FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Editing Reality

A Tool for Interactive Segmentation and Manipulation of 3D Reconstructed Scenes

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Christoph Huber
Matrikelnummer 0826569

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Mitwirkung: Dipl.-Ing. Dr. Christian Schönauer

Wien, 19. Jänner 2016

Christoph Huber	Hannes Kaufmann

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Editing Reality

A Tool for Interactive Segmentation and Manipulation of 3D Reconstructed Scenes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Media Informatics and Visual Computing

by

Christoph Huber
Registration Number 0826569

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Assistance: Dipl.-Ing. Dr. Christian Schönauer

Vienna, 19th January, 2016

Christoph Huber	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Christoph Huber
Philippovichgasse 16/3, 1190 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Jänner 2016

Christoph Huber

Acknowledgements

I would like to thank Christian Schönauer for his patient and thoughtful guidance throughout the different phases of this project.

Thanks to Hannes Kaufmann for giving me the opportunity to work in such an interesting field.

Thanks to Alina Messner for proofreading.

Finally, a big thank you to all the people who participated in the user study.

Kurzfassung

Die zunehmende Verbreitung von günstigen 3D-Scannern hat vielen Menschen Zugang zu Technologie verschafft, die dreidimensionale Umgebungen in Echtzeit rekonstruieren kann. Derzeit verfügbare Anwendungen zur Rekonstruktion von Szenen zielen hauptsächlich darauf ab, ein einzelnes, verarbeitetes 3D-Modell für anschließenden 3D-Druck zu produzieren. Es gibt kaum Möglichkeiten zur natürlichen und intuitiven Navigation und Manipulation von Szenen. Zu diesem Zweck muss die virtuelle Umgebung in individuelle Objekte segmentiert werden, die ihren echten Ebenbildern so gut wie möglich gleichen. Um die Intuitivität eines solchen Systems noch weiter zu erhöhen, sollten gängige Ansätze zur Navigation von Szenen überdacht und verbessert werden.

In diesem Forschungsprojekt wurde ein Prototyp für die virtuelle Rekonstruktion und Manipulation von dreidimensionalen Szenen entwickelt. Das System ermöglicht schnelle und intuitive Rekonstruktion durch 3D-Scanning mit Hilfe eines Tablets in Verbindung mit einer Tiefenkamera. Individuelle Flächen und Objekte werden durch eine Kombination von automatischen und manuellen Segmentierungsmethoden identifiziert und voneinander getrennt. Die nachfolgende Verarbeitungsphase erlaubt das Schließen von Löchern und das Entfernen von kleinen, unerwünschten Komponenten. Anschließend werden die Tracking-Fähigkeiten des 3D-Rekonstruktions-Algorithmus ausgenutzt, um Anwendern die Navigation in der virtuell rekonstruierten Szene durch Bewegen des Tablets zu ermöglichen, dessen Bildschirm als Fenster in die virtuelle Welt fungiert. Der Touchscreen des Handhelds kann dazu verwendet werden, zuvor segmentierte Objekte herumzutragen und an jeden beliebigen Ort in der Szene zu verschieben. Vervielfältigung, Transformation und Entfernung von Gegenständen werden durch die zur Verfügung gestellten Werkzeuge ebenfalls ermöglicht. Bearbeitete Szenen können in verschiedene Dateiformate exportiert werden, um anschließend in anderen Applikationen weiterverwendet zu werden.

Zur Evaluation des Prototyps wurde im Anschluss an die Entwicklung eine kleine Benutzerstudie durchgeführt. Die Ergebnisse zeigen, dass das System eine hohe Benutzerfreundlichkeit aufweist und die Bedienung leicht gelernt werden kann. Teilnehmer konnten Szenen ohne größeren Aufwand in ausreichender Qualität rekonstruieren und segmentieren. Darüber hinaus haben sich die verwendeten Methoden zur Navigation und Manipulation von Szenen als höchst intuitiv und natürlich erwiesen. Die Evaluation hat zudem eine Reihe von Möglichkeiten aufgezeigt, wie eventuelle zukünftige Versionen noch weiter verbessert werden könnten.

Abstract

The rise of consumer devices capable of 3D scanning has given many people access to technology that is able to reconstruct three-dimensional environments in real-time at a relatively low cost. Available consumer solutions for scene reconstruction, however, are mostly focused on producing a single, cleaned watertight mesh that is suitable for 3D printing. There are no or very limited ways for a user to navigate and manipulate the reconstructed scene in a natural and intuitive manner. To this end, the virtual scene must be segmented into distinct objects that should resemble their real-world counterparts. To further aid the design of an intuitive system, prevalent approaches to scene navigation have to be revisited and improved upon.

In this research project, a prototype for the virtual reconstruction and manipulation of three-dimensional scenes has been developed. The system allows for quick and intuitive 3D scanning and reconstruction by using a tablet display in combination with a depth camera. Distinct planes and objects are identified and separated from each other by utilizing a combination of automatic and manual segmentation techniques. The subsequent processing stage allows users to fill possible gaps and remove small, unwanted components. Afterwards, the tracking capabilities of the 3D reconstruction algorithm enable users to navigate through their reconstructed virtual scene by physically moving the tablet. The mobile display functions as a window into the virtual world. By utilizing the touch capabilities of the screen, previously segmented objects can be carried around and repositioned anywhere in the scene. Duplication, transformation and removal of items is also possible with the provided tools. Edited scenes can be exported to one of several common file formats for use in other applications.

After development, a small user study was conducted in order to evaluate the prototype. The results show that the system has a high usability and can be learned easily. Participants were able to reconstruct and segment scenes in reasonable quality without much effort. Moreover, the methods used for scene navigation and interaction proved to be highly intuitive and natural. The evaluation also revealed several possible areas of improvement for future releases.

Contents

Kurzfassung **ix**

Abstract **xi**

Contents **xiii**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Aim of the Work . . . 2
1.4 Outline . . . 3

2 Theoretical Background and Related Work **5**
2.1 Microsoft Kinect . . . 5
2.2 3D Reconstruction with KinectFusion . . . 7
2.2.1 Depth Map Conversion . . . 7
2.2.2 Camera Tracking . . . 8
2.2.3 Volumetric Integration . . . 9
2.2.4 Surface Reconstruction . . . 10
2.3 3D Segmentation . . . 11
2.3.1 Plane Model Segmentation . . . 11
2.3.2 Euclidean Cluster Extraction . . . 12
2.3.3 Region Growing Segmentation . . . 14
2.3.4 Other Techniques . . . 15
2.4 3D Interaction . . . 18
2.5 Similar Tools . . . 18
2.5.1 Kinect for Windows SDK Examples . . . 19
2.5.2 ReconstructMe . . . 19
2.5.3 Skanect . . . 20

3 System Design and Workflow **21**
3.1 Hardware . . . 21
3.1.1 Components . . . 21
3.1.2 Setup . . . 23

3.2 Workflow Overview . . . 23
3.3 User Interface . . . 26
3.3.1 Overview . . . 26
3.3.2 User Guidance . . . 27
3.4 Preparation . . . 28
3.5 Scanning . . . 31
3.5.1 Scene Capture . . . 31
3.5.2 Reconstruction . . . 32
3.6 Segmentation . . . 33
3.6.1 Plane Segmentation . . . 33
3.6.2 Object Segmentation . . . 36
3.6.3 Plane Cut Segmentation . . . 38
3.7 Processing . . . 40
3.7.1 Small Component Removal . . . 41
3.7.2 Hole Filling . . . 41
3.8 Manipulation . . . 42
3.8.1 Navigation . . . 43
3.8.2 Interaction . . . 45
3.8.3 Export . . . 48

