MASTER

Real-time 3D environment mapping
occlusion and collision in augmented reality

van der Ven, F.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor’s or master’s), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
Real-time 3D Environment Mapping: Occlusion and Collision in Augmented Reality

Master’s Thesis

F. van der Ven

Supervisors:
Prof. dr. ir. R. van Liere
Dr. K.A. Buchin
Dr. A.C. Jalba

Final Version

Eindhoven, November 2016
Abstract

Achieving realistic augmented reality gives us several complex challenges to solve. Past work has shown that in most cases the technology available was not up to the task. However, the advances made in recent years have drastically changed the possibilities, and in this thesis we investigate them. To aid in this, we categorize augmented reality systems based on the type of information they use. The categorization shows us that we have not yet achieved a truly realistic merging of the virtual world with our real world. This thesis examines the basic operations we need for that purpose. By solving the collision and occlusion problems, we lay the groundwork for further work in achieving realistic augmented reality. We create a testbed and define metrics for quantifying the performance of our system, with respect to collision and occlusion. This testbed can be used to quantify the performance of any augmented reality system, and we apply it to our own implementation.

The results show that we have defined a working testing methodology, and using the testbed we show that the Kinect V2 depth-sensor is suitable for 3D environment mapping. The collision and occlusion results become worse when too much smoothing is applied, or when upsampling introduces too much noise. Limitations of this thesis are that we implemented the system on the CPU only, when a GPU approach would produce better running time performance. Based on this thesis, future work can be performed on merging virtual lighting with the real world.
Preface

This thesis is the culmination of my five years of study at the Eindhoven University of Technology. I am happy to have been part of the community and I have enjoyed my time greatly.

First of all, I would like to thank my parents for the incredible love, support, and confidence they have given me over the years. They have been the solid foundation of all my endeavors so far.

I would also like to express my sincere gratitude to my supervisor Robert van Liere, for his great support during the execution of this thesis. His advise and critical eye on my work proved to be invaluable.

Next, I would like to thank my great friend Menno Veen for letting me live at his place during a difficult time personally. Without his support the final weeks of work on my thesis would have been much more difficult.

Finally, the work for this thesis was performed at the office of The Line in ’s-Hertogenbosch. I would like to acknowledge and thank my colleagues, who showed support and patience throughout the whole project. My thanks go out to Hans Hagendoorn, also for his support and for the opportunity to combine my work and study at The Line.
Contents

List of Figures ix
List of Tables xi

1 Introduction 1
1.1 Motivation ........................................ 2
1.2 Objectives .......................................... 3
1.3 Research Questions ................................ 4
1.4 Applications ....................................... 4
1.4.1 Pokémon Go ................................... 4
1.4.2 Layar ........................................... 5
1.4.3 Machine Maintenance ............................. 5
1.4.4 Virtual Object Interaction ........................ 6
1.4.5 Healthcare ....................................... 7

2 Environment 3D-Mapping Techniques 9
2.1 Point Clouds ........................................ 9
2.2 Sensing Devices ................................. 10
2.2.1 Passive Methods ................................ 10
2.2.2 Active Methods ................................ 13
2.2.3 Conclusion ....................................... 15
2.3 Microsoft Kinect .................................... 15
2.3.1 Kinect V1 ....................................... 15
2.3.2 Kinect V2 ....................................... 16
2.3.3 Multiple Device Setup ........................... 17

3 System Design 19
3.1 Overview ........................................... 19
3.2 System Architecture ............................... 20
3.3 Implementation ................................... 22
3.3.1 Program Structure ............................... 23
3.3.2 World Cloud Creation ........................... 24
3.3.3 World Cloud Manipulation ....................... 27
3.3.4 Geometry Reconstruction ....................... 30

4 Parameter Tuning 33
4.1 Individual Parameters ............................. 34
4.1.1 Downsampling .................................. 34
4.1.2 Smoothing ...................................... 35
4.1.3 Upsampling .................................... 36
4.1.4 Triangulation .................................. 38

Real-time 3D Environment Mapping: Occlusion and Collision in Augmented Reality vii
## CONTENTS

4.2 Combination of Parameters ........................................ 39
4.3 Object Mapping Examples ........................................... 41
  4.3.1 Room ......................................................... 41
  4.3.2 Solid Material ................................................ 41
  4.3.3 Transparent Material .......................................... 41

5 Testbed ............................................................................ 45
  5.1 Bouncing Ball ......................................................... 46
  5.2 Viewing an Occluded Ball ........................................... 49

6 Conclusions ..................................................................... 53

Bibliography ..................................................................... 55

Appendix ............................................................................ 57

A Technologies Used ....................................................... 57
  A.1 Hardware .............................................................. 57
  A.2 Software .............................................................. 57

B Application Threads ..................................................... 59

C ICP Screenshots .......................................................... 60

D Tuning Screenshots ...................................................... 62
  D.1 Downsampling ....................................................... 62
  D.2 Smoothing ............................................................ 64
  D.3 Upsampling ........................................................... 66

E Pipeline Running Times ................................................. 70
## List of Figures

1.1 Reality-virtuality continuum. Image source: [1]. .................................................. 1  
1.2 Extent of World Knowledge. Image source: [1]. .................................................. 2  
1.3 Augmented reality using Microsoft HoloLens. The user sees the virtual objects through the headset, but for demonstration purposes we see them from a third-person view. Image source: [2]. ................................................................. 2  
1.4 A virtual Pokémon overlayed over the camera capture. Image source: [3]. ........ 4  
1.5 Finding available houses to buy through the Layar application in (a), and projecting a virtual model of a car over a marker (b). Image source: [4]. ...................... 5  
1.6 Instructions for opening the printer paper tray. Image source: [5]. ............... 6  
1.7 A virtual bunny sits in a real-world hand. Image source: [6]. ......................... 6  
1.8 Natural vision combined with a CT scan. Image source: [7]. ....................... 7  
  
2.1 A point cloud of a sailboat with varying density. Image source: [8]. ............ 10  
2.2 A taxonomy of depth scanning techniques. Image source: [9]. ................. 11  
2.3 An example photogrammetry setup where a single camera is used to capture an object. The camera moves around the object, but stays focused on the same point. Image source: [10]. .............................................................. 12  
2.4 Differing aperture values produce different depths of field. Image source: [11]. 13  
2.5 Structured pattern projected on a bust. Image source: [12]. ..................... 14  
2.6 Kinect V2 depth sensor system. Image source: [13]. ........................... 16  
  
3.1 Top-down schematic view of sensors aimed at a marker. .......................... 20  
3.2 Diagram of our geometry reconstruction pipeline. .................................... 21  
3.3 From hardware to geometry reconstruction pipeline. .................................. 22  
3.4 Program structure, showing the pipeline, steps, models, and their relations. .... 23  
3.5 ArUco marker detection. A coordinate system is layed over the markers. .... 24  
3.6 Sketch of two sensors, $c_1$ and $c_2$ aimed at a marker $m$. .................... 25  
3.7 Two iterations of the Iterative Closest Point algorithm. ............................ 26  
3.8 ICP rejection criteria. The left image shows rejection based on point distance, the right image shows rejection based on duplicate targets. The source points are denoted by $p$, and the target points by $q$. The green pairs are kept, and blue pairs are rejected. Image source: [14] ................................................................ 27  
3.9 A voxel grid layed over a point cloud. Image source: [15]. ....................... 28  
3.10 Least Squares approximation. From top to bottom we have a polynomial degree of 8, 15, and 25. A higher polynomial degree better approximates the shape of the underlying data. ................................................................. 29  
3.11 Upsampling. The left image shows the upsampled point cloud, the right image shows the original. Image source: [16] ............................ 30  
3.12 A triangulated hand. Having more vertices allows us to create more triangles, and this means the hand has more detail. Image source: [17]. .......... 31  
3.13 Different collision shapes around a graphical model of a monkey head. ....... 32
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Overview of the physical testbed setup.</td>
</tr>
<tr>
<td>4.2</td>
<td>The view of the individual sensors in the tuning setup. The boxes are used to visually examine the quality of the captured geometry.</td>
</tr>
<tr>
<td>4.3</td>
<td>Voxel grid upsampling, with a search radius of 0.005 and step size of 0.001.</td>
</tr>
<tr>
<td>4.4</td>
<td>Triangulation results. The scene captured is that of figure 4.1.</td>
</tr>
<tr>
<td>4.5</td>
<td>Geometry captures with different levels of detail. Figures 4.6a, 4.6b, and 4.6c have settings 0.0, 0.5, and 1.0 respectively. The scene captured is that of figure 4.1.</td>
</tr>
<tr>
<td>4.6</td>
<td>Virtual geometry capture of a small box.</td>
</tr>
<tr>
<td>4.7</td>
<td>Virtual geometry capture of a lock.</td>
</tr>
<tr>
<td>4.8</td>
<td>Virtual geometry capture of a kettle.</td>
</tr>
<tr>
<td>4.9</td>
<td>Virtual geometry capture of a small glass.</td>
</tr>
<tr>
<td>4.10</td>
<td>Virtual geometry capture of a transparent plastic bottle.</td>
</tr>
<tr>
<td>4.11</td>
<td>Virtual geometry capture of a small glass.</td>
</tr>
<tr>
<td>4.12</td>
<td>Virtual geometry capture of a transparent plastic bottle.</td>
</tr>
<tr>
<td>4.13</td>
<td>Virtual geometry capture of a room.</td>
</tr>
<tr>
<td>4.14</td>
<td>Virtual geometry capture of a small box.</td>
</tr>
<tr>
<td>4.15</td>
<td>Virtual geometry capture of a lock.</td>
</tr>
<tr>
<td>4.16</td>
<td>Virtual geometry capture of a kettle.</td>
</tr>
<tr>
<td>4.17</td>
<td>Virtual geometry capture of a small glass.</td>
</tr>
<tr>
<td>4.18</td>
<td>Virtual geometry capture of a transparent plastic bottle.</td>
</tr>
<tr>
<td>4.19</td>
<td>Before and after situation of a collision. An imaginary ball is shot from point u to v, ending up in w. Vector n represents the normal of the surface.</td>
</tr>
<tr>
<td>4.20</td>
<td>Three frames of a virtual ball collision test.</td>
</tr>
<tr>
<td>4.21</td>
<td>Bounce angle mean for varying detail levels.</td>
</tr>
<tr>
<td>4.22</td>
<td>Running times for varying detail levels.</td>
</tr>
<tr>
<td>4.23</td>
<td>Our occlusion setup. Figure 5.5a shows the view of the virtual camera, the front view of our setup. Figure 5.5b is a top-down view.</td>
</tr>
<tr>
<td>4.24</td>
<td>Occlusion tests, with detail levels of 0.0, 0.5, and 1.0 for the top, middle, and bottom image respectively.</td>
</tr>
<tr>
<td>4.25</td>
<td>Plot of the mean occlusion error.</td>
</tr>
<tr>
<td>4.26</td>
<td>Plot of the occlusion error standard deviation.</td>
</tr>
<tr>
<td>B.1</td>
<td>Threads in the implementation of our system. Each sensor has its own thread, and feed their individual point clouds to the geometry reconstruction pipeline which also has its own thread.</td>
</tr>
<tr>
<td>C.1</td>
<td>Top-down and side view of a scanned table. We see the two point clouds being misaligned. In the bottom image, the table appears broken.</td>
</tr>
<tr>
<td>C.2</td>
<td>Side view of a scanned table. The two point clouds are aligned properly and now form a proper table.</td>
</tr>
<tr>
<td>D.1</td>
<td>Downsampled point clouds using a voxel grid. Box sizes are 0.02 and 0.08 for the upper and lower image respectively.</td>
</tr>
<tr>
<td>D.2</td>
<td>Downsampled point clouds using a voxel grid. Box sizes are 0.06 and 0.2 for the upper and lower image respectively.</td>
</tr>
<tr>
<td>D.3</td>
<td>Moving Least Squares smoothing with search radii 0.02 and 0.08 for the upper and lower image respectively.</td>
</tr>
<tr>
<td>D.4</td>
<td>Moving Least Squares smoothing with search radii 0.1 and 0.4 (polynomial fitting enabled) for the upper and lower image respectively.</td>
</tr>
<tr>
<td>D.5</td>
<td>Local plane upsampling with search radii of 0.005 and 0.02 respectively. The step sizes are 0.001 and 0.004.</td>
</tr>
<tr>
<td>D.6</td>
<td>Local plane upsampling with search radii of 0.04 and 0.08 respectively. The step sizes are 0.008 and 0.016.</td>
</tr>
<tr>
<td>D.7</td>
<td>Voxel grid dilation upsampling with dilation voxel size of 0.02 and 0.08 respectively. The number of dilation iterations is set to 2.</td>
</tr>
<tr>
<td>D.8</td>
<td>Voxel grid dilation upsampling with dilation voxel size of 0.02 and 0.08 respectively. The number of dilation iterations is set to 4.</td>
</tr>
</tbody>
</table>
# List of Tables

4.1 Results of downsampling with varying voxel grid box sizes. .......................... 35
4.2 Results of smoothing with varying search radii. ........................................... 35
4.3 Results of local plane upsampling, with varying search radii and step sizes. We have combined two steps here, the upsampling (US) and the subsequent downsampling (DS) step. ........................................... 36
4.4 Experiments for voxel grid dilation upsampling, with varying voxel dilation sizes and dilation iterations. We have combined two steps here, the upsampling step (US) and the subsequent downsampling (DS). ................................. 37

5.1 Bounce angle mean, standard deviation, and errors. We consider an experiment an error when no collision occurs at all. .................................................. 48
5.2 Mean and standard deviation of the occlusion error. ...................................... 51

E.1 Mean running times for each step in the pipeline in milliseconds. ...................... 70
E.2 Standard deviation of the running times for each step in the pipeline in milliseconds. 70
Chapter 1

Introduction

In this thesis we focus on merging virtual objects with the real world; we want to create realistic augmented reality. For this, we need to first have a proper definition of augmented reality. We turn to the paper "Augmented Reality: A class of displays on the reality-virtuality continuum" [1] by Milgram et al., where we see that augmented reality is defined as belonging to the reality-virtuality continuum. See figure 1.1.

