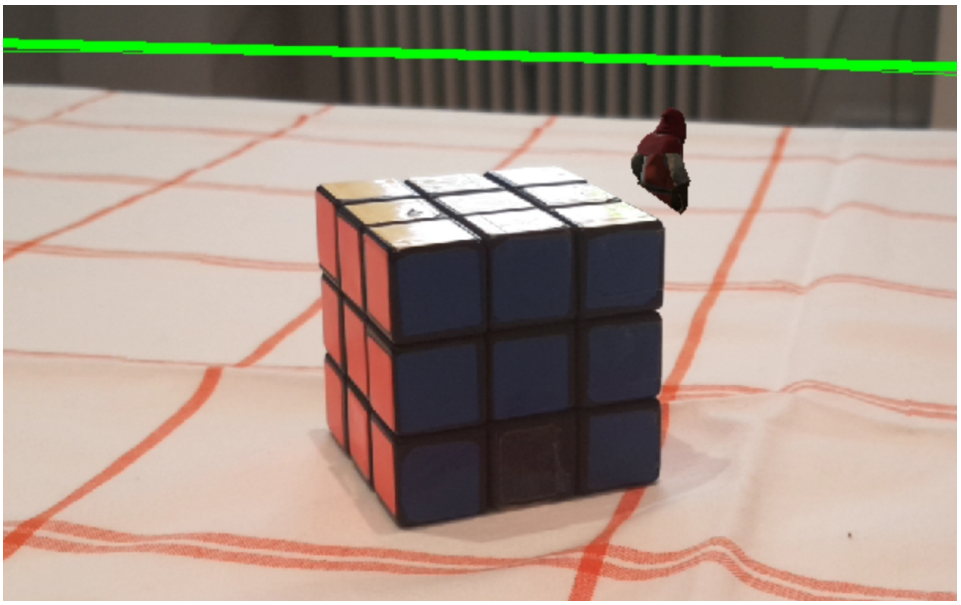


Figure 4.2.: The occlusion material is used to partially occlude the archer with the background image, making him appear to stand behind the cube.

For debugging purposes a second material has been created, that just displays the meshes in a plain white color. This material is set to respond to lighting, so triangles are rendered with different brightness levels according to their surface normals. Thereby it is easier to recognize the mesh topology when using this material. A button in the application's main menu can be clicked to change between these two materials.

4.3. Sample Game

A sample game was implemented to showcase the usage of the proposed object reconstruction pipeline. The game is a rudimentary member of the *Artillery* genre. It is a turn-based multiplayer game for two players on a single phone. Each player controls an archer that can move freely around on the table and shoot projectiles. The objects that were placed on the table need to be navigated around and can be used as cover from enemy projectiles. Detailed Screenshots showing the game can be found in appendix B.1.

When starting the game, the first player needs to choose a starting location for his archer. This can be done by tapping anywhere on the table plane. After the soldier is spawned, the player is prompted to pass the phone on to the second player. He needs to select a starting location as well.

After both players have spawned their archers, the actual game starts and it is player 1's turn. A player has three actions available during his turn: Moving, Shooting, and ending

his turn. The player can move his archer around by tapping anywhere on the table plane. The archer will then move towards the tapped location in a straight line. If he encounters an obstacle on the way, for example one of the generated meshes or the other player, he stops short of his goal. To shoot an arrow, the player needs to aim with his phone. The archer will always aim towards the intersection point of the table plane and the camera view direction. In addition to aiming, the player needs to hold down the "Fire" button to control the launch strength of the arrow, the arrow fires on release. Next to the Fire button is an indication of the current strength. While charging, a prediction for the first few centimeters of the projectile path is also shown in form of a dotted line. After shooting, the full projectile path is shown and the hit result is displayed via a text message.

Figure 4.3.: A Screenshot from within the sample game with both players facing each other next to the reconstructed objects on the table. In the upper left corner the player is informed about his successful hit. The projectile path is shown in the form of the big red dots.

The total distance a player can move per turn is limited, and he can only fire a single arrow, so each action should be planned carefully. The remaining movement distance is displayed in the top left corner of the screen and the Fire button changes its color to red once the shot is fired. The intended way of playing with these limitations is to first walk out of cover, shoot at the enemy and then walk back into cover. Due to the movement restriction players are forced into worse positions after taking a good shot, which should ultimately prevent a stalemate of both players not being able to hit one another without standing in the open after their turn.

5. Evaluation

This chapter describes the results of the implemented reconstruction pipeline. Furthermore, the quality of the reconstruction and the performance of the various processing steps are measured and analyzed. Finally, the given results are discussed with regard to the strengths and weaknesses of the proposed system. Performance measurements in this chapter were taken on a Huawei Honor 10 running Android 9.0 Pie. The smartphone is powered by a HiSilicon Kirin 970 CPU (4x 2.4 GHz and 4x1.8 GHz), 4 GB RAM and an ARM-Mali-G72-MP12 GPU, which represents similar performance as most last-generation flagship smartphones at the time of writing.

5.1. Evaluation Method

The evaluation of surface reconstruction techniques can be quite challenging. One part of most evaluations is measuring the performance of the system. This can be done by reconstructing the surface from a specific data set and measuring the time the algorithm needs. Repeating this step with other algorithms as well results in easily comparable performance numbers. However, as mentioned before, the actual performance of most surface reconstruction systems only plays a subsidiary role. The more important property to evaluate is the quality of the reconstructed mesh, i.e. how closely the mesh matches the real object.

Due to the broad variety in surface reconstruction approaches and their differentiating application areas, there is no standardized way of measuring this quality. Probably the most common and obvious method is to directly compare the reconstruction result to the surface from which the input data was obtained [21]. However, with most data sets this cannot be done easily, as there is no correct digital representation of the real-world object which could be used to compare the own reconstruction results to. Other approaches to evaluate the reconstruction quality completely omit an automatic measure for it. Instead, similar to the performance evaluation, they use the same data set to run multiple reconstruction algorithms on it [39]. The results are then compared visually to identify the areas where one or the other approach provided better results.