4 Implementation **51**
4.1 System Overview . . . 51
4.2 Dependencies and Foundations . . . 53
4.2.1 Frameworks and Libraries . . . 53
4.2.2 Win32 API . . . 55
4.2.3 OpenGL . . . 58
4.3 Controller . . . 63
4.3.1 Main Controller . . . 64
4.3.2 Graphics Controller . . . 66
4.4 2D User Interface . . . 67
4.4.1 UI Panels . . . 67
4.4.2 Style and Layout Management . . . 69
4.5 Rendering . . . 71
4.5.1 Renderables . . . 71
4.5.2 Scene Data . . . 74
4.5.3 Text . . . 75
4.5.4 Shaders . . . 75
4.5.5 Renderer . . . 77
4.6 Navigation . . . 77
4.6.1 Orbital Camera . . . 78
4.6.2 Sensor Camera . . . 79
4.7 Scanning . . . 80
4.8 Selection . . . 81

4.8.1 Color Picking . . . 82
4.8.2 Raycasting . . . 83
4.8.3 Duplication and Manipulation . . . 84
4.8.4 Highlighting and Transformation . . . 85
4.8.5 Plane Cut Interaction . . . 86
4.9 Segmentation . . . 86
4.9.1 Point Cloud Conversion . . . 87
4.9.2 Plane Segmentation . . . 87
4.9.3 Object Segmentation . . . 89
4.9.4 Plane Cut Segmentation . . . 91
4.9.5 Triangle Mesh Conversion . . . 91
4.10 Export . . . 92

5 Evaluation **95**
5.1 Participants . . . 95
5.2 Procedure . . . 95
5.3 Test Scenarios . . . 96
5.3.1 Bowling . . . 96
5.3.2 Stacked Boxes . . . 98
5.4 Data Collection . . . 99
5.4.1 Screen Capture . . . 99
5.4.2 Application Log . . . 99
5.4.3 Exported Model . . . 99
5.4.4 Questionnaire . . . 99
5.5 Pilot Study . . . 100
5.6 Results and Discussion . . . 101
5.6.1 Observations . . . 102
5.6.2 Questionnaire Analysis . . . 105
5.6.3 Created Reconstructions . . . 112

6 Conclusion and Future Work **117**

A Questionnaire **121**

List of Figures **133**

List of Algorithms **137**

Bibliography **139**

CHAPTER 1

Introduction

1.1 Motivation

In recent years, 3D reconstruction of real-world objects and scenes has become increasingly popular not only for research purposes, but also for personal use. This is largely attributable to the increasing availability of affordable devices capable of 3D scanning such as the Microsoft Kinect [49] and the Structure Sensor [59]. They enable a wide range of people to create 3D models of physical objects with reasonable quality. Having digital reconstructions of real-world items is useful for a variety of applications, ranging from consumer-level 3D printing to clinical analysis, cultural heritage and entertainment production.

The ability to interact with and navigate through 3D scenes is essential to many applications dealing with processing or editing of 3D models. Traditional approaches with virtual cameras that can be controlled with mouse and keyboard have their advantages for certain editing tasks, but are not entirely intuitive. *Natural User Interfaces* (NUIs) aim to overcome this limitation by focusing on human abilities such as touch, voice and movement [34][76]. Multi-touch displays, for example, enable a more direct manipulation of 3D objects through touch gestures [41]. However, interaction techniques using only a touchscreen are still restricted to a 2D surface. Several attempts have been made to counteract this problem by utilizing additional RGB or depth cameras. These devices enable intuitive scene navigation and seamless 3D object manipulation through head and hand tracking [20][22][29][61].

However, little research has been done on 3D interaction designed specifically for models obtained through 3D scanning. Algorithms used for 3D reconstruction typically involve tracking of the current camera's position and orientation in order to align the captured depth data of subsequent frames. These capabilities can be utilized even after completing a 3D scan, which makes it possible to navigate a scene through physical movement and reorientation of the depth camera. Using the device in combination with a touch display enables interactions with virtual representations of real objects. This

introduces a number of interesting possibilities for more intuitive 3D scene interaction techniques.

1.2 Problem Statement

There are several commercially available applications that allow users to capture and digitally reconstruct three-dimensional scenes [58][66]. Most of these toolkits are focused on producing a single, cleaned watertight mesh suitable for 3D printing. Even though some of them offer limited tools for model processing, almost no effort is put into designing intuitive and natural ways of manipulating a reconstructed scene. One example of this would be the ability to move specific scanned objects, e.g. a chair, around and place it somewhere else on the ground.

In order to enable this type of interaction, distinct planes and objects in the reconstructed scene have to be identified and segmented first. The resulting virtual clusters should resemble corresponding physical objects in the real world as closely as possible. Currently, there is no easy way for users to scan and subsequently segment their scene using only one toolkit. Proper mesh segmentation is crucial for more realistic scene manipulation and enables reconstructed 3D object models to be processed and exported individually for further use in different applications (e.g. game engines).

Existing tools for scanning and reconstruction usually offer scene navigation in the form of a mouse-controlled mesh viewer with cleaning and basic editing capabilities. While this classic approach has its advantages (e.g. for applying different filters and searching for holes), more intuitive and natural ways for scene navigation should be explored. Mobile touch displays and the tracking abilities of the 3D scanner can be utilized even outside the reconstruction process to navigate through virtual scenes by physically moving scanner and display in the real-world counterpart. An editor with the previously mentioned capabilities would introduce a new approach of interacting with reconstructed scenes and enable several novel possibilities for 3D object manipulation.

1.3 Aim of the Work

The goal of this research project was the development and subsequent evaluation of a system prototype that allows users to interactively scan, reconstruct, segment and process three-dimensional scenes. Furthermore, it should enable virtual scene manipulation through interaction with a tablet. The primary focus was on intuitive navigation and interaction design as well as an easy-to-use interface.

For the reconstruction and segmentation process, suitable methods that are fast, reliable and adjustable had to be selected and implemented. Due to large differences between scanned point clouds (e.g. size and quality), it must be possible to change certain parameters or even perform additional manual segmentations to achieve satisfactory results for an even larger variety of input data.

By continuing camera tracking after the reconstruction has finished, users should be able to move through the digital representation of their scene by using the tablet display

as a window into the reconstructed virtual world. Carrying around and repositioning objects through touch input in combination with physical movement in the real scene should also be possible. Further requirements for the prototype included tools for rotation, scaling, duplication and removal of objects. These capabilities should enable intuitive reorganization and manipulation of reconstructed scenes. In addition, users ought to have the option of exporting their segmented and processed models for further use in other applications, such as game engines or advanced mesh editors.

A small user study was conducted in order to evaluate the usability and functionality of the developed prototype. Participants were required to complete prepared test scenarios by using the different implemented tools. Afterwards, they were asked to provide feedback regarding specific aspects of their interaction with the system. The results of this user study should expose strengths and weaknesses as well as possible areas of improvement.