![Reality-Virtuality Continuum](image-source-1)

Figure 1.1: Reality-virtuality continuum. Image source: [1].

On the left end of the spectrum we have a real environment, where no virtual information is added whatsoever. It can be observed directly, through some kind of window, or by watching a video of the real world. On the other end, we have a completely virtual environment, where our perception of the real environment is replaced by only virtual information. Between these two extremes lies what we call mixed reality. This is where we merge virtual information with the real environment, and which can then be perceived through for example a monitor or a head-mounted display. Augmented reality is a subclass of mixed reality. It lies closer to the ‘real’ end of the spectrum, which means that we mostly perceive the real environment, but with added information such as virtual text or virtual objects. In short, the definition of augmented reality is "augmenting natural feedback to the operator with simulated cues."

Depending on the augmented reality application, we require different levels of information from the world. For example, to place a virtual object on top of a physical table, we need to know where the table is and what shape it has. In the paper by Milgram et al., we find a classification for the amount of world information we have, called the Extent of World Knowledge (EWK). See figure 1.2. It defines how much we know about objects and the world in which they are displayed. On the left end of the spectrum we have the unmodeled world. The augmented reality system knows nothing about its contents, and it is simply shown unaltered to the user. We can overlay information on it, but we can not realistically add new information. In this case we cannot place a virtual object on the real table. On the other end of the spectrum we have the completely modeled world. Here we know everything about the objects in the world, including the user himself. We know their position, their surface properties, their geometry, and we can use this information to augment the scene with new information. Having a fully modeled world allows us to use its
CHAPTER 1. INTRODUCTION

information to reason about the physical environment. In this case we know where the physical table is, and what geometry it has. This allows us to correctly place the virtual object on it. Note that as we move right on the scale, our augmented reality application becomes more complex, but also gives us more possibilities.

![Figure 1.2: Extent of World Knowledge. Image source: [1].](image1)

We say that the real world contains two types of objects, namely static and dynamic. For static objects, properties such as shape and scale are fixed. For dynamic objects, these properties can change. A hand is an example of a dynamic object. Its shape is not fixed, as the fingers can move freely and unpredictably.

1.1 Motivation

Our goal is to achieve realistic augmented reality, where virtual objects interact with the real environment. Figure 1.3 is an example which motivates our goal. We see a user wearing an augmented reality headset, viewing a living room with a floor, table, and couch, and virtual objects are correctly projected onto their surfaces. The user sees the virtual objects in the scene, through a projection onto the glass of the headset. When the user moves to a different position, the real and virtual objects will be seen from a different perspective.

![Figure 1.3: Augmented reality using Microsoft HoloLens. The user sees the virtual objects through the headset, but for demonstration purposes we see them from a third-person view. Image source: [2].](image2)

This scene contains both static and dynamic objects. For example, the real floor contains patches of green virtual grass, and both the real floor as well as the virtual grass are static. They are fixed in place, and do not change due to any form of interaction. The scene also has dynamic
objects. On the floor, we see small virtual cows walking. They collide with both virtual as well as with real objects, and they do not pass through them. The user himself is also an example of a dynamic real object. His hands move and can interact with virtual objects. For example, he can pick up the virtual cows from the floor. In this thesis we aim to achieve the following:

1. A realistic augmented reality scene, as in figure 1.3.
2. The scene has both static and dynamic objects, real and virtual.
3. Interaction between the real and virtual worlds must occur in real-time.

To achieve the mixed scene in figure 1.3, there are two main operations we need to perform. First, we need to be able to collide virtual objects with real surfaces, and for this we need to be able to detect such collisions. Second, we need to know whether real or virtual objects occlude each other. Occlusion occurs when an object closer to the viewer obscures the view of objects further away along the line-of-sight. In other words, a virtual object might be partially behind a real object, and only part of it is visible. In short, there are two fundamental problems to be solved:

1. **Collision** We want virtual objects to collide with real objects realistically.
2. **Occlusion** Real and virtual objects can partially or fully occlude each other.

For solving the collision and occlusion problems we need to place the real and virtual worlds into a common computational framework. Since the real world is dynamic, we need to perform real-time 3D environment mapping to make it fit into the framework. In this thesis we focus on the collision and occlusion problems, and for that we only require the geometry of the environment. In other words, we limit our Extent of World Knowledge to only geometric information. Our system assumes that the world consists of a single, solid material and we thus do not have to obtain material properties of the world. Also, we do not need to know semantics of the real world.

In the paper "Interactive occlusion and collision of real and virtual objects in augmented reality" [18] by Breen et al., we see that our goals are not new. In fact, more than 20 years ago they were already attempted to be solved. In this thesis we use recent techniques and devices to reexamine the possibilities and results.

### 1.2 Objectives

The main objective of this thesis is to get an understanding of the currently available techniques for 3D geometry mapping, to find which of them are viable options for solving the collision and occlusion problems in augmented reality, and to implement an application that maps the real world to a virtual model. We divide the main objective into several sub objectives as follows.

1. **Categorize 3D mapping techniques and select the most promising one** to implement and evaluate.
2. **Implement** the technique and create a setup where we can tune the parameters that influence performance.
3. **Create a testbed** for measuring the performance of any augmented reality system. In particular, we want to quantify collision and occlusion performance of the system.

The different chapters in this thesis address these objectives. In the current chapter we have defined augmented reality and what we require for achieving it. In chapter 2 we investigate and categorize the various 3D geometry mapping techniques. The strengths and weaknesses of the techniques are discussed and based on these we pick one to implement. The architecture, implementation, and setup of our implementation are described in chapter 3. In chapter 4 we
tune the system parameters to balance model detail and running time performance. Next, to verify the strengths and weaknesses of the chosen technique, we create a testbed and perform several tests. These are described in chapter 5. Finally, we discuss the main objective in the conclusions in chapter 6.

1.3 Research Questions

The following two research questions act as the driving force behind this thesis.

"How suitable is the Kinect V2 depth-sensing device for 3D environment mapping, and for tackling the collision and occlusion problems in Augmented Reality?"

"Is there a suitable testing methodology for quantifying depth sensor performance?"

1.4 Applications

In this section we look at several applications, and we determine their relative positions on the EWK scale. The more an application lies to the right on the EWK scale, the more information it requires and the more complex it is.

Figure 1.4: A virtual Pokémon overlayed over the camera capture. Image source: [3].

1.4.1 Pokémon Go

The Pokémon Go game has brought a very simple form of augmented reality entertainment to the masses. It is entirely based on the location of the user. If the user is in the vicinity of Pokémon, they pop-up on a map and he or she can then interact with them. It features a see-through mode, where the virtual creature is overlayed on the real world, and which you can then ‘capture’ by throwing virtual balls at him.
For the application, the world does not change dynamically. It is a static object. The application only needs to know the location of the user, and the pose of the smartphone. There is no notion of real world geometry, and thus it can merely overlay the virtual objects. The virtual creatures and the balls you can throw are dynamic virtual objects, but they do not bounce on real objects. Therefore, since the application needs to know little about the real world, it lies far on the left of the EWK spectrum.

1.4.2 Layar

The Layar application is another example of an augmented reality application that only requires the location of the user and the pose of the smartphone. Based on this, the user can see information projected onto the real world when looking through the camera of the smartphone. See figure 1.5a. Besides relying on this static information, Layar also has support for detecting fiducial markers. Fiducial markers are static objects in the world, which the application can track in real-time and can be used to display 3D models on. This can be seen in figure 1.5b, where a virtual model of a car is projected onto a marker. Layar is primarily being used for advertising purposes. Markers are added to billboards, magazines, and posters for Layar users to get more information about products. The Layar application needs to track fiducial markers in the real world, and thus it lies more to the right on the EWK spectrum than the Pokémon app.

![Figure 1.5: Finding available houses to buy through the Layar application in (a), and projecting a virtual model of a car over a marker (b). Image source: [4].](image)

1.4.3 Machine Maintenance

A typical example of where augmented reality is useful, is in maintenance. Take a car engine for example. It is incredibly complex, with numerous components. Augmented reality could assist the mechanic in diagnosing problems and in disassembling the engine. Early work has focused on smaller scale devices. The paper "Knowledge-based Augmented Reality" [5] by Feiner et al demonstrated a system for assisting the operator in performing maintenance on a printer. Instructions for servicing the printer are overlaid on the vision of the user, as seen in figure 1.6. The printer has a fixed location in the world, and it has sensors that provide the system with information about its state. The system assumes known information about the printer, and it is therefore a static object. The information about the real environment is limited, we only know information through sensors in the printer. For accurately overlaying information to the user, the system also knows his position and pose. Since the location of the printer is fixed, we know where it is in relation to the user. The information we have about the printer is limited to sensors and known data, so we are on the left of the EWK spectrum.
1.4.4 Virtual Object Interaction

If we want to interact with virtual objects using our hands, we could use peripherals such as tracking gloves or pointers. Although this is a useful way to provide input to the system, it puts restrictions on the types of interactions we can do. Instead, we want to be able to have any kind of interaction, using any part of our body or even by using other real-world items. For example, if we want to pick up a virtual bunny and place it on a table, we require both a virtual model of our hands, and a virtual model of the table. See figure 1.7. This is an example where we require a geometrical model of the real world that is generated in real-time, as the hands are a real dynamic object. Because of this, we cannot assume anything a priori. We now require quite a lot of real-time properties of the real world, and we are thus located on the right of the EWK spectrum.

Figure 1.7: A virtual bunny sits in a real-world hand. Image source: [6].
1.4.5 Healthcare

The final application lies in the field of healthcare. By merging different scans of a patient and projecting these onto him, a surgeon could effectively see through the body. In figure 1.8 we see an example of this. Here, a live CT scan is merged with the view of the surgeon, and he can quickly see if there are issues. This information is generated in real time, and is thus dynamic. A CT scan provides depth information, and goes one step further than geometry information. Therefore, we are even further to the right on the EWK spectrum than the previous example.

Figure 1.8: Natural vision combined with a CT scan. Image source: [7].
Chapter 2

Environment 3D-Mapping Techniques

When merging virtual objects with dynamic real-world objects, we need an accurate virtual representation of the real-world. In this section we look at this representation, and we consider ways to map the real world to it. This mapping needs to be generated fast, for it to be useful for augmented reality. There are various techniques for capturing a geometry model of the world, and when discussing them we need to keep in mind the requirements of our system. The technique we choose needs to have at least the following properties.

- **Affordable** The solution needs to be affordable, preferably below 1,000 euros.
- **Available** The hardware should be readily available for consumers.
- **Real-time** Our application is focused on collision and occlusion, and both of these require real-time processing of a scene. Therefore, the technique we choose needs to be fast.

We note that even though the techniques and requirements we have are similar to those in the field of robotics, the goals are different. In robotics, environment information is used to manipulate the robot itself. Our primary goal is to augment the real world with virtual information.

Many of the techniques we discuss give output in the form of depth maps, which are commonly used to represent scanned 3D information. By combining several depth maps from different points of view, it is possible to create a complete virtual reconstruction of a real scene. Before discussing the different types of devices that are available, we first examine the depth maps they produce.

### 2.1 Point Clouds

Depth sensors can produce enormous amounts of depth data, and we need a way to efficiently store and manage this. A point cloud is a depth map containing a set of points in a typically 3-dimensional space, with each point having its own set of properties. In the most minimal case, a point has a position in space, an $xyz$-property. This is necessary for actually having meaning in the 3D space. Besides that, a point can have an $rgb$-colour value, a normal value, and for example an age parameter (i.e. when it was added to the data). Scanners such as the Microsoft Kinect produce point clouds. In the case of geometry mapping, a point cloud is used as a discretized representation of the real world, where every point represents a depth measurement. Because of this, a point cloud by definition always lacks at least some information. The space between points is ’unknown’, but can be approximated by interpolating between points that are known. It follows from this, that the density of a point cloud is directly related to the amount of detail it has. See figure 2.1. We see that the more points a cloud has, the higher the density and the more detailed the visual representation. Since a point cloud consists of discrete elements, we can apply
the typical mathematical operations to it. In chapter 3 we discuss several of them, for example downsampling and smoothing.

![Figure 2.1: A point cloud of a sailboat with varying density. Image source: [8].](image)

### 2.2 Sensing Devices

In this section we discuss devices and techniques for scanning the real world into a virtual model. We will primarily focus on non-contact techniques, but it is worth mentioning there are also techniques based on contact. These techniques, as the name implies, use physical touch to measure real objects. However, for our application this is not suitable, as this would be too slow. A taxonomy of 3D sensing can be seen in figure 2.2.

From this taxonomy, we pick the techniques that have the most potential for our application. We start our discussion with a section on passive methods: photogrammetry and depth-of-field manipulation. They are called passive, because they do not require changing the scene in any way. Photogrammetry is concerned with using multiple photographs to reconstruct a scene in 3D. This technique identifies the common features of multiple photographs and uses the displacement between them to create a reconstruction. It is also possible to acquire depth maps through the use of shape detection. However, this requires knowledge about what shapes to look for, and this is not a luxury we have in our dynamic real-world environments. Next, we discuss active methods such as structured light and laser, which both involve emitting light into the scene.

#### 2.2.1 Passive Methods

Passive methods of geometry capturing do not alter the lighting or other properties of the scene. Instead, only visual cues are used. One of the most obvious techniques is that of stereoscopy, since it closely mimics how our human eyes work. In stereoscopy, two images are captured, where the 'camera-position' of each image is slightly different from the other, but both are focused on the same point. Through the use of triangulation, an estimation can be made of the position of the
Figure 2.2: A taxonomy of depth scanning techniques. Image source: [9].
point in 3D. Taking this idea one step further, we can involve more viewpoints. This technique is called photogrammetry and is discussed in the next section.

**Photogrammetry**

Photogrammetry is based on using multiple photographs from different perspectives, identifying overlapping features, and stitching the images together into a 3-dimensional model. See figure 2.3. An advantage of this technique is that no special equipment is required; regular cameras can be used. The quality of the camera influences the final result greatly. One aspect that is especially important is the resolution. The higher the resolution, the more data is in the photos and the more accurately a 3D model can be reconstructed. Other than that, one can use cameras ranging from the ones in smartphones to Digital Single Lens Reflex cameras.

