HAW HAMBURG

Bachelorarbeit

Hugo Protsch

Detecting Geometric Primitives in Depth Data from the Google ARCore Depth API

Faculty of Engineering and Computer Science
Department Computer Science

Hugo Protsch

Detecting Geometric Primitives in Depth Data from the Google ARCore Depth API

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Philipp Jenke
Supervisor: Prof. Dr. Olaf Zukunft

Submitted on: July 26, 2024

Hugo Protsch

Thema der Arbeit

Detecting Geometric Primitives in Depth Data from the Google ARCore Depth API

Stichworte

Augmented Reality (AR), Oberflächenrekonstruktion, Punktwolken, Tiefenbilder, Extraktion geometrischer Primitive, Objektrepräsentation, Mobile Anwendungen

Kurzzusammenfassung

Diese Arbeit stellt ein System vor, das geometrische Primitive in der Umgebung des Nutzers auf mobilen Geräten ohne spezialisierte Tiefensensor-Hardware erkennt. Das System nutzt Tiefendaten von der Google ARCore Depth API, um eine Punktwolke zu erstellen. Um die Punktwolkeninformationen effizient in Echtzeit zu speichern und zu aktualisieren, wird eine eigens entwickelte Octree-Implementierung verwendet. Primitive werden innerhalb der Punktwolke mithilfe der RANSAC-Implementierung von [SWK07] erkannt. Die resultierenden Parametrisierungen der Primitive werden verwendet, um Dreiecksnetze ihrer konvexen Hüllen zu erzeugen. Diese Netze werden schließlich gerendert und in Echtzeit auf das Kamerabild überlagert, sodass der Nutzer sie in einer Augmented Reality (AR) Anwendung sehen kann.

Hugo Protsch

Title of Thesis

Detecting Geometric Primitives in Depth Data from the Google ARCore Depth API

Keywords

Augmented Reality (AR), Surface Reconstruction, Point Clouds, Depth Maps, Primitive Extraction, Object representation, Mobile Applications

Abstract

This thesis presents a system that detects geometric primitive in the user's surroundings on mobile devices without specialized depth-sensing hardware. The system utilizes depth

data from the Google ARCore Depth API to create a point cloud. To efficiently store and update the point cloud information in real-time, a custom octree implementation is employed. Primitives are detected within the point cloud using the RANSAC implementation by [SWK07]. The resulting parameterizations of the primitives are used to generate triangle meshes of their convex hulls. These meshes are finally rendered and overlayed onto the camera feed, accessible to the user through an Augmented Reality (AR) application.

Contents

1 Introduction

In recent years, advancements in computer vision technology and hardware for mobile devices have enabled the development of mobile applications that can understand and interact with the real world. Augmented Reality (AR) is a popular application of this technology, allowing for the overlay of digital information onto the real world. For instance, in the ecommerce industry, companies like Amazon are leveraging AR to enable customers to visualize furniture in their homes before making a purchase. Similarly, mobile mapping applications utilize AR to provide directions and information about the environment through the camera feed.

Virtual Reality (VR) on the other hand creates fully immersive experiences that isolate users from the real world. Through the use of specialized head-mounted display (HMD) headsets, users can explore virtual worlds, play games, and watch movies in environments completely separate from their physical surroundings. This technology has gained traction in the gaming and entertainment industries with platforms like Oculus Rift, SteamVR and PlayStation VR.

Building upon the advancements of VR and mobile AR, a recent trend in the industry is the development of devices that combine the two technologies, allowing to seamlessly blend virtual objects into the real world and vice versa, creating the illusion of a singular, unified world. This technology is named differently by different companies, such as Mixed Reality (MR) by Microsoft and Meta or Spatial Computing by Apple. MR / Spatial Computing allows for immersive experiences while still enabling users to interact with the real world. Similarly to AR, its potential applications span across various industries, from healthcare to education to entertainment. Examples of devices include Microsoft's HoloLens and more recently the Meta Quest 3 and Apple Vision Pro.

Figure 1.1 shows a screenshot of a trailer for a game on the Quest 3 that mixes the real world with virtual objects. You can see virtual boxes and enemies blended into the real

Figure 1.1: Screenshot of trailer for the game Espire 2 by Tripwire Interactive showcasing a player using the Meta Quest 3 playing the game blended with their real environment.

environment of the player. The player can interact with these objects with their body and hands, using controllers or hand tracking.

Current AR applications use a variety of techniques to blend virtual objects into the real world and make them interactable. As an example, the ARCore Depth SDK by Google provides features like

- Detection of flat surfaces
- Placement of virtual objects on surfaces
- Depth information that can be used to occlude objects behind real-world objects
- Lighting estimation for realistic shading of virtual objects

This however, does not allow for understanding of the environment for the purpose of complex logic outside of rendering, such as navigation of AI agents through the environment or creating customized game logic based on the layout of the environment, as the API does not provide a mesh of the environment or similar features. To enable these kinds of applications, it is essential to have an accurate understanding of the real world surrounding the user.

This thesis aims to provide an implementation of a system that can create a mesh of the environment from data provided by the ARCore Depth API using primitive detection algorithms and overlay this mesh onto the camera feed for visualization. The information about the environment (both the primitive parameterization and the resulting mesh) can then be used for various applications as mentioned above.

2 Primitive Detection Algorithms

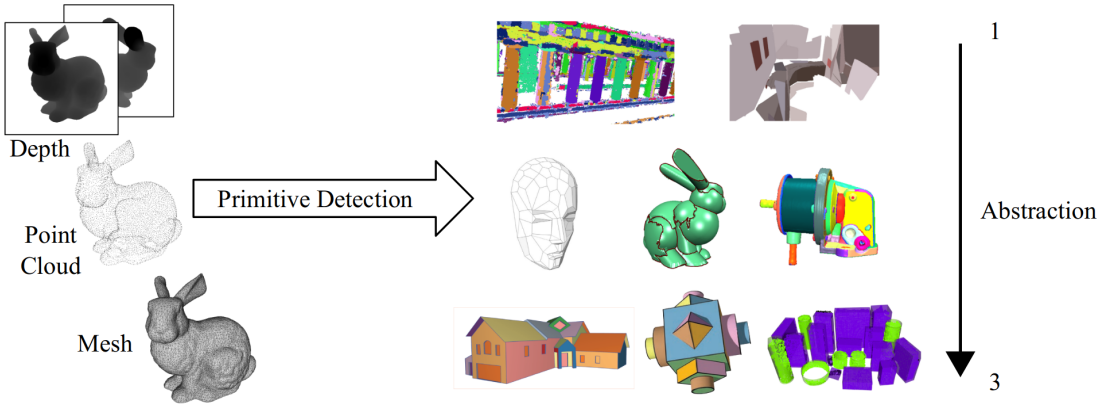


Figure 2.1: Process of detecting primitives [KYB19]

Primitive detection is a well-established area in computer vision that aims to detect simple geometric shapes like planes, spheres or cylinders in given input data. These methods are model-fitting algorithms that identify the most likely model that fits (a subset of) the input data. The result of these algorithms is a set of parameters that describe the detected shape. These shapes are an abstraction of the input data, offering a simplified compact representation of the data, allowing for higher performance and the ability to perform higher-level tasks such as object recognition or scene reconstruction. The input data representation varies by algorithm, with the most common types being:

- **Point Clouds:** A set of 3D points in space representing the sampled surface of the real-world
- **Depth Images:** A 2D image where each pixel represents the distance to the camera, typically obtained from depth sensors

- **Image Sequences:** A sequence of 2D images from different viewpoints that are typically first convert to a point cloud or depth image for further processing, further discussed in section 4.1.2
- **Meshes:** A polygonal representation of surfaces, consisting of vertices, edges and faces, typically generated from surface reconstruction algorithms. Not further considered in this thesis. For an explanation of meshes, see section 4.4.1. [KYB19]

This chapter provides an overview of the most common primitive detection algorithms by categorizing them based on their underlying methodology. The chapter then compares the two most widely used base methodologies, the Hough Transform and the Random Sample Consensus (RANSAC) algorithm.

2.1 Categorization

Kaiser, Ybanez Zepeda, and Boubekeur [KYB19] reviewed over 70 detection algorithms, evaluating them based on their input/output data types, underlying methodology, supported primitive types, context of application and provide a rating for multiple criteria. The authors categorize the underlying methodology of the algorithms into three categories:

- **Stochastic**: algorithms that use random sampling to detect primitives, such as RANSAC
- **Parameter Space**: algorithms that use a parameter space to detect primitives, such as the Hough Transform
- **Clustering**: algorithms that aggregate data in a local-to-global fashion based on similarity constrains, such as primitive-driven region growing

The following sections provide an overview of the most common algorithms in each category.

2.1.1 Stochastic (RANSAC)

The Random Sample Consensus (RANSAC) algorithm is a widely used stochastic model parameter estimation algorithm first introduced in 1981 by Fischler and Bolles [FB81]. "The basic principle of the algorithm is to try many possible randomized models that could fit the data and evaluate how good this model is in order to find a consensus, i.e. an agreement of most of the data samples." [KYB19]

For a given shape that requires n points to be defined, RANSAC follows the following steps to detect the shape in a set of data points P [FB81]:

1. Randomly select a subset $S1$ of n data points from P
2. Fit the model $M1$ to the selected points
3. Determine the subset of inliers $S1^*$ of P that fit the model $M1$ within a predefined tolerance, this is called the consensus set
4. If size of the consensus set $|S1^*|$ is greater than a predefined threshold t, re-fit the model to $S1^*$, resulting in new model $M1^*$
5. If size of the consensus set $|S1^*|$ is smaller than a predefined threshold t, repeat until a model with a consensus set of size t is found or a predefined number of iterations is reached

The algorithm has 3 main parameters that need to be set:

- **Error tolerance**: the distance between a data point and the model under which the data point is considered an inlier
- **Threshold** t: the number of inliers required to consider a model valid
- **Number of iterations**: the number of times the algorithm will try to find a model with a consensus set of size t

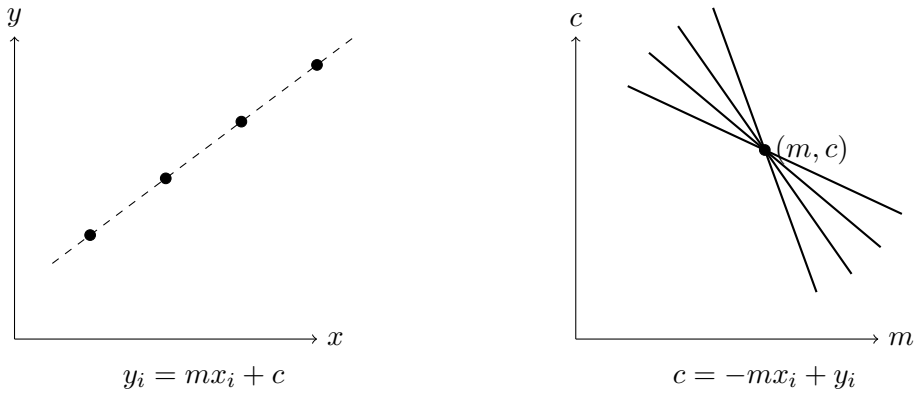
2.1.2 Parameter Space (Hough Transform)

The parameter space refers to a mathematical space which is defined by the parameters of the shape instead of their coordinates. The Hough transform (HT) introduced in 1962 [Hou62] takes advantage of this concept. The algorithm was originally designed to detect lines in images, but has since been generalized to detect more complex shapes like circles [Bal81] and 3D shapes [Woo+14].

The HT works by creating a voting space based on parameters where similar shapes overlap[KYB19]. The following explanation of the HT is based on [Shr21]. Figure 2.2 illustrates the concept of the Hough Transform for detecting lines. On the left, the image space is shown, where 4 points are located on a line. The right side shows the parameter space, where the axes represent the parameters of the line equation

$$y = mx + c \tag{2.1}$$

Each point in image space creates a line in parameter space, as there are an infinite number of lines that pass through a point. All points on the line represent a valid parameterization of the line. When all points are plotted in parameter space, local maxima represent the parameters of possible lines in image space.



(a) Image Space (b) Parameter Space

Figure 2.2: Concept of Hough Transform illlustrated for the usage of detecting lines. Reconstructed from [Shr21].

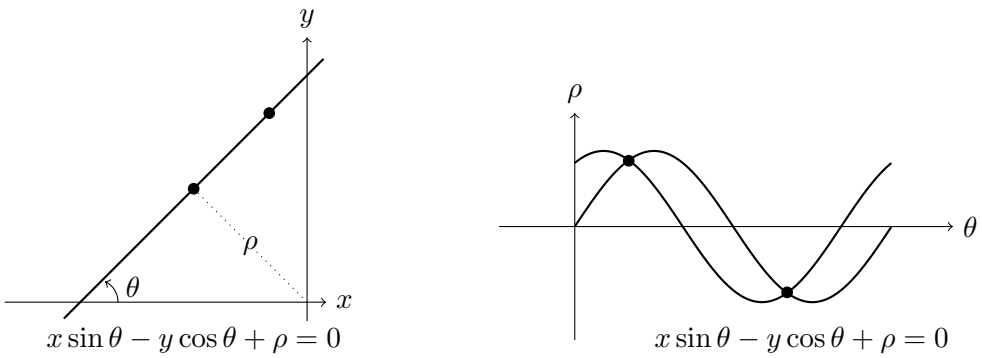
The Hough Transform is implemented by discretizing the parameter space into uniform sections using a two-dimensional array to store the votes for each parameter, called the

accumulator [DH72]. Each data point contributes votes to the shapes it aligns with, indicating the likelihood of a specific shape being present in the input data. By accumulating these votes, the parameters corresponding to the shape with the highest number of votes can be identified. After all data points have voted, the parameters with the highest votes reveal the shape that most accurately fits the input data.

A problem that arises when using the Hough Transform with the parameterization of a line $y = mx + c$ is that the slope m can be infinite for vertical lines, which would require the parameter space to be infinite. A more suitable parameterization for the Hough Transform is achieved by using the angle θ from the x-axis and the distance ρ from the origin:

$$x \sin\theta + y \cos\theta + \rho = 0 \tag{2.2}$$

This parameterization allows for the representation of vertical lines without the need for



(a) Image Space (b) Parameter Space

Figure 2.3: Hough Transform using polar coordinates. Reconstructed from [Shr21].

an infinite parameter space. The angle θ ranges from 0 and π, representing all possible orientations of lines, while the distance ρ can take any value between 0 and the size of the image space. This parameterization not only allows for the representation of vertical lines but decreases the size of the accumulator array, as the parameter space is now finite. To detect lines using this parameterization, the same voting process as described above is used.

2.1.3 Clustering (Primitive-Driven Region Growing)

Clustering or segmentation is a fundamental concept in data analysis that groups similar data points into clusters. "Most commonly, segmentation is treated as a local-to-global aggregation problem with similarity constraints employed to control the process." [LMM98] Algorithms start by selecting a small subset of data points as initial regions, which are then expanded by adding similar points to the region. Homogenous regions are finally merged to form the final clusters. [LMM98]

One common paradigm in the context of 3D geometric primitive detection is primitive-driven region growing. This approach starts by assigning a seed label to a point, and then iteratively checks neighboring points to see if their characteristics like color, Euclidean distance or normal orientation match the seed labels characteristics within a certain threshold. Generic solutions often use a heuristic like the primitive fitting error, which is the mean square error between the point and the potential primitive. If the neighboring point matches, the label is assigned to the neighboring point. The process is repeated until no more points can be added to the region. [KYB19]

For an extensive overview of other clustering paradigms like automatic clustering see survey by [KYB19].