However, such approaches to the evaluation as they are described above are not feasible to be applied to our proposed reconstruction pipeline. Due to the limited scanning density of the mobile AR point cloud and the device's limited computation power, our results should not be directly compared to those of other algorithms. Instead, we decided to adopt the general evaluation criteria of quality and performance and defined our own measures for them. In the following, the method of acquiring those measurements is described in detail.

To measure the quality of the reconstruction, we decided to use our application to recon-

struct a single object with simple geometric properties. Due to the object's simple geometry it is easy to obtain a correct digital representation of the object to compare our reconstruction results to. This digital representation can also be referred to as the ground-truth. After the object has been scanned and reconstructed on the smartphone, the data was saved to the device and then transferred to a PC. Inside Unreal Engine, this data was loaded next to the object's ground-truth mesh. Both meshes were then placed at the same location and aligned to the coordinate axis so they could be compared by a script that was written for that purpose. The script fires a bunch of raytraces along the direction of one of the coordinate axis against both meshes, measuring the distance between them. The raytraces are ordered in a grid with a spacing of 0.1 cm between each of them. Such a grid of raytraces is fired a total of six times, twice for each coordinate axis, and for each axis in positive and negative direction respectively. By recording the distance results of all raytraces that hit both meshes, the mean and the median of the absolute distance between the reconstruction result and the ground-truth as well as the standard deviation can be calculated.

The performance measure was split into two parts. The measurements were conducted on the afore-mentioned smartphone, so the obtained results are significant for a possible consumer application. First, the same object that was used for quality measures was reconstructed, recording the time needed for each calculation step. Additional statistics regarding the size of the input and output data for each step were gathered. These statistics include the number of processed points, the size of the cluster for the sample object and the number of triangles in the final mesh. In the second part of the performance evaluation, a more complex scene consisting of multiple objects on a table was scanned and reconstructed. Mostly the same data as in the first step was recorded, the differences are described in the following. Due to the triangulation task being run in parallel, time and triangle count are not measured for each object individually, but rather the time until all objects were triangulated as well as the number of reconstructed objects, as those numbers should be more meaningful for the given scenario. It was observed that in most cases a single cluster took a lot longer to triangulate than the rest, hence the number of points in that cluster is also recorded.

The tests were conducted on a test setup under preferably good conditions. The table was positioned in a brightly lit room and covered by a white table cloth with a checked pattern, to offer good contrast and a decent amount of feature points for the VIO system and plane detection to work well. Also, the test objects were selected by the criteria to offer many feature points themselves to ensure that enough data for the reconstruction can be collected.

5.2. Results and Discussion

Our object of choice for the quality evaluation was a Rubik's cube. As it is demonstrated in this section, our system is very limited in the amount of small details it can capture. Therefore the rounded corners and small gaps on the sides are ignored and the Rubik's Cube is approximated with an actual cube. For the quality evaluation, the cube's edge length was measured to be 5.6 cm. This length was then used to scale a cube mesh in the engine to match the size of the real one. Figure 5.1 showcases an example reconstruction result for the Rubik's

Cube. Before the reconstruction, the cube was scanned for around 3-5 seconds, which was long enough to move the phone around the cube once to obtain feature points from all sides. Longer scanning times resulted in a lot of noisy points being captured, which sometimes also altered the shape of the reconstructed cube.

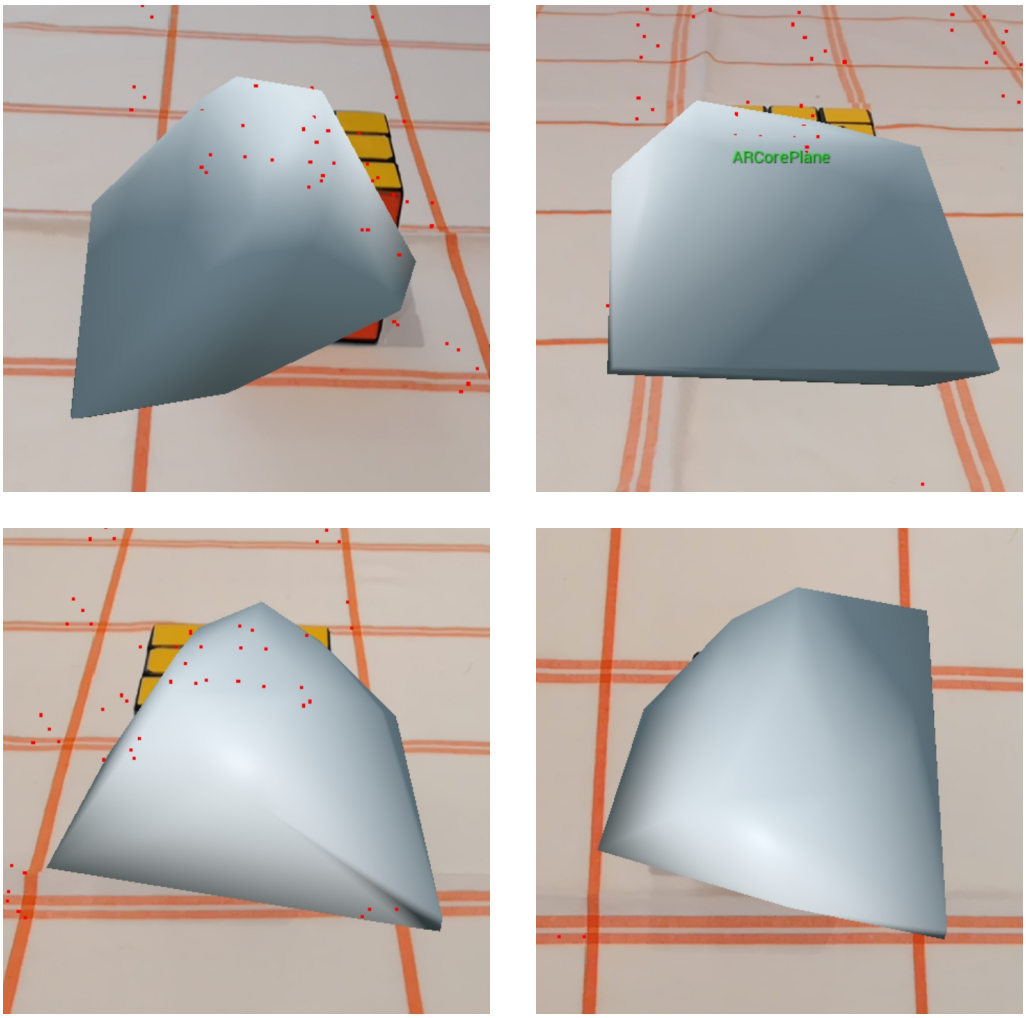


Figure 5.1.: A reconstructed mesh for the Rubik's Cube. a) Front view. b) Side view. c) Back view. d) Top view.