A drawback of photogrammetry is that it requires a precise setup. One can either use a single camera in different viewpoints or multiple cameras focused on the same point. However, the properties of the cameras should remain the same, that is, same focal length, same type of lens and same zoom level. Small distortions in the photos can result in inaccurate model reconstruction.

The types of objects that can be captured by photogrammetry are also limited. Model reconstruction relies on feature matching in the photos, and this is only possible with sufficient overlap. Only when the scene is static, this can be done reliably. When the scene is dynamic, that is with moving objects or camera position, there might not be enough overlap in the photos. Not only that, but the environment’s surface materials also influence the resulting 3D model. Looking at a (semi-)transparent object from different viewpoints results in slight distortions in the glass, making it difficult for matching algorithms to find common features between photographs. Objects with minimal texture, shiny surface, or little detail in general are difficult to reconstruct with photogrammetry.

A significant amount of research has been performed on photogrammetry. An overview of the available techniques can be found in "A comparison and evaluation of multi-view stereo reconstruction algorithms" [19] by Seitz et al. The technique could be useful for our application, however it is entirely based on matching visual cues, which is subject to a lot of limitations. As Lanman describes in his paper [12], “Flat or periodic texturing prevent robust matching.” and “Under controlled circumstances, such as known or constant background, the boundaries of foreground objects can be reliably identified.” This does not bode well for our application, as it should handle dynamic scenes, where there could be a lot of movement.

![Figure 2.3: An example photogrammetry setup where a single camera is used to capture an object. The camera moves around the object, but stays focused on the same point. Image source: [10].](image-url)
Depth-of-field Manipulation

Another passive approach is to take multiple photographs of a scene from the same viewpoint, and manipulate the depth-of-field. By taking photographs with differing aperture values, foreground and background objects are either focussed or defocussed. These focus differences can then be analyzed, and depth can be determined. See figure 2.4 for an example. This approach requires multiple photographs, but there are also techniques to infer it from a single one. An interesting approach is to deliberately introduce patterns in the aperture, allowing discrimination in depth of field using only a single image capture. One paper describing this principle is "Image and Depth from a Conventional Camera with a Coded Aperture" [11] by Levin et al.

Depth-of-field manipulation is not suitable for our application because of our real-time requirements. In the first case, taking multiple photos with differing aperture is too slow. But when using a single photograph and introducing an aperture pattern, we lose too much information. The method is not accurate enough for creating a detailed depth map. However, for discerning larger surfaces it is sufficient.

Figure 2.4: Differing aperture values produce different depths of field. Image source: [11].

2.2.2 Active Methods

Another class of sensing devices is based on emitting light or radiation. By actively manipulating the scene, we can create artificial features in the scene to track, instead of relying on environmental visual cues only. Since these methods rely on projecting into the scene, they are sensitive to differences in surface materials. For example, projecting onto reflecting material makes it difficult to measure results.

Various active methods have been developed. One of the first attempts was to scan single points at a time. A light source is projected onto an object through the use of one or more movable mirrors. Next, a stationary camera is used to track the position of the point of light as it moves over the object. After every scan the mirror setup needs to be adjusted to move the light point to a new position. This made the scanning slow, and thus unsuitable for our application. As camera and laser technology improved, the process became faster but still limited. A new technique was developed, involving a large structured pattern being projected onto an image. This essentially removed the mechanical limitation of having to move a laser dot over an object, which leads us to the structured light approach.
Structured Light

Two excellent articles on structured light 3D scanning are "A low cost 3D scanner based on structured light" [9] by Rocchini et al, and "Build Your Own 3D Scanner: 3D Photography for Beginners" [12] by Lanman and Taubin. The technique is centered around two components: a digital camera and an image projector. The projector is used to send out patterns of structured light, and the camera captures the result. As the structured patterns are projected onto the environment, they get distorted based on the environment’s shapes. See figure 2.5. The camera captures these distortions and we can then reconstruct a 3D model based on them. There are variations in how the patterns are projected. The pattern can be coded spatially, but also temporally. Spatial encoding uses a single camera frame to capture the scene, and is thus suitable when the scene is highly dynamic. Temporal encoding uses multiple frames and the differences in the captured patterns. An overview of different pattern and coding techniques can be found in "Pattern codification strategies in structured light systems" [20] by Salvi et al.

When using the structured light pattern technique, a projector overlays a certain light pattern over the environment. Using a single scanner is feasible, however using multiple scanners in a single scene can become problematic due to the patterns overlapping and interfering with each other.

The structured light technology has several advantages. First off, it is affordable and since 3D scanning is becoming more popular, consumer technology has become widely available. Secondly, structured light scanners provide high accuracy and resolution, improving the resulting 3D scan. Finally, the technology is easy to use and flexible. These factors together make structured light a good candidate for our application.

![Figure 2.5: Structured pattern projected on a bust. Image source: [12].](image)

Laser

Laser is an abbreviation for Light Amplification by Stimulated Emission of Radiation. A laser is simply a concentrated beam of light, which can be shone onto a scene. There are several ways to use a laser to perform 3D environment mapping.

- **Triangulation** is one of the earliest active 3D mapping techniques. We shine a laser onto an object, and look for the resulting laser dot with a camera. With careful calibration, the dot’s location on the object could be calculated and used for reconstructing a 3D model. The term triangulation is used because the laser projector, the dot, and the camera form a triangular setup. This scanning process is slow - a single dot needs to go over the object’s whole surface, and even more, the laser has to be mechanically repositioned after every scan.
• **Time-of-flight** scanners use the time it takes for a laser pulse to hit an object and to reflect back to the sensor. The farther away an object, the longer this round-trip time is. Just as with triangulation, this can be done one point at a time. The laser emitter needs to be mechanically repositioned after every scan. However, it is also possible to emit a raster of dots, and measure the round-trip-time of all the dots at once. This allows for faster scanning of a scene.

• **Phase shift** is a similar approach to that of time-of-flight. However, instead of pulsing the laser, its intensity is modulated. The phase shift between the measured intensity and the received intensity is measured, and based on the offset the distance can be measured.

Time-of-flight laser scanning is the most suitable candidate for our application, as it uses a simple principle, is easy to use, and is fast.

2.2.3 Conclusion

We have seen several techniques, passive and active. The photogrammetry, structured light, and laser time-of-flight methods are suitable for our application, as they are affordable, available, fast, and easy to use. There is one consumer-grade device that also has these properties: the Microsoft Kinect. Our geometry mapping implementation as created in this project works for any device, but we chose the Kinect because it is cheap and widely available. In the next section, we look at the features of the Kinect.

2.3 Microsoft Kinect

The Microsoft Kinect has been around for quite some time, and there are two versions of it, simply dubbed V1 and V2. The first version came out in 2010, and found great use in the academic society. It is cheap, accurate, and because it is widely used, the support from both Microsoft and the community is good. It has been used for tracking persons, pose and gesture recognition, and for robot vision. Even though for this project we use the Kinect V2, in this section we look at the V1 as well. Our geometry mapping implementation works with both versions. For an in depth comparison of the V1 and V2 cameras see “Performance evaluation of the 1st and 2nd generation Kinect for multimedia applications” [21] by Zennaro et al.

2.3.1 Kinect V1

The Kinect V1 primarily uses the structured light technique to generate depth maps. It contains an RGB camera, an infrared camera, and an infrared projector. The camera resolutions are 1280x960 and 640x480 respectively, captured at 30 Hz. The range of the measurements are reliable between 0.5 and approximately 8 meters. The infrared projector sends out light with a certain pattern, similar to what can be seen in figure 2.5. However, instead of stripes, it sends out a speckle pattern. Next, triangulation is performed on the dots, by intersecting the rays from the projector and the capture from the infrared camera. A major drawback of the Kinect V1 is that it is very susceptible to noise. It is greatly affected by other light sources, which interfere with the projected pattern of the Kinect.
2.3.2 Kinect V2

The Kinect V2 uses the time-of-flight principle. To this end, the Kinect V2 contains an infrared emitter, and both a color camera and an infrared/depth camera. The cameras have resolutions of 1920x1080 and 512x424 respectively. Images are captured at a rate of 30 Hz. Measurement range lies between 0.5 and approximately 10 meters. For more details on the Kinect V2 sensor, see "Kinect v2 for Mobile Robot Navigation: Evaluation and Modeling" by Fankhauser et al [22] and "The xbox one system on a chip and kinect sensor" by Sell et al [13]. As mentioned, the depth map is generated through the use of the time-of-flight laser scanning principle. The Kinect’s strobed infrared light source illuminates the scene, the infrared is reflected on the physical objects, and this reflection is captured by the infrared camera. Depth is calculated by measuring the time it takes between light emission and camera capture, and performing a time-of-flight calculation based on this. See figure 2.6.

The Kinect V2 produces three different output streams, one 1920x1080 RGB image stream, one 512x424 infrared image stream, and one 512x424 depth stream. All streams run at 30 frames per second. A single frame is thus quite large:

\[
\begin{align*}
1920 \times 1080 \times 8 \times 3 &= 49,766,400 \text{ bits} = 6.2208 \text{ megabytes} \quad (2.1) \\
512 \times 424 \times 8 &= 1,736,704 \text{ bits} = 0.217088 \text{ megabytes} \quad (2.2) \\
512 \times 424 \times 8 &= 1,736,704 \text{ bits} = 0.217088 \text{ megabytes} \quad (2.3)
\end{align*}
\]

Given that both streams run at 30 frames per second, the total output of the Kinect V2 is 6,654,976 * 30 = 199,64928 megabytes per second. The USB 3 standard [23] states an available bandwidth of 5Gbit per second, which is approximately 625MB per second. However, this is a theoretical limit. In practice, the bandwidth is limited by the USB controller both on the device and on the host computer. During our experiments, we found that a single Kinect V2 completely saturates a single host USB 3 controller. In order to use multiple devices, we would thus require a dedicated USB 3 controller per Kinect.
2.3.3 Multiple Device Setup

Using a single sensor, the occlusion problem can be partially solved. When we have a single sensor that is mounted on the head of a user, the depth information follows the field of view of the user. We can then determine whether virtual objects occlude one another, but only from the single user point of view. However, for solving the collision problem a single camera is not sufficient. When we drop a virtual ball onto a mapped real-world table, it will bounce away and collide with geometry that is possibly outside our field of view. In other words, we need to know about geometry that the user might not see.

A possible solution is to move the single sensor around, and merge new depth information with old. A drawback of this is that you merge with possibly old data - only the newly obtained data is guaranteed to be real time. In a highly dynamic environment this would make scanned information old quite fast. We have to know the location of the sensor in every frame, as to correctly interpret the depth data. This would require us to either have some positional device in the sensor or track features in the environment and use them to determine the sensor location.

Another option is to have multiple sensors at fixed locations, aimed at the same point. The main advantage of this setup is that we can place the cameras in such a way that a whole room is captured, and we have no 'unknown' parts. This is also its biggest disadvantage: we can only map the room we have the setup in. We lose the unrestrained properties of using a single camera and moving it around. However, since we have a controlled setup, we can perform more reliable mapping. We have a ground truth - the static environment - and can determine movement based on this. Another advantage is that having multiple cameras allows us to capture more of the environment simultaneously, as opposed to the single viewpoint of the single camera. This is of great importance to our application. We want to have real-time collision between virtual and real objects. Only by having up-to-date information about real geometry can we determine these collisions. If a scene would change because of hand movement for example, we need to know this as soon as possible to correctly determine collisions. Finally, worth mentioning here is that when using multiple Kinect sensors, interference can occur. This is especially the case when using a Kinect V1, due to the structured light technique it uses. The Kinect V2 suffers much less from this problem, as the time-of-flight technique is less susceptible.

In short, for this thesis project we decided to use multiple cameras at a fixed location in a room. This allows us to perform controlled experiments, and focus on the geometry mapping instead of tracking the camera location.
Chapter 3

System Design

In this chapter, the design of our system is described. We examine our architecture and implementation, and we do so in a top-down approach. In section 3.1 we first summarize the system at the highest level, and in section 3.2 we dive deeper into it. We describe the individual components of the system and how they interact with each other. We look at the geometry reconstruction pipeline, from point cloud to collision model. As a reference, we list the technologies used in our implementation in appendix A.

3.1 Overview

The goal of our system is to generate a virtual geometry model of the real environment, which we can then use to generate a collision model. Our system consists of several global steps.

1. Sensor registration. We use a multi-sensor setup, with several Kinect V2 devices placed at fixed positions in a room aimed at the same point. The goal is to have each Kinect capture as much of the room as possible. To correctly concatenate the resulting point clouds, we need to know the transformation matrices of the cameras in world space. We use a single marker placed in the center of the room, from which we can calculate the transformation matrix of each camera. This sensor registration step needs to occur only once at the start of the system.

2. Point cloud concatenation. The individual depth maps generated by the sensors are converted to point clouds. We use the transformation matrices calculated in the previous step, to concatenate the individual sensor point clouds into a single world point cloud. This concatenation happens once in each iteration of the pipeline.

3. The single world point cloud is converted to a triangle mesh, which we then use to generate a collision model.

4. We perform collision detection between virtual objects and the generated virtual geometry model.

5. Render the virtual geometry model into the Z-buffer.

6. Render virtual objects. The information in the Z-buffer from the previous step will cull parts of the virtual world that are occluded by the real world.

In this thesis we focus on generating the virtual geometry model, and leave the actual rendering for future work.
CHAPTER 3. SYSTEM DESIGN

3.2 System Architecture

In this section we consider the architecture of the system. We start with the physical setup of the sensors. The number of sensors we use influences the amount of mapped space in the final virtual geometry model. They need to be placed strategically, such that each one maps as much of the scene as possible, with minimal overlap. A single sensor can only scan the room from one viewpoint, and can only capture what is in its field of view. Another sensor from a different viewpoint can then scan the missing information. Depending on both the application and the scene the number of cameras needs to be determined.