2.2 Choosing an Algorithm: Hough Transform vs. RANSAC

[KYB19] lists over 70 detection algorithms, many of which are specialized for specific application contexts like indoor scenes, outdoor scenes, urban buildings or individual objects. As this thesis aims to develop a first end to end implementation for detecting and rendering primitives in AR, it is important to consider algorithms that are widely used and applicable to a wide range of applications.

Among the various methodologies, the Hough Transform and the Random Sample Consensus (RANSAC) algorithm emerge as the most widely used in the field of primitive detection. [SWK07]. On the other hand, primitive growing is also a commonly used methodology but is considered to be the least robust to outliers and one of the slowest methods compared by [KYB19]. This might pose a problem for the use-case of this thesis, as the depth data may contain a significant amount of noise and the performance of

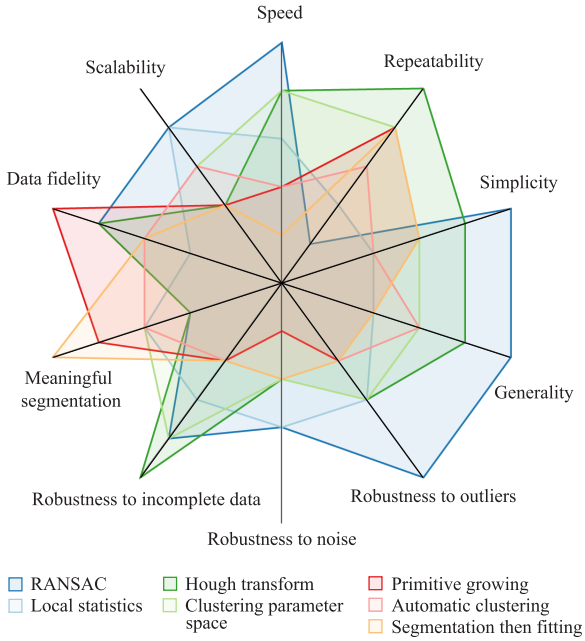


Figure 2.4: Compairison of Primitive Detection algorithm performance [KYB19]

a mobile device is limited. Therefore, it is necessary to compare the Hough Transform and RANSAC and make decision on which algorithm to use for the implementation.

Both algorithms show varying performance depending on the context of the application. With optimizations, both algorithms are suitable for a wide range of applications, but it is important to consider that the Hough Transform is more computationally expensive than RANSAC [KYB19]. In a study, [TLG07] found that RANSACs processing time is negligible in comparison to the Hough Transform. Additionally, RANSAC is more robust to noise and outliers [KYB19]. It is also a simpler algorithm which makes it easier to extend and adapt to different contexts [TLG07; KYB19]. The main drawback of RANSAC is that results are not repeatable, as the algorithm is based on random sampling [KYB19].

In a specific application of detecting building roofs as planes in 3D Lidar scans of cityscapes, [TLG07] found that by default both RANSAC and Hough-Transform yield similar, insufficient results for their use-case. “That can be explained by the use of a pure

mathematical principle, without taking into account the particularity of the building Lidar data. [...] That is why [they] may detect a set of points which represents several roof planes or which belongs to several planes."

Additionally, determining the optimal parameters for the Hough Transform proved challenging, as the optimal values heavily vary based on the characteristics of the point cloud. Motivated by RANSAC's speed, Tarsha-Kurdi, Landes, and Grussenmeyer extended the algorithm and achieved satisfying results.

In the end, both the Hough Transform and RANSAC are suitable algorithms for detecting primitives. However, the Hough Transform is more computationally expensive, harder to tune for specific contexts and less robust to noise and outliers. RANSAC also has a publicly available reference implementation in C++ by [SWK07], which is e.g. used in the open source tool CloudCompare [Dan+]. This allows for easy testing on ones own pointclouds without the need to write custom code, making it useful for evaluation purposes, as can later be seen in chapter 5.

Due to these reasons, the Random Sample Consensus (RANSAC) algorithm will be used for the implementation of the primitive detection in this thesis.

3 Implementation Context

This chapter goes over the technical background of the implementation and the context in which it is developed.

3.1 Potential Pitfalls

During the planning phase of this thesis, a potential pitfall of the system was identified. The development device, Google Pixel 7, lacks a depth sensor. Extracting depth information using the Depth API is still possible, however the Depth API will rely solely on Depth from Motion techniques to derive depth information from camera images as later described in section 4.1.2. It is important to note that camera-based Depth from Motion has limitations when it comes to detecting depth in objects with minimal texture, such as walls. This drawback could potentially present challenges, especially when detecting walls or furniture with minimal texture, where the accuracy of depth information obtained from the Depth API may not be sufficient for accurate recognition. [Goo] This pitfall is addressed in the evaluation, chapter 5.

3.2 Hard- and Softwarestack

The mobile application is developed for Android and tested using a Google Pixel 7. The implementation is carried out in Java/Kotlin using the Android Studio integrated development environment (IDE). The Google ARCore SDK is used to access depth information about a scene. The SDK is available by default, and no additional libraries are required.

Algorithms are implemented in C++ in the `procedural-augmented-reality` project provided by Prof. Dr. Phillipp Jenke. The code of this thesis is integrated into the application and interfaced with Kotlin through a binding layer.

3.3 Libraries and External Code

The following libraries and/or publicly available code are used in this thesis:

- The RANSAC implementation by Schnabel, Wahl, and Klein is used for primitive detection and is further examined in this thesis [SWK07]
- *ARCore Raw Depth* provides a reference implementation for using the ARCore Raw Depth API. It is used as a basis for unprojecting depth image pixels into world space in this thesis. [GT22]
- The monotone chain implementation in C++ to compute the convex hull available on Wikibooks[1]

3.4 Testing

All algorithms implemented in the `procedural-augmented-reality` project are unit tested using Google Test, a C++ testing framework. To achieve this, a CMake library target *backend* is defined, that contains all functionality of the `procedural-augmented-reality` project. A second target that contains the tests, *backend-test*, is defined, which links against *backend* and the Google Test library.

The testing process involves writing individual test cases for each function and class to ensure they behave as expected under various conditions. For instance, the integrity of the Octree data structure is verified by recursively checking the bounds of each node and its children. Other tests validate the correct insertion and retrieval of points within the Octree, ensuring that nodes are placed in the correct sub-octants. Edge cases are also considered, such as testing the behavior of the Octree when searching for points within a radius. Additionally, the deletion functionality is extensively tested through various scenarios, including deleting nodes from the root and upwards or downwards in the hierarchy.

[1] *Algorithm Implementation/Geometry/Convex hull/Monotone chain - Wikibooks, open books for an open world.* URL: https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monotone_chain#C++ (visited on 04/22/2024).

4 Solution

This chapter goes into detail about the solution of the whole detection pipeline. All concepts and algorithms that are used will be explained in detail as needed.

In the system, the first step involves collecting depth information via depth images and confidence images from the ARCore Raw Depth API, as explained in section 4.1. Next, in section 4.2, the depth images are converted into a point cloud. The RANSAC algorithm is then applied to the point cloud to detect geometric primitives in section 4.3. Finally, rendering the primitives is explained in section 4.4. For the scope of this thesis, the implementation focuses on detecting planes. However, it is possible to extend the pipeline to detect all other primitives detected by the RANSAC implementation by [SWK07].

4.1 Capturing Depth Images

The first step is acquiring the depth data required for primitive detection. Google ARCore [Goo] is an SDK for developing augmented reality applications on Android and iOS devices and provides a wide range of features, such as motion tracking, environmental understanding, and light estimation. For the purpose of this project, it is used to access depth information from the camera in form of depth images.

4.1.1 ARCore Depth APIs

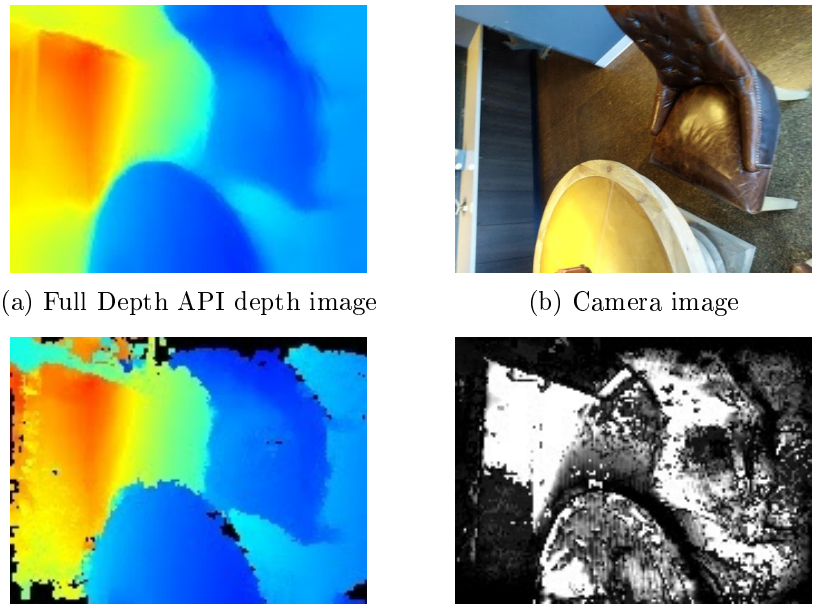
Google ARCore provides two APIs to access depth information – the Depth API and the Raw Depth API. Both APIs work by estimating depth information from a sequence of monocular camera images using a technique called *Depth from Motion*. When the user moves their device, the system takes a sequence of images and estimates the depth information by comparing the differences between these images to the position of the device at the time the images were taken. The technical details of this system are further

discussed in section 4.1.2. Google claims that that one "can get accurate results from 0 to 65 meters away, with best results between 0.5 and 5 meters." [Goo]

Both the Raw Depth API and Full Depth API provide depth information for a given frame of a camera image using depth images, but they differ in the level of detail they provide:

The Raw Depth API provides depth images and confidence images, where some pixels may not have any depth information. The depth image provides the distance from the camera of a given pixel in millimeters. The confidence image indicates the reliability of the depth information for each pixel, ranging from 0 (no confidence) to 255 (high confidence).

In contrast, the Full Depth API provides a single depth image, where each pixel has a depth value. To achieve this, values for pixels without depth information are interpolated. No confidence image is provided.



(a) Full Depth API depth image (b) Camera image

(c) Raw Depth API depth image (d) Raw Depth API confidence image

Figure 4.1: Full Depth vs. Raw Depth. Source: [Goo]

Both Depth APIs have their use cases – the Full Depth API is preferred in cases where it is crucial to have a depth value for every pixel, such as calculating if an object should be occluded by the scene in an AR application, while the Raw Depth API is preferred if

accuracy of the depth information is crucial. As accuracy is crucial for primitive detection and depth information for every pixel is not required, the Raw Depth API is used in this thesis.

4.1.2 Technical Background: Depth From Motion

Depth from motion is a technique developed by Google that estimates depth information from a sequence of monocular camera images and is used by the ARCore Depth API to provide depth information. Its primary purpose is to enable AR applications that rely on depth information on devices lacking a dedicated depth sensor or multiple cameras. This sections provides a brief overview of the workings of the Depth from Motion system by [Val+18].

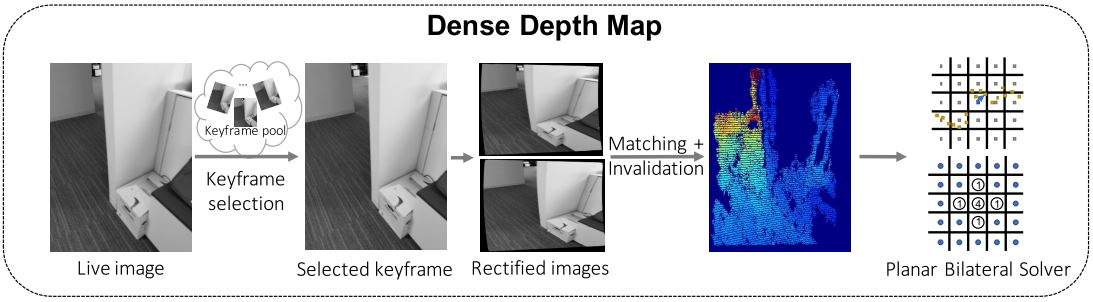


Figure 4.2: Overview of the Depth from Motion System. Source: [Val+18]

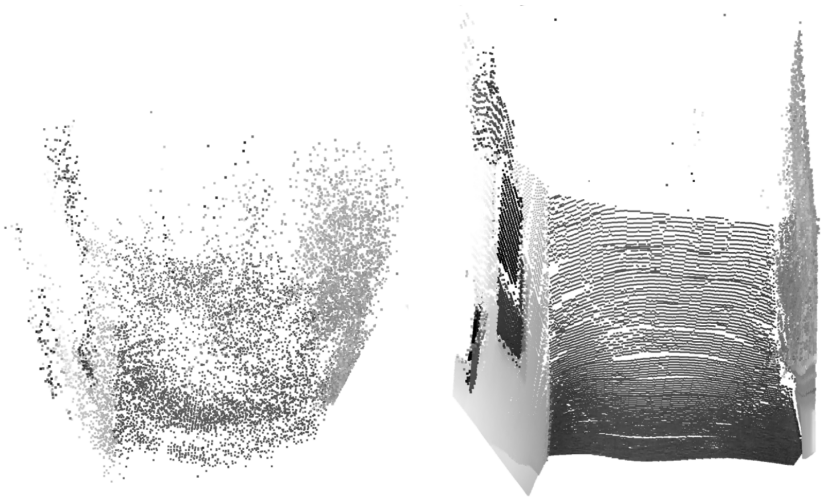
When users move their smartphones, the system uses ARCore's visual-inertial odometry (VIO) to determine its position and orientation in six dimensions (6DoF): up/down, left-/right, forward/backward, and tilt/swivel/rotate. After activating tracking and acquiring the most recent camera image (in black and white for faster processing), a reference image or keyframe from the past is chosen to compare with the current image.

A process known as polar rectification then aligns the keyframe and current frame onto the same plane based on their differences in position and orientation. This alignment simplifies the process of finding matching points in both images by focusing on comparable horizontal lines.

Next, they use an image correspondence algorithm to identify matching points, which generates disparity maps that indicate the positional differences between matched points in the two images. As scenes may contain low textured objects or repetitive patterns

which can lead to incorrect matches, they use an invalidation step, which removes points that are likely to be incorrect matches. This step also provides a confidence value, which is later used in the interpolation step. Triangulation is used to compute a sparse depth map based on the disparity maps.

The missing values in the sparse depth maps are then interpolated using a variation of an algorithm called the bilateral solver (see figure 4.3), which yields an intermediate data structure known as a bilateral depth grid. This grid is a structured representation of depth information that can be converted into a full depth map on demand, assessable through the Full Depth API.



(a) Raw point cloud (b) Interpolated

Figure 4.3: Interpolation with Bilateral Solver. Source: [Val+18]

From the paper it is unclear, which part of the pipeline are omitted in the Raw Depth API, but it is reasonable to assume that the Raw Depth API accesses the sparse depth map and confidence values before the interpolation step.