For the performance measurements, a more complex scene has been reconstructed. Multiple different objects have been placed on the table. The exact scene that was used can be seen in figure 5.2, along with screenshots of the reconstruction result. This time, the scene was scanned for about 10 seconds while moving the phone slowly around the table once. In figure 5.2 d) a lot of extra meshes were reconstructed. This was observed to have happened due to the table plane being initially detected too low and therefore many feature points that actually lie on the table were recorded as being above the plane. Those feature points then

satisfied the *MinPoints* condition and valid clusters were formed. Therefore these noisy extra objects were reconstructed.

Those feature points were then enough to form valid clusters with regard to the clustering parameters and be reconstructed themselves.

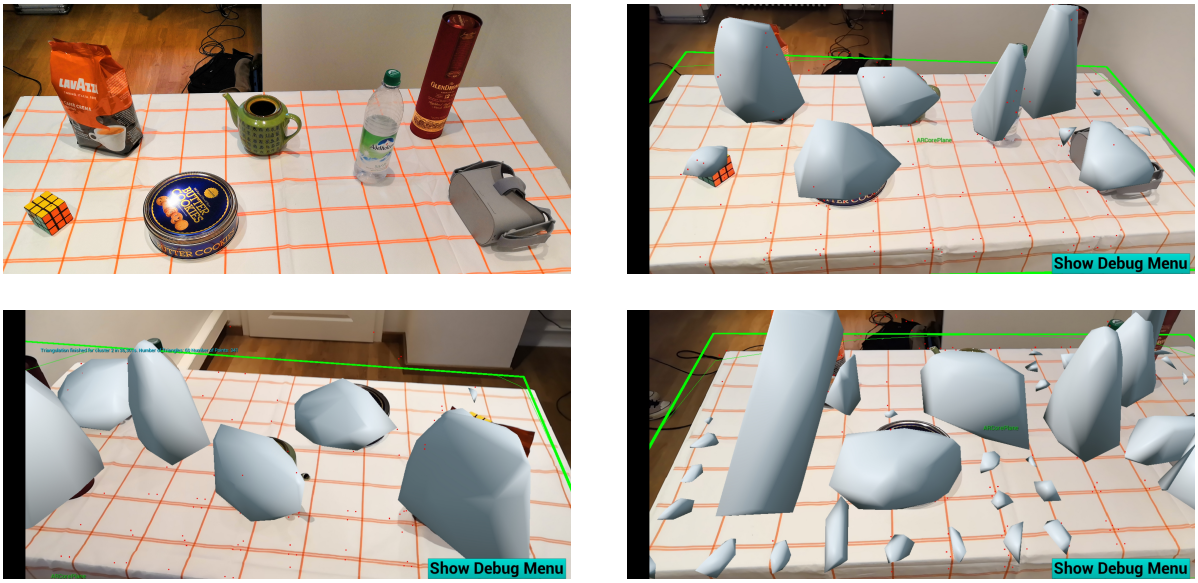


Figure 5.2.: Reconstruction result for a scene with multiple objects. a) The complete scene setup. b) Front view of the reconstructed scene. c) Back view of the reconstructed scene. d) Example for a very noisy reconstruction result.

5.2.1. Quality

For the quality measurements, the process described in 5.1 was repeated a few times to obtain a variety of different scans of the same scene. In addition, each recorded point cloud was reconstructed three times with the *confidenceThreshold* parameters of 0.4, 0.6 and 0.8, as this parameter strongly influences the number of points in the point cloud after the processing step. Please refer to appendix A.1 for the full table with all evaluation data.

To make a statement about the reconstruction quality based on the measured metrics, they first need to be set in relation to the reconstructed object. The Rubik's Cube that was used has an edge length of 5.6 cm. As the distance measures are taken from all sides, a mean distance of 1.0 cm means, that the reconstructed mesh could be a cube with edge length 7.6 cm. In the best measurements, the mean distance was as low as 0.6 cm, in the worst case up to 2 cm. This confirms the visual impression, that the reconstructed mesh is not a very accurate representation of the real object. It should also be noted, that the standard deviation tends to be quite large in relation to the mean distance. This means, that the actual measured distances fluctuate a lot around the mean distance, so the reconstruction might be pretty accurate in some regions, while it is extremely far from the real object in others. The high

Table 5.1.: The results of the quality evaluation for the Rubik's Cube with mean distance between real object and reconstruction $\bar{d}$ (in cm), median distance $\tilde{d}$ (in cm) and standard deviation of the distance measurements σ (in cm).

#	$\bar{d}$	$\tilde{d}$	σ
1	0.61	0.60	0.41
2	1.27	1.12	0.89
3	1.82	1.73	1.42
4	1.77	1.29	1.83
5	1.57	1.46	1.18
6	0.80	0.67	0.65
7	0.64	0.61	0.46
8	0.96	0.73	0.89
9	1.33	0.88	1.36
10	1.24	1.19	0.86
11	1.33	1.20	1.11
12	0.67	0.50	0.72
13	1.68	1.41	1.13
14	1.11	0.73	1.12
15	1.24	0.84	1.30
16	1.92	1.66	1.42

standard deviation also indicates, that the reconstructed object is not particularly cubic, but rather shapeless.