1.4 Outline

This thesis is separated into six chapters. Chapter 1 introduces the topic and states the goal of the project. Chapter 2 explains several technologies, concepts and methods that were used and discusses relevant related work. Chapter 3 presents the used hardware setup as well as the expected workflow and design of the developed system. The different application stages as well as the general interface are described. Chapter 4 is concerned with the actual implementation of the developed system, including used libraries and frameworks. Chapter 5 explains the design and setup for the conducted user study. Results of the performed tests are presented and discussed. Finally, Chapter 6 summarizes the key elements of this thesis and discusses open problems as well as possible areas of improvement for future work.

CHAPTER 2

Theoretical Background and Related Work

This chapter covers the underlying technology of the system and the techniques used for three-dimensional reconstruction and segmentation. In addition, related literature as well as software similar to the developed prototype is presented and discussed. The system utilizes a variety of supplemental basic interaction and manipulation techniques, such as 3D selection, rotation and translation. However, due to the scope of this thesis, their theoretical foundations are not included in this chapter.

2.1 Microsoft Kinect

Figure 2.1: The Microsoft Kinect and its main components. Modified after Zhang [85].

The Kinect is a widely available motion sensing device developed by Microsoft. It has originally been created for use with the Xbox 360 gaming console in order to give users the ability to interact with games through speech and physical movement. The

Kinect consists of a depth sensor, RGB camera, a microphone array and a motorized rack that can be used to adjust the device's orientation [49]. The components relevant for this thesis are shown in Figure 2.1.

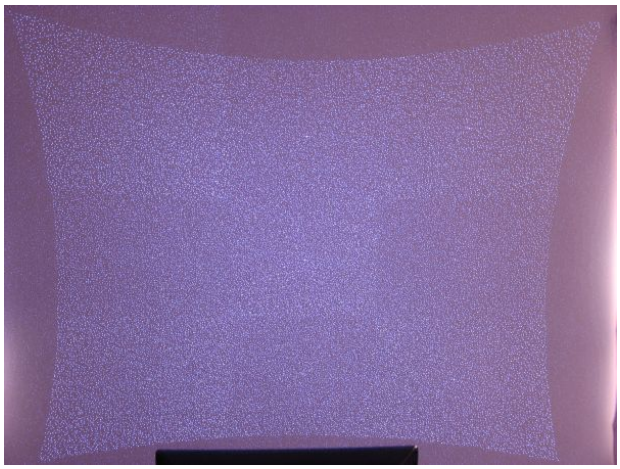


Figure 2.2: IR dot pattern projected onto a scene. Retrieved from [23].

The depth sensor consists of an infrared (IR) projector combined with an IR camera. A structured light technique developed by PrimeSense [21] is used to create a depth map as follows. The emitter projects a known IR dot pattern onto the scene, which is captured by the IR camera (see Figure 2.2). The observed dots are then matched with the known pattern. Since the exact distance between the projector and the camera is established, the depth value (i.e. the distance from the sensor) of any dot can be calculated using triangulation [14][21].

The Kinect has a depth operation range of 800mm to 4000mm. The accuracy decreases with the distance between a surface and the camera. Microsoft has released a Kinect specifically for Windows that supports *Near Mode*, providing a range of 500mm to 3000mm [45]. Due to the depth sensor utilizing IR light in order to calculate 3D positions, the system is able to work in many different lighting conditions. The Kinect for Xbox 360 has the following additional specifications [47][49]:

- **Color Stream:** 32-bit at a resolution of 640x480 and 30 frames/sec
- **Depth Stream:** 16-bit at a resolution of 640x480 and 30 frames/sec
- **Field of View (horizontal):** 57 degrees
- **Field of View (vertical):** 43 degrees

A newer version of the motion sensing device, the Kinect for Xbox One, has been released in 2013. It uses a time-of-flight sensor, which measures the time a light signal takes to travel between the camera and an object's surface. The device has a higher resolution, an increased field of view and an overall better accuracy than its predecessor

[50]. However, due to its availability and a higher resolution not being vital to this research project, the original Kinect for Xbox 360 was used.

2.2 3D Reconstruction with KinectFusion

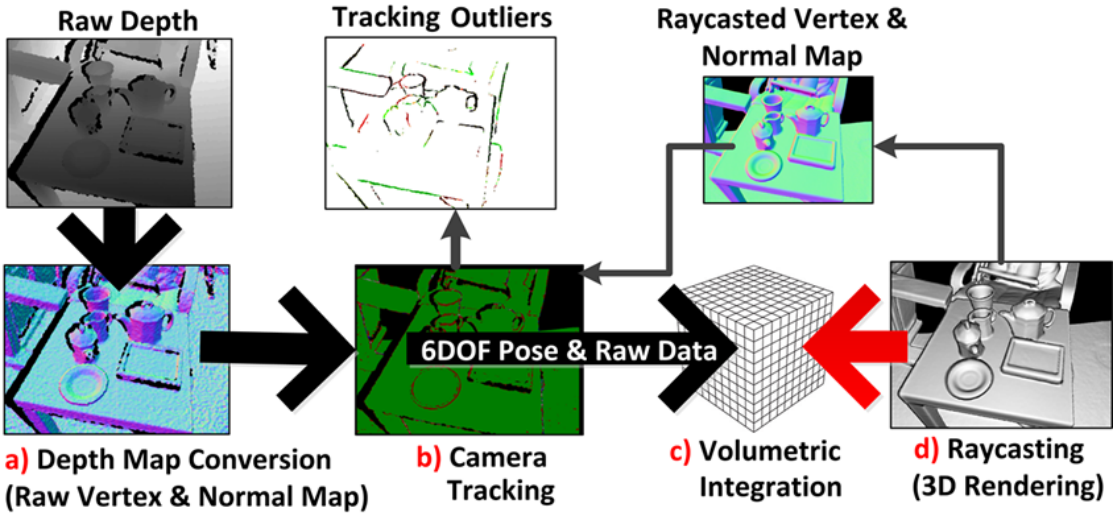


Figure 2.3: KinectFusion pipeline. Retrieved from Newcombe et al. [31].

KinectFusion is an algorithm developed by Microsoft Research [31]. This technique uses a temporal sequence of depth data collected from different frames and merges them to a single three-dimensional surface model of a specific scene. The implementation has optimized several steps of its pipeline (outlined in Figure 2.3) by using parallel computing on the graphics processing unit (GPU) and by assuming very little movement between two consecutive frames. This allows KinectFusion to run at interactive rates on most modern systems. Users can thereby preview the ongoing reconstruction in real time [31]. The specific steps performed by the algorithm are explained in detail in the following sections.

2.2.1 Depth Map Conversion

For each frame captured during the scanning process, the Microsoft Kinect produces a depth map, which contains the distance between the camera and the nearest object for every pixel in the depth sensor's current field of view [46]. As the produced map is typically noisy and inconsistent, KinectFusion uses a bilateral filter, which averages depth values around a pixel, to smooth the resulting values and preserve edges [31].