An important consideration here is that the sensors need to know the position of each other. This is necessary for concatenating their individual point clouds into a single world cloud. Determining the position of each sensor is done using a common visual marker. See figure 3.1, where we see a top-down view of a scene where four sensors are aimed at one. The marker should have distinguishable features, such that each sensor can determine its pose based on it. For this project we use ArUco markers, as can be seen in the figure. These markers have the nice property that even a single one provides enough information for a sensor to obtain its pose. By using a whole board of markers, we can detect the sensor pose even when the board is partially occluded. For more information on ArUco markers, see "Automatic generation and detection of highly reliable fiducial markers under occlusion" by Garrido-Jurado et al. Now that we have a common visual reference, we can determine the relative pose and position of each sensor, and we call this sensor registration. When we receive a new point cloud from a sensor, we transform it such that it has the marker at its origin and has a corresponding rotation.

As soon as new sensor data is available, the system starts processing it. The data from every sensor is put through a geometry reconstruction pipeline, which can be seen in figure 3.2. We see three major operations, that are divided in steps. In steps 1 to 3 we create a single world cloud from the data of individual sensors.

1. In step 1, the system receives data from multiple sensors. Each sensor produces a point cloud, an rgb-image, and an infrared image. The point clouds are in sensor space, which means that the origin of the points lies at the sensor. Instead, we want the origin for all
sensors to be the same, and thus we need to apply a transformation to get them into the same coordinate space. From sensor registration, we know the transformation matrix of each sensor relative to the marker. We apply this transformation to the output point cloud of this step.

2. Step 2 of the pipeline consists of cleaning the data. This is done by using a pass-through filter. The Kinect sometimes produces invalid points, where the $xyz$ values are all 0. The filter removes these points. Furthermore, sometimes the data will contain one or more points that are very far out of the usual detection bounds of the Kinect. For example, points that are 20 meters away from the sensor. Through both experiments and the Kinect specifications we find that the sensor has a range of approximately 0.5 to 10 meters. Therefore, we consider the point at 20 meters noise and remove it.

3. The Kinects each run in their own thread, and produce their own point clouds. We need to concatenate them into a single world cloud, and this is done in step 3. Steps 1 and 2 of the pipeline can run in parallel, and in step 3 these parallel operations are joined. The concatenation step waits for all sensors to have produced new data, merges that, and then outputs this to the next step. We can simply concatenate the point clouds into a new global one, since we already transformed the clouds into a common world space in step 1.

See section 3.3.2 for details. Next, in steps 4 to 6 we manipulate the world cloud such that it is suitable for geometry reconstruction.

4. Each Kinect can produce more than 100,000 points. Combining several clouds can quickly result in a very large dataset. We need to downsample the number of points, which is done in step 4. The merging can also produce 'double surfaces'. For example, when we scan a table with two sensors, there will be some overlap in the two merged surfaces. In other words, the sensors scan the same surface twice, and the concatenated point cloud contains points close to each other for the same surface. The downsampling step merges these points into one.

5. After downsampling, we smooth the data. One can think of ironing clothes - we remove the wrinkles and make the clothes smooth again. However in this case, we adjust points to better be in line with their neighbors. In other words, we move points around so together they better approximate the shape they truly represent. We do so by finding a mathematical function
CHAPTER 3. SYSTEM DESIGN

that describes the points best, and by fitting them to it. This also allows us to perform upsampling, which introduces new points along the fitting function. The data generated by the sensors can never be complete, as a point cloud is a discrete representation of the real world. By upsampling, we fill the gaps between known points with new interpolated points. For example, say a sensor did not detect a small part of a wall, and as a result there is a hole. The upsampling process will attempt to fill this gap by interpolating its neighbors.

6. In the previous step, we performed upsampling in order to fill holes. This introduces new points, increasing the size of the point cloud. We need to downsample again to bring this number down and improve running time performance. See section 3.3.3 for details. Finally, in steps 7 and 8 we transform the world cloud into geometry and collision models.

7. We now have a processed world point cloud and we convert it to a triangle mesh. This mesh consists of small triangles that are all connected to each other, forming a solid geometry model.

8. In order to apply physics to the generated virtual world, we have to create a collision object for it. In this step, we convert the triangle mesh into a collision model. More details about these final steps can be found in section 3.3.4.

3.3 Implementation

In this section we discuss the implementation of our system. First of all, we have the connection between the sensors, the host computer, and the software. The depth sensor, in our case a Kinect V2, is connected to the computer through a USB 3 connection. Every Kinect produces a huge amount of data, and consequently every Kinect needs its own dedicated USB 3 controller on the host. See section 2.3.2. When the Kinect has new data available, the operating system is signaled, and makes it available to the geometry reconstruction pipeline. See figure 3.3. Worth mentioning here is that our pipeline implementation makes heavy use of the Point Cloud Library (PCL). This software library contains tools and algorithms for 2D/3D image and point cloud processing.

Figure 3.3: From hardware to geometry reconstruction pipeline.
3.3.1 Program Structure

In figure 3.4 an overview of the implemented system can be seen. The pipeline receives point clouds from the sensors, and pipes these through processing steps to come to a final geometry model. The pipeline then sends it to the scene model, which updates both the collision model and the main UI window.

![Diagram of program structure]

Our system has a strict performance requirement. We want to make use of all the available processing power, which means we have to make use of threads. To simplify the implementation, we make heavy use of the Qt signal/slot mechanisms. In the Qt framework, signals and slots are used for communication between components. Classes can derive from the QObject base class, after which they can emit and receive signals. Say there are two objects $A$ and $B$ that derive from QObject, with $A$ having a signal $a$ and $B$ having a slot $b$. Now the instance of class $A$ can emit an event $e$ from $a$ to $b$ of $B$. The signal $a$ looks like a regular function call with parameters, while the slot $b$ looks like a regular function that can receive parameters.

The Qt signal/slot system can be used to synchronize multiple threads, by combining the QObject and QThread classes. A QThread can have an event loop, and if a QObject is then assigned to that thread, every signal it receives will be pushed onto that event loop. When the thread is working on something, the thread’s event loop will block. When the thread is done processing, the event loop continues and the signal is handled. The event that is transmitted from $a$ to $b$ can carry data, just as with a regular function call. However, for this to work reliably between multiple threads, the parameters need to be serialized. This means that, for example when event $e$ contains point cloud data, the data needs to be copied as to leave the original data intact.

The different threads used in the implementation can be seen in figure B.1 in appendix B. We see a UI thread, a pipeline thread, and one thread per sensor. In order for the user interface to stay responsive, long-running or frequently occurring tasks need to be handled in the background. For this reason, the sensors and the pipeline have to be separated from the UI thread. Furthermore, the sensors and pipeline each have their own thread, because the sensor data needs to be handled as soon as possible. The sensors continually produce data, and if not handled in time the program starts producing IO errors. The data is sent to the pipeline, and if it is not ready, it simply ignores it. Otherwise if the pipeline is idle, it starts processing the new data. As soon as the data is processed by the pipeline, it is sent to the scene object. The scene class is responsible for keeping track of the different objects in the scene, such as the reconstructed mesh of the real world and any virtual objects such as balls. The scene class interacts with the collision model, to determine whether virtual objects collide with each other and with the generated model of the real world.
CHAPTER 3. SYSTEM DESIGN

3.3.2 World Cloud Creation

The Kinect sensors are placed at fixed locations in a room, aimed at a single focus point. To construct a virtual geometry model of the real world, the individual datasets of the sensors need to be combined into one. This occurs in steps 1 to 3 of figure 3.2. We first discuss the marker detection, and how the transformations are calculated.

As discussed in section 3.2, the system uses ArUco markers for finding the initial camera poses. Each sensor in the setup sees the ArUco marker, and based on this the pose of the sensor can be determined. A single marker provides enough information to determine the position and orientation. A whole board of ArUco markers provides more accurate information, and even when some markers are occluded the detection still works. See figure 3.5 for an example. We see two captures of the scene where the marker boards are correctly detected, and a coordinate system visual is overlayed. The axes correspond with the colors, i.e. xyz is mapped to rgb and thus the x-axis is colored red. We see that in each image, the orientation of the coordinate system corresponds with the orientation of the marker board. If we move the camera around the marker, the coordinate system will keep the same position on the markers. The specifics of how the detection algorithm works can be found in "Automatic generation and detection of highly reliable fiducial markers under occlusion" [24] by Garrido-Jurado et al.

![ArUco marker detection. A coordinate system is laid over the markers.](image)

OpenCV contains an implementation for the detection of ArUco markers. The algorithm produces a rotational vector \( \mathbf{rvec} \) and a translational vector \( \mathbf{tvec} \) that represent the camera’s pose in relation to the marker. We need to apply some modifications to them. First of all, we want to convert the rotation vector to a rotation matrix. The rotation is stored in a compact vector form, and we can use Rodrigues’ rotation formula to convert \( \mathbf{rvec} \) to a rotation matrix \( R \). Secondly, the coordinate systems of OpenCV and PCL do not match. We need to invert the Y-axis. This is simply a negation of the Y-component of the coordinates. Finally, these vectors are in camera coordinate space. This means that the origin \((0,0,0)\) of the transformation is at the camera itself. We want all cameras to have the same ‘notion’ of the room, that is, they should all know where they are in relation to each other and the marker. Therefore, we want to have all the cameras in the same coordinate space, with the same origin. We call this world space. We choose the marker to be this common origin.
See figure 3.6. Say we receive a new, unprocessed point cloud from sensor 1. It will have its origin at $c_1$. Instead, we want its origin to be at $m$, as to coincide with the other sensors. To go from the origin at $c_1$ to the origin at $m$ we do the following. Say we have a rotation matrix $R$ and translation vector $tvec$ which represent the transformation from marker to camera with origin $c_1$. To move the origin from $c_1$ to $m$, we perform inverse transformations based on $R$ and $tvec$. First we rotate the point cloud such that the marker is ‘flat’ in world space. To do so, we need the inverse of $R$. For a rotation matrix, its inverse is precisely its transpose, therefore

$$R_{inv} = R^T$$

Next, we need to determine what translation to apply such that $m$ is at $(0,0,0)$. We get

$$tvec_{inv} = R_{inv} \ast -tvec$$

Finally, we can create a transformation matrix that, when applied to the point cloud of sensor 1, transforms it so it has its origin at the marker. We get

$$finalTransform = R_{inv} \ast tvec_{inv}$$

During development, we noticed that the transformation matrices produced by the marker detection algorithm do not align the clouds well enough. This can be seen in appendix C, where we see the two captured point clouds of the table not being properly aligned with each other. In the top image of figure C.1 we see two circles, with the left circle showing two square shapes in the point cloud. These represent the same object, but do not overlap while they should. In the right circle we see two ‘pillars’ on top of the table. This is in fact a person sitting at the table, but who is now split in half. The two halves should be connected and form the shape of a person.

To correct this misalignment, we first apply a translation offset to each matrix. We move the point clouds a factor 0.8 towards the center. This was determined experimentally, and is roughly consistent for each capture. The offset causes the clouds to align better, but still the rotation does not match. Now that we have a rough estimate of the transformation needed for properly concatenating the two clouds, we can use the Iterative Closest Point (ICP) algorithm to determine the final transformation.

ICP is an iterative algorithm, which tries to find a transformation from one point cloud to the other. It starts off with a rough estimate of the transformation, which is refined in every iteration, and used in the next. In figure 3.7 we can see the general idea. Say we have a real-world piece of
string and have captured two point clouds of it with two different sensors. These point clouds are taken from two different viewpoints, and they both contain the same string. We want to find a common part in both clouds that we can use for aligning. In the figure we see this common part. Two iterations of the ICP algorithm can be seen, where in every iteration the string is aligned better.

![Figure 3.7: Two iterations of the Iterative Closest Point algorithm.](image)

The ICP algorithm has been studied extensively, and a good starting point in the literature is "Efficient variants of the ICP algorithm" [25] by Rusinkiewicz et al. There are many variants of ICP, and each variant typically affects one or more stages of the algorithm. PCL provides several algorithms for each stage, as outlined in "Registration with the Point Cloud Library PCL" [14] by Holz et al. We give a short overview of its workings.

In each iteration, the algorithm tries to find corresponding pairs of points between the source and target clouds, starting from a transformation estimate. By minimizing the distance between these point pairs, a new transformation is determined and used in the next iteration. This way, on every iteration the transformation estimate is improved. There are six stages in the algorithm:

1. **Select** source points from the source point cloud. PCL supports several sampling methods. The default is sampling in index space, where we simply take every \( n \)th point in the data. This is a fast way of sampling, but the form of the underlying data is ignored. In our implementation we use this default. Using a different sampling method could make the algorithm converge sooner.

2. **Match** corresponding (closest) points in the target point cloud. For finding these, PCL uses nearest neighbor search with a KD-Tree.

3. **Weigh** the corresponding pairs. A weight is assigned to each pair, and for this we simply use their Euclidean distance.

4. **Reject** outlier point pairs, whose weights are larger than a certain threshold. This threshold can be pre-defined, or based on the weights of other point pairs. In our implementation, we use a pre-defined threshold of 10 centimeters. We also have other rejection criteria, for example based on duplicate target points. In figure 3.8 we can see two examples. The
left image shows rejection based on point distance. The blue pair is rejected because the Euclidean distance crosses a certain threshold. In the right image, the blue pairs are rejected because they contain duplicate target points.

5. **Assign an error metric** to matched pairs. This allows us to say whether an iteration of the algorithm produces a better transformation than the previous. An example of an error metric is the sum of squared distance between corresponding pairs. This is the default in PCL and is also what we use for our implementation.

6. **Minimize** the error metric with respect to the transformation.

![Figure 3.8: ICP rejection criteria. The left image shows rejection based on point distance, the right image shows rejection based on duplicate targets. The source points are denoted by \( p \), and the target points by \( q \). The green pairs are kept, and blue pairs are rejected. Image source: [14]](image)

After every iteration we have a transformation matrix that makes the point clouds move closer together, and eventually the algorithm converges. We can then apply this final transformation to the two point clouds, and concatenate them. Now that we know how ICP works, we see that this algorithm is suitable for determining the final transformation for aligning two point clouds. We first use the marker to give us a rough estimate of the transformation matrix, and then use that as a starting point for the ICP algorithm to have it converge sooner. The result of the ICP algorithm in our system can be seen in figure C.2 of appendix C. We see that the two point clouds of the table are now properly aligned, and the table does not appear ‘broken’ anymore.