Another observation that is worth pointing out is that the median distance between the reconstruction and the ground-truth is smaller than the mean distance in all of the measurement series. This implies that the individual distance measurements are not normally distributed. Instead, many of the measured distances are actually smaller than the computed mean, while the distance at a few sample points is a lot larger. This can also be confirmed visually, as for example the lower-left corner of the mesh seen in figure 5.1 a) is a lot further away from the cube than other parts of the reconstruction. As the Delaunay triangulation that is used for the reconstruction always produces the convex hull around the point cloud as a mesh, those strong outliers can only be located outside the object. Such outliers expand the mesh in a large area, leading to a tendency of the reconstruction being larger than the real object instead of smaller. This tendency could also be noticed and confirmed visually during the evaluation. A smaller reconstruction than the real object can only happen in regions of the object where the sampling density was too low and hence all feature points in that area were discarded as noise.

Table 5.2 shows the averaged quality measurements sorted by the three different confidence threshold values that were used during the evaluation. This data should be taken with a grain of salt, as the overall sample size for the confidence thresholds was quite low. Nonetheless, the data shows, that the confidence threshold value of 0.6 provided the most accurate

Table 5.2.: The averaged results for the different confidence thresholds used for reconstructions with confidence threshold c and average mean distance between real object and reconstruction $\overline{d}$

c	$\overline{d}$
0.4	1.31
0.6	1.03
0.8	1.49

reconstruction with a mean distance of about 1.0 cm across the different measurement series. As the other two values achieved less accurate results, this indicates that there might be a local minimum of the reconstruction error for a confidence threshold of around 0.6. Note, however, that this observation stands in contrast to the default value for the confidence threshold that was extrapolated in section 4.1.1. This discrepancy is probably a result of the test setup providing more and better feature points than the setup used during development, raising the need to filter out points more aggressively. Having to adjust the reconstruction parameters in dependence of the environment and setup in which the application is used states a substantial problem. While it is no big deal to use the configuration menu for this purpose during development, in a consumer application the user should not be bothered with such technical detail for the app to work properly. Therefore it is necessary to design an efficient way to automatically determine the best parameter values during runtime, instead of setting them to a fixed value during development.

An important aspect to note is that the proposed system is also highly dependent on the used mobile AR framework, ARCore in our application. The only input needed for the object reconstruction is the point cloud of visual features that was detected by the SDK's internal VIO system. Due to technical limitations, the VIO system is not able to detect sufficient feature points on reflective or uniformly colored surfaces, or if the environment is only dimly lit. In such cases, the system will not be able to produce a good reconstruction of the real objects, as it just has too few and inaccurate sampling data.

Another issue with ARCore's VIO system that we encountered were situations where the room-scale tracking was completely lost for a short amount of time and then recovered at the wrong location. When this happens after the scene has already been scanned and the objects reconstructed, an additional locational offset between the real object and its reconstruction is introduced, leading to completely inaccurate collision and occlusion behavior. Figure 5.3 shows some examples for this offset. These issues primarily occurred after the camera image was not able to detect any feature points for a few frames, maybe due to its lens being covered, the camera only capturing a plain white wall, or the application being halted for a short amount of time, e.g. when taking a screenshot or switching to another application.

While evaluating the reconstruction quality, we also encountered an interesting phenomenon that has the potential to flaw the entire process of using the VIO tracking data for a better scene understanding. In figure 5.4, the red dots are drawn at the locations of the currently detected feature points, as they can be accessed from ARCore. According to

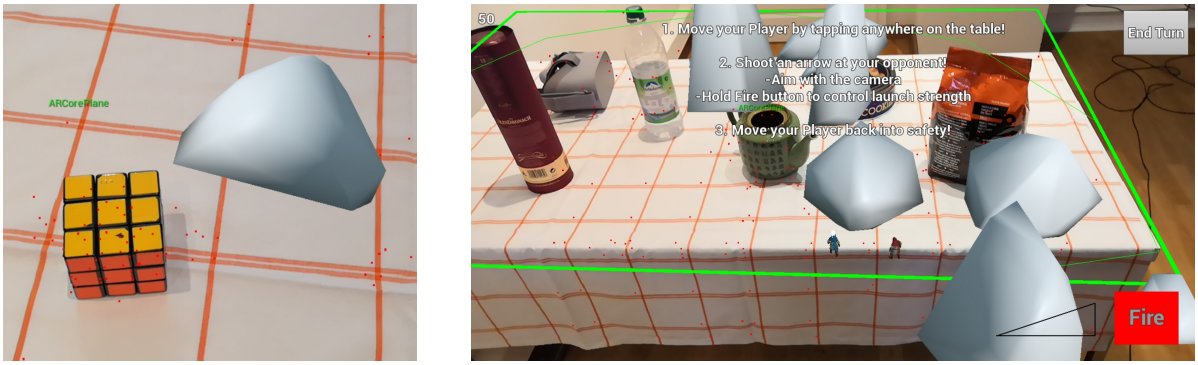


Figure 5.3.: Examples for the real and virtual world being misaligned after AR tracking was lost for a short amount of time.

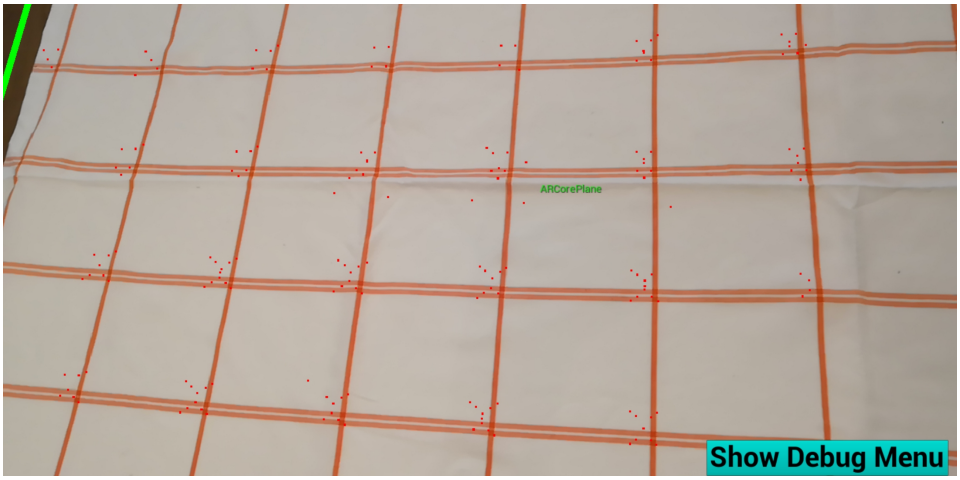


Figure 5.4.: The feature points detected by ARCore (red dots) are not located at the location of the actual visual feature, but slightly offset.