Using the known intrinsic parameters of the Kinect camera, each depth measurement can be reprojected as a 3D vertex in the camera's coordinate system to obtain a vertex map. A normal map, which contains the normals of each point in the vertex map, can be estimated by calculating the cross product of two neighboring reprojected vertices. To

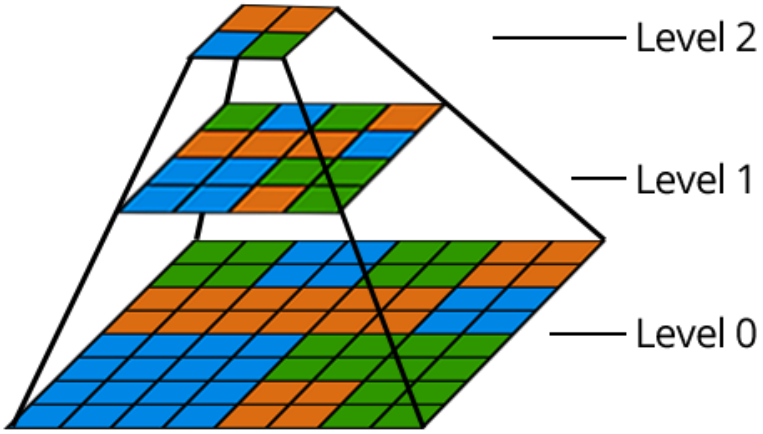


Figure 2.4: A multi-resolution pyramid with three levels. Modified after Pirovano [44].

optimize subsequent steps of the processing pipeline, KinectFusion stores these maps at different resolutions by using sub-sampling. The resulting structure is a multi-resolution pyramid, where each successive higher level has half the resolution of the one below it (see Figure 2.4) [44]. To convert the vertices and normals into global coordinates, the six degree of freedom transform of the camera pose is used. The result is a point cloud composed of a 3D position and a normal vector for each vertex [31][87].

Microsoft Research has highly optimized this step of the KinectFusion processing pipeline. Each 3D point and each normal vector are calculated by separate GPU threads in parallel and are then merged into maps [31].

2.2.2 Camera Tracking

In the first frame, the point cloud gathered in the previous step is regarded as the whole model. In each subsequent frame, the newly calculated 3D positions and normals have to be aligned and merged with that model in order to iteratively improve and refine it. To this end, the translation and rotation matrices of the current camera, i.e. its pose, in world space have to be found. KinectFusion uses a modified version of the *Iterative Closest Point* (ICP) algorithm, which takes the calculated cloud from the current and the last frame and tries to find the translation and rotation that will best align the two [8][31].

For each iteration, the original ICP method finds correspondences between the two data sets by matching points in a source cloud to the closest points in a reference cloud. The algorithm then estimates a transformation that minimizes the distances (*alignment errors*) between the corresponding points. This transformation is applied to the source cloud and the next iteration begins. The algorithm terminates when the sum of squared alignment errors between two consecutive iterations is below a specified threshold [8].

Since this method is too slow for real-time scenarios, Newcombe et al. adjusted the procedure to their specific needs. Due to KinectFusion running at interactive rates, it

has to make the assumption of very small movements between two subsequent frames. Therefore, the first cloud is transformed into camera coordinate space and then projected into image coordinates. This projection is then compared to the current vertex and normal maps. The algorithm considers two points to be corresponding if they are projected onto the same pixel. Outliers are rejected by testing whether or not the Euclidean distance and angle between two corresponding points in world space are within a chosen threshold. In this implementation, the algorithm tries to minimize the distance between a point in the current cloud and a tangent plane at the corresponding point in the previous cloud [87]. This is significantly faster than the original implementation. These calculations are performed at all three resolutions of the multi-resolution pyramid shown in Section 2.2.1. These modifications to the original ICP algorithm allow KinectFusion to parallelize the correspondence calculations by performing them in a separate thread for each point. After a sufficient number of iterations (e.g. maximum passes, error below a certain threshold), the modified ICP generates a relative camera transform which can be used to calculate the current global camera pose [31][87].

2.2.3 Volumetric Integration

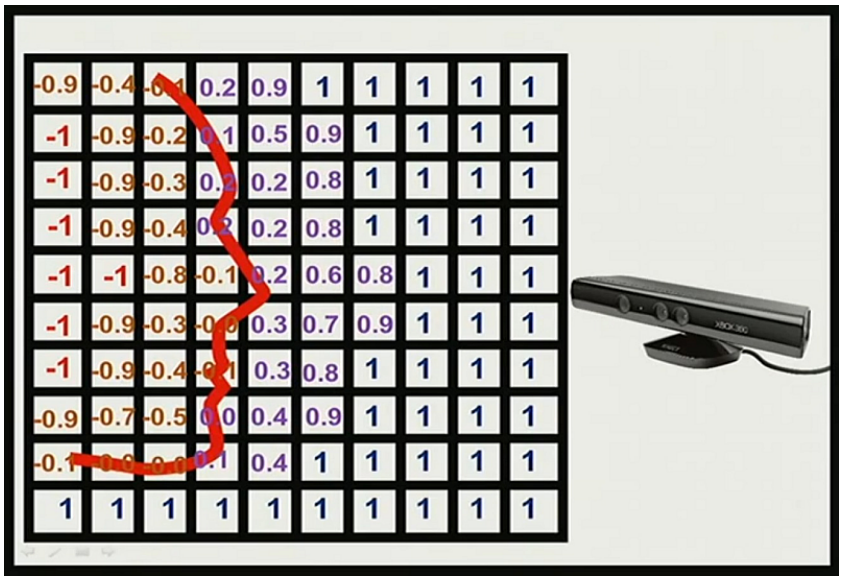


Figure 2.5: One slice of a 3D TSDF volume representation, where each square represents a voxel with the corresponding TSDF value inside. Positive values are in front of the surface, while negative values are behind it, with the surface itself being at the zero-crossing. Retrieved from Zhaoyang et al. [87].

After a global camera pose estimation has been found, the clouds can be merged into a single volume. The unfiltered depth data from the current frame is used to preserve details. KinectFusion uses a *Truncated Signed Distance Function* (TSDF) for its volumetric representation (see Figure 2.5). A TSDF volume is a uniformly subdivided

3D grid of voxels stored on the GPU. The calculated 3D vertices in world space are integrated into these voxels by applying a signed distance function, which maps every point to a value between -1 and 1 indicating their relative distance to a surface in the viewing direction of the camera. Negative values are either behind the surface or have not been measured yet, while positive values are in front of the surface, increasing as they are getting closer to the camera. The surface itself is at the zero-crossing, i.e. the point where the values change sign. This has the advantage of implicitly storing surface geometry and easily filling holes as new measurements are added. In addition to the signed distance value, each voxel also has a weight assigned to it, indicating the uncertainty of the surface measurement. This allows the system to merge the data to a single model by using the weighted average of all TSDF volumes [31][87].

This specific representation has also been chosen due to its ease of parallelization on the GPU. To construct and continuously update the fused volume, the algorithm launches one GPU thread per (x, y) position on the front surface. Each one of them then moves through each subsequent slice along the z-axis and updates the weight and averaged TSDF of the voxels along the way [87]. Although this technique is very inefficient in terms of memory usage, Newcombe et al. accepted this drawback due to its real-time capabilities [31].