### 3.3.3 World Cloud Manipulation

**Downsampling**

When concatenation is finished, we have a single world cloud to perform further processing on. The point clouds produced by the Kinect contain approximately 150,000 points, depending on the scene. This data can be very dense, with a lot of points packed into a small space. When downsampling, points are removed from a point cloud while retaining as much information as possible. The approach we use is to overlay a voxel grid over the point cloud. A voxel grid can be seen as a 3-dimensional grid of small boxes (voxels). See figure 3.9. Next, we determine what points fall in what box, and if multiple fall into the same box they will be merged into one. In this case, the centroid of the points in the voxel will become the new point. The centroid can be seen as the center of the points in the box in 3D space. In other words, each voxel can only have at most one point.

A voxel grid has a clear quality parameter - the size of the boxes. The smaller the boxes, the more accurately a point cloud is filtered. In theory, if the boxes become infinitely small, the point cloud will be approached exactly and in fact no downsampling occurs.
CHAPTER 3. SYSTEM DESIGN

Figure 3.9: A voxel grid layed over a point cloud. Image source: [15].

Smoothing

The sampled point cloud data contains noise and measurement errors. These errors become clear in the final geometry model, either as bumpy surfaces or as messy triangulation. We use a smoothing algorithm to fix this. For our implementation, we chose the Moving Least Squares algorithm, as it is widely used and is readily available in PCL. The paper "An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation" by Andrew Nealen is a good introduction to the ideas behind the Moving Least Squares approach, but for completeness we describe it here.

We start off with the regular Least Squares technique. Say we have \( N \) points, where each point \( x_i \) with \( 0 \leq i \leq N \) is a 2-vector containing its \( xy \)-coordinates. In order to smooth this data, we want to fit it to a function as best as possible. This fit can be done in two ways. First, it can be done using a tangent estimation. We consider neighborhood points around \( x_i \) and we determine their surface tangent. We then use that tangent as the function to fit to. A second approach is to fit to a polynomial function of any degree, which we specify a priori. We find the best-possible fit by minimizing the distance between the desired polynomial function and the currently fitted polynomial. We call this the error function, which we want to minimize. This is done using the method of least squares, hence the name. In figure 3.10 we see three example fittings in 2 dimensions. We see some points scattered around, that seemingly follow a certain pattern. We want to find a polynomial, such that we get the actual curve underlying the data. In the top image, we attempt to fit the data to a polynomial of degree 8. We see that it does not fit with the original data well. In the middle image we fit to a polynomial of degree 15, and this produces a good approximation. Finally, in the bottom image we fit to a polynomial of degree 25, which is also a good fit. The higher order polynomial we have, the more processing power we need to approximate it. Depending on the data, this might be necessary. If we have a lot of curvature
variance on the surface, a high polynomial will approximate the surface better. In short, we want to find a minimal polynomial degree that produces good results.

In the regular Least Squares method, we iterate over the points and use all other points to determine a good fit. This approach is not suitable for large datasets such as ours. Therefore, our implementation uses the Moving Least Squares variation. Instead of considering all the points in the data when determining a good fit, we consider only a subset. We iterate over all the points and for each point we apply the least squares method to that point and only some of its neighbors.

![Least Squares approximation](image)

**Figure 3.10:** Least Squares approximation. From top to bottom we have a polynomial degree of 8, 15, and 25. A higher polynomial degree better approximates the shape of the underlying data.

### Upsampling

In the previous step, we performed smoothing by approximating mathematical functions. Having such a function, we can also calculate normals, calculate curves, and perform upsampling. The latter is most interesting to us. The Kinects will fail to see everything in the scene, leaving us with gaps in the data. Using the approximated function, we can interpolate and fill these gaps. See figure 3.11 for an example. In the left image we clearly see gaps in the data. The right image shows that the gaps are upsampled and filled. The Point Cloud Library supports two primary methods of upsampling, when using the Moving Least Squares smoothing algorithm.
• **SAMPLE_LOCAL_PLANE** For each point, the local plane is sampled using the calculated normal of the point and its neighbors. An imaginary plane is then drawn over that point, with a normal average to that of the point and its neighbors. That local plane is then filled with points. We have two parameters that we can influence when performing this step. They are the the local upsampling search radius and the local plane step size. The first parameter influences how many neighbors of a point will be sampled to determine the local plane, and the latter influences the step size as we move over the local plane to create new points.

• **VOXEL_GRID_DILATION** The input cloud is inserted into a voxel grid with small box size. Say we are looking at point \( x \) and we have fitted it to a certain function \( f \). When using voxel grid dilation, the point falls into a box \( b \). Say \( b \) has an empty neighbor box \( b_2 \), but the function \( f \) still crosses it. On every iteration of the upsampling algorithm, the non-empty boxes of the voxel grid are dilated. This means that since \( b \) contains a point \( x \), a new point will be added to \( b_2 \). This process is repeated for a predefined number of times. As a result, we get a new grid with constant density and filled gaps.

In PCL, upsampling only works when polynomial fitting is enabled. After upsampling, the number of points in the cloud has increased again. Depending on how big this number is, we might want to downsample again.

![Figure 3.11: Upsampling. The left image shows the upsampled point cloud, the right image shows the original. Image source: [16]](image)

### 3.3.4 Geometry Reconstruction

**Triangulation**

The next objective is to convert the point cloud to a representation that is useful for collision detection. There are several different forms, and a point cloud is actually one of them. The algorithm we use for collision detection requires another: a collection of connected triangles or quads that form a mesh. See figure 3.12. Here we see a hand modeled with triangles. Our implementation uses triangles because the tools and libraries we use support this best. Furthermore, the physics library we use, Bullet, supports creating collision models directly from the triangle data. In the figure we also see that more triangles gives a more detailed model. This comes at a price though, as this increases the processing power we need to render and perform further processing. Yet again, this is a balancing act between model accuracy and running time performance.

After the smoothing and upsampling step, we have a point cloud that is cleaned up from noise, and ready to turn into a triangle mesh. Our implementation uses the Greedy Projection Triangulation (GP3) approach. Besides GP3, there are several other triangulation approaches in
CHAPTER 3. SYSTEM DESIGN

PCL, but these did not produce satisfactory results. Greedy Projection Triangulation is based on the paper "On Fast Surface Reconstruction Methods for Large and Noisy Datasets" [26] by Marton et al, which in turn is based on "A fast and efficient projection-based approach for surface reconstruction" [27] by Gopi and Krishnan. We give a short overview of the algorithm here. Triangulation is performed incrementally. For each point \( p \), its \( k \) nearest-neighbors are found using a search radius \( r \), and an approximate surface plane is calculated, from which we get the surface tangent. Next, all neighbors of \( p \) are ordered around it and projected onto the plane. Redundant points that add nothing to the surface are then pruned. Finally, the neighbor points are connected to \( p \) to form triangles. In this step minimum and maximum triangle angles are taken into account. For example, if a triangle has an angle greater than 120 degrees to another triangle, we consider it a bad one as the resulting surface would be malformed. We have several parameters that we can tweak for this algorithm. First of all we have \( \mu \), which adjusts the search radius \( r \) based on the density around the point. The relation between the search radius and \( \mu \) is \( r = \mu \cdot d_0 \), where \( d_0 \) is the distance from the point to its closest neighbor. A higher value for \( \mu \) makes the search radius bigger. Note that the search radius could be different for each point, based on the point density. However, the use of a voxel grid in the downsampling step as well as in the voxel grid dilation upsampling step produces uniform point densities. Therefore, the effect of \( \mu \) is limited. Next, we have a minimum and maximum angle, which restrict the angle between triangles. Finally, we have \( r_{\text{max}} \) and \( k_{\text{max}} \), which place an upper bound on the search radius and the number of considered neighbors respectively.

Collision Model

We now have a virtual geometry model of the real environment that we can use for collision detection. In our implementation, we maintain a scene graph that contains a collection of all virtual objects we currently have, including the virtual geometry model itself. Besides that, our scene can have virtual balls, cubes, and basically any other 3D shape. We want these objects to interact with each other, and to make this realistic we require a physics system. Every virtual object in the scene has both a graphical model and a collision model. Our graphical system uses the Visualization ToolKit (VTK) library, which has an OpenGL backend. Our collision system uses the Bullet Physics library.
• **VTK Triangle Mesh** The VTK triangle mesh holds the generated triangles, and is used to render the 3D model to the screen. This is the 3D graphical model of the virtual object.

• **VTK Transform** The VTK transform contains position, rotation, and scaling and is used to transform the graphical model.

• **Bullet Collision Shape** The collision shape is used for determining collisions between virtual objects. The collision shape can be an approximation of the graphical model, in order to keep acceptable performance.

• **Bullet Transform** The Bullet transform is used to position, rotate, and scale the collision shape.

![Figure 3.13: Different collision shapes around a graphical model of a monkey head.](image)

When we perform collision detection between two objects, we determine if surfaces intersect. The more surfaces we have, the longer this collision detection will take. Our virtual geometry model of the environment can contain hundreds of thousands of triangles. For each object, we can either use the graphical representation itself as the collision shape or we can try to approximate it. In the first case, we would use the triangles in the mesh as the collision surfaces. In the latter case, we would approximate the model using simpler shapes such as cubes and spheres. In figure 3.13 we see an example of different collision shapes. The monkey is the graphical model and the red lines form the collision shape. In the leftmost image, we see an approximated sphere collision shape around the monkey. In this case, when we would determine if there is a collision, we would actually determine whether two spheres intersect. The rightmost image shows a collision shape that is based on the triangles of the underlying graphical model and thus has the same level of detail.

We want to use our virtual geometry model of the environment for collision and occlusion, and therefore we want the collisions to be as precise as possible. For the environment graphical model we use collision shapes based on its triangles. For other objects, we approximate using simpler shapes.
Chapter 4

Parameter Tuning

We want our system to be as fast as possible, and for augmented reality it must perform in real-time. Running time performance of the system is heavily dependent on the number of points in the point cloud. Intuitively, the more points a cloud has, the more calculations need to be performed in the pipeline and the worse the running time performance becomes. Each step in the pipeline either increases or decreases the number of points in the resulting point cloud. We can influence this by adjusting the parameters of our system. In the parameter tuning described below, we investigate the effects that the parameters have on the different processing steps of our system.

For each step in the pipeline we decide on what parameter value has the best trade-off between accuracy and running time performance. Eventually, after finding an optimal value for each step, we look at the combination of these parameters. Finally, we perform geometry captures on various objects to see the results of our tuned parameters. The dimensions in our virtual world are in meters. This means that when two virtual points are 0.5 units apart from each other, they are 0.5 meters apart in the real world. This one-to-one mapping of virtual units to real-world units is useful when reasoning about parameter settings. For example, when considering neighbors around a point and we specify a search radius of 0.1, we look for all points in the point cloud that are within 10 centimeters of the original point.

During tuning, we use the setup as shown in figure 4.1. The view of the cameras can be seen in figure 4.2.

Figure 4.1: Overview of the physical testbed setup.
CHAPTER 4. PARAMETER TUNING

Figure 4.2: The view of the individual sensors in the tuning setup. The boxes are used to visually examine the quality of the captured geometry.

4.1 Individual Parameters

In this section, we look at the parameters of each step in the pipeline and how they influence the point cloud.

4.1.1 Downsampling

The first step we perform is downsampling of the point cloud. This is necessary to make the performance acceptable, but the amount of downsampling we perform greatly affects the detail the resulting virtual geometry model will have. As described in section 3.3.3 we use a voxel grid approach. We have a single parameter we can influence, namely the size of the voxel grid cells. Several experiments were performed to determine the influence of the voxel grid. We have two Kinects aimed at an object, and vary the size of the voxels of the grid.

In table 4.1 we see the resulting cloud sizes for the varying box sizes. The minimal box size that we can use is 0.02, because for lower values PCL would throw errors such as "Leaf size is too small for the input dataset. Integer indices would overflow." This is a known issue with PCL, and happens when the cloud dimensions are large. We see that the larger the box size, the more points are removed from the cloud, and thus we get a smaller cloud size. We also see that the time spent in the downsampling step is roughly the same for every box size, indicating that the initial grid size determines the running time. As described earlier, a box size of 0.02 effectively means a box size of 2 centimeters, and a box size of 0.2 means a box size of 20 centimeters. If we think about it in terms of centimeters, we immediately find that 20 centimeters is simply too big. We would lose too much detail. The parameter value we choose depends on the application, but for our purpose a box size of 2 centimeters is reasonable. It is enough to distinguish features such as surfaces, but not too small as to create unnecessary detail. We also see that the point cloud size stays reasonable. With a box size of 0.02, approximately 365000 points are reduced to around 58000.

In appendix D.1 screenshots can be found of the varying downsample box sizes. We clearly see that the model rapidly loses detail and point density as the box size increases. We also confirm that 20 centimeters makes the cloud lose too much information.

To conclude, downsampling is an important step and the box size is a key running time performance variable. In the following experiments, the downsampling size is fixed to 0.02.


**Table 4.1:** Results of downsampling with varying voxel grid box sizes.

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Box Size</th>
<th>Cloud Size Before</th>
<th>Cloud Size After</th>
<th>Time Spent (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.02</td>
<td>366841</td>
<td>58554</td>
<td>77</td>
</tr>
<tr>
<td>2</td>
<td>0.04</td>
<td>365878</td>
<td>22440</td>
<td>81</td>
</tr>
<tr>
<td>3</td>
<td>0.06</td>
<td>366750</td>
<td>11771</td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>0.08</td>
<td>367386</td>
<td>7285</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>0.1</td>
<td>367520</td>
<td>5059</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>0.2</td>
<td>363641</td>
<td>1752</td>
<td>65</td>
</tr>
</tbody>
</table>

4.1.2 Smoothing

After the initial downsampling we perform smoothing. This is to remove outliers, and to better approximate the true shape the point cloud represents. In this step there is one variable that has the greatest influence on the outcome. We use the Moving Least Squares algorithm, and the main parameter is the search radius. This parameter influences the neighbors that are considered when traversing the points in the point cloud.