Google's documentation, the detected feature points are visually distinct points in the camera image [12]. This statement leads to the assumption, that the feature points in the given image should be positioned on the locations with the highest contrast, i.e. the corners where the lines of the table cloth intersect. However, some of the feature points are located amidst the white squares, a region that can not be described as visually distinct. These offset feature points appear repeatedly on many of the line intersections, so it is not just a singular error. We even encountered a similar offset for some of the feature points on the Rubik's Cube, where they appeared slightly above or outside the cube, hence causing some inaccuracies in the reconstruction. We have come up with two possible explanations for these offsets: Either ARCore detects feature points that are not visible to the human eye, which is fairly unlikely due to the limited resolution of smartphone cameras, or the location that is saved for the feature points is not the actual location of the feature but rather some kind of anchor point

near the feature, that might be needed for internal VIO calculations. If the second explanation is true, the feature points achieved from ARCore cannot be reliably used for precise scene reconstruction, as they are not necessarily a part of the real object's surface. However, this does not state a big problem for our current reconstruction approach, as it suffers from greater inaccuracies than those imposed by such offset feature points.

5.2.2. Performance

In the first part of the performance evaluation, the time it takes the system to reconstruct a single object was measured. For this purpose, the same Rubik's Cube as before was used. Table 5.3 shows the time measurements of a few reconstructions and the respective amount of data that was used. The application was restarted between each of the recorded measurements.

Table 5.3.: Calculation times for the reconstruction of a single object with point cloud processing time t_p (in s), number of processed points N_{pP}, clustering time t_C (in s), number of points in the cluster N_{PC}, triangulation time t_Δ (in s), number of triangles N_Δ and the total time needed for the reconstruction t_Σ (in s). $\overline{x}$ depicts the mean values for each column.

#	t_p	N_{pP}	t_C	N_{PC}	t_Δ	N_Δ	t_Σ
1	0.9	46	0.1	44	0.6	36	1.6
2	0.1	157	0.9	103	0.15	54	1.15
3	0.2	277	0.28	187	0.9	62	1.38
4	0.22	274	0.22	188	0.37	42	0.81
5	0.1	98	0.15	89	0.78	44	1.03
6	0.12	267	0.1	137	0.28	52	0.5
7	0.2	248	0.7	76	0.11	38	1.01
$\overline{x}$	0.26	195	0.35	118	0.46	47	1.07

In most cases, the application took around 1-1.5 seconds to completely reconstruct the cube. This time frame is short enough to keep the user's attention inside the application and not distract him too much due to long loading times. It appears that the triangulation step tends to take longest, while the point cloud processing is the fastest of the reconstruction steps.

It should be noted, that for each calculation step there were measurements that took an unusually long time compared to the average time. For the point cloud processing, this outlier was sample 1, for the clustering samples 2 and 7, and for the triangulation samples 3 and 5. These outliers did not occur in correlation with exceptionally high numbers of input data, so there might be another reason for such spikes in calculation time. A probable cause for them could be the use of async tasks (see 4.2.1) to decouple the reconstruction from the game thread. The other threads that are created for the calculation steps are presumably run with a lower priority so they do not take up valuable game resources. This might lead to scheduling issues that further delay the execution of that thread, especially while other

apps on the phone demand processing time as well. This could be confirmed by repeating the taken measurements multiple times to rule out any issues with the data sets, as it was done for some recordings of the complex scene setup in the second part of the performance evaluation.

The DBSCAN clustering algorithm has a runtime complexity that is slightly higher than linear [34], while the triangulation algorithm runs in $\mathcal{O}(N^{2.5})$ [25]. In theory, the point cloud processing algorithm has a worst case complexity of $\mathcal{O}(N \cdot m^2)$, with N being the number of different pointIDs and m the number of samples per point ID. However, the number of input data stays in the same order of magnitude for all reconstructions, so the complexity is not as relevant as the actual factor that gets applied to it for each algorithm. Unfortunately, due to the low sample size, it is not possible to extrapolate any correlation between the size of the input data and the resulting calculation time for any of the calculation steps. Therefore it is also not possible to make any further statements about the expected performance of our application for other reconstructions.

In the second part of the performance evaluation, the time needed to reconstruct the more complex scene that can be seen in figure 5.2 was measured and analyzed. The results of these measurements are shown in table 5.4. Some of the data recordings were reconstructed twice with slightly different clustering parameters to change the number of clusters. As before, the application was restarted between each of the recordings.

The reconstruction of the complex scene took longer than the reconstruction of a single object, which should be no surprise due to the fact that more objects have to be reconstructed. The mean and median values show, that the processing time t_p and the clustering time t_C did not change noticeably in comparison to table 5.3. The triangulation time t_Δ, on the other hand, increased a lot, taking multiple seconds on average to triangulate all clusters. Naturally, this also increased the total time needed to reconstruct the scene. On average, the full reconstruction time t_Σ is still low enough to be accepted by the user as necessary and keep his attention. In the worst cases, however, reconstructing the scene took more than half a minute. We even experienced one case, where the triangulation algorithm did not terminate at all, though we were not able to track down the cause for this. These worst case times are well beyond the time where users will lose their attention and probably uninstall the app immediately due to the waiting time [40].

As it was already observed with the Rubik's Cube, sometimes one of the calculation steps took a significantly longer time than the average. For t_p and t_C these outliers are in the same magnitude as they were before despite the far larger datasets, indicating that the previous assumption of this being a scheduling issue was correct. This assumption is further strengthened by the recorded times in lines 8 and 9. Both were taken from the same dataset, with only the clustering parameters being altered between them. The point cloud processing was the exact same for both reconstructions, but still t_p is nearly ten times higher for measurement number 9.