2.2.4 Surface Reconstruction

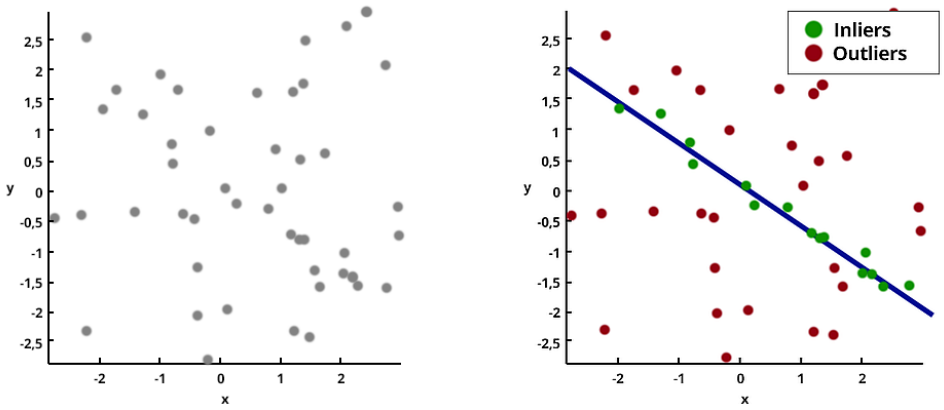
To reconstruct and render the calculated model, raycasting is performed. This technique chooses a point of view, in this case the estimated focal point of the camera, and casts rays into the scene. KinectFusion assigns one GPU thread to each pixel in the output image, each of which performs raycasting from that position. The ray traverses voxels along the ray and detects the zero-crossing, i.e. the change of sign in the TSDF volume. The algorithm calculates the true surface intersection point by linearly interpolating between the points on either side of the sign change. The 3D position of the intersection is obtained and assigned to the pixel the ray originated from. As each ray has a known starting position and direction, the surface normals can be estimated as a result. To compute shadows, a secondary ray is cast from the surface intersection to the light source. If it hits another surface before reaching its destination, the vertex is shadowed. In a similar fashion, other light calculations like reflections can be determined by using additional rays. The color of the output pixel is the highest resolution texture available for the voxel at the intersection point [31].

The result of this algorithm performed on the chosen TSDF volume is a visualized surface reconstruction of the 3D model from the camera's point of view, with correct occlusion handling and all the necessary requirements for virtual scene navigation [31]. In addition, the vertex and normal maps calculated in this step can be used as refined input data for the ICP algorithm during camera tracking in the next loop of the pipeline (as described in Section 2.2.2) due to the used camera pose corresponding to the physical camera transform. These maps contain considerably less noise than the raw data obtained by the Kinect camera (Section 2.2.1) and therefore result in a significant error reduction during ICP tracking [55].

2.3 3D Segmentation

3D segmentation refers to the task of dividing a 3D model (point cloud or mesh) into distinct submodels based on geometric properties of the available vertices. In the case of 3D reconstructed environments, each of the objects retrieved through segmentation should correspond to a physical object in the real scene. Approaches to 3D segmentation of scanned point clouds and reconstructed models have been widely studied [1][7][19][26][27][42][65][78]. Due to the scope of this thesis, only the methods and algorithms that are used in the implemented prototype as well as closely related techniques are introduced in the following sections. As the proposed system works with the underlying cloud data of the obtained 3D reconstruction during segmentation, only techniques used to segment point clouds are discussed. More information on the closely related mesh segmentation can be found in a variety of literature [2][11][73]. For a more exhaustive study on point cloud segmentation, refer to the survey performed by Nguyen et al. [56].

2.3.1 Plane Model Segmentation



(a) 2D point set with noisy data.

(b) Fitted line with RANSAC.

Figure 2.6: RANSAC fitting example for two-dimensional data.

Plane Model Segmentation is the process of finding and extracting planar surfaces, such as tables, walls, ceilings or floors, in a specific point cloud. This is a particularly important technique, as many other algorithms, such as those discussed in Section 2.3.2 and Section 2.3.3, rely on preliminary plane extraction to produce useful results.

Random Sample Consensus (RANSAC) was introduced by Fischler et al. [19] and is a well established algorithm that is often used for plane detection. This iterative method estimates parameters of a specific model (e.g. lines, planes, cylinders) from a series of measurements containing outliers. It is highly robust against small measurement errors, which are common in reconstructed scenes obtained through the methods presented in Section 2.2 [19][84]. A simple application of RANSAC for a 2D set of data is shown in

Figure 2.6. In this project, the method is used to detect planes in unorganized point clouds obtained by the Microsoft Kinect. The algorithm performs the following steps to find the best plane in a set of observed data values [19][70][72]:

Algorithm 2.1: Plane Model Segmentation using RANSAC.

Input: Point cloud P, max distance d, min inliers min_il, max iterations max_it
Output: Best plane parameters M_{best}, best inliers I_{best}

```
while number of iterations <= max_it do
    Randomly select three points p^k ∈ P and add them to inliers I_curr;
    Calculate parameters M_curr for a plane defined by p^k;
    foreach p_i ∈ P do
        if distance of p_i to the plane M_curr < d then
            Add p_i to I_curr;
        end
    end
    if Count(I_curr) > min_il then
        Reestimate plane parameters M_curr using all p_i ∈ I_curr;
        if Count(I_curr) > Count(I_best) then
            I_best = I_curr;
            M_best = M_curr;
        end
    end
end
```

The distance threshold has to be chosen according to the specific point cloud data and is therefore typically provided manually. In addition, due to the random nature of the inlier selection at the beginning of the main loop and the fixed amount of iterations, RANSAC is a non-deterministic algorithm that does not always provide an optimal solution. The greater the number of iterations, the higher the probability of a correct model [19].

2.3.2 Euclidean Cluster Extraction

A simple technique to divide an unordered point cloud into distinct objects is the *Euclidean Cluster Extraction*. This method assigns points to different clusters based on their Euclidean distance to each other. Rusu [69] used this technique in conjunction with RANSAC plane fitting (see Section 2.3.1) to segment different physical objects in a 3D reconstruction of a typical indoor kitchen table in order to allow robots to interact with them. Essentially, if the minimum gap between two sets of vertices is larger than a user-defined threshold, they belong to two different clusters [69][79].

One key task of this method is to estimate the distance between a point and its neighbors. To efficiently solve this problem, the algorithm makes use of the k-d tree representation, which is a special form of a binary search tree. Every node represents a