4.2 Building the Point Cloud

With the depth image and confidence image collected from the Raw Depth API, the next step is to convert the depth image into a point cloud, as RANSAC is a point-based algorithm. The process consists of three steps:

1. Filtering low confidence points

2. Transforming depth image pixels to world coordinate points
3. Inserting new points into the point cloud

4.2.1 Filtering Low Confidence Points

The Raw Depth API provides a confidence image that indicates the reliability of the depth information for each pixel. To improve the accuracy of the point cloud, points with low confidence are filtered out. A threshold value is set, below which points are discarded. Filtering out low confidence points early in the pipeline also improves performance, as fewer points need to be processed in the following steps.

4.2.2 Transforming Depth Image Pixels to World Coordinate Points

The Depth API provides depth images where each pixel holds the distance from the camera in millimeters. Before inserting points into the point cloud, the pixel values of the depth image first need to be converted into 3-dimensional coordinates relative to a fixed origin in the world. This section will first provide a brief overview of the mathematical concepts required to understand the transformation process. Then, the process of transforming a point from the depth image into world space will be explained.

Transformations

In computer graphics matrices are used to represent geometric transformations like translation, scaling, rotation, shearing, reflection and projection. While it is out of scope to cover all of these transformations in detail, this section will use the example of the translation to explain the concepts of transformations and homogenous coordinates based on [Dör+19].

To apply transformations to a point, the points is represented in homogenous coordinates, which is a 4x1 matrix with the fourth w element set to 1.

$$p = \begin{bmatrix} w \cdot x \\ w \cdot y \\ w \cdot z \\ w \end{bmatrix} \tag{4.1}$$

The homogenous coordinates are then multiplied with a transformation matrix M to apply the transformation.

$$p' = M \cdot p \tag{4.2}$$

In the case of a translation, the translation matrix is a 4x4 matrix with the translation vector t as the fourth column.

$$p' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} w \cdot x \\ w \cdot y \\ w \cdot z \\ w \end{bmatrix} = \begin{bmatrix} w \cdot (x + t_x) \\ w \cdot (y + t_y) \\ w \cdot (z + t_z) \\ w \end{bmatrix} \tag{4.3}$$

Note that transformation can also effect the fourth element w, as later explained in the section about perspective projection. To account for this, the resulting matrix is then divided by the fourth element w. The cartesian coordinates of the point are then the first three elements of the resulting matrix.

One advantage of using matrices to represent transformations is that multiple transformations can be combined by multiplying the transformation matrices. The resulting matrix will then apply all transformations in the order they were multiplied.

$$p' = (M_n \cdots M_3 \cdot M_2 \cdot M_1) \cdot p \tag{4.4}$$

If multiple transformations were to be applied to thousands of points, it would be more efficient to multiply the transformation matrices once and then apply the resulting matrix to all points [Vri20]. Graphics Processing Unit's (GPUs) also contain hardware implementation of 4x4 matrix operations, which further increases the performance of matrix operations over other methods [Dör+19]. Transformation matrices can also be inverted (M^{-1}) to apply the inverse transformation, or in other words 'undo' the transformation [Dör+19].

Coordinate Systems and Basis Change

Different coordinate systems, also known as spaces [Vri20], are commonly utilized to represent points in space, allowing for simplified calculations. To illustrate this, consider a camera inside a moving car, filming a person inside the car. In order to implement smooth camera movement around the person, calculations based on coordinates relative to the

world (world space) would need to account for the car's movement in each frame. By using the car's local coordinate system, where all points are relative to the clarity. [Vri20]

To transform points between these systems, their positions are multiplied with a corresponding transformation matrix. To improve performance, these matrices are often combined to a singular matrix, called the MVP-Matrix, that transforms points from local space to screen space. [Vri20]

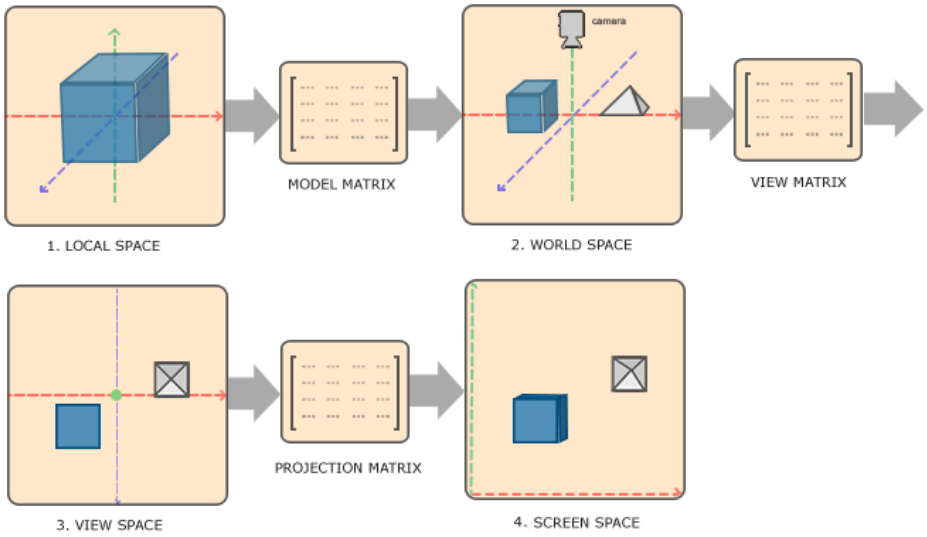


Figure 4.4: Coordinate systems. Adjusted from [Vri20].

Perspective Projection and Camera Intrinsics

In the real world, objects appear smaller the further away they are from the viewer. The same concept applies to images captured by cameras. To illustrate this, consider a simpel pinhole camera model, as shown in figure 4.5. The camera is represented as a box with a small hole (aperture) on one side. This aperture allows light rays from the scene to pass through and form an inverted image on the opposite side of the box In the diagram, the red lines depict the light rays corresponding to the top and bottom of the tree. The closer the tree is to the camera, the smaller it needs to be to fit inside the two rays, or in other words to appear the same size on the image plane. This concept is known as perspective projection and can be achieved by dividing the x and y coordinates of a point by its z coordinate. During this process, the z coordinate is lost and can't be recovered.

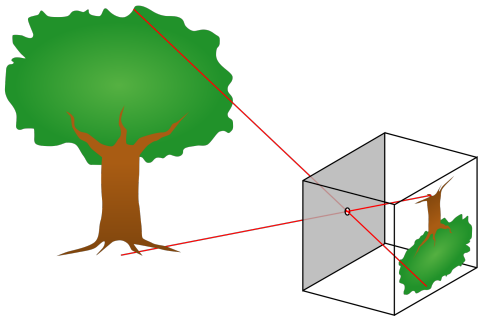
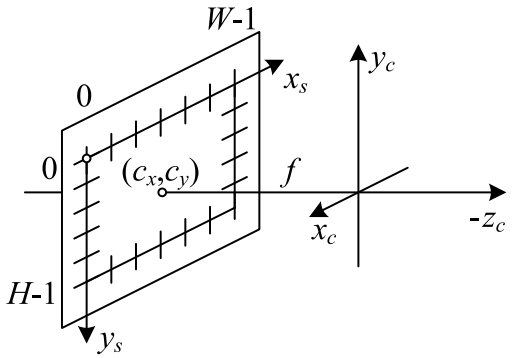


Figure 4.5: Pinhole camera model. Source: Wikimedia Commons

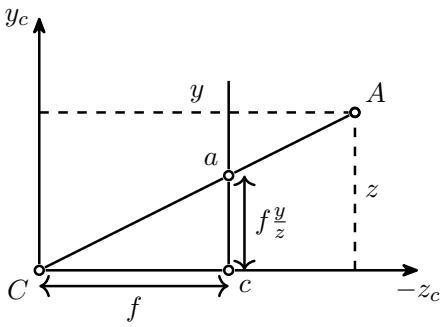
[Sze22]

$$P_s = \begin{bmatrix} x/z \\ y/z \\ 1 \end{bmatrix} \tag{4.5}$$

This however, does not account for the geometry of the camera itself – the camera intrinsics. These intrinsics are a set of parameters that describe the camera's geometry and are used to calculate the projection [Sze22]. The two most important parameters are the focal length f and the image center c, as shown in figure 4.6a.



(a) Geometry of a camera with an image plane height H, width W, focal length f, and principal point c. Source: [Sze22]



(b) Projection of an observed point A onto the image plane in the y-direction. Based on [Kri17] and [Goo]

Figure 4.6: Camera intrinsics

The focal length f is the distance between the image plane and the aperture of the camera. The image center c (also called principal point) is the point where the principal axis z_c intersects the image plane, denoted in pixel coordinates. Figure 4.6b shows the

projection of an observed point A onto the image plane. Note that the image plane is displayed in the $-z_c$ direction, as opposed to $+z_c$ direction. It shows how the focal length f affects the y-coordinate of the projected point. [Sze22; Goo; Kri17] This can be simulated by using ones fingers to form a square that represents a virtual image plane and moving it closer or farther to the eye. The farther away the square is (higher focal length), the less field of view the square covers.

Mathematically, the projection is often represented as a matrix, called the calibration matrix

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.6}$$

with independent focal lengths f_x and f_y to account for aspect ratios other than 1:1. As different focal lengths in each dimension do not reflect the real geometry of a camera, a more intuitive way to understand different focal lengths can be used by introducing the aspect ratio a:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & af & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{4.7}$$

[Sze22]

Applying the transformation to the depth pixels

With the mathematical concepts clarified, the transformation of depth image pixels into world space can be explained. First, it is necessary to understand how the depth values are represented in the depth image.

In terms of coordinate systems, the depth values are in screen space, as shown in figure 4.4. To convert these points into world space, the transformations need to be applied backwards:

1. From screen space to view space (‘unprojecting‘)
2. From view space to world space

Another way to think about this procedure is to conceptualize the unprojecting step as transforming the pixels into a local space relative to the camera, as defined by the camera’s position, up and look vector. Then, the point is transformed from the cameras

local space into world space using the cameras model matrix. For simplicity and to avoid confusion when transforming into different directions, the latter approach is used going forward.

The first step is to unproject the pixels from screen space to the cameras local space K_l, as explained by [GT22; Goo]. Referring to figure 4.6 and "given point A on the observed real-world geometry and a 2D point a representing the same point in the depth image, the value given by the Depth API at a is equal to the length of CA projected onto the principal axis. This can also be referred as the z-coordinate of A relative to the camera origin C." [Goo]

Using the camera intrinsics, the depth value is transformed to the local space relative to the camera K_l. The camera intrinsics can be retrieved from the API using the method `frame.getCamera().getTextureIntrinsics()`. However, this method returns the intrinsics of the camera image, which differs from the depth image, as the depth image usually has a lower resolution. To calculate the focal length f and camera center c from the provided intrinsics K, depth image D and camera image dimensions x_{Dim}, y_{Dim}, the provided focal length and principal point are scaled by the ratio of the dimensions of the depth image and the camera image:

$$S = \begin{bmatrix} \frac{\dim_x(D)}{x_{Dim}} & 0 \\ 0 & \frac{\dim_y(D)}{y_{Dim}} \end{bmatrix} f = S \cdot K_f c = S \cdot K_c \tag{4.8}$$

These values are then used to unproject the point into local space relative to the camera K_l using the equation

$$p_l = \begin{pmatrix} d \cdot (x - c_x)/f_x \\ d \cdot (c_y - y)/f_y \\ -d \\ 1 \end{pmatrix} \tag{4.9}$$

as evident from figure 4.6.

As the resulting point is in a local space relative to the camera K_l, it needs to be transformed into world space K_w. To transform it, the camera pose is retrieved and converted to a matrix: `cameraPoseAnchor.getPose().toMatrix(modelMatrix, 0)`. By multiplying the model matrix T_{wl} with p_l, the point is transformed into world space.

$$p_w = T_{wl} \cdot p_l \tag{4.10}$$

4.2.3 Inserting New Points into the Point Cloud

With all pixels from the depth image transformed to world coordinates, the final step involves adding these points to the point cloud. However, simply storing a list of all points and appending new data to it is not feasible. As a new depth image is captured every frame, the number of points would grow endlessly. This approach would also result in a massive number of points representing the same point on real-world geometry, but with slightly different position values, as subsequent depth images will show the same real-world geometry from different angles, with depth values varying slightly. To address this issue, a spatial data structure that partitions the space into smaller regions and allows to store only one point per region can be used. One possible approach is to use a three-dimensional grid, where each cell represents a region in space. Alternatively, a more advanced approach involves utilizing a spatial data structure such as an octree, which can provide improved performance.

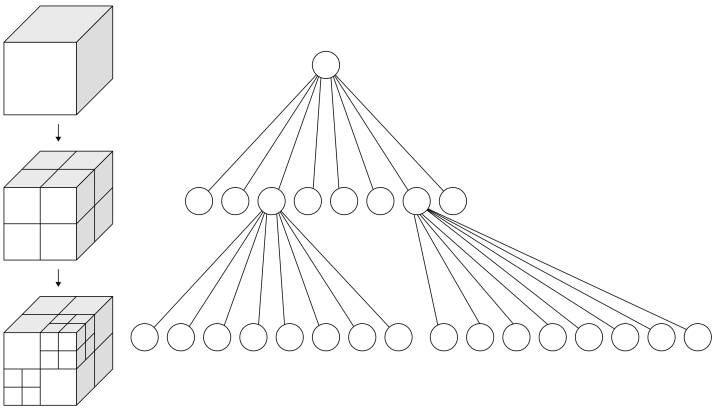


Figure 4.7: Octree. Source: Wikimedia Commons

Octree An octree is a spatial tree data structure where each node has exactly eight children. The tree can be used to sparsely partition three-dimensional space into smaller cubes and allows for efficient insertion and traversal in logarithmic time. "For the definition a simple recursive splitting of [cubes] is continued until there is only one point in a [cube]." [GE02]

Fixed Depth Octree

A straightforward approach to utilize an octree for point cloud storage is to use a variation of the octree with a fixed depth. Points are always inserted at the defined depth of the tree, creating all nodes up to that depth if they do not yet exist. The center coordinate of the leaf node can then be inferred as the point in space, thus saving the coordinates of the point explicitly is not required. Instead, to find the coordinates of a given node, the tree needs to be traversed while keeping track of the extent and center coordinate of the current node. This approach will naturally provide quantization of the point cloud, with the resolution of the point cloud determined by the depth of the octree and extend of the root. This is functionally equal to a three-dimensional grid, but with the advantage of logarithmic time complexity for insertion and traversal. Using this approach, duplicate points are removed and adjusting the resolution of the octree allows for fine-tuning of the threshold distance between points under which points are considered duplicates.

One advantage of an octree with a fixed width is simplicity, as points are always inserted at the same depth and deletion of points is not required. The biggest drawback is that the resolution of the point cloud is fixed and needs to be chosen beforehand. As the accuracy of the depth values differs based on conditions like lighting and texture of the captured surface, it is difficult to choose a fixed resolution that works for all values.

Quantization might also lead to worse detection results for surface that do not align with the axis of the octree. For example, a plane that is tilted by a couple of degrees will lead to aliasing, meaning that the distance between the points and the fitted plane will vary across the plane, as demonstrated in figure 4.8.