The same kind of outliers also appeared for t_Δ, however, the spikes were a lot higher here. This can easily be seen with the mean $\overline{t_\Delta}$ being more than four times higher than the median $\widetilde{t_\Delta}$. The scheduling issue observed for the other calculation steps is probably also at least in

Table 5.4.: Calculation times for the reconstruction of a sample scene consisting of multiple objects with dataset number D, point cloud processing time t_p (in s), number of processed points N_{pP}, clustering time t_C (in s), number of clusters N_C, triangulation time t_Δ (in s), number of points in the cluster that took the longest to triangulate N_{PC} and the total time needed for the reconstruction t_Σ (in s). $\overline{x}$ depicts the mean value and $\widetilde{x}$ the median value for each column.

#	D	t_p	N_{pP}	t_C	N_C	t_Δ	N_{PC}	t_Σ
1	1	0.1	1141	0.73	14	0.26	67	1.09
2	2	0.17	2311	0.21	36	6.96	434	7.34
3	3	0.1	1436	0.1	19	1.48	403	1.68
4	3	0.16	1436	0.22	11	1.37	404	1.74
5	4	0.12	1793	0.19	15	1.41	340	1.72
6	4	0.14	1793	0.34	8	35.3	341	35.78
7	5	0.19	1909	0.42	14	21.44	315	22.05
8	6	0.1	1393	0.13	17	12.73	282	12.96
9	6	0.96	1393	0.17	12	0.42	282	1.56
10	7	0.87	929	0.86	8	0.95	199	2.68
11	8	0.11	1029	0.92	10	0.31	178	1.34
12	9	0.25	2409	0.33	42	0.63	333	1.2
13	9	0.31	2409	0.61	27	8.95	431	9.86
14	10	0.1	1313	0.15	11	3.49	348	3.74
15	11	0.11	1424	0.19	15	0.33	345	0.63
16	12	0.12	1475	0.16	14	13.11	307	13.4
$\overline{x}$		0.24	1600	0.36	17	6.82	313	7.42
$\widetilde{x}$		0.13	1436	0.21	14	1.44	337	2.21

parts responsible for these spikes, however, as the effect is a lot stronger here, we suspected that there might be another issue. To investigate this, some datasets with a high triangulation time were reconstructed multiple times without changing any reconstruction parameters. The measurements from this test are not included in table 5.4 as they would corrupt the mean and median calculations, but can be found in appendix A.1. If long calculation times are caused by a scheduling issue they should be significantly shorter in the repeated reconstructions, as it was seen with t_p before. However, in all of those repeated reconstructions, the triangulation steps took roughly the same time as in the original measurement, hence indicating that there is another issue causing the high calculation time in addition to the scheduling issue. Figure 5.5 shows that extremely high t_Δ values only appeared for a large number of points that were triangulated, but there are also clusters with a high number of points that were triangulated really fast. These findings lead to the conclusion, that the problem is not bad scaling of the algorithm with large input data. We rather suspect the existence of some degenerate cases in the point cloud data where our specific implementation of the Delaunay triangulation struggles for some reason and takes a lot longer to terminate. Fixing this issue

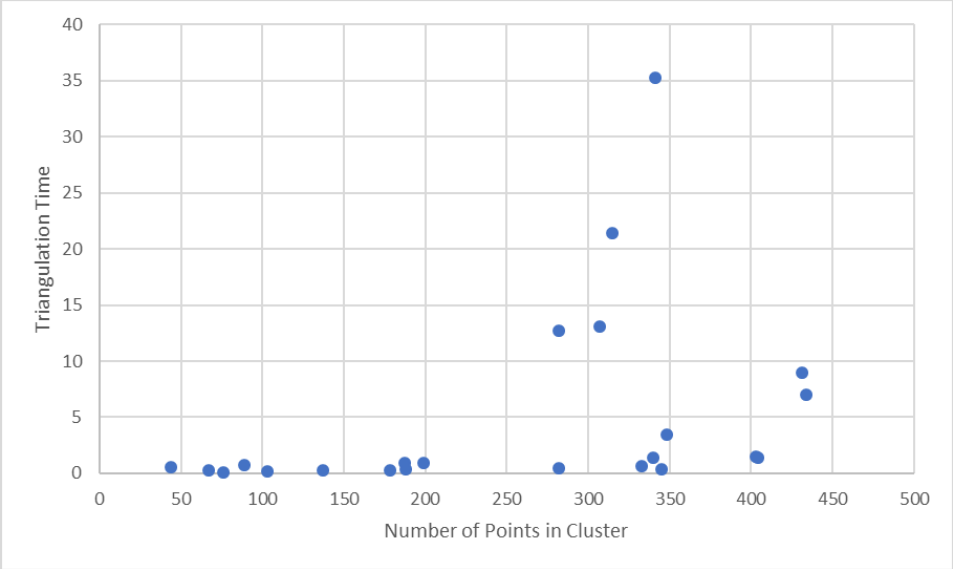


Figure 5.5.: The relation between the number of points in a cluster and the resulting triangulation time. Data used from both parts of the performance evaluation.

by reworking the implementation may bring the total reconstruction time for a complex scene consistently down to about 2-3 seconds, which should be far more acceptable for users than the currently measured times.

6. Conclusion

6.1. Summary

The main goal of this thesis was to enhance the current capabilities of the existing mobile AR frameworks by generating a more detailed scene understanding that can then be used in a game environment. This should be achieved without the use of any specialized hardware, instead only using the internal tracking data of the AR framework. Both of these goals were achieved, but it has to be kept in mind that the used approach heavily limited the application area by putting hard constraints on the scene that can be reconstructed. Nonetheless, our solution should be seen as a proof of concept that a better scene understanding can be achieved on current smartphone hardware.

For the limited scenario of a tabletop game with a few arbitrary objects on top, our proposed solution was able to provide a decent approximation of those objects and generate a mesh that could be used for collision calculation. The reconstructed mesh does not claim to be a realistic representation of the real object, so the collisions should not be expected to realistically reflect the real collision behavior as well. Nevertheless, it was demonstrated that the collision can be used in a beneficial manner for a game scenario. While the sample game that has been implemented during this thesis was designed to make good use of the collisions, we believe that it is possible to design other interesting game concepts that utilize the proposed system as well.

We have also shown that the reconstructed meshes can be used for the occlusion of virtual content behind real objects. The effect of this occlusion on the immersion of the user is greatly restricted by reconstruction inaccuracies, as small discrepancies between the shape of the reconstructed mesh and its real counterpart can easily be seen. However, we believe that inaccurate occlusion is still better than no occlusion at all, as it represents an additional tool to play around with while developing an AR application.