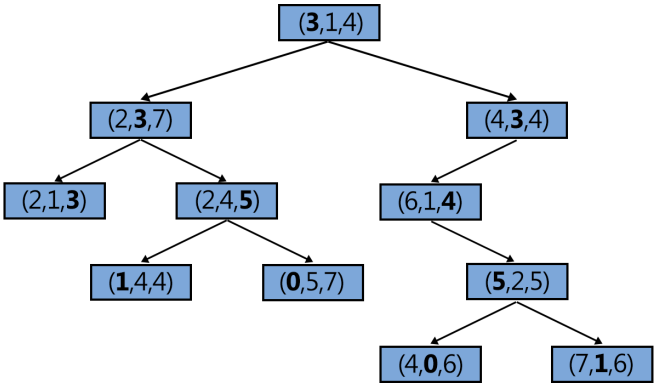


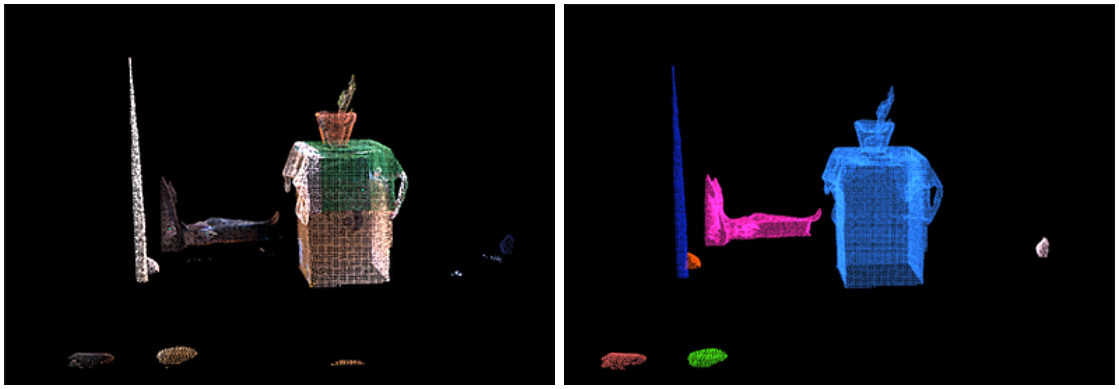
Figure 2.7: K-d tree representation for three-dimensional points. Each node represents a single vertex. The bold value indicates the dimension used for splitting the child trees. Modified after Cibils [12].

single vertex in the point cloud. Each level of the tree splits all children into a left and right subtree by using a hyperplane that is perpendicular to a specific dimension. The dimension depends on the current tree level. For example, if one starts with the median as the root node and chooses a split along the x-axis, all vertices with a smaller x-value than the median are assigned to the left subtree, while those with bigger x-values are put in the right one. On the next level, the two newly created subtrees are split again using the y-axis. The third level is then split employing z-values, while the fourth returns to the x-axis, and so on. Figure 2.7 illustrates a three-dimensional k-d tree constructed using the described process. This data structure can be used to efficiently perform k-nearest neighbor searches in point clouds, which finds the k closest points to a specified position. The steps performed for Euclidean Cluster Extraction are shown in Algorithm 2.2 [69].

Algorithm 2.2: Euclidean Cluster Extraction.

Input: Point cloud P, maximum distance d
Output: Segmented clusters C

```
Construct a k-d tree for P;
Create a queue Q;
foreach point p_i ∈ P that is not in a cluster do
    Add p_i to Q;
    Create a cluster C_temp with p_i;
    foreach p_i ∈ Q do
        Using the k-d tree, get all neighbors P_k of p_i in a sphere with radius d;
        Add P_k to Q;
    end
    Add Q to C_temp, add C_temp to C and clear Q;
end
```



(a) Input point cloud without ground plane. (b) Resulting clusters.

Figure 2.8: Euclidean Cluster Extraction example.

As discussed in Section 2.2.4, KinectFusion typically produces a densely populated point cloud that represents a single surface. Distances between vertices are generally very small [31]. Since almost all objects are connected in some way, Euclidean Cluster Extraction can not produce useful results on its own. Plane model segmentation, as described in Section 2.3.1, has to be performed as a preliminary step. After the planes have been identified, their individual vertices can be neglected during the cluster extraction. As a result, objects very close to these surfaces can be segmented into distinct clusters. Figure 2.8 shows an example of this procedure.

2.3.3 Region Growing Segmentation

While Euclidean Cluster Extraction focuses only on the spatial distance between a point and its nearest neighbors, *Region Growing Segmentation* uses additional information when deciding if neighbors should be added to the same cluster. It starts by selecting a seed point from the input cloud and subsequently adds neighbors to the seed point's cluster if they satisfy a given criterion. This routine is iterated until there are no unclustered points left in the provided point cloud [5][7][69]. The general procedure of this technique is outlined in Algorithm 2.3.

The key difference between popular region-based approaches lies in the selection of initial seed points and in the choice of criteria that are used to decide if point neighbors should be added to the same cluster. The most frequently used method for seed point selection was introduced by Besl et al. [7] in the original implementation of the algorithm for region growing segmentation, in which they identified seed points based on their curvature. The regions continue to grow based on criteria such as close distance between points and estimated planarity of surfaces [7].

The prototype developed during this thesis uses the improved constraints presented by Rabbiani et al. [68], who added points to the region if their differences in estimated surface curvature and normal orientation lie within a specified threshold. This method is also utilized by Rusu [69], who, in general, tries to solve segmentation problems that are

Algorithm 2.3: Region Growing Segmentation.

Input: Point cloud P, criteria *crit*
Output: Segmented clusters C

```
Create a queue Q;
while not all p ∈ P have been clustered do
    Add a seed point s ∈ P to Q;
    Create a cluster C_temp;
    foreach s_i ∈ Q do
        if s_i can be added to the cluster C_temp based on crit then
            Add s_i to C_temp;
            Add nearest neighbors of s_i to Q;
        end
    end
    Add C_temp to C;
end
```

very closely related to the ones presented in this thesis. Figure 2.9 shows a point cloud that has been segmented with the explained technique.

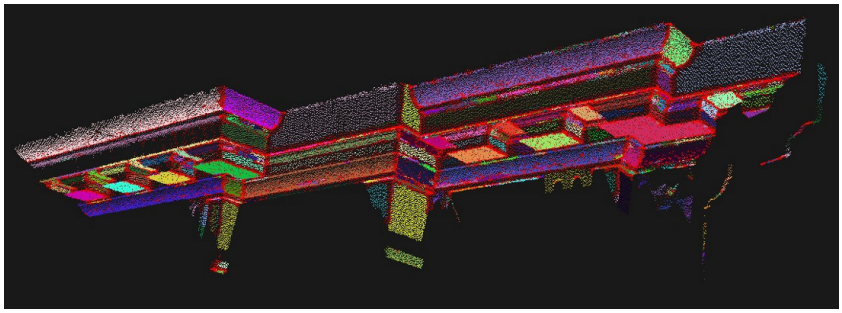


Figure 2.9: Region Growing Segmentation example using smoothness and curvature constraints. Red points have been rejected, other colors represent different resulting regions. Retrieved from [64].

The quality of a segmentation performed with region-based methods is highly dependent on the choice of initial seed points. A bad selection can lead to an abundance of seeded regions resulting in over-segmentation [69].

2.3.4 Other Techniques

The following paragraphs give a quick overview of some other relevant and popular state-of-the-art mesh and point cloud segmentation techniques.

The *Min-Cut Based Method* proposed by Golovinskiy et al. [26] segments a point cloud into a foreground and background by using a nearest-neighbor graph similar to the

Figure 2.10: Result of a Min-Cut Based Segmentation. Black pixels represent the foreground cluster (a street light). Retrieved from Golovinskiy et al. [26].

one described in Section 2.3.2. The algorithm assumes that points in the foreground are better connected to each other than to those in the background. A user-provided expected object radius is used to assign a weight to each node in the graph that encourages points farther away to be in the background. Afterwards, foreground and background points are added as hard constraints either automatically or interactively by users. The algorithm then performs a minimum cut of the graph, which minimizes the sum of weighted nodes while obeying the hard constraints. A foreground and background cluster for the provided point cloud can then be generated, as illustrated in Figure 2.10. While this method is primarily focused on outdoor scans, it can also be used to segment indoor reconstructions by using appropriate parameters [26].