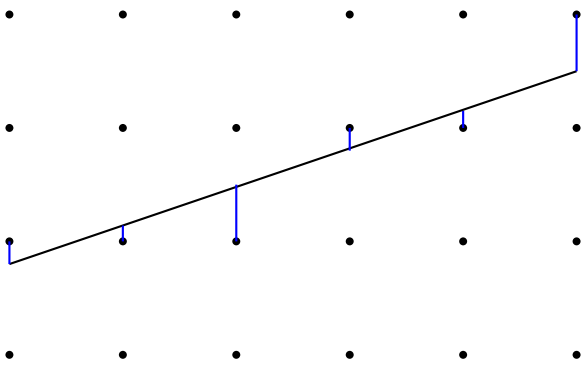


Figure 4.8: Example of aliasing, distance between the line and cell centers varies across cells

Octree with Neighborhood

To improve upon the fixed depth octree, the actual point coordinates can be saved in a given node of the octree. Furthermore, the confidence value can also be stored, in order to determine how accurate it's position is. This allows to improve the accuracy of the data as new, higher confidence data arrives. In order to achieve this, custom logic on inserting new points is required: When inserting a new point with a certain confidence value, a range query with a radius calculated from the confidence value is performed to find all nodes within a certain distance of the point to be inserted.

- If no node is found, a new leaf node is created and the point is inserted.
- If a node is found with a lower confidence value, the new point is inserted and the old node is removed.
- If a node is found with a higher confidence value, the new point is not inserted and discarded.

Using this approach, duplicate points are removed and adjusting the multiplier for the radius of the range query allows for fine-tuning of the threshold under which points are considered duplicates. From here on, this approach will be referred to as the *Epsilon Octree*.

Range Query To check if an octree node, which is an axis-aligned bounding box (AABB) with equal sides, and a sphere intersect, the square distance between the center of the sphere and the closest point on the AABB is calculated. If the square distance is smaller than the square of the radius of the sphere, the sphere and the AABB intersect.

Calculating the square distance between a sphere and the closest point on the AABB can be achieved by summing up the squared distance in each dimension: If the sphere's center is outside the extent of the box on a given axis, the distance is the amount by which it exceeds the box's boundary; otherwise, the distance is zero. In n dimensions, this can be expressed as

$$d^2 = \sum_{i=1}^{n} \left\{ \begin{array}{ll} (C_i - B_i - s)^2 & \text{if } C_i > (B_i + s) \\ (C_i - B_i + s)^2 & \text{if } C_i < (B_i - s) \\ 0 & \text{otherwise} \end{array} \right\} \tag{4.11}$$

where C is the center of the sphere, B is the center of the AABB, and s is the half-size of the AABB. [Gla94]

Deleting nodes Deleting nodes from an octree is a non-trivial task [Sam89; FB74], as it may require restructuring the tree to maintain the octree properties. Finkel and Bentley [FB74] suggest reinserting all child nodes of the deleted node, while [Sam80] propose a more efficient method that tries to replace the deleted node with a suited node, such that the octree properties are maintained. For simplicity, the first approach is used. In addition, an optimization is made: As nodes are only deleted when a new point is inserted with a higher confidence value, it is possible to simply update the old node with the new position, in case the octree properties are not violated. This is the case when the new point is within the same cell as the old point. The tree is first traversed once to find the node that needs to be deleted. If the node is in the same cell as the new point to be inserted, the old node is updated with the data of the new point. Otherwise, the old node is deleted as by [FB74] and the new point is inserted.

4.3 Detecting Primitives using RANSAC

To detect primitives in the point cloud, the Random Sample Consensus (RANSAC) algorithm is used, as discussed in section 2.2. As [KYB19] considers [SWK07] to be the reference implementation of RANSAC and its code is publicly available in C++, it will be used in this thesis.

4.3.1 Schnabel's Efficient RANSAC Algorithm

Schnabel, Wahl, and Klein [SWK07] extended the RANSAC algorithm to be more efficient and robust for detecting primitives. Their approach is specifically designed for detecting planes, spheres, cylinders, cones and tori in 3D point clouds. The key improvements over the original RANSAC method are in the following areas:

- **Sampling strategy**: "Since shapes are local phenomena, the a priori probability that two points belong to the same shape is higher the smaller the distance between the points" [SWK07]. To take advantage of this fact, they employed non-uniform sampling based on locality, which increases the probability of selecting points that

belong to the same shape. They provide an example of the magnitude of this improvement: For a point cloud consisting of 341,587 points that contains a cylinder with 1066 points, uniform sampling would require 151,522,829 shapes candidates to be drawn, while their method only requires 64,929 candidates.

- **Score function**: They introduce a scoring function that evaluates the quality of a shape candidate. The score function is based on 3 parameters: The support of the shape (number of inliers), the deviation of the normals and a connectivity measure, that discards shapes that are not the largest connected component on the shape.
- **Refitting**: After a shape is detected, they refit the shape to the inliers using a least-squares method [Sha98], and include points that are within a certain distance of the shape to declutter the point cloud.

To find connected components in the point cloud, they use a bitmap in the parameter domain of the shape. The parameter domain refers to a parameters defining the shape, e.g. u, v for a plane. They then set a pixel in the bitmap for each point that projects into it. The points of the largest connected component in the bitmap are then considered inliers of the shape, while the rest are outliers and discarded / kept in the dataset for further iterations. Optimally, the bitmaps resolution should be set to the sampling resolution of the point cloud. For irregularly sampled point clouds, they recommend choosing the minimal resolution that is satisfied throughout the point cloud as the bitmap resolution [SWK07]. Later on this will prove to be a challenge in combination with data from the Depth from Motion algorithm, as described in section 5.3.2.

The algorithm requires the following parameters to be set:

- **Epsilon** ϵ: The maximum distance between a point and the shape to be considered an inlier.
- **Bitmap Epsilon** β: The resolution of the bitmap used to determine connected shapes.
- **Normal threshold** α: The maximum deviation of the normals of the points to the shape.
- **Minimum support** n: The minimum number of inliers a shape needs to have to be considered a valid shape.

- **Overlook probability**: The probability that a point is overlooked by the sampling strategy. A higher value increases the number of shape candidates drawn but also the probability of finding the best candidate shape.

4.3.2 Wrapping C++ Code in Java/Kotlin Using SWIG

As the RANSAC algorithm is implemented in C++ and the application is developed for Java/Kotlin, the C++ code needs to be wrapped to be used in the application. To automatically generate the necessary code to interface with Java/Kotlin, the Simplified Wrapper and Interface Generator (SWIG) is used. SWIG is "a program development tool that automatically generates the bindings between C/C++ code and common scripting languages" [Bea96]. It takes an interface file that describes the functions and classes to be wrapped as input and generates the necessary code to interface with the target language.

An example interface file is shown in codeblock 4.1. First, the module name is defined using the `%module` directive. Then, in a block defined by the `%{` and `%}` directives, the C++ code that is necessary for compilation and should be included is defined. Finally, the `%include` directive is used to declare all classes and functions that should be wrapped by SWIG. In the case of template functions and classes, the `%template` directive also needs to be used to explicitly instantiate a template using a specific name. In the example codeblock, a `std::vector<Vector3f>` is wrapped as `Vector3fVector` in the target language.

Listing 4.1: Example SWIG interface file

```
%module backend
%{
#include "math/Vector3f.h"
%}

%include "std_vector.i"
%include "math/Vector3f.h"
%template(Vector3fVector) std::vector<Vector3f>;
```

In the case of generating interfaces in Java, multiple files will be generated by SWIG:

- `backendJNI.java` contains the Java Native Interface (JNI) function declarations (denoted by the keywords `final static native`) that are used to interface with the C++ code
- `backend_wrap.cxx` contains the native C++ implementation of the JNI function declarations above
- One java file for each wrapped C++ class that internally calls the JNI functions declared in `backendJNI.java`

4.3.3 Porting the Efficient RANSAC Algorithm Library to the ARM Architecture

To compile the library for ARM, a few adjustments to the code are necessary. The library uses the `xmmintrin.h` header file, which is an x86-specific header file that provides access to the Streaming SIMD Extensions (SSE) instruction set. In this case it is only used to allocate and free memory using the functions `_mm_malloc` and `_mm_free`. A conditional compilation directive is added to include the header file only if the target architecture is x86. If the target architecture is ARM, the standard library functions `malloc` and `free` are used instead.

Furthermore, the bundled compiler used by Android Studio (clang) is stricter than the gcc compiler and requires some minor adjustments to the code. Most notably, the error `explicit qualification required` is fixed by adding the `this` keyword to all member function calls. Other adjustments include fixes like removing obsolete keywords and changing deprecated functions of the C++ standard library with their modern counterparts.

This example illustrates that wrapping code into another architecture is not always straightforward and comes with its own set of challenges.

4.4 Rendering the Primitives

With the primitives detected using the RANSAC algorithm, the next step is to render them in the AR scene. This section first provides an overview of the OpenGL rendering pipeline in section 4.4.1 and the necessary steps to render the planes in section 4.4.2.

As planes are infinite in size, section 4.4.3 finally explains how to create a mesh that constrains the planes to the area where the points are located, using the convex hull algorithm.

4.4.1 OpenGL Rendering Pipeline

This section provides an overview of the OpenGL rendering pipeline and the necessary steps to render the primitives. Explanations are based on the book *Learn OpenGL* by Vries [Vri20] unless stated otherwise.

Polygon Meshes

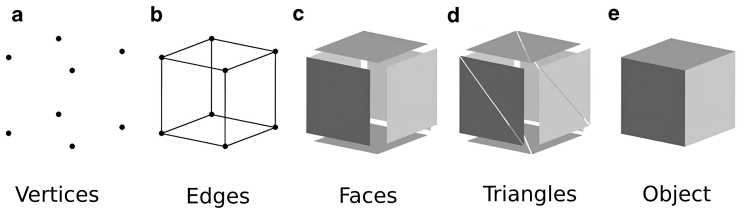


Figure 4.9: Mesh overview. Source: [Dör+19]

Rendering an object is achieved by creating a polygon mesh, which is a collection of vertices that define the faces of it. The vertices are connected by edges to form polygons, which in turn create the surface of the object. Polygons can be of any shape, but need to be planar, meaning that all vertices lie on the same plane. The most commonly used type of polygon is the triangle, as it is the simplest polygon that can define a surface and is guaranteed to be planar. The graphics pipeline is also optimized for triangles, as they are easy to rasterize and interpolate. Figure 4.9 shows the relationship between vertices, edges, and polygons in a mesh. [Dör+19]

Polygon meshes can be represented many ways. The simplest representation is a list of vertices, where the vertices are stored in a specific order to form the polygons. In OpenGL, this can be achieved by using the `glDrawArrays` function, which takes a buffer of vertices and draws them as triangles. However, a drawback of this approach is that vertices are often shared between multiple polygons, resulting in redundant vertex data, as is apparent in figure 4.9. To render the cube in the figure 4.9, only 8 vertices are

required, but since each of the 6 faces is rendered with two triangle, a total of 36 vertices would be required when using this approach. [Dör+19; Vri20]

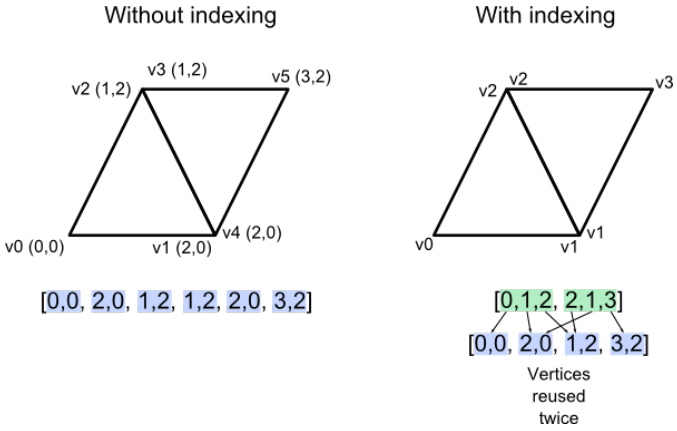


Figure 4.10: The concept of indexing. VBO in blue, EBO in green.

To address this issue, a common solution is the use of an indexed face-set, which is a datastructure consisting of two lists – one for the vertices and one for the indices. The vertices list contains all unique vertices of the mesh, while the indices list contains the indices of the vertices that form the polygons. In OpenGL, the `glDrawElements` function is used to draw indexed meshes It which requires two buffers to be bound: The Vertex Buffer Object (VBO) for the vertices and the Element Buffer Object (EBO) for the indices. [Dör+19; Vri20] Figure 4.10 illustrates the concept of indexing.

Shaders

This section provides an overview of shaders and their role in the rendering pipeline as detailed by [Vri20].

Shaders are isolated programs that run on the GPU and can be used to render objects. In OpenGL, they are written in the OpenGL Shading Language (GLSL). Two types of shaders are required to render an object: Vertex shaders and fragment shaders. See figure 4.11 for an overview of the graphics pipeline.

Vertex shaders are executed for each vertex defined in the vertex buffer, which is defined on the CPU and passed to the GPU by copying the data to the GPU's memory, e.g. by using the `glBufferData` function. On mobile devices, a separate buffer is often not needed, as Android devices usually only have a unified memory architecture, which

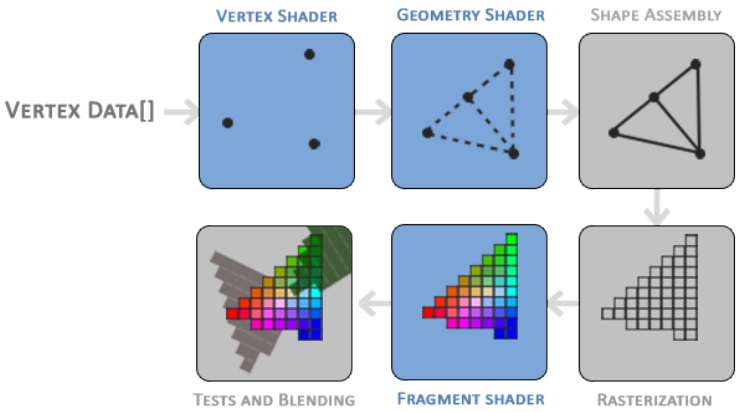


Figure 4.11: Graphics Pipeline. Blue boxes represent programmable stages. The Geometry Shader is optional and not covered in this thesis. Source: [Vri20]

allows for direct access to memory declared on the CPU from the GPU without copying the data. For example, the Pixel 7 is equipped with a proprietary Google Tensor G2 system-on-chip (SoC) Processor with integrated a Mali-G710 MP7 GPU[1] and 8GB of LPDDR5X RAM[2].

The vertex data can contain any attributes of the vertex, such as position, color, texture coordinates, or normals. The vertex shader is then executed for each vertex and can be used to transform the vertex position between different coordinate systems, or manipulate the vertex attributes on a per-vertex basis. To pass data back to the pipeline, the vertex shader can define `out` variables. In case of the vertex position, the output position is passed back to the pipeline by assigning it to the `gl_Position` variable using homogeneous coordinates. OpenGL expects the vertex position to be in normalized device coordinates (NDC), which range from -1 to 1 in all dimensions [Dör+19]:

$$(x, y, z) \in [-1, 1] \times [-1, 1] \times [-1, 1] \tag{4.12}$$

All coordinates outside this cube are considered outside the view of the camera and subsequently clipped, which means they will not be rendered. Note that in other implementations, the cube might be defined with $z \in [0, 1]$ [Dör+19]

[1] Thomas Claburn. *Google introduces Pixel 7 phones, related watch and Pixel.* URL: https://www.theregister.com/2022/10/06/made_by_google_pixel_phones/ (visited on 07/23/2024).
[2] *Pixel phone hardware tech specs - Pixel Phone Help.* URL: https://support.google.com/pixelphone/answer/7158570?hl=en#zippy=%2Cpixel (visited on 07/23/2024).