The evaluation has shown that the proposed system in its current state is not ready to be used in an end user application. The optimal reconstruction parameters have been observed to strongly depend on real world environmental conditions and hence need to be changed accordingly to achieve good results. Having to control such a technical detail of the implementation for the application to work properly would surely be overwhelming for many users and should therefore be avoided.

During the evaluation, it has also become clear that there are two main problems with our current implementation of the proposed reconstruction pipeline. First, there seems to be an issue with Unreal Engine's Async Tasks which we used to realize asynchronous computation. Probably due to a scheduling issue some calculations take a lot longer than usual, increasing the total time needed for the reconstruction. The second problem we encountered is that

our implementation of the Delaunay triangulation takes very long to complete for some data sets, while it performs well in most other cases of a similar size. If those two issues appear concurrently, the triangulation of a cluster might take up to several minutes, which definitively needs to be prevented before using the system in real applications.

6.2. Future Work

While working on this thesis some topics arose that might be approached in the future. The first and most obvious of those is further development and refinement of the proposed reconstruction system. In a first step, the given implementation could be reviewed to fix the critical bugs that were discovered during the evaluation such as the extremely spiking triangulation times. The evaluation has also shown, that while being able to achieve usable results, the system in its current state needs a lot of manual adjustment to work properly in the given environment. A solution for this might be to dynamically determine the best reconstruction parameters based on the perceived camera image.

To further enhance the current system, each reconstruction step needs to be treated isolated and optimized individually. Additional analysis of the ARCore feature point cloud might reveal a better way to process the points to gain a higher precision. Especially the surface reconstruction algorithm used in this thesis has some problems impacting the quality of the reconstruction. As the Delaunay triangulation always outputs the convex hull around the input points, it currently is not possible to reconstruct concave regions, as they often appear in real-world objects, also a single noise point has the potential to corrupt the reconstruction result in a big area. Other surface reconstruction techniques can be reviewed and tested with the given system to see if they are able to achieve better results.

There are also a bunch of other approaches to a better scene understanding that should be further examined. For the best user experience, it is necessary for future system to omit the recording step that is required for our proposed solution. Instead, the reconstruction should run in real-time and dynamically update the mapping of the real world. Such a system would also be able to achieve a significantly higher quality in the long run as errors can be corrected automatically. With mobile devices having more and more computation power being built each each year, real-time reconstruction systems that were previously only running on high-end desktop hardware may be adapted to work on smartphones as well in due future.

Finally, Google and Apple are constantly developing updates for their mobile AR frameworks. Each time they add new features developers and researchers are given new ways and starting points to further enhance the scene understanding.

A. General Addenda

A.1. Evaluation Data

Google Spreadsheet - `http://bit.ly/BA_Data`

B. Figures

B.1. Game Screenshots

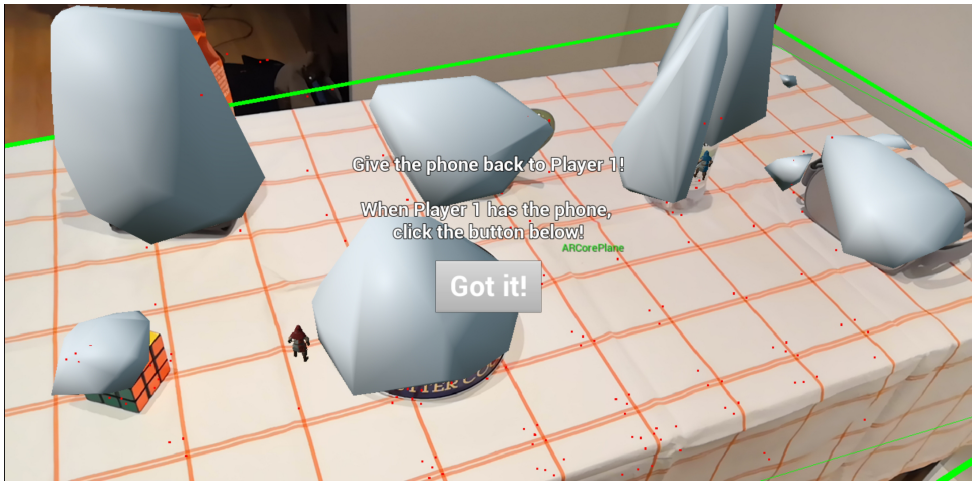


Figure B.1.: Both players spawned their archer and are hiding behind the reconstructed objects. Player 2 is prompted to pass the phone back to player 1, whose turn is next.

Figure B.2.: Screenshot while a player is currently shooting. The white dots show the predicted projectile path for the current launch strength.

Figure B.3.: The player just missed his shot, indicated by the red dots showing that the projectile has passed over the other archer. The fire button is now disabled and rendered red as the player has already used his shot this turn.

Figure B.4.: A successful hit. In the top left corner the player is informed about his successful hit. The big red dots show the projectile path hitting the enemy archer.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Google. *ARCore*. 2018. URL: https://developers.google.com/ar/ (Retrieved 04/25/2019).

[2] Apple. *ARKit*. 2018. URL: https://developer.apple.com/arkit/ (Retrieved 05/03/2019).

[3] A. B. Craig. *Understanding augmented reality: Concepts and applications*. Newnes, 2013.

[4] R. T. Azuma. "A survey of augmented reality". In: *Presence: Teleoperators & Virtual Environments* 6.4 (1997), pp. 355–385.

[5] P. Milgram and F. Kishino. "A taxonomy of mixed reality visual displays". In: *IEICE TRANSACTIONS on Information and Systems* 77.12 (1994), pp. 1321–1329.

[6] T. P. Caudell and D. W. Mizell. "Augmented reality: An application of heads-up display technology to manual manufacturing processes". In: *Proceedings of the twenty-fifth Hawaii international conference on system sciences*. Vol. 2. IEEE. 1992, pp. 659–669.

[7] C. Arth, R. Grasset, L. Gruber, T. Langlotz, A. Mulloni, and D. Wagner. "The history of mobile augmented reality". In: *arXiv preprint arXiv:1505.01319* (2015).