Hierarchical clustering techniques, as described by Attene et al. [1] and Garland et al. [25], are used for mesh segmentation and try to address over-segmentation problems of region-based algorithms as discussed in Section 2.3.3. A cluster is initialized for each unique mesh element (e.g. a face) and a graph consisting of cluster nodes is constructed. Adjacent segments are assigned a cost for merging them into a single cluster, based on a given criterion. Regions with the lowest cost are then continuously merged until no more adjacent clusters fulfill the specified criterion. Garland et al. used Euclidean distance and the angle between triangles and representative planes as merge costs for face clusters [25]. Attene et al., on the other hand, evaluated adjacent clusters by fitting them to primitive models (planes, spheres and cylinders). They then estimated the minimum approximation error, measured as the average Euclidean distance between the cluster and the best fitting primitive [1].

While all of the algorithms mentioned so far have a resulting cluster number that is unknown in advance, several *iterative clustering methods* exist [43][65]. These techniques

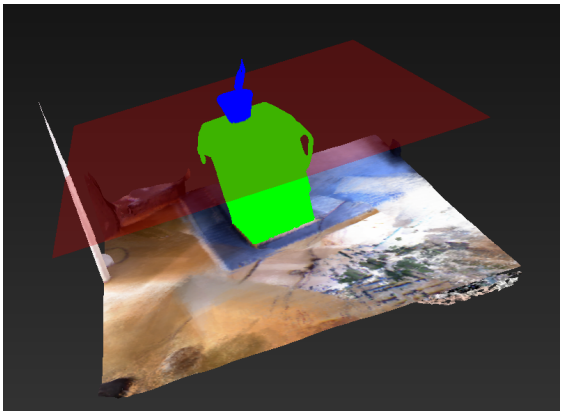


Figure 2.11: Segmentation using an interactively placed plane. The model is cut along the plane, with blue indicating the submesh above the plane and green the one below it.

are based on the algorithm by Lloyd [40] and require the desired number of segments k as an input parameter. The process begins by creating k clusters, each defined by a centroid, and subsequently assigns mesh elements (e.g. vertices, faces) to a cluster based on a distance criterion. After all elements have been assigned, the center points of the regions are updated and a new iteration begins. The process terminates when the cluster centroids stay constant [43][65].

Interactive segmentation techniques rely on user interaction to divide a 3D model into meaningful parts. The easiest approach is to allow users to explicitly cut a mesh by using predefined primitives such as lines, planes or curves. When a plane is positioned through a given model, the mesh is divided into the two submodels on either side of it (see Figure 2.11) [16]. This technique is also used in this research project.

(a) Paint Mesh Cutting (b) Dot Scissor

Figure 2.12: Segmentation techniques based on interactive user input. Figure 2.12a retrieved from Fan et al. [17], Figure 2.12b retrieved from Zheng et al. [88].

Fan et al. [17] introduced a more versatile interactive method. Users draw different sketches on a mesh and the algorithm subsequently merges vertices close to the drawings and includes vertex neighbors based on the angular distance, until all vertices are

clustered [17]. Zheng et al. [88] proposed an even simpler interactive interface for mesh segmentation that only requires a single click to perform a cut. The system then highlights a segment based on considerations of shape geometry and user intent [88]. Both of these techniques are illustrated in Figure 2.12.

2.4 3D Interaction

Interaction with 3D scenes is a very active field of research [20][37][41][61][74][82]. Due to the scope of this work, this section will only give a short overview of some recent and particularly interesting projects that are related to the methods used in this thesis. For more detailed explanations of the basic 3D interaction and navigation techniques used in the developed system (e.g. ray casting, color picking, transformations, orbital cameras), refer to the works of Hearn et al. [28] and Jankowski et al. [32].

Mossel et al. [54] introduced two novel 3D object manipulation techniques for mobile devices. *3DTouch* combines 2D touch information with the device orientation to reduce the overall number of touch gestures needed for object manipulation. *HOMER-S*, on the other hand, is an adaption of the well-known *HOMER* interaction technique introduced by Bowman et al. [9]. It requires only a single touch gesture for object selection and maps the position and orientation of the handheld onto the selected object [54].

Fritz et al. [22] developed a mobile application using a tablet with an attached Kinect that uses head tracking to allow users to freely look around in a virtual scene. In addition, the depth sensor is used to detect hand and finger gestures to provide a natural interface for 3D object manipulation [22].

Holodesk by Hilliges et al. [29] is an interactive system that uses a Kinect in combination with a semi-transparent display allowing users to seemingly reach into a scene and directly manipulate three-dimensional objects.

Zhang et al. [86] focused on integrating 3D scanned models into a game-based virtual laboratory. The aim of this work was for users to be able to look at and interact with the imported object. However, their research was very narrowly focused on integrating one mesh into an existing premodeled virtual environment instead of solely interacting with the reconstruction [86].

Sugiura et al. [77] developed a system consisting of an optical see-through head-mounted display and an attached RGB-D camera. It enables dynamic 3D interactions with virtual objects that superimpose a real environment. A physics engine provides collision detection and allows virtual items to move in a more realistic way [77]. However, the system focuses solely on interaction with superimposed virtual objects and does not provide any tools for real-world scene reconstruction or manipulation.

2.5 Similar Tools

There are several commercially available products that allow consumers to scan and digitally reconstruct three-dimensional scenes with low-cost depth cameras. Some of

them offer additional limited mesh processing and cleaning abilities. The most popular tools for real-time reconstruction of 3D environments are shortly described in this section.

2.5.1 Kinect for Windows SDK Examples

Microsoft Research built a number of example applications using the Microsoft Kinect that are publicly available as part of their *Kinect for Windows Software Developer Toolkit* [48]. The kit provides source codes for several examples that utilize the KinectFusion algorithm (see Section 2.2), including applications for basic scene reconstruction with or without color as well as head scanning. The *KinectFusion Explorer* is the most complete example project in that collection. In addition to an implementation of the basic KinectFusion algorithm, it offers features and options regarding different resolutions, model exports, as well as adjustment of depth thresholds and several other parameters used during the reconstruction process. All applications are available either in C# or in C++ [48].

2.5.2 ReconstructMe

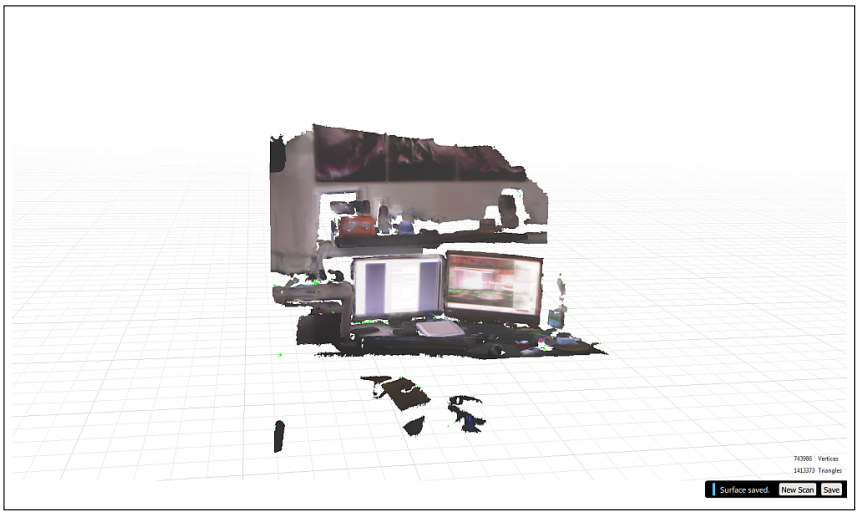


Figure 2.13: Interface after a finished scan with the free version of the ReconstructMe software [66].