The graphics pipline then uses the NDC and performs the perspective divide, which divides the x, y, and z coordinates by the w coordinate of the homogenous coordinates, resulting in a reduction of dimension from 4 to 3. Then, the viewport transformation, which maps the NDC to screen space, parameterized by the screen width and height in pixels, is performed. Rasterization and interpolation of the vertex positions alongside all other vertex attributes is then performed across the primitive. The resulting elements are called fragments. The fragment shader is then executed for each rasterized fragment of the primitive and is expected to output a color by setting the `out vec4 FragColor` variable. The screen position of the fragment and all variables that were passed as `out` from the vertex shader can also be accessed in the fragment shader, with their values interpolated across the primitive. This allows for smooth color transitions or texture mapping across the primitive. Before the final color is written to the framebuffer, a depth test is performed to determine if the fragment is visible or covered by fragments from other primitives, in which case it is discarded. [Vri20]

Perspective Projection in OpenGL

As discussed in section 4.2.2, perspective projection is used to simulate the effect of objects appearing smaller the further away they are from the viewer. This transformation from 3D space to a 2D projection is handled by a projection matrix. In OpenGL rendering however, the projection matrix does not directly transform from view-space to screen-space. Instead, an intermediate coordinate system known as clip space is introduced, as shown in figure 4.12. Coordinates in clip space are in normalized device coordinates (NDC), which is a cube with coordinates ranging from -1 to 1 in all dimensions, as discussed in the previous section.

The projection matrix defines a frustum, which, depending on the projection type, can be a truncated pyramid or a cube, that defines the volume of space visible through the camera lens. In the case of the perspective projection, the frustum is a truncated pyramid, as shown in figure 4.13. The parameters of the frustum are defined by the field of view, aspect ratio, and near and far clipping planes. Multiplication of the vertex position with the projection matrix transforms the vertex into clip space. "The projection matrix [...] also manipulates the w value of each vertex coordinate in such a way that the further away a vertex coordinate is from the viewer, the higher this w component becomes" [Vri20]. When the coordinates are later divided by w in the perspective divide, it results in the desired perspective scaling effect. Points closer to the viewer have a

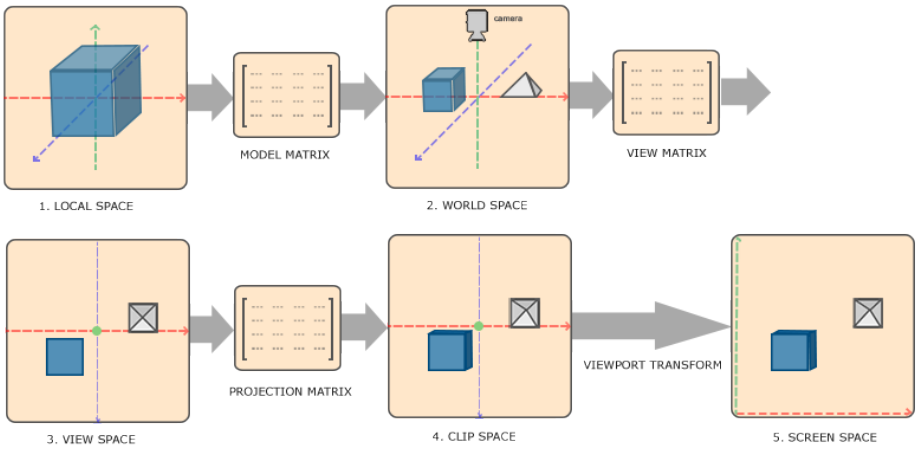


Figure 4.12: Extension of coordinate systems by clip space. Source: [Vri20]

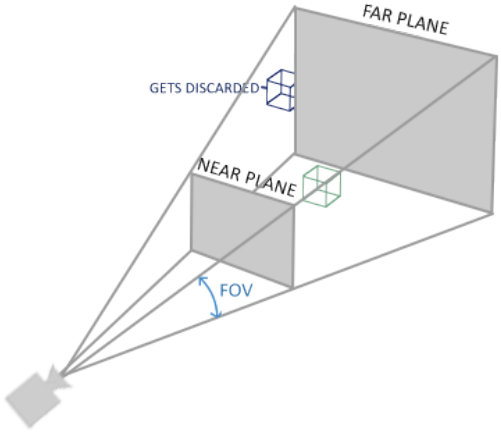


Figure 4.13: Frustrum defined by perspective projection. Source: [Vri20]

smaller w and are less affected by the divide, while points further away have a larger w and are reduced in size more significantly.

4.4.2 Rendering Planes

The RANSAC algorithm provides the parameterization of any detected plane using a normal vector n and the point p relative the worlds origin. Using OpenGL, a plane can be rendered by creating two triangles composed of 3 vertices each, with two corner vertices shared between the triangles. To render a plane from the parameterization, one

can first find two arbitrary vectors u and v that are perpendicular to each-other and to the normal vector n. Using u and v, the four corner vertices of the plane can then be calculated by adding and subtracting $u * size/2$ and $v * size/2$ from the point p.

Calculating an Arbitrary Perpendicular Vector The cross product of two vectors a and b is a vector that is perpendicular to both a and b, as long as a and b are not parallel. To calculate an arbitrary perpendicular vector to a given vector n, one can use any of the 3 basis vectors b_1, b_2, and b_3. Choosing any of the basis vectors that is not parallel to n will result in a perpendicular vector. To minimize floating point errors, which are largest for planes where n almost aligns with the chosen basis vector, the basis vector with the smallest dot product with n can be chosen. The normalized cross product of two vectors n and $b_{smallest}$ then yields a perpendicular vector to n.

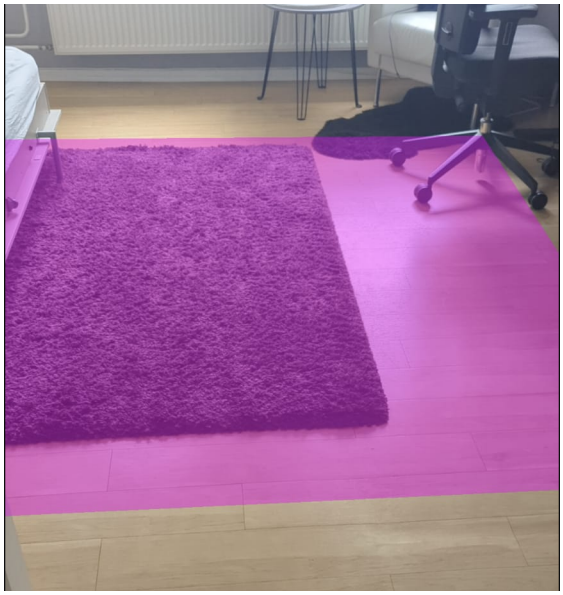


Figure 4.14: Rendered RANSAC plane with size of $2 \cdot 2$m

4.4.3 Constraining Planes to the Area Where the Points Are Located

As surfaces in the real world are not infinite, the planes detected by the RANSAC algorithm should also be constrained to a finite area. To achieve this, a bounding volume can be used. A bounding volume is a geometric shape that encloses a set of points or other shapes, and is often used to simplify collision detection or culling [GE02]. In this

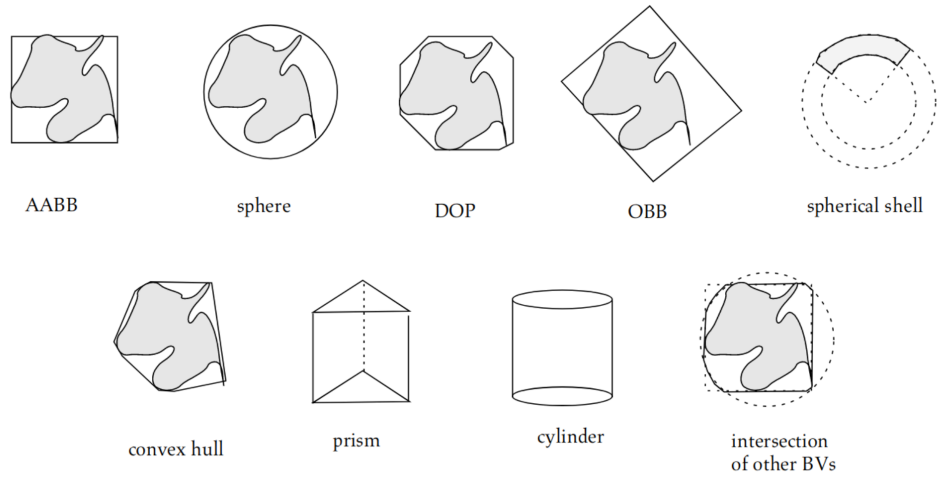


Figure 4.15: Most commonly used bounding volumes. Source: [GE02]

case the bounding volume will be used to generate a mesh that represents the plane, constrained to where its points are located. Common bounding volumes include axis-aligned bounding boxes (AABB), spheres, or oriented bounding boxes (OBB). A more complex bounding volume is the convex hull, which is the smallest convex polygon that contains all the points [GE02]. A rubber band can be used to illustrate the concept: If a rubber band is stretched around a set of points represented by nails in a board, the convex hull is the shape a rubber band takes when it is released, as illustrated in figure 4.20 [De +08]. As the convex hull is the most accurate commonly used bounding volume, it will be used for constraining the planes going forward.

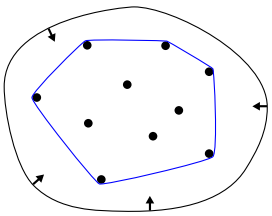
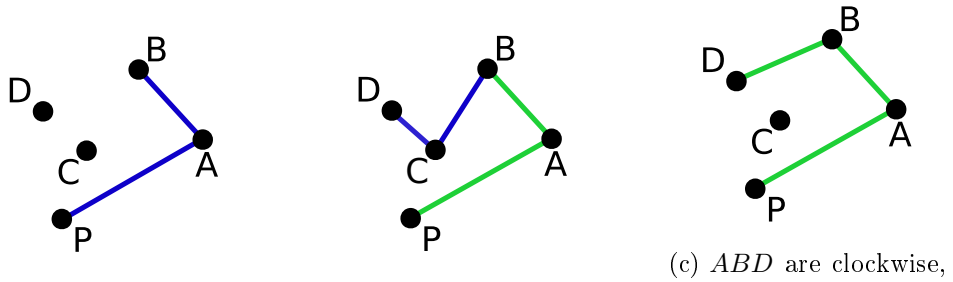


Figure 4.16: Rubber band analogy for the Convex Hull. Source: Wikimedia Commons

Convex Hull Algorithms

There are many algorithms available for calculating the convex hull, as it is a well-known problem in computational geometry. This section will mainly cover the Graham Scan. For a more comprehensive overview of available algorithms, see [De +08] section 1.4.

In 1972 [Gra72] proposed an algorithm to calculate the convex hull of n points in $O(n \log n)$ time, the Graham's Scan. The Graham Scan algorithm begins by selecting the point with the lowest y-coordinate (or the leftmost point in the case of a tie). Note that the original paper describes using the center of the points as the starting point, but common implementations use the lowest y-coordinate[3,4] – the core concepts remain the same. After selecting the starting point, it then sorts the remaining points in ascending order of their polar angles relative to the starting point. If two points have the same polar angle, only the farthest point is kept. Finally, it iteratively considers each point in the sorted order and determines whether it makes a clockwise or counterclockwise turn relative to the previous two points on the convex hull. If it makes a clockwise, the algorithm removes the previous point from the convex hull and repeats the process until all points have been considered.



(a) PAB are counterclockwise, A is kept

(b) BCD are clockwise, C is removed

(c) ABD are clockwise, D is kept, resulting hull

Figure 4.17: Example of the Graham Scan algorithm applied to a set of points. Source: Wikimedia Commons

[And79] later proposed a variation of the algorithm called the monotone chain in 1979, which sorts the points by their x-coordinate and y-coordinates instead of their polar angle. It then constructs the upper and lower hulls separately, which are then merged to form the convex hull.

[3] *Graham scan*. In: *Wikipedia*. Page Version ID: 1219560976. Apr. 18, 2024.
[4] *Convex Hull using Graham Scan*. GeeksforGeeks. Section: DSA. July 24, 2013. URL: https://www.geeksforgeeks.org/convex-hull-using-graham-scan/ (visited on 06/22/2024).

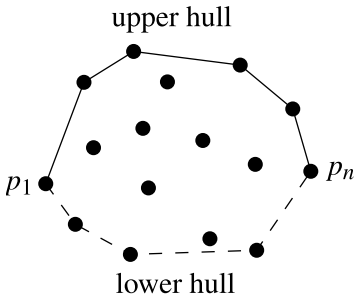


Figure 4.18: Upper and lower hull of Andrew's monotone chain algorithm. Source: [De +08]

More recent algorithms like Chan's algorithm [Cha96] combine the Graham Scan with other techniques to solve the problem in $O(n \log h)$ time, where h is the number of points of the convex hull. Thus, in case h is much smaller than n, these algorithms are faster than the Graham's Scan.

For the purpose of this thesis, the Chan's algorithm would be the most efficient choice, as the number of points of the convex hull is expected to be much smaller than the total number of points. However, as Chan's algorithm is more complex to implement and the performance impact of the convex hull algorithm is small compared to running the RANSAC algorithm, Andrew's Monotone Chain Algorithm is used instead.

Rendering Planes Constrained by the Convex Hull

To render a plane constrained by the convex hull, a triangle mesh can be created with triangles consisting of two subsequent vertices of the hull and the centroid of the hull as the third vertex each, as shown in figure 4.19.

Figure 4.20 shows the result of the convex hull algorithm implemented into the application. Prior to detecting the planes, the smartphone has been moved around the scene to capture the points from different angles.

To make it easier to distinguish between different planes when detecting multiple planes, as will be shown in the evaluation, section 5.4, the planes are rendered in random colors by assigning a color vertex attribute to each vertex of a plane.