[8] D. Chatzopoulos, C. Bermejo, Z. Huang, and P. Hui. "Mobile augmented reality survey: From where we are to where we go". In: *Ieee Access* 5 (2017), pp. 6917–6950.

[9] M. Mohring, C. Lessig, and O. Bimber. "Video see-through AR on consumer cell-phones". In: *Third ieee and acm international symposium on mixed and augmented reality*. IEEE. 2004, pp. 252–253.

[10] G. Klein and D. Murray. "Parallel tracking and mapping on a camera phone". In: *2009 8th IEEE International Symposium on Mixed and Augmented Reality*. IEEE. 2009, pp. 83–86.

[11] M. Li and A. I. Mourikis. "High-precision, consistent EKF-based visual-inertial odometry". In: *The International Journal of Robotics Research* 32.6 (2013), pp. 690–711.

[12] Google. *ARCore Fundamental Concepts*. 2018. URL: https://developers.google.com/ar/discover/concepts (Retrieved 06/05/2019).

[13] Apple. *ARKit Documentation*. 2018. URL: https://developer.apple.com/documentation/arkit (Retrieved 06/06/2019).

[14] Niantic. *Pokemon GO*. 2016. URL: https://itunes.apple.com/us/app/pok%C3%A9mon-go/id1094591345?mt=8 (Retrieved 05/16/2019).

[15] 5minlab. *Brickscape*. 2017. URL: https://itunes.apple.com/us/app/brickscape/id1233962836?mt=8 (Retrieved 05/16/2019).

[16] A. T. Networks. *Knightfall AR*. 2017. URL: https://play.google.com/store/apps/details?id=com.aetn.games.android.history.knightfall.ar&rdid=com.aetn.games.android.history.knightfall.ar (Retrieved 05/16/2019).

[17] D. Design. *AR Smash Tanks!* 2017. URL: https://play.google.com/store/apps/details?id=com.dumpling.smashtanks&hl=en (Retrieved 05/16/2019).

[18] D. Weibel and B. Wissmath. "Immersion in computer games: The role of spatial presence and flow". In: *International Journal of Computer Games Technology* 2011 (2011), p. 6.

[19] M. Slater, M. Usoh, and A. Steed. "Depth of presence in virtual environments". In: *Presence: Teleoperators & Virtual Environments* 3.2 (1994), pp. 130–144.

[20] M. K. Coomans and H. J. Timmermans. "Towards a taxonomy of virtual reality user interfaces". In: *Proceedings. 1997 IEEE Conference on Information Visualization (Cat. No. 97TB100165)*. IEEE. 1997, pp. 279–284.

[21] M. Berger, A. Tagliasacchi, L. Seversky, P. Alliez, J. Levine, A. Sharf, and C. Silva. "State of the art in surface reconstruction from point clouds". In: *EUROGRAPHICS star reports*. Vol. 1. 1. 2014, pp. 161–185.

[22] R. Sitnik and M. Karaszewski. "Optimized point cloud triangulation for 3D scanning systems". In: *Machine Graphics & Vision International Journal* 17.4 (2008), pp. 349–371.

[23] E. F. Ohou. "Surface Reconstruction: Techniques and Methods from 3D Points Data". In: ().

[24] D.-T. Lee and B. J. Schachter. "Two algorithms for constructing a Delaunay triangulation". In: *International Journal of Computer & Information Sciences* 9.3 (1980), pp. 219–242.

[25] D. F. Watson. "Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes". In: *The computer journal* 24.2 (1981), pp. 167–172.

[26] F. Cazals and J. Giesen. "Delaunay triangulation based surface reconstruction". In: *Effective computational geometry for curves and surfaces*. Springer, 2006, pp. 231–276.

[27] J.-D. Boissonnat. "Geometric structures for three-dimensional shape representation". In: *ACM Transactions on Graphics (TOG)* 3.4 (1984), pp. 266–286.

[28] H. Edelsbrunner and E. P. Mücke. "Three-dimensional alpha shapes". In: *ACM Transactions on Graphics (TOG)* 13.1 (1994), pp. 43–72.

[29] N. Amenta, M. Bern, and D. Eppstein. "The crust and the β-skeleton: Combinatorial curve reconstruction". In: *Graphical models and image processing* 60.2 (1998), pp. 125–135.

[30] N. Amenta and M. Bern. "Surface reconstruction by Voronoi filtering". In: *Discrete & Computational Geometry* 22.4 (1999), pp. 481–504.

[31] E. Piazza, A. Romanoni, and M. Matteucci. "Real-time CPU-based large-scale 3D mesh reconstruction". In: *arXiv preprint arXiv:1801.05230* (2018).

[32] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, et al. "KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera". In: *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM. 2011, pp. 559–568.

[33] R. A. Newcombe and A. J. Davison. "Live dense reconstruction with a single moving camera". In: *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE. 2010, pp. 1498–1505.

[34] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. "A density-based algorithm for discovering clusters in large spatial databases with noise." In: *Kdd*. Vol. 96. 34. 1996, pp. 226–231.

[35] Epic Games. *Unreal Engine*. 1998. URL: `https://www.unrealengine.com` (Retrieved 05/22/2019).

[36] L. Valente, A. Conci, and B. Feijó. "Real time game loop models for single-player computer games". In: *Proceedings of the IV Brazilian Symposium on Computer Games and Digital Entertainment*. Vol. 89. 2005, p. 99.

[37] Unreal Engine Wiki. *Using Async Tasks*. 2016. URL: `https://wiki.unrealengine.com/Using_AsyncTasks` (Retrieved 05/29/2019).

[38] Unreal Engine Wiki. *Procedural Mesh Component*. 2018. URL: `https://wiki.unrealengine.com/Procedural_Mesh_Component_in_C%2B%2B:Getting_Started` (Retrieved 05/30/2019).

[39] M. Kazhdan, M. Bolitho, and H. Hoppe. "Poisson surface reconstruction". In: *Proceedings of the fourth Eurographics symposium on Geometry processing*. Vol. 7. 2006.

[40] J. Nielsen. "Website response times". In: *Nielsen Norman Group* 21.06 (2010).