ReconstructMe [66] is an easy-to-use 3D real-time scanning application that supports a wide range of depth cameras including the Microsoft Kinect. The software uses its own implementation for the reconstruction process, which is very similar to the KinectFusion method described in Section 2.2. ReconstructMe is able to provide real-time previews during the scanning process, but optionally supports offline reconstruction for computers with older hardware as well [66].

The focus of this application lies on the reconstruction of a single three-dimensional object that can then be exported to common file formats for 3D printing or further

editing in external mesh editors [66]. It offers no user-driven processing or segmentation capabilities whatsoever and does not provide any methods for the user to directly interact with the scene. As a result, the user interface (see Figure 2.13) is very clean, seeing that there are very few customizable options available.

ReconstructMe offers a Software Development Kit that allows developers to implement custom applications for three-dimensional scene reconstruction using their provided algorithm. The SDK also provides basic post-processing operations such as noise removal, hole filling as well as reduction of the overall triangle count [67].

2.5.3 Skanect

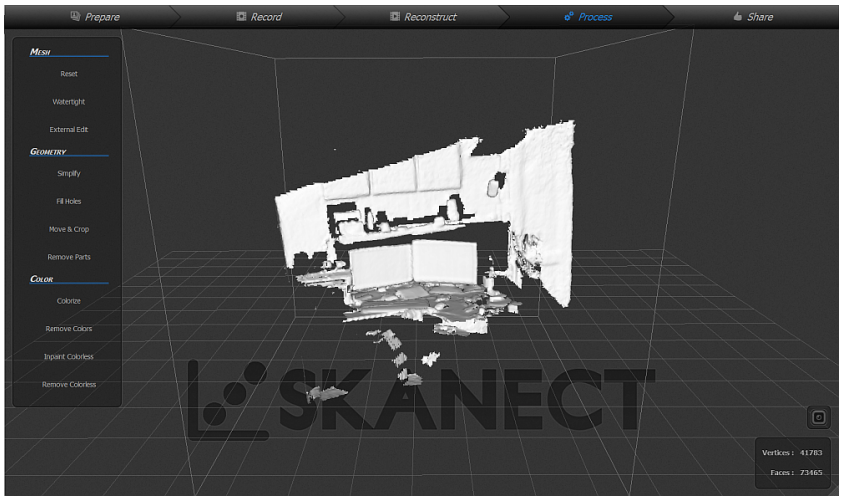


Figure 2.14: Skanect interface during the processing step [58].

Skanect [58] is another very popular solution for obtaining fast and reliable 3D scans. It supports many different depth cameras and utilizes ICP (see Section 2.2.4) during the reconstruction process. Similar to the other applications mentioned in this section, the software provides a very convenient way to capture high-quality colored 3D scans while showing real-time previews of the ongoing reconstruction [58].

In addition to this basic functionality, Skanect includes a step dedicated to mesh processing right after the scanning procedure (see Figure 2.14). Users can close holes with a specified minimum hole size and make the mesh watertight by connecting all distinct components. The geometry of the reconstructed model can be simplified by reducing the overall number of triangles and by removing small components. In addition, the whole mesh can be moved around and even cropped using the ground plane of the editor. The final model can be colorized and exported [58].

However, the application does not offer any way to perform mesh segmentation and does not utilize the depth sensor for subsequent scene navigation and manipulation.

CHAPTER 3

System Design and Workflow

This chapter is primarily concerned with the overall design and workflow of the developed prototype. First, the used hardware setup is presented. Then, a short overview of the expected system workflow is given. The chapter continues by taking a closer look at the overall composition of the user interface and the displayed guidance messages. Afterwards, the usage of each individual application stage is explained in depth. Details on the technical implementation can be found in Chapter 4.

3.1 Hardware

This section provides an overview of the different hardware components as well as the general setup of the developed prototype. Since the current system is primarily a proof of concept, the hardware setup could be subject to change in future releases.

3.1.1 Components

The developed prototype consists of a Microsoft Kinect, a tablet computer and a Windows PC. The depth camera is attached to the tablet and has to be plugged into the desktop PC and a power outlet. Section 3.1.2 describes the setup in more detail. The different components and their purpose in the system are described in the following sections.

Microsoft Kinect

The Kinect is a low-cost motion sensing device developed by Microsoft. It consists of a depth sensor, RGB camera, microphones and a motorized rack that can be used to adjust the orientation of the device. The first-generation Kinect was originally introduced in 2010 for Microsoft's gaming console Xbox 360, while the second generation became available in 2014 for the Xbox One. Both iterations can be used on Windows PCs. Details about its general functionality are discussed in Section 2.1. In this prototype,

due to lower cost and availability, a first-generation Kinect for Xbox 360 is being used. As the only relevant differences between the generations are increased resolution and depth fidelity, the chosen version only makes a difference in terms of quality, not in terms of functionality.

The Kinect is primarily used to perform 3D scanning and reconstruction by utilizing the KinectFusion algorithm described in Section 2.2. In addition, the data gathered during the capturing process, together with the current estimated position of the camera, will be used to navigate through the reconstructed scene during the manipulation step.

Windows PC

Due to the high hardware requirements for the KinectFusion algorithm (see Section 2.2) and the lack of officially supported drivers for tablets, the primary application has to be executed on a dedicated desktop PC running Microsoft Windows. In particular, KinectFusion requires a strong graphics card with a high amount of memory to produce results in good resolutions. The exact hardware specifications of the computer used for this prototype are as follows:

- **Operating System:** Windows 7 Professional 64-bit
- **CPU:** Intel Core i5-3570K Processor, 4 Cores @ 3.4 GHz
- **Graphics:** NVIDIA GeForce GTX 970 with 4 GB Memory
- **Memory:** 8 GB DDR3
- **Storage (of Application):** Samsung SSD 840 with 256 GB

Android Tablet

The prototype requires touch input as well as a handheld display to create the illusion of a window into the virtual world. An Android tablet is used as the primary display output. It is the only device that end users interact with directly. Since the primary application runs on the Windows PC and is streamed to the tablet, the only hardware requirements are Wi-Fi, a multi-touch sensitive touchscreen and an appropriate display size and resolution. The specifications of the tablet used for this prototype are as follows:

- **Model:** Samsung Galaxy Tab 10.1
- **Operating System:** Android 4.0.4
- **CPU:** NVIDIA Tegra 2, 2 Cores @ 1 GHz
- **Display Size:** 10.1 inch (25.65cm) widescreen
- **Display Resolution:** 1280x800 pixels
- **Memory:** 1 GB RAM
- **Wi-Fi:** 802.11 a/b/g/n, Dual-band support (2.4GHz, 5GHz)

3.1.2 Setup