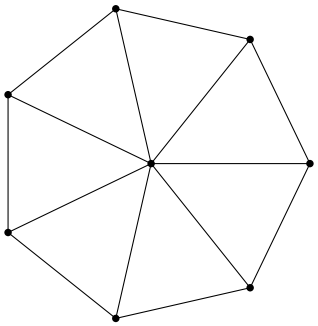
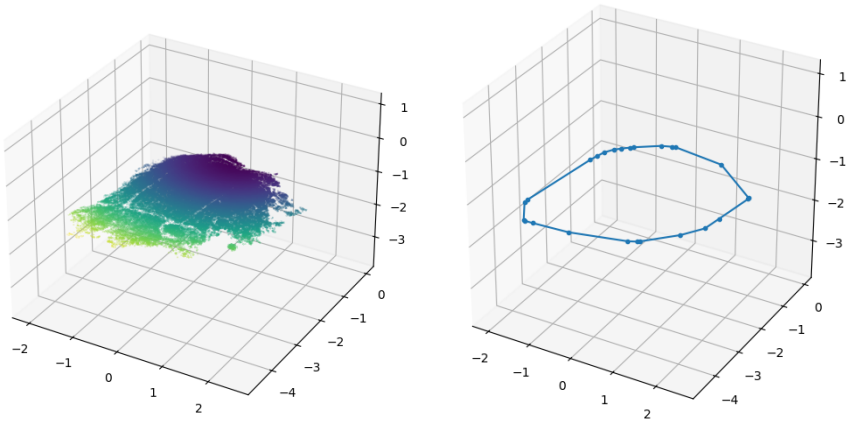


Figure 4.19: Triangle mesh for convex hull polygon with 7 vertecies using centroid



(a) Point data

(b) Convex Hull

(c) Rendered Primitive

Figure 4.20: Convex Hull with real data

5 Evaluation

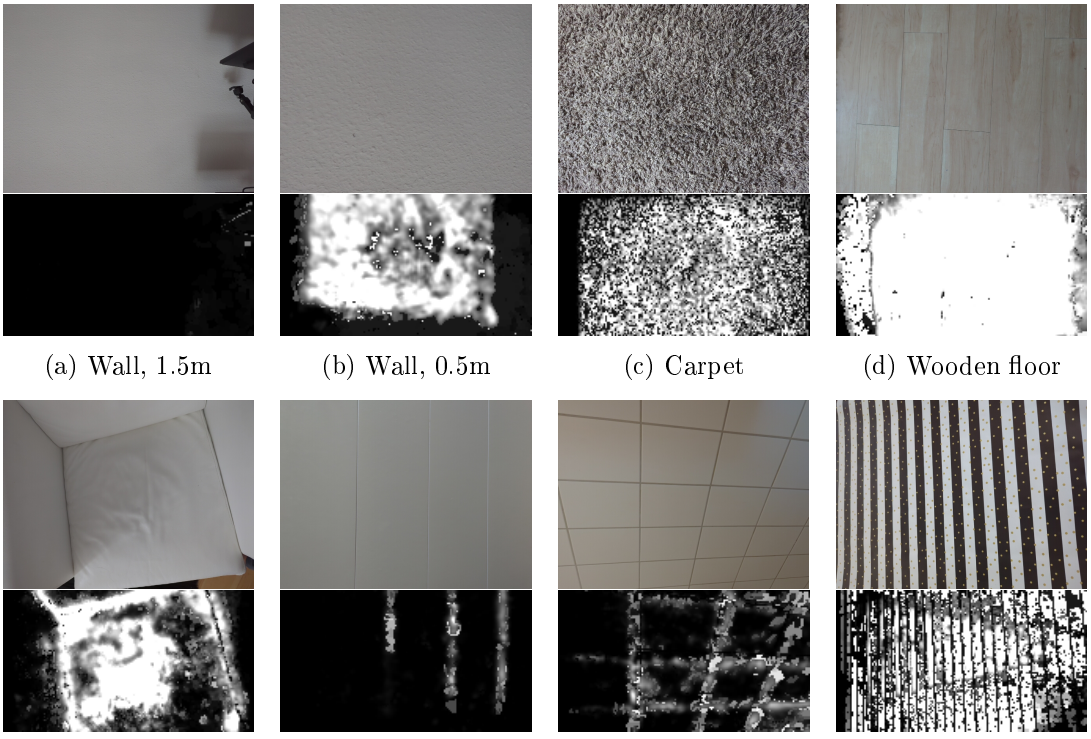
In this chapter, the individual algorithms, as well as the final performance of the combined system, will be evaluated. First, the depth from motion algorithm will be evaluated based on different materials and their resulting confidence maps in section 5.1. Section 5.2 will then evaluate the two point cloud datastructures as described in section 4.2.3. In section 5.3, the RANSAC algorithm will be evaluated based on synthetic data, as well as real world data. Finally, in section 5.4.2, the application performance will be evaluated on the use case of a full room scan. CloudCompare [Dan+], an open-source tool for working with 3D point clouds, will be used for the evaluation of the point clouds and the RANSAC algorithm.

5.1 Depth from Motion

As the depth from motion algorithm greatly differs in its performance based on the amount of texture in the captured scene [Goo], this section will evaluate different materials based on their resulting confidence maps, as retrieved from the ARCore Raw Depth API. In figure 5.1, the confidence maps of different materials are shown - the camera image on top, the confidence map on the bottom.

In figure 5.1b and 5.1a, the confidence maps of a wall at different distances is shown. It is apparent that the confidence map of the wall at 1.5m distance is almost completely black, meaning low confidence. This is due to the lack of any texture on the wall at this distance. When moving closer to the wall, the structure of the wall becomes more apparent and the confidence increases greatly.

In figure 5.1c, a carpet with a lot of texture is shown, the resulting confidence map has fairly high confidence on average, but also has dark sports throughout the map. This could be a result of the repetitive pattern of the carpet.



(a) Wall, 1.5m (b) Wall, 0.5m (c) Carpet (d) Wooden floor

(e) Leather armchair (f) White furniture (g) Bathroom tiles (h) Gift paper

Figure 5.1: Comparison of confidence maps of different materials. Camera image on top, confidence map on bottom.

The highest confidence of all tested materials is achieved on a wooden floor, as shown in figure 5.1d. This can be attributed to the high amount of texture and value variance throughout the wood, while not being repetitive. The seams between the wooden planks are also clearly visible.

Figure 5.1e shows a leather armchair, which has high confidence on the seams of the chair and creases of the leather, with low confidence in other areas, where the leather is smoother. A similar pattern is visible on white furniture and bathroom tiles, in figures 5.1f and 5.1g respectively, where the white surface has very low confidence, while the seams between the surfaces have increased confidence. The gift paper shown in figure 5.1h shows a similar effect, but less pronounced. As the surface itself is striped with dots randomly placed throughout, the confidence is higher overall, but the striped pattern is still visible in the confidence map, as the stripes themselves contain little texture.

These examples show how the amount of texture on a given surface greatly influences the confidence of the depth from motion algorithm, as expected. The algorithm performs

best on surfaces with high texture, while struggling with low texture surfaces. Small repetitive patterns can also lead to low confidence, as visible in the carpet example. It is also important to note that the texture captured by the camera is important, not the texture of the material itself. This can be seen in the first example of the wall: When capturing from afar, the cameras resolution is too low to capture the texture, but when moving closer to the wall, the texture of the wall becomes more apparent and the confidence increases. Lighting conditions will also influence the confidence of the algorithm, as angled light will create shadows that will increase the texture of the surface [Goo] (not shown in the comparison).

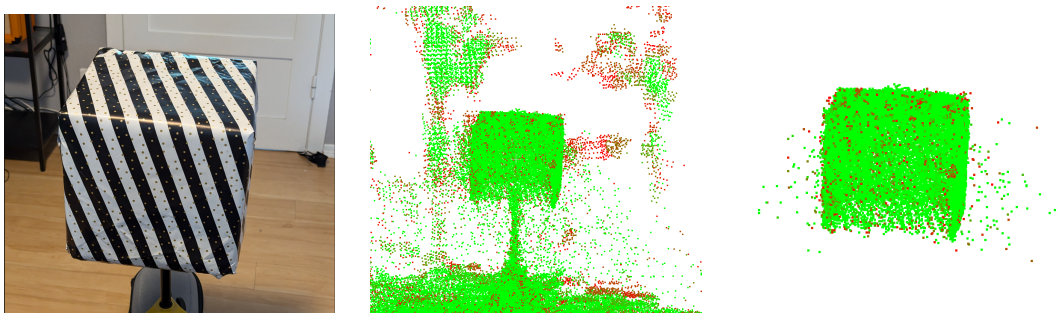
5.2 Point Clouds

This section will evaluate the two point cloud datastructures as described in section 4.2.3. To evaluate the datastructures, the point cloud is transferred from the mobile application to a computer using a simple HTTP server with a single endpoint, that accepts a file using a POST request and saves it to disk. The Android application will then stream the point cloud data, serialized as a PLY file, to the server on a push of a button. A PLY file is a simple file format for storing 3D point cloud data in plain text and allows for saving custom properties for each vertex, such as the confidence value of the point. In all the following images of the point clouds, the confidence value is color-coded, with green being high confidence and red being low confidence.

Listing 5.1: Example PLY file

```
ply
format ascii 1.0
element vertex 6
property float x
property float y
property float z
property float confidence
property uchar red
property uchar green
property uchar blue
end_header
0.1851168 -0.77074146 -0.92582846 1.0 0 255 0
```

```
−0.061887983 −0.7755803 −0.79551744 1.0 0 255 0
−0.058628604 −0.7638883 −0.7860476 0.80784315 245 10 0
−0.07720285 −0.75610626 −0.80394375 0.81960785 230 25 0
−0.19774273 −0.6645372 −2.5003948 1.0 0 255 0
−0.09499544 −0.028413996 −2.553247 0.99607843 5 250 0
```



(a) Color image of setup (b) Captured point cloud (c) Segmented point cloud

Figure 5.2: Test setup for capturing point clouds

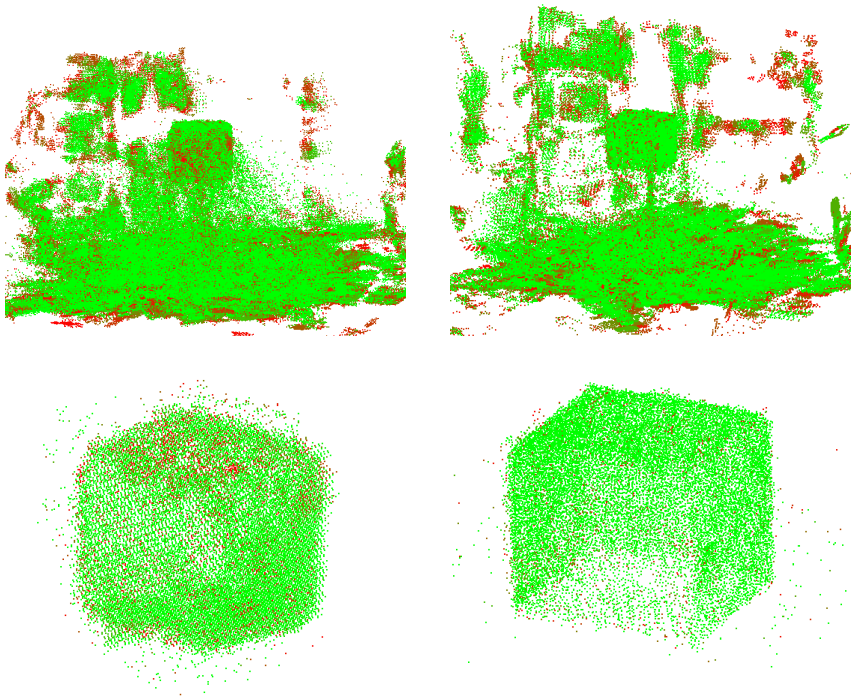
For test data, a cube wrapped with the same gift-paper as in figure 5.1h is used, called *test cube* from now on. The bottom of the test cube is hollow, so in total, 5 faces are visible. The cube is propped up on stand, to allow for easy segmentation of the point cloud later on, in order to remove the floor and other objects from the point cloud. Reproducibility is ensured by using controlled lighting conditions and a fixed setup. To capture a scan of the test cube, the camera is moved around the cube 3 times, while also moving the camera up and down, to capture multiple angles for the depth from motion algorithm to work to its full potential. The full setup is shown in figure 5.2.

5.2.1 Quantization Octree vs. Epsilon Octree

A point cloud with a resolution of $s = 0.005$m or 5mm is captured for both the quantization octree and the epsilon octree. In the resulting segmented point cloud, the quantization octree has a total of about 90.000 points, while the epsilon octree only has about 20.000 points. This can be explained due to the quantization octree always ending up with the maximal resolution with enough data added, as it is functionally a three-dimensional grid. The epsilon octree, on the other hand, will only add points that are within the epsilon threshold of the current point, which means that gaps between points are at least epsilon wide, but can be wider. For example, if two points almost 2 epsilon apart, no

points will be added between them and the gap will be approximately 2 epsilon wide. As epsilon is set to s, the epsilon octree will have fewer points than the quantization octree with the same resolution. To compare the two datastructures, the quantization octree will be used with a lower resolution of $s = 0.010$m, which results in a number of points of about 22.000, which is comparable to the epsilon octree at $s = 0.005$m.

Figure 5.3 shows the raw and segmented point clouds of both datastructures. From inspecting the raw point cloud visually, it is apparent that the quantization octree has a higher amount of noise. While the cube is easily recognizable in the point cloud of the epsilon octree, the point cloud of the quantization octree has a lot of noise around the cube. The quantization octree also has more low-confidence points than the epsilon octree. This is due to the quantization octree always adding points to the cloud that fall within the threshold, while the epsilon octree only adds / updates points based on a condition that takes the confidence of the point into account and ignores lower confidence points that are close to higher confidence points. As such, the epsilon octree improves the accuracy of the point cloud when new, higher confidence points are added.

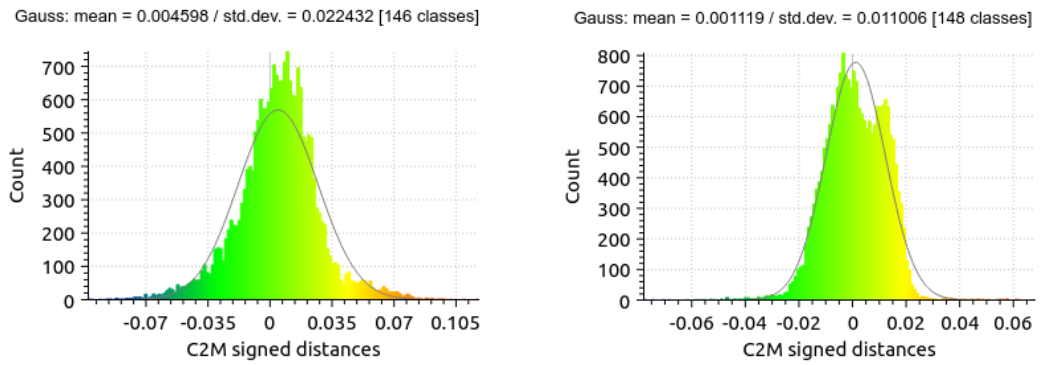


(a) Quantization Octree, $s = 0.010$m (b) Epsilon Octree, $s = 0.005$m

Figure 5.3: Comparison of Quantization Octree and Epsilon Octree. Raw point cloud on top, segmented on bottom.

To calculate statistics on the point clouds, the real box is measured and a primitive box with matching dimensions is created in CloudCompare. Using the *fine registration (ICP)* feature, it is possible to align the point clouds to the primitive box. The underlying algorithm of the fine registration feature is the Iterative Closest Point (ICP) algorithm, which iteratively minimizes the mean square distance between a "model" shape and a "data" shape [BM92]. Once the point clouds are aligned to the primitive, the distance between the points of the point cloud and the primitive box can be calculated using the *Cloud/Mesh Distance* feature in CloudCompare. The result of this calculation can be seen in figure 5.4. From the histogram, it is apparent that the quantization octree has higher noise than the epsilon octree. This can also be seen in the standard deviation, which is 0.022 for the quantization octree and 0.011 for the epsilon octree. This confirms the first suspicion from visual inspection of the point clouds. The mean distance is also higher for the quantization octree, with 0.0046 compared to 0.0011 for the epsilon octree. This can be understood as points being 0.5cm and 0.1cm away from the real surface in mean, respectively. If these points were to be used for primitive detection, this would indicate how far off the detected primitives would be from the real surface.