In table 4.2 we see the cloud sizes and experiment running times for the varying search radii. We see that the point cloud sizes stay roughly the same. This makes sense - although the algorithm does remove outliers, its primary goal is to move points around to fit them to a function. We see that the time spent rapidly increases, the bigger the search radius. This is to be expected. If the search radius is very big, for each point, every other point would be considered. In experiment 8, we enable the use of polynomial fitting. This has a big impact on running time.

In the previous downsampling step, we chose 0.02 as the box size. Because of this, points are at least 0.02 units apart and we have to take this into account when choosing the search radius for the smoothing step. In figures D.3 and D.4 the influence of different search radii is shown. In the first image of figure D.3, a search radius of 0.02 is used. We see that it has almost no effect. The search radius is too small, and not enough neighbors are considered. If we compare this with a radius of 0.1, as seen in figure D.4, we clearly see the effect of the search radius. With a search radius of 0.1 more points are considered and more smoothing is applied. In the last image, we see that the object on the table, the box, is smoothed to a lump resembling a parabola. We need to find a balance between too little and too much smoothing. The value 0.02 produces too little smoothing and 0.4 too much. A value producing good visual results is 0.08, which we will also use for the following experiments. We see that some noise is smoothed away; the box still resembles a box, and the surface of the table is smooth as well.
CHAPTER 4. PARAMETER TUNING

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Radius</th>
<th>Step Size</th>
<th>Size Before</th>
<th>Size After US</th>
<th>Size After DS</th>
<th>Time US</th>
<th>Time DS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.005</td>
<td>0.001</td>
<td>79443</td>
<td>5703782</td>
<td>95071</td>
<td>3409</td>
<td>729</td>
</tr>
<tr>
<td>2</td>
<td>0.01</td>
<td>0.002</td>
<td>79702</td>
<td>5721667</td>
<td>114699</td>
<td>3045</td>
<td>729</td>
</tr>
<tr>
<td>3</td>
<td>0.02</td>
<td>0.004</td>
<td>79800</td>
<td>5727434</td>
<td>140116</td>
<td>2902</td>
<td>596</td>
</tr>
<tr>
<td>4</td>
<td>0.04</td>
<td>0.008</td>
<td>79698</td>
<td>5724076</td>
<td>177798</td>
<td>2745</td>
<td>655</td>
</tr>
<tr>
<td>5</td>
<td>0.08</td>
<td>0.16</td>
<td>79745</td>
<td>5728383</td>
<td>309577</td>
<td>2895</td>
<td>712</td>
</tr>
</tbody>
</table>

Table 4.3: Results of local plane upsampling, with varying search radii and step sizes. We have combined two steps here, the upsampling (US) and the subsequent downsampling (DS) step.

4.1.3 Upsampling

There are two main approaches to upsampling, as previously described in section 3.3.3. We first look at the local plane upsampling approach. We have two parameters that influence the algorithm: the local plane search radius and the local plane step size. In figure 4.3 we see an example of their effects. When we performed the downsampling step, we picked a voxel size of 0.02, or 2 centimeters. This means that when upsampling, we need to have a search radius bigger than that to find sufficient points. As can be seen in the figure, a search radius of 0.005 is too small to fill any gaps. However, it does show the workings of local plane upsampling. For every point, an imaginary circle is drawn around it, such that the normal of the plane is consistent with the point normal. This circle is then filled with new points, based on the step size. In table 4.3 and in figures D.5 and D.6, experiment results can be seen. We see that the cloud sizes after upsampling stay roughly the same. This is because the radius and step size are scaled equally - when the radius doubles, the step size doubles as well. However, the size after downsampling does increase. The reason for this is that the larger search radius covers gaps in the data, with new data that did not exist before.

Figure 4.3: Local plane upsampling, with a search radius of 0.005 and step size of 0.001.
Table 4.4: Experiments for voxel grid dilation upsampling, with varying voxel dilation sizes and dilation iterations. We have combined two steps here, the upsampling step (US) and the subsequent downsampling (DS).

We now look at the voxel grid dilation approach. In table 4.4 we see the experiment results. The parameters of interest are the voxel grid dilation size, and the number of dilation iterations. We see that the upsampled grid sizes become very large for small dilation voxel sizes. With a size of 0.01 the grid contains almost 2.5 million points after upsampling. We also see that depending on these parameters, the computation times can be quite high. In figures D.7 and D.8, we see the resulting point clouds. We see that the larger the dilation voxel size, the less actual upsampling is performed. This makes sense, if we consider the voxel size in centimeters again. This approach is quite resource intensive, as even with large dilation sizes the computation times are high. For some parameter values it produces artifacts, which can be seen in figure 4.4. It is unclear why these occur.

Figure 4.4: Voxel grid dilation upsampling. Artifacts occur when dilation size is set to 0.08 and with 4 iterations.
4.1.4 Triangulation

In the triangulation step we transform the point cloud to a triangle mesh. As discussed in section 3.3.4, we have several parameters that influence the resulting mesh:

- Search radius $r$, adjusted for density by $\mu$.
- Maximum search radius $r_{\text{max}}$.
- Maximum number of nearest neighbors $k_{\text{max}}$.
- Minimum and maximum angles between triangles.

We can reason about appropriate values for $\mu$, $r_{\text{max}}$ and $k_{\text{max}}$. As mentioned before, $\mu$ has almost no effect since the point cloud has uniform density. The value for $r_{\text{max}}$ needs to be set to a value that is at least as high as the voxel grid size in the downsampling step. In the paper by Marton et al [26] we can find default values for these parameters. We leave $\mu$ to a value of 3 and we set the maximum number of nearest neighbors $k_{\text{max}}$ to 100. The maximum search radius we can vary between 0.2 and 1. During the experiments however this did not produce any noticeable effect. Finally, we could vary the minimum and maximum angles between triangles. However, as Marton et al. state, the minimum is optional and the maximum is to not create malformed triangles. Therefore, we leave them to 10 and 120 degrees respectively. This produces the results seen in figure 4.5, where we can clearly see the produced triangles.

Figure 4.5: Triangulation results. The scene captured is that of figure 4.1.
4.2 Combination of Parameters

We now know, for every step in the geometry reconstruction pipeline, what parameters there are and what effect they have on the point cloud. We have also found relations between the processing steps. Changing a parameter in one step influences one in another. If we use a large voxel size, we get points that are farther apart, and we thus need a higher search radius in the following steps. When we perform downsampling with a higher box size, we get better performance but at the cost of detail. As a result, the smoothing parameters have to change as well.

We can use the relations we found to create a global detail setting, that balances running time performance and amount of detail. It influences the size of the voxel grid cells, the smoothing parameters, whether we use ICP, and whether we use upsampling. We define it as a value between 0.0 and 1.0, and behaves as follows:

- 0.0: Low level of detail, good performance. Downsampling voxel size is large.
- 0.5: Balanced level of detail and performance. Downsampling voxel size is small.
- 0.6: Same as 0.5, but now ICP is enabled for better cloud alignment with multiple sensors.
- 0.7: Enable polynomial fitting and upsampling for the MLS step.
- 1.0: High level of detail, bad performance.

Between 0.0 and 0.5 we decrease the voxel grid size, and between 0.7 and 1.0 we increase polynomial order for smoothing. The relation between voxel grid size and the search radii of the subsequent steps we set as follows, with voxel grid box size being the primary variable that is adjusted by the detail level.

\[
\begin{align*}
\text{VoxelGridBoxSize} & = 0.02 \\
\text{MLSSearchRadius} & = \text{VoxelGridBoxSize} \cdot 5 \\
\text{GP3SearchRadius} & = \text{VoxelGridBoxSize} \cdot 2 \\
\text{MLSGlobalPlaneUpsamplingRadius} & = \text{MLSGlobalPlaneUpsamplingRadius} / 4 \\
\text{MLSGlobalPlaneStepSize} & = \text{VoxelGridBoxSize} \cdot 2
\end{align*}
\]

We have \(0.02 \leq \text{VoxelGridBoxSize} \leq 0.06\), since it does not make sense to have voxel sizes larger than 6 centimeter in the real world. Furthermore, we have that \(2 \leq \text{MLSPolynomialOrder} \leq 8\). A polynomial order higher than that produced no noticeable effects, but the running time increased dramatically. Note that a high detail setting does not necessarily mean the results are better. We could introduce noise in the data that makes the resulting mesh look bad. See figure 4.6. We see that a low level of detail smooths away the structures, and a higher level of detail introduce too much noise. A setting of 0.5 produces visually attractive results.
Figure 4.6: Geometry captures with different levels of detail. Figures 4.6a, 4.6b, and 4.6c have settings 0.0, 0.5, and 1.0 respectively. The scene captured is that of figure 4.1.
4.3 Object Mapping Examples

Now that we have defined a level of detail parameter, we set it to 0.5 and give examples of geometry capturing results for various objects. We first look at a capture of a room, and after that of individual objects. We use objects with different materials, to see how well they are captured.

4.3.1 Room

In figure 4.7 we see a geometry capture of a room, using a single Kinect sensor. The room is approximately 8 by 5 meters, with the Kinect positioned in one corner, facing the other. We see that the geometry at the opposite corner of the room contains holes. This is because the range of the Kinect is insufficient, and we cannot capture enough detail. Using upsampling would help here.

4.3.2 Solid Material

In figures 4.8, 4.9, and 4.10 we see virtual geometry captures of a small box, a lock, and a kettle respectively. We see mixed results here. First of all, we notice the noise on the edge of the table itself. This could have several causes, such as inaccurate sensor registration, noise introduced by pipeline processing, or through interference between multiple sensors. Next, we consider the small box in figure 4.8. Since it is a proper solid we see that it is captured, albeit noisy. We also see a lock in figure 4.9, which shows the effect of the smoothing. Since the lock is not very high, it is smoothed away, and we see it as a small bump on the table. This is an example of too much smoothing. In figure 4.10 we see a kettle with a shiny material. Shiny materials could produce bad results due to the time-of-flight technology of the Kinect. We see this is indeed the case, as the kettle appears distorted and noisy. The infrared light is reflected on the shiny surface of the kettle and does not return to the sensor properly.

4.3.3 Transparent Material

In figures 4.11, and 4.12 we see virtual geometry captures of a glass and a bottle. We see that the captures are very bad, in fact, the objects are almost not captured at all. This was expected, as a transparent material does not reflect the infrared of the Kinect, and thus does not reach the sensor. Therefore, the glass is not captured at all. The plastic bottle is captured, but only some parts of it such as the lid.
CHAPTER 4. PARAMETER TUNING

Figure 4.7: Virtual geometry capture of a room.

Figure 4.8: Virtual geometry capture of a small box.

Figure 4.9: Virtual geometry capture of a lock.
CHAPTER 4. PARAMETER TUNING

Figure 4.10: Virtual geometry capture of a kettle.

Figure 4.11: Virtual geometry capture of a small glass.

Figure 4.12: Virtual geometry capture of a transparent plastic bottle.
Chapter 5

Testbed

We are interested in how well our system performs in certain aspects, and for this we require a testing methodology. In this chapter we describe a generic testbed, that can be used for quantifying the performance of any augmented reality system. Having such a testbed, we can compare systems and their components, examine geometry capturing performance, and determine the suitability of an augmented reality system for different applications.

Our testbed focuses on a small set of generic tasks, which should relate to the actions one wants to perform with an augmented reality application. The task must be possible both in the real world, as well as in the virtual world. We first perform the task in the real world, and the information we get from this forms our ground truth. This ground truth is what we consider the ideal situation, and performing the same task in our augmented reality setup should ideally give similar results. The metric of a task must express the distance between what happens in the real world and what happens in the virtual world. Note that no current augmented reality system will be accurate enough to truly copy the real world, as there are simply too many variables in the real world to take into account. However, we can approximate it, and our metric should give an indication of how far we are from the ideal situation. Each task in the testbed has the following framework:

- **What** The name of the test.
- **Why** Why the test is useful.
- **How** How the test is performed.
- **Metric** Define the metric that we use to quantify the task.
- **Results** Define good and bad results.

To demonstrate the idea behind the testbed, we choose two tasks for our system implementation and evaluate them. In one task we focus on collision, and in the other on occlusion. Based on the results of performing these tasks, we can say something about how well our system handles the collision and occlusion problems. The first task is bouncing a ball on a table, which tests the collision property of our system. The second task involves occlusion, and quantifies how well we can see a virtual ball behind a box.
CHAPTER 5. TESTBED

5.1 Bouncing Ball

The first task we use for evaluating our system, is dropping a ball on a flat table. We first perform this task in the real world. We drop a physical ball straight down onto a table, and determine how it bounces away. In an ideal situation, the ball would bounce straight back up. Next, we use our geometry capturing system to create a virtual model of the table. We now drop a virtual ball on the table in the virtual environment. If the table would be captured properly, the virtual ball would exhibit the same behavior as its real counterpart, and also bounce straight up. However, due to irregularities in the captured geometry of the table, the ball will bounce away with a certain angle. In our framework this looks as follows:

- **What** Bouncing a virtual ball on a real table.
- **Why** We want to quantify the collision performance of our system. A bouncing virtual ball on a real table requires accurate collision detection.
- **How** First bounce a real ball on a real table, and examine the angle the ball bounces away with. Then do the same with a virtual ball and the virtual geometry capture of the table.
- **Metric** The bounce angle is the difference between the real and virtual angles the balls bounce away with.
- **Results** When the bounce angle is close to 0, it means the virtual ball drop mimics the real ball drop closely, which is desired. When the bounce angle is far from 0, we consider it bad as the virtual ball drop does not mimic the real world.

This task involves the collision problem, and to see why we first need to define collision. In figure 5.1, we see the before and after situation of a collision between an imaginary ball at point \( u \) and a surface. We start by shooting a ball from point \( u \) onto the surface, which it hits at point \( v \). It bounces away from the surface and ends up in \( w \). This is an ideal situation where we have no friction, and we thus get that angle \( a = c \). The collision with the surface makes the ball bounce away with a certain angle. We say that the ball collides with the surface, if the distance between \( u \) and \( p \), namely \( d(u, p) \) with \( d \) being the euclidean distance, is equal to 0.