These results suggest that the epsilon octree is an improvement over the quantization octree in terms of accuracy, both when it comes to the amount of noise and the distance to the real surface. As such, the epsilon octree will be used for following evaluations.



(a) Quantization Octree (b) Epsilon Octree

Figure 5.4: Histogramm of distances between segmented point cloud points and primitive mesh

5.3 RANSAC Algorithm

In this section, the RANSAC algorithm will be evaluated based on a model of a cube, first using synthetic data in section 5.3.1 and then using real world data in section 5.3.2.

5.3.1 Tests on Synthetic Cube

A unit cube will be used as a test object to evaluate the RANSAC algorithm. The cube has been generated with a side length of 1 and a sampling rate of 0.01. This would equate to a cube with a side length of 1m and a distance of 1cm between points in the real world, which is comparable to what was used in section 5.2.1.

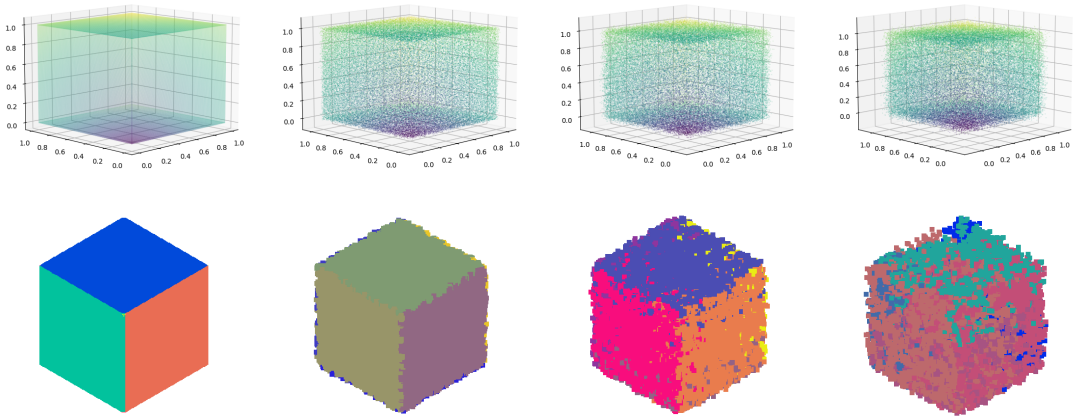
Resilience to Noise

In figure 5.5, the unit cube is shown with varying noise levels using gaussian noise. The noise level is defined as the standard deviation of the noise added to the points.

With a noise level of 0.01, the cube is perfectly reconstructed. Increasing the noise level to 0.02, the cube is still recognized mostly correctly. Starting from noise level to 0.03, the default parameters do not yield correct results. To achieve correct results, the epsilon parameter of the RANSAC algorithm has to be increased to 0.2. This leads to 6 faces being recognized correctly, but some points not being assigned to the correct faces, as the with each primitive extraction pass, points are extracted that lie within 0.2 of the recognized plane.

Resilience to Missing Data

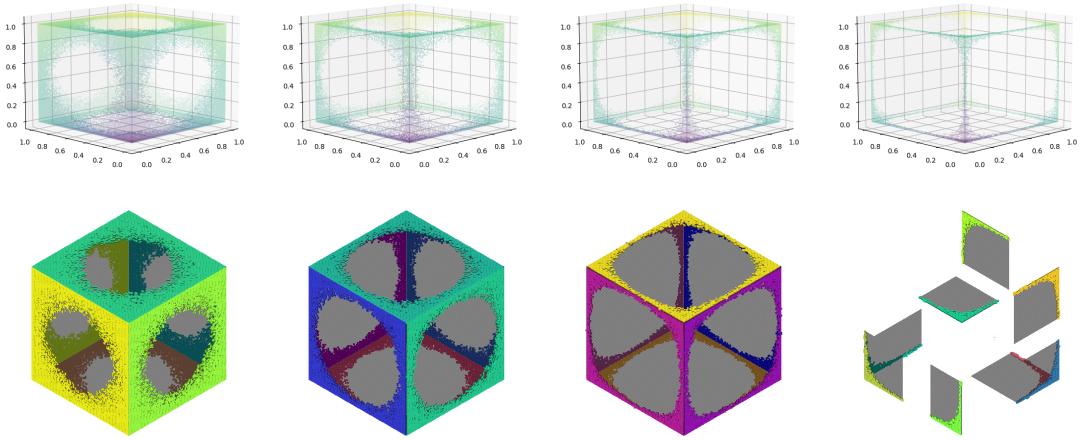
As a big problem of depth from motion techniques is the lack of depth information in areas with minimal texture, the resilience to missing data is crucial. In figure 5.6, points towards the center of the cubes surfaces have been removed. This mimics the structure of real world data from the Depth API, as edges are often detected more accurately than surfaces, as visible in the confidence maps of the bathroom tiles and white furniture in figure 5.1. To simulate this effect, points have been removed with an increasing probability based on the quadratic distance to the center of the face the points belongs to.



(a) Noise level 0.00 (b) Noise level 0.01 (c) Noise level 0.02 (d) Noise level 0.03

Figure 5.5: Unit cube with varying noise level

The algorithm is able to correctly recognize the faces of the cube with a missing level up to 24. With a missing level of 48, the algorithm still detects the planes, but doesn't assign all points to the faces. With increasing missing level, the n parameter, which defines the minimum number of points required to fit a primitive, is also required to be lowered. In real world applications this would lead to more false detections, primitives being detected where there are none.

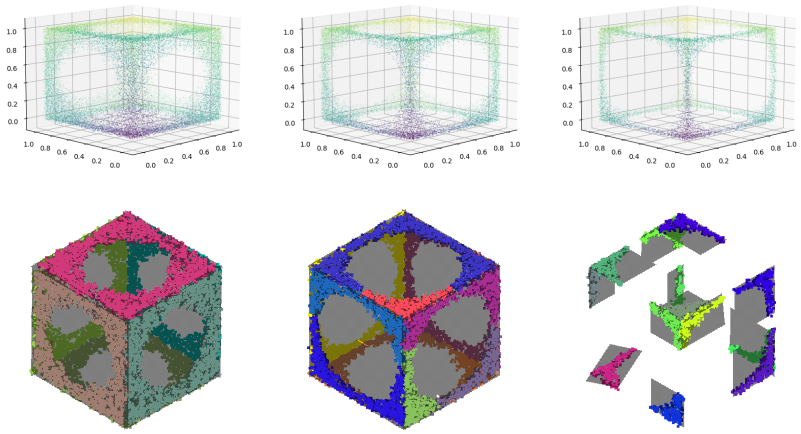


(a) Missing level 6 (b) Missing level 12 (c) Missing level 24 (d) Missing level 48

Figure 5.6: Unit cube with varying missing level

Resilience to Noise and Missing Data Combined

When combining both noise and missing data, the quality of the detection is much worse. In figure 5.7, the cube is shown with a noise level of 0.01 and varying missing levels. With noise level 0.01 and missing level 6, the cube is still recognized correctly. With missing level 12, more than 6 faces are being recognized, but all the points are still assigned to primitives. This might be due to the shapes not being recognized as connected shapes, which causes the algorithm to only fit the largest connected component of the points and will create a new parameterization for the remaining points. To achieve results with a missing level of 24, the epsilon parameter had to be increased to 0.015 to achieve any results at all. As visible from the figure, the cube is not recognized correctly, with only small parts of the cube being recognized. When increasing the noise level to 0.02, the algorithm yields no satisfactory results with any missing level, even with tweaking the parameters.



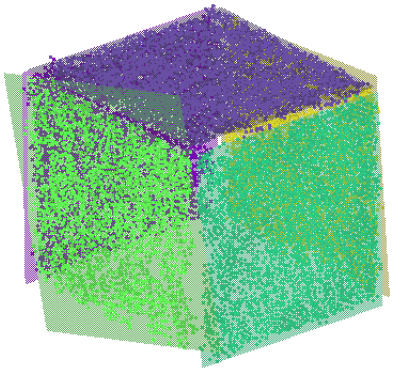
(a) Missing level 6 (b) Missing level 12 (c) Missing level 24

Figure 5.7: Unit cube with noise level = 0.01 and varying missing level

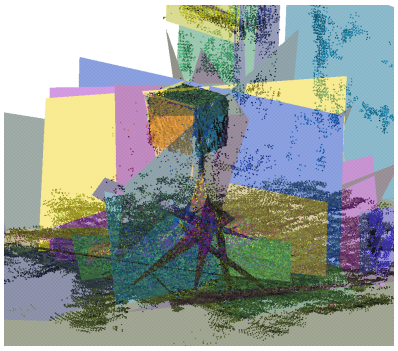
5.3.2 Tests on Real World Data

Figure 5.8 show the results of the RANSAC algorithm on the dataset created with the epsilon octree, as seen in figure 5.3b. Note that CloudCompare uses AABBs to represent the planes, instead of the convex hull as used in the application. In the segmented dataset, the cube is recognized correctly, with all 5 faces being recognized. In the raw dataset,

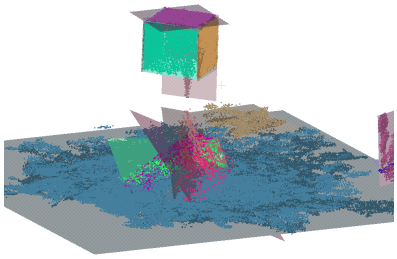
the faces of the cubes are still recognized, but the algorithm also recognizes other planes in the dataset, some of which contain a subset of the points of the cube. This shows a limitation of the RANSAC algorithm, which is similar to [TLG07] findings, where plane primitives are constructed from points which do not belong to the same plane. [SWK07] solution to this problem is using a bitmap to detect connected shapes, as mentioned in section 4.3.1. Only the largest connected shape in a given shape candidate is kept. As the points that are not part of the largest connected shape are kept in the dataset, the algorithm will still consider them in the next iterations. However, if the resulting separate primitives then do not cross the support threshold, they are filtered out. In other words, the bitmap epsilon parameter β can be used to adjust the resolution of the bitmap, with higher values leading to shapes being detected as separate shapes quicker, while the minimum support threshold n can be used to filter out shapes that are not supported by enough points.



(a) Segmented dataset, default parameters



(b) Raw dataset, $n = 400, \epsilon = 0.024, \beta = 0.047$



(c) Raw dataset, $n = 400, \epsilon = 0.024, \beta = 0.005$

Figure 5.8: RANSAC results on real data set collected using the epsilon octree

In the case of the dataset visible in figure 5.8, the default value of the bitmap epsilon parameter was set to $\beta = 0.024$. Ideally, the bitmap epsilon parameter would be set to the sample resolution of the dataset [SWK07]. When raising the resolution to $\beta = 0.005$, the detection results of the cube are improved, while still detecting some artifacts that shouldn't be detected. With these parameters, the planes in the background are also no longer detected, as their effective sampling resolution is too low, which causes them to be detected as many separate shapes that do not cross the support threshold. This shows that applying the algorithm to a dataset with vastly varying sampling resolutions will lead to suboptimal results, as the parameters need to be set for the whole dataset. This is problematic for the use case in combination with the depth from motion algorithm, as the Raw Depth API will have low confidence in areas with low texture, which will cause lower sampling rates, or completely missing data, which will in turn lead to connected shapes not being present and being filtered out. It is therefore difficult to find a parameterization for the algorithm that fits both highly textured and low-textured objects.

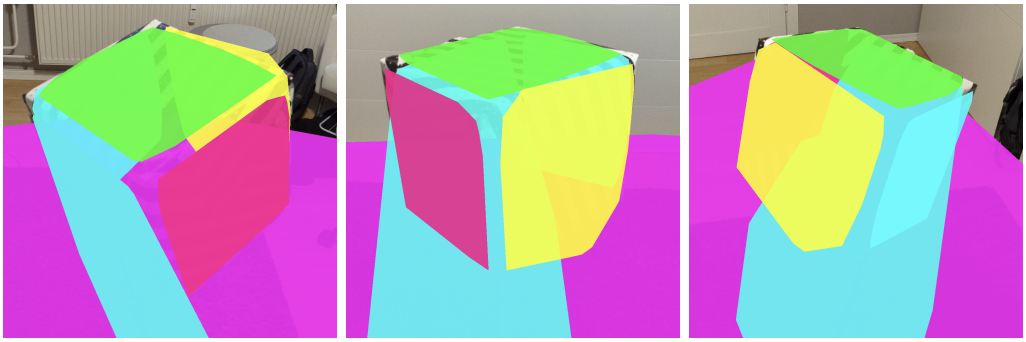
5.4 Application Performance

This section will evaluate the combined performance of the application, first on the test cube, then on a full room scan.

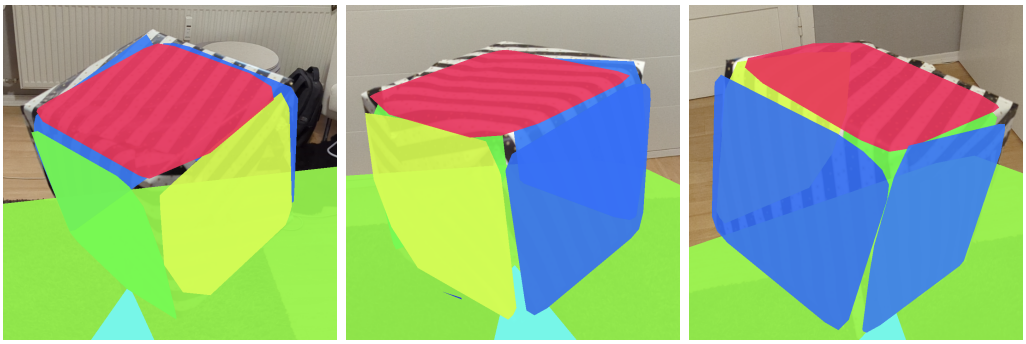
5.4.1 Test Cube

Figure 5.9 shows the results of a scan of the test cube under the same procedure as in section 5.2 with varying sample resolution s. Note that the point cloud is not segmented in the application, thus other planes (namely the floor) are also detected in the background. The parameters used aside from s will be discussed after describing the results.

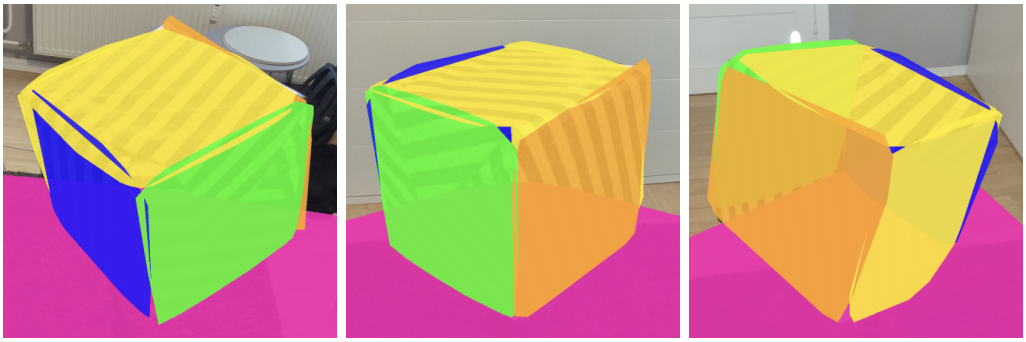
With an epsilon octree resolution of $s = 0.015$, one of the faces is merged with points from the stand the cube is placed on. Starting with a resolution of $s = 0.010$, all the 5 faces are recognized correctly and the convex hull algorithm correctly creates a matching polygon for each face, with small gaps between the faces. Increasing the resolution further to $s = 0.005$ improves the accuracy of the edges of the planes slightly. While there are no longer gaps between the faces, some amount of clipping is now visible in the polygons.