![Figure 5.1: Before and after situation of a collision. An imaginary ball is shot from point u to v, ending up in w. Vector n represents the normal of the surface.](image)

In this task, there is collision between the ball and the surface of the table. The virtual geometry capture of the table consists of triangles, and we create a collision model from them as seen in section 3.3.4. The triangles of the virtual table surface might be irregular, and not form a completely smooth surface. This could be due to errors caused by the geometry reconstruction pipeline. If the virtual ball hits such a triangle, it will bounce away with an angle that does not correspond to its real counterpart. In other words, if the surface is bumpy, the bounce angle will be large and irregular. In figure 5.1 this would mean that \( c \) is not equal to \( a \).
CHAPTER 5. TESTBED

We have performed the bouncing ball task with our implementation of the system. In our tests, the bounce angle will never be exactly 0. This is because the geometry of the table will never be captured perfectly. To account for this, we perform the experiments repeatedly on different geometry captures and consider the average and standard deviation of the results. We examine varying levels of detail, from 0.0 to 1.0 with increments of 0.1. Per detail level, we have captured at least 300 different geometries. In figure 5.2 we see three screenshots of a single ball drop.

![Figure 5.2: Three frames of a virtual ball collision test.](image)

We first look at the resulting bounce angles, which we can see in table 5.1, and figure 5.3. We can see that the mean bounce angle comes closer to 0, the higher the detail level. Note the detail levels influence several parameters of the pipeline. At level 0.0, we have a large voxel size, and thus get a small number of points in the cloud. We lose information about the surfaces. When we then perform triangulation, the triangles get bigger and the vertices of the triangles are further apart from each other. This introduces errors, such that the balls bounce away with an incorrect angle. We see that the mean of the bounce angle becomes smaller when the detail level increases. At 0.1 we see that the bounce angle is lower than the trend, and can be considered an outlier. The parameters we pick influence each other, and depending on the specific geometry we try to capture, certain parameter settings work better than others. For our specific test setup, the particular settings at detail level 0.1 might produce good results. We also see, that a detail level of 0.8 and higher suddenly produce higher means. At these levels we perform upsampling, in the hope of creating a more accurate point cloud. However, this instead introduces noise, which makes the surface less smooth. Finally, we see the errors column in the results table. We classify an error as a virtual ball drop that does not collide with the table at all. This is most likely an implementation error, and could have to do with the collision detection algorithms we use from the Bullet Physics library. The problem could lie in precision, where the triangles are too small to perform collision detection on. No attempts were made to solve this, as these errors occurred only infrequently.

In figure 5.4, and tables E.1 and E.2 in appendix E, we see the running times of the steps in the pipeline. We have seen that if we have a high detail level, we need more processing time. This is confirmed, a high detail level means a high total running time. At detail level 0.6 and higher we enable ICP, which is clearly reflected in the running times for the WorldCloud step. ICP takes approximately 3000 milliseconds, which makes it a costly feature. Polynomial fitting and upsampling are enabled at a level of 0.8 and up, which is seen in the MLS step, and suddenly takes much longer. The detail level influences the running times in a range from approximately 150 milliseconds to 10000 milliseconds. This is evidenced by the mean running times and the standard deviation. For higher detail levels, the standard deviation becomes quite high. This means that the running times are inconsistent - at higher detail levels the running time can vary with thousands of milliseconds between consecutive frames.
Table 5.1: Bounce angle mean, standard deviation, and errors. We consider an experiment an error when no collision occurs at all.

<table>
<thead>
<tr>
<th>DL</th>
<th>Mean</th>
<th>StdDev</th>
<th>Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>16.3</td>
<td>10.5</td>
<td>0</td>
</tr>
<tr>
<td>0.1</td>
<td>6.9</td>
<td>2.8</td>
<td>0</td>
</tr>
<tr>
<td>0.2</td>
<td>14.5</td>
<td>12.5</td>
<td>11</td>
</tr>
<tr>
<td>0.3</td>
<td>9.6</td>
<td>12.0</td>
<td>0</td>
</tr>
<tr>
<td>0.4</td>
<td>6.5</td>
<td>8.1</td>
<td>0</td>
</tr>
<tr>
<td>0.5</td>
<td>5.8</td>
<td>4.4</td>
<td>0</td>
</tr>
<tr>
<td>0.6</td>
<td>5.9</td>
<td>5.0</td>
<td>0</td>
</tr>
<tr>
<td>0.7</td>
<td>3.7</td>
<td>9.0</td>
<td>119</td>
</tr>
<tr>
<td>0.8</td>
<td>27.8</td>
<td>21.8</td>
<td>158</td>
</tr>
<tr>
<td>0.9</td>
<td>23.7</td>
<td>19.8</td>
<td>121</td>
</tr>
<tr>
<td>1.0</td>
<td>7.1</td>
<td>6.8</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 5.3: Bounce angle mean for varying detail levels.

Figure 5.4: Running times for varying detail levels.
To conclude, we have shown that the detail level influences both the bounce angle and the running times. From 0.0 to 0.7 the bounce angles improve. Higher than that bounce angles become worse, because of polynomial fitting and upsampling. Furthermore, running times become too high for our purposes. We see an outlier at 0.1, but it is unclear why this occurs. The bounce angle is best at a detail level around 0.5. This was expected, as we optimized our parameters to this in section 4.1. The results of this test confirm that this setting performs well, balancing accuracy with performance.

5.2 Viewing an Occluded Ball

The second task we describe is viewing a partially occluded ball. Again, we first describe what would happen with this task in the real world, and then compare that to what happens in our virtual environment. Our setup is as follows. We place a box on a table, and place one ball in front of it and one ball behind it. The balls are fixed in place, and positioned such that when looking at the box from the front, the top halves of the balls are visible above the top of the box. See figure 5.5a, where we look at the setup from the point of view of the camera.

![Figure 5.5a](image1.jpg)

(a) V

![Figure 5.5b](image2.jpg)

(b) A, B

Figure 5.5: Our occlusion setup. Figure 5.5a shows the view of the virtual camera, the front view of our setup. Figure 5.5b is a top-down view.

Ball A is placed in between the viewpoint and the box, and ball B is placed behind the box, such that it is partially occluded. See figure 5.5b, where we see a top down view of the setup, with v being the viewpoint of the camera. From the point of view of the camera, the visible surface area of ball A acts as a reference, and the visible surface area of ball B is our metric. If we would perform this task in the real world, we would place the camera such that we see precisely half of ball B. We take a photo using the camera, and compare the number of pixels belonging to balls A and B. If we have correctly set up our scene, only the top half of ball B would be visible such that \( p(B) = p(A)/2 \) with \( p \) the surface pixel count of the ball.

Next, we perform the task in our virtual environment. We use our virtual geometry capturing system to capture the scene. The virtual geometry will have irregularities, due to noise in the data and errors caused by the geometry processing pipeline. Because of this, the top of the box will also contain irregular triangles and not appear smooth. This influences the amount of surface area of B that is visible, and we can use this as a metric. We take a picture using our virtual camera, and examine the pixels of both balls A and B. From the real world we know that half of the surface area of B must be visible, in other words that \( p(B)/p(A) = 0.5 \). We define the occlusion error metric as \( oe(B) = |0.5 - p(B)/p(A)| \), with \( 0 \leq oe(B) \leq 0.5 \). We consider both the cases, with B completely visible and B completely occluded, as equally bad. Both produce an occlusion error of 0.5. We say that the occlusion is good, when the surface of the top of the box is smooth. This is the case when the occlusion error is close to 0, and we see precisely half of ball B.
CHAPTER 5. TESTBED

This test looks as follows in our framework.

- **What** Viewing an Occluded Ball
- **Why** We want to quantify the occlusion performance of our system.
- **How** A virtual ball is positioned, such that exactly the top half is visible. Depending on the virtual geometry capture, more or less of this half will be visible due to triangle irregularities. When the geometry capture is good, surfaces should be smooth and this should improve occlusion performance.
- **Metric** Half of the ball should be visible. The *occlusion error* is the absolute value of the difference between 0.5 and the percentage of the surface that is visible.
- **Results** An occlusion error close to 0 is good, since that means half of the ball is visible. The further away from 0, we either see too much of the ball or too little and the worse we consider the occlusion.

As an example, we have performed this test on our implementation of the system. We have set up the scene as described, but instead of performing the task in the real world we have made certain assumptions.

- In our virtual scene, we place the camera manually such that it roughly sees half of ball $B$. We assume it is exactly 0.5, but in reality it is slightly different due to calibration errors.
- We fix the camera location relative to the registration marker. We assume the different geometry captures have exactly the same transformation, but in reality the captured point clouds are never the same, which cause the geometry to be offset slightly. This causes the ball to either be more visible or less visible, depending on the geometry capture. We assume this offset is negligible.

Per detail level, we have taken 6 different geometry captures and determined the occlusion error. We did so by taking a screenshot with the virtual camera and counting the pixels of both balls. See figure 5.6. In table 5.2 we see the results of performing this task with different detail levels. We see that at levels 0.4, 0.5, and 0.6 the mean occlusion error is the lowest, with a small standard deviation as well. For the other levels, we see that the occlusion error is slightly higher. In figure 5.6 sample screenshots can be seen. We see that the results partially overlap with the collision tests from the previous section. The error starts high at a detail level of 0.0, becomes better at 0.5, and becomes worse again for higher detail levels. In figure 5.6 we can see three screenshots of the test. The leftmost image has a detail level of 0.0, and we see that the right ball is partially behind a noisy triangle. The middle image shows a good capture, where roughly half of the ball is visible. We classify this as good occlusion, which is confirmed by the values in table 5.2. In the right image, we see a geometry capture with a high detail level. We see that the surface of the box is not smooth, and as a result the top half of the right ball is behind several noisy triangles.

![Figure 5.6: Occlusion tests, with detail levels of 0.0, 0.5, and 1.0 for the top, middle, and bottom image respectively.](image-url)
Table 5.2: Mean and standard deviation of the occlusion error.

<table>
<thead>
<tr>
<th>DL</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>0.28</td>
<td>0.11</td>
</tr>
<tr>
<td>0.1</td>
<td>0.15</td>
<td>0.11</td>
</tr>
<tr>
<td>0.2</td>
<td>0.17</td>
<td>0.10</td>
</tr>
<tr>
<td>0.3</td>
<td>0.14</td>
<td>0.04</td>
</tr>
<tr>
<td>0.4</td>
<td>0.10</td>
<td>0.07</td>
</tr>
<tr>
<td>0.5</td>
<td>0.12</td>
<td>0.05</td>
</tr>
<tr>
<td>0.6</td>
<td>0.15</td>
<td>0.09</td>
</tr>
<tr>
<td>0.7</td>
<td>0.24</td>
<td>0.04</td>
</tr>
<tr>
<td>0.8</td>
<td>0.15</td>
<td>0.05</td>
</tr>
<tr>
<td>0.9</td>
<td>0.24</td>
<td>0.10</td>
</tr>
<tr>
<td>1.0</td>
<td>0.26</td>
<td>0.10</td>
</tr>
</tbody>
</table>

Figure 5.7: Plot of the mean occlusion error.

Figure 5.8: Plot of the occlusion error standard deviation.
Chapter 6

Conclusions

Our motivation for the work in this thesis is that we want to create a realistic and convincing Augmented Reality environment. For this we need to merge virtual objects with the real physical world. There are several challenges to be solved, but the most fundamental ones are collision and occlusion between virtual objects and the real environment. To solve these, we require a virtual geometric model of the real environment. Only then can we place virtual objects within it, and only then can we provide adequate solutions for the collision and occlusion problems.

Our work was focused on getting a good understanding of the available techniques for creating a virtual geometry model, to implement one of them, and to create a testbed for determining the properties of the system. We first examined the definition of Augmented Reality, and what makes it realistic. Next, we examined several categories of depth sensors, and listed their advantages and disadvantages. Based on these, we decided on what kind of sensor to use for our implementation. The device we chose had to be affordable, available, and real-time. We chose the Microsoft Kinect V2 to form the basis of our real-world geometry mapping system.

Next, we discussed the design, architecture, and implementation of our system. We started from an overview, and gradually went into detail about the different steps in the pipeline, in going from point cloud to an actual virtual geometry and collision model. We examined the effects of the different parameters in our system and tuned them to make the system suitable for our goal of realistic Augmented Reality. For each individual step in the pipeline we found the best settings. For downsampling, this was a voxel diameter of 2 centimeters. This gives a significant speedup to the pipeline and retains enough detail for our purposes. Next, we looked at the smoothing radii, and found that a value that is 3 to 5 times higher than the voxel diameter works well. The next step, upsampling, introduces noise. After tuning all parameters, we defined a setting that influences the level of detail and running time performance. Our testbed results showed that it works as intended. Values between 0.0 and 0.5 produce acceptable running times with less than 500 milliseconds per frame, at the cost of detail, which is smoothed away. ICP is enabled at 0.6, which improves the alignment of point clouds coming from individual sensors and concatenated into the single world cloud. At a level of 0.7 we start upsampling, after which performance drops dramatically due to noise that is introduced. Higher levels introduce more detail but much higher running times, up to 10,000 milliseconds per frame. These results were as expected, as the level of detail was tuned to a setting of 0.5.

We created a testbed based on Augmented Reality tasks, to quantify both collision and occlusion. For this, we defined two quantitative measures, namely the bounce angle and the occlusion error. A small bounce angle and a small occlusion error mean we can perform realistic simulation in augmented reality applications. We then used this testbed to examine our implementation. We have discovered that our methodology for quantifying collision produces satisfying results. The smoother the surface, the smaller the bounce angle. Therefore, we can use it as a quantitative measure for further work. The occlusion error we defined is in agreement with the results of the collision tests. We have seen that certain materials are captured better than others. Solid materials are captured better than reflective or transparent ones. The camera of the Kinect does not
receive the projected infrared, as it is not reflected properly by the material.

Finally, we answer our two research questions: "How suitable is the Kinect V2 depth-sensing device for 3D environment mapping and for tackling the collision and occlusion problems in Augmented Reality?" and "Is there a suitable testing methodology for quantifying depth sensor performance?" We have shown that it is possible to create a virtual geometry model of a real-world environment using one or more Kinect V2 sensors. The suitability depends on the intended augmented reality application. It is possible to create an accurate virtual geometry model, but accuracy and detail greatly influence running time performance. We have shown that, as expected, there is a clear relationship between point cloud size and processing times. To keep running times acceptable, we decrease the detail of the captured point clouds. By having more powerful computing hardware, we can perform less downsampling, and retain more information. We have created a testbed for quantifying the captured geometry of any augmented reality system, and we defined tasks and metrics for collision and occlusion performance. Using these metrics, we can compare different devices and systems to each other, and say something meaningful about the quality of their geometry captures.

Limitations

- The first limitation is that we exclusively implemented the geometry reconstruction pipeline on the CPU. Previous work has shown, that the GPU is especially suited for the kind of processing we perform, notably for processing point clouds due to the large number of points. The ICP algorithm for example could be used more efficiently, with running times under 50 milliseconds, while it now costs thousands of milliseconds. Using the GPU could improve running times greatly. This would allow us to increase the quality parameters even further, in order to create a more accurate model.

- Another limiting factor is that we only used two Kinect V2 cameras. This has to do with the available hardware, and the fact that we require a separate USB 3 controller per Kinect. Ideally, we would like to experiment with more Kinects. There will always be blind spots in the room, and having additional sensors from different viewpoints minimizes these.

- We mostly use pre-existing techniques for manipulating point clouds, that were available in software libraries. It would be interesting to examine their features more in depth, and to quantitatively compare their properties. There are many algorithm parameters that can be tweaked further, and they can be tweaked to match the intended application.

Future Work

- In chapter 1 we have seen the Extent of World Knowledge scale. Our work focused on a specific part of it, namely on capturing geometry. However, there is more information we could obtain from the real world, that could be used to improve our augmented reality system. An example of possible future work could be to realistically merge real and virtual lighting. Virtual lighting is calculated using raytracing, where one shoots a ray of light into the virtual environment and follows it as it bounces from object to object. If we want to do this in our augmented reality world, we first of all need our captured geometry. This is not enough; we need material properties of the surfaces to realistically determine the reflection angle of the light. In other words, light is scattered differently based on material. Having material properties, we can also perform more realistic physics interactions. A virtual ball bounces differently on a hard table than on a pillow.

- Another possibility for future work could be to capture semantics of objects in the real world. Using this information, we could improve point cloud processing. If we know we have a square table with a plastic surface, we know that its geometry should be very smooth. We can then account for this in the smoothing step of the pipeline. In the same vein, if we know what object we have, we also know of what materials it consists. This could also be used in the future work of the previous point.
Bibliography


Real-time 3D Environment Mapping: Occlusion and Collision in Augmented Reality
Appendix A

Technologies Used

A.1 Hardware

The following hardware was used during the project.

- **Lenovo W520 Notebook** For portability during the development of the application, a notebook computer was used. A disadvantage of this was that it is not powerful enough for the application in most cases. The Lenovo W520 is more than 5 years old as of writing and has an Intel Core i7-2630QM CPU @2.00GHz, 8GB RAM, nVidia Quadro 1000M.

- **Custom Desktop Computer** A desktop computer is usually more powerful than a notebook. The main computer we performed experiments with contained a Intel Core i7-4770S CPU at 3.10GHz, 32 GB RAM, and a GeForce GTX970.

- **Kinect V2** The Kinect V2 depth camera by Microsoft is relatively cheap, readily available, and easy to use. The Kinect V2 is the successor to the popular first version, and has several upgraded features such as higher resolution cameras. It is quite a large and heavy device, but it can be mounted to a surface using a stand. In our case we used tripods and mounted the Kinects on them.

A.2 Software

The following programming language, frameworks, and tools were used to realize the application.

- **Operating System** We initially developed the application for use on a Windows system. This choice was made because the Kinect is made by Microsoft and thus the support for the Kinect on a Windows system was best. Later, it proved difficult for certain software to compile on Windows. More importantly, Windows does not properly support multiple Kinects attached to the same system. Therefore we switched to Linux, and the Arch Linux distribution specifically. Arch Linux is rolling-release distribution, which means it receives updates regularly and often. During the project we kept updates to a minimum as not to disrupt development by breaking things. We used Linux kernel version 4.6.4.

- **C++** The C++ programming language was chosen for various reasons. It is a relatively low-level language, giving a lot of control and if used well provides a high performance compared to higher-level languages such as Java or Python. The downside is that it is more complex to use, mostly in the areas of memory management and building/compiling. The language itself is mature, proven, and stable. As a result, there are a lot of libraries one can use with it. This was an especially important advantage - the Point Cloud Library and OpenCV libraries are invaluable and aimed at C++. Although there are wrappers for higher level languages, using C++ itself is the most flexible option.
• **CMake** The CMake build system is widely used and proved very useful. Libraries usually have helper functions that make them easy to use with CMake - this was also the case with most of the libraries described here.

• **Kinect SDK** Microsoft provides a driver and an SDK for Windows, that worked quite well. However, this driver does not support multiple Kinects attached to the same computer and was thus dropped.

• **Libfreenect2** On Linux, we used the libfreenect2 Open Source Kinect V2 driver. It has support for multiple Kinects on the same system. [https://github.com/OpenKinect/libfreenect2](https://github.com/OpenKinect/libfreenect2)

• **Point Cloud Library** The Point Cloud Library (PCL) has various functions to manipulate point clouds. It contains an open source implementation of KinectFusion, but we did not use that during the project as it was not mature enough and contained bugs. PCL contains a large collection of algorithms and during the project we used these extensively.

• **OpenCV** The OpenCV library was used for marker tracking in the initial camera calibration phase. It contains functions for manipulating images, extracting features, and has a module for marker tracking.

• **Visualization ToolKit** VTK was used for the 3D visualization aspects of the project. We use it to show the point clouds and generated meshes. VTK is a dependency of the Point Cloud Library and was thus available to us for 'free'.

• **Bullet Physics** One requirement we had for a physics library was that it provides a function for generating collision objects based on a set of triangles. Bullet physics has such a feature, and this worked nicely for us.

• **Qt** We used Qt for the user interface and multi-threading aspects of the application. Using Qt's signal and slot mechanisms, a lot of boilerplate code and manual synchronization of variables was not necessary anymore.
Appendix B

Application Threads

Our geometry capturing implementation needs to perform in real-time, and therefore it needs to be as fast as possible. Certain operations can be performed simultaneously and we use threads to do so. In figure B.1 we see the different threads we use in our implementation. For more details, see section 3.3.1.

![Diagram of threads in the implementation of our system](image)

Figure B.1: Threads in the implementation of our system. Each sensor has its own thread, and feed their individual point clouds to the geometry reconstruction pipeline which also has its own thread.
Appendix C

ICP Screenshots

Figure C.1: Top-down and side view of a scanned table. We see the two point clouds being misaligned. In the bottom image, the table appears broken.
In figure C.1 we see two concatenated point clouds, that do not align properly. The scene is similar to that of figure 4.1, but with the table in the center of the room and a person sitting at it. The top image shows a table, with a person sitting at it. In the left yellow circle, two square shapes can be seen. In reality, these shapes form a single box and thus should also appear as one in the point cloud. In the right circle we see a person sitting at a desk, that is now split in half. At the start of our system, we perform sensor registration using the common visual marker. This process by itself is not accurate enough to properly align two point clouds. For the final alignment we use the ICP algorithm. The result can be seen in figure C.2, where the two clouds are now properly aligned. For more detail see section 3.3.2.

Figure C.2: Side view of a scanned table. The two point clouds are aligned properly and now form a proper table.
Appendix D

Tuning Screenshots

D.1 Downsampling

Figure D.1: Downsampled point clouds using a voxel grid. Box sizes are 0.02 and 0.08 for the upper and lower image respectively.
When downsampling, points are removed from a point cloud while retaining as much information as possible. We see that as the voxel size increases, detail is lost. This is most notable in the bottom image of figure D.2, where the outline of the captured table is barely visible anymore. See section 3.3.3 for details about downsampling.

Figure D.2: Downsampled point clouds using a voxel grid. Box sizes are 0.06 and 0.2 for the upper and lower image respectively.
D.2 Smoothing

When we perform smoothing, we try to fit points to a mathematical function. In these figures we see the results of that. In the top image of figure D.3 we see that the box on the table is still rectangular. As we apply more smoothing in figure D.4, the box starts to deform, eventually resembling a lump on the table. We also see that the surface of the table contains more bumps. See section 3.3.3 for details about smoothing.

Figure D.3: Moving Least Squares smoothing with search radii 0.02 and 0.08 for the upper and lower image respectively.
Figure D.4: Moving Least Squares smoothing with search radii 0.1 and 0.4 (polynomial fitting enabled) for the upper and lower image respectively.
APPENDIX D. TUNING SCREENSHOTS

D.3 Upsampling

Upsampling introduces new points in the data to fill holes. In figures D.5 and D.6 we see examples of upsampling with the local plane approach. For each point, its local surface normal is determined. Within a specified radius of the point, new points are added corresponding with this surface. This can be seen in the bottom image of figure D.6, where we see circular groups of points. See section 3.3.3 for details about upsampling.

Figure D.5: Local plane upsampling with search radii of 0.005 and 0.02 respectively. The step sizes are 0.001 and 0.004.
Figure D.6: Local plane upsampling with search radii of 0.04 and 0.08 respectively. The step sizes are 0.008 and 0.016.
APPENDIX D. TUNING SCREENSHOTS

In figures D.7 and D.8 we see upsampling with the voxel grid dilation approach. Most notably in the bottom image of figure D.5 we see small lines of points. On every iteration of this upsampling algorithm, the non-empty boxes of the voxel grid are dilated. This means an empty neighbor voxel of a point gets a new point. We have restricted the number of dilation iterations to 2 and 4, which cause the lines to stop expanding. If we would increase this number, the lines would appear longer. This can already be seen between the bottom images of figures D.7 and D.8.

Figure D.7: Voxel grid dilation upsampling with dilation voxel size of 0.02 and 0.08 respectively. The number of dilation iterations is set to 2.
Figure D.8: Voxel grid dilation upsampling with dilation voxel size of 0.02 and 0.08 respectively. The number of dilation iterations is set to 4.
Appendix E

Pipeline Running Times

In tables E.1 and E.2 we can see the mean and standard deviation of pipeline running times. For different detail levels from 0.0 to 1.0, we have captured the running times per pipeline step. In chapter 4 we see the tuning of the parameters of individual steps, where we balance level of detail with running time performance.

<table>
<thead>
<tr>
<th>DL</th>
<th>Merging</th>
<th>Downsampling</th>
<th>Smoothing</th>
<th>Triangulation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>3</td>
<td>15</td>
<td>24</td>
<td>43</td>
<td>84</td>
</tr>
<tr>
<td>0.1</td>
<td>3</td>
<td>15</td>
<td>31</td>
<td>53</td>
<td>102</td>
</tr>
<tr>
<td>0.2</td>
<td>3</td>
<td>15</td>
<td>41</td>
<td>69</td>
<td>129</td>
</tr>
<tr>
<td>0.3</td>
<td>3</td>
<td>15</td>
<td>58</td>
<td>96</td>
<td>173</td>
</tr>
<tr>
<td>0.4</td>
<td>3</td>
<td>16</td>
<td>89</td>
<td>152</td>
<td>259</td>
</tr>
<tr>
<td>0.5</td>
<td>3</td>
<td>18</td>
<td>151</td>
<td>291</td>
<td>462</td>
</tr>
<tr>
<td>0.6</td>
<td>1551</td>
<td>17</td>
<td>145</td>
<td>282</td>
<td>1995</td>
</tr>
<tr>
<td>0.7</td>
<td>1368</td>
<td>18</td>
<td>145</td>
<td>285</td>
<td>1810</td>
</tr>
<tr>
<td>0.8</td>
<td>1371</td>
<td>17</td>
<td>486</td>
<td>296</td>
<td>2171</td>
</tr>
<tr>
<td>0.9</td>
<td>1385</td>
<td>17</td>
<td>767</td>
<td>314</td>
<td>2484</td>
</tr>
<tr>
<td>1.0</td>
<td>1481</td>
<td>17</td>
<td>1510</td>
<td>332</td>
<td>3334</td>
</tr>
</tbody>
</table>

Table E.1: Mean running times for each step in the pipeline in milliseconds.

<table>
<thead>
<tr>
<th>DL</th>
<th>Merging</th>
<th>Downsampling</th>
<th>Smoothing</th>
<th>Triangulation</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>6</td>
<td>29</td>
<td>48</td>
<td>86</td>
<td>169</td>
</tr>
<tr>
<td>0.1</td>
<td>7</td>
<td>29</td>
<td>62</td>
<td>107</td>
<td>205</td>
</tr>
<tr>
<td>0.2</td>
<td>7</td>
<td>29</td>
<td>83</td>
<td>140</td>
<td>259</td>
</tr>
<tr>
<td>0.3</td>
<td>7</td>
<td>29</td>
<td>117</td>
<td>193</td>
<td>346</td>
</tr>
<tr>
<td>0.4</td>
<td>6</td>
<td>29</td>
<td>178</td>
<td>306</td>
<td>521</td>
</tr>
<tr>
<td>0.5</td>
<td>6</td>
<td>29</td>
<td>303</td>
<td>584</td>
<td>927</td>
</tr>
<tr>
<td>0.6</td>
<td>3215</td>
<td>30</td>
<td>291</td>
<td>564</td>
<td>4081</td>
</tr>
<tr>
<td>0.7</td>
<td>2847</td>
<td>30</td>
<td>292</td>
<td>572</td>
<td>3715</td>
</tr>
<tr>
<td>0.8</td>
<td>2855</td>
<td>29</td>
<td>974</td>
<td>593</td>
<td>4415</td>
</tr>
<tr>
<td>0.9</td>
<td>2896</td>
<td>30</td>
<td>1537</td>
<td>630</td>
<td>5044</td>
</tr>
<tr>
<td>1.0</td>
<td>3082</td>
<td>29</td>
<td>3023</td>
<td>665</td>
<td>6740</td>
</tr>
</tbody>
</table>

Table E.2: Standard deviation of the running times for each step in the pipeline in milliseconds.