(a) $s = 0.015, n \approx 60.000$



(b) $s = 0.010, n \approx 125.000$



(c) $s = 0.005, n \approx 250.000$

Figure 5.9: Detection results of the test cube using the mobile application

Using a trial-and-error approach, the parameters of the RANSAC algorithm were adjusted to achieve the best trade-off between performance, the amount of detected planes and noise. The final parameters used are:

- $0.010 \geq s \geq 0.005$ – The resolution of the epsilon octree. Higher resolutions than 0.005 lead to a significantly higher amount of points and thus worse runtime performance without noticeable improvements in the detection results, while lower resolutions than 0.010 lead to visibly worse results, with outliers having a bigger impact on the detection results, as visible in figure 5.9a.
- $n = \left(\frac{1}{s}\right)^2 \cdot 0.15^2 = 900$ – The minimum number of points required to fit a primitive. Equal to the amount of points on a $15cm^2$ plane, if uniformly sampled with s. Realistically, this number will be lower, as the points are scattered across the normal of the plane. However, the epsilon octree also has a lower actual resolution than s, as discussed in section 5.2.1, which should cancel out the effect. The calculation is only a rough estimate and proved to show good results.
- $\epsilon = 0.022$ – The epsilon parameter of the RANSAC algorithm. Lower values lead to planes being detected as multiple planes along the planes normal. This parameter varies depending on noise of the data. Two standard deviations contain 95.5 percent of all data, which is used as a starting point. As the epsilon octree showed a standard deviation of 0.011, a value of 0.022 is chosen and shows good results.
- $\beta = 1.5s$ – The epsilon parameter of the bitmap. As discussed in 5.3.2, settings this parameter is a trade of between detecting shapes that have missing data and filtering out shapes that are not connected in reality. $1.5s$ is a rather strict value that filters out most shapes with gaps in their data due to missing texture, however it greatly reduces the amount of artifacts.

Other parameters are kept at their default values. As all parameters except ϵ are relative to s, it should be easy to adapt the parameters to different devices that might offer depth maps with other accuracies - only the resolution of the epsilon octree has to be adjusted. As for ϵ, the parameter needs to be adjusted based on the noise level of the data, which might vary based on the device and the lighting conditions.

As for runtime performance, the device remains responsive with $\epsilon \geq 0.005$, however it heats up quickly and the application experiences a significant slowdown after a few

scans. This is due to both the creation of the point cloud using the epsilon octree and the RANSAC algorithm being very computationally expensive.

5.4.2 Full Room Scan

For a full room scan, the device is moved around the room in a similar fashion as described in section 5.2. The room is scanned from multiple angles to capture all surfaces. In total, about 500.000 points are collected and processed by the RANSAC algorithm. The processing time of the RANSAC algorithm is about two minutes, which is significantly longer than for the test cube. Figure 5.10 shows the results of a full room scan. In general, the results can be described as mixed. The walls and floor are mostly recognized correctly, while some parts of them are missing. The decent results on the wall can be explained by the lighting conditions, as the walls are lit from a window at an angle, which creates shadows that increase the texture of the wall. In contrast, white furniture is almost not recognized at all, as the low texture of the material leads to missing data. When it comes to smaller objets, the algorithm struggles to correctly recognize them as is visible from the capture of the desk, where artifacts are visible in the detection results.

In conclusion, the application is able to detect primitives in a full room scan, however the results are not sufficient to build a full mesh of the room. The processing time of the RANSAC algorithm is also too high to be used in most real-time applications.

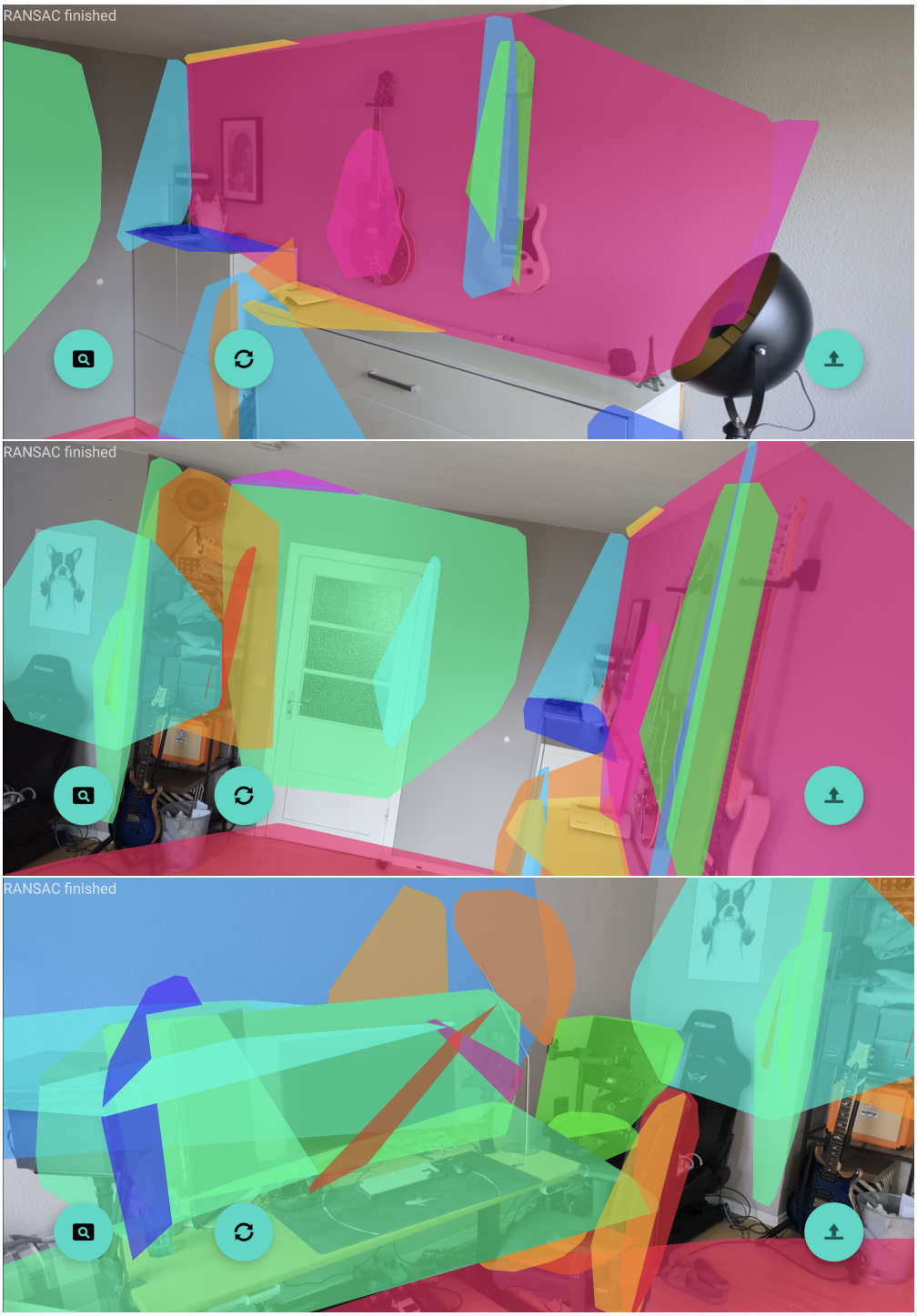


Figure 5.10: Detection results of a full room scan using the mobile application. $s = 0.010, n \approx 500.000$

6 Conclusion and Outlook

This thesis introduced a system for detecting primitives in a point cloud generated from depth data obtained from the Google ARCore API. The system utilizes a custom octree implementation to efficiently store and update the point cloud information in real time, that is then used to detect planes using the RANSAC algorithm. The resulting planes are then used to construct a triangle mesh representing their convex hulls, which is overlaid onto the camera feed.

The system demonstrates high accuracy in detecting planes and constructing meshes for highly textured objects, such as the test cube. However, the data quality of the Depth API with low-textured objects like white furniture prove to be a challenge for the system, as the depth data for these materials contains gaps, resulting in artifacts in the detection results. Mitigating these artifacts requires increasing the bitmap resolution, but this approach causes lower-textured materials to go undetected. For these reasons, the system is not suitable for creating a full mesh-representation of a scene.

Nevertheless, the system developed in this thesis provides a solid foundation for further research and development in processing depth data from ARCore or similar APIs to create meshes from a scene. One major limitation is missing datapoints in the data obtained through the Depth from Motion technique used by ARCore. Therefore, evaluating the use of interpolated data from the Full Depth API and its potential for improving the results of the RANSAC algorithm would be worthwhile. Exploring the data quality of different devices, with and without depth sensors, and its impact on the performance of the primitive detection algorithm would also be an interesting direction for future research.

Testing different primitive detection algorithms is another area for further research. Chapter 2 provides an overview of various primitive detection algorithms, while [KYB19] provides an extensive survey of algorithms specialized for different contexts and application. The design of the system allows for easy integration of different algorithms without

affecting other components and the ability to collect point data with a smartphone and transfer it to another device makes it easy to evaluate and compare different algorithms on real data collected from the Depth API. As a concrete example, [KYB19] lists that the Hough Transform is more robust to incomplete data than RANSAC. Consequently, it would be valuable to investigate whether the Hough Transform is more suitable for the depth data obtained from the Depth API.

As far as expanding the system, implementing the ability to detect more complex primitives, such as cylinders and spheres, would be a logical next step. Addressing the issue of data quality also opens up opportunities for further research, such as automatic segmentation, classification, and labeling of objects in a scene.

Bibliography

[And79] A. M. Andrew. „Another efficient algorithm for convex hulls in two dimensions“. In: *Information Processing Letters* 9.5 (Dec. 16, 1979), pp. 216–219.

[Bal81] D. H. Ballard. „Generalizing the Hough transform to detect arbitrary shapes“. In: *Pattern Recognition* 13.2 (Jan. 1, 1981), pp. 111–122.

[Bea96] D. Beazley. „SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++“. In: *Tcl/Tk Workshop*. Vol. 43. July 10, 1996, p. 74.

[BM92] P.J. Besl and Neil D. McKay. „A method for registration of 3-D shapes“. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (Feb. 1992). Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence, pp. 239–256.

[Cha96] T. M. Chan. „Optimal output-sensitive convex hull algorithms in two and three dimensions“. In: *Discrete & Computational Geometry* 16.4 (Apr. 1, 1996), pp. 361–368.

[Dan+] Daniel Girardeau-Montaut et al. *CloudCompare [GPL software]*. Version 2.x. Retrieved from http://www.cloudcompare.org/.

[De +08] Mark De Berg et al. *Computational Geometry: Algorithms and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[DH72] Richard O. Duda and Peter E. Hart. „Use of the Hough transformation to detect lines and curves in pictures“. In: *Communications of the ACM* 15.1 (Jan. 1, 1972), pp. 11–15.

[Dör+19] Ralf Dörner et al., eds. *Virtual und Augmented Reality (VR/AR): Grundlagen und Methoden der Virtuellen und Augmentierten Realität*. Berlin, Heidelberg: Springer, 2019.

[FB74] R. A. Finkel and J. L. Bentley. „Quad trees a data structure for retrieval on composite keys“. In: *Acta Informatica* 4.1 (Mar. 1, 1974), pp. 1–9.

[FB81] Martin A. Fischler and Robert C. Bolles. „Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography“. In: *Communications of the ACM* 24.6 (June 1, 1981), pp. 381–395.

[GE02] Gabriel Zachmann and Elmar Langetepe. „Geometric Data Structures for Computer Graphics“. In: *The Eurographics Association* (2002).

[Gla94] Andrew S. Glassner. *Graphics Gems*. Reissue edition. San Diego: Morgan Kaufmann, Jan. 5, 1994. 864 pp.

[Goo] Google LLC. *ARCore Developer Documentation*. Google for Developers. URL: https://developers.google.com/ar/develop (visited on 02/09/2024).

[Gra72] R. L. Graham. „An efficient algorith for determining the convex hull of a finite planar set“. In: *Information Processing Letters* 1.4 (June 1, 1972), pp. 132–133.

[GT22] Google LLC and Eric Turner. *ARCore Raw Depth*. Google Codelabs. May 11, 2022. URL: https://codelabs.developers.google.com/codelabs/arcore-rawdepthapi (visited on 02/09/2024).

[Hou62] Paul V. C. Hough. „Method and means for recognizing complex patterns“. U.S. pat. 3069654A. Individual. Dec. 18, 1962.

[Kri17] Kris Kitani. „Computer Vision - Lecture 8.2: Camera Matrix“. Carnegie Mellon University Robotics Institute, 2017.

[KYB19] Adrien Kaiser, Jose Alonso Ybanez Zepeda, and Tamy Boubekeur. „A Survey of Simple Geometric Primitives Detection Methods for Captured 3D Data“. In: *Computer Graphics Forum* 38.1 (Feb. 2019), pp. 167–196.

[LMM98] Gabor Lukács, Ralph Martin, and Dave Marshall. „Faithful least-squares fitting of spheres, cylinders, cones and tori for reliable segmentation“. In: *Computer Vision — ECCV'98*. Ed. by Hans Burkhardt and Bernd Neumann. Berlin, Heidelberg: Springer, 1998, pp. 671–686.

[Sam80] Hanan Samet. „Deletion in two-dimensional quad trees“. In: *Commun. ACM* 23.12 (Dec. 1, 1980), pp. 703–710.

[Sam89] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Reading, Mass: Addison Wesley, Aug. 1, 1989. 510 pp.

[Sha98] C.M. Shakarji. „Least-squares fitting algorithms of the NIST algorithm testing system“. In: *Journal of Research of the National Institute of Standards and Technology* 103.6 (Nov. 1998), p. 633.

[Shr21] Shree K. Nayar. „First Principles of Computer Vision - Hough Transform“. Department of Computer Science, Columbia University in the City of New York, 2021.

[SWK07] R. Schnabel, R. Wahl, and R. Klein. „Efficient RANSAC for Point-Cloud Shape Detection“. In: *Computer Graphics Forum* 26.2 (June 2007), pp. 214–226.

[Sze22] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 2nd ed. 2022 edition. Cham: Springer, Jan. 5, 2022. 947 pp.

[TLG07] Fayez Tarsha-Kurdi, Tania Landes, and Pierre Grussenmeyer. „Hough-Transform and Extended RANSAC Algorithms for Automatic Detection of 3D Building Roof Planes from Lidar Data“. In: (2007).

[Val+18] Julien Valentin et al. „Depth from motion for smartphone AR“. In: *ACM Transactions on Graphics* 37.6 (Dec. 4, 2018), 193:1–193:19.

[Vri20] Joey de Vries. *Learn OpenGL - Graphics programming: Learn modern OpenGL graphics programming in a step-by-step fashion*. Erscheinungsort nicht ermittelbar: Kendall & Welling, 2020. 522 pp.

[Woo+14] Oliver Woodford et al. „Demisting the Hough Transform for 3D Shape Recognition and Registration“. In: *International Journal of Computer Vision* 106 (Feb. 1, 2014).

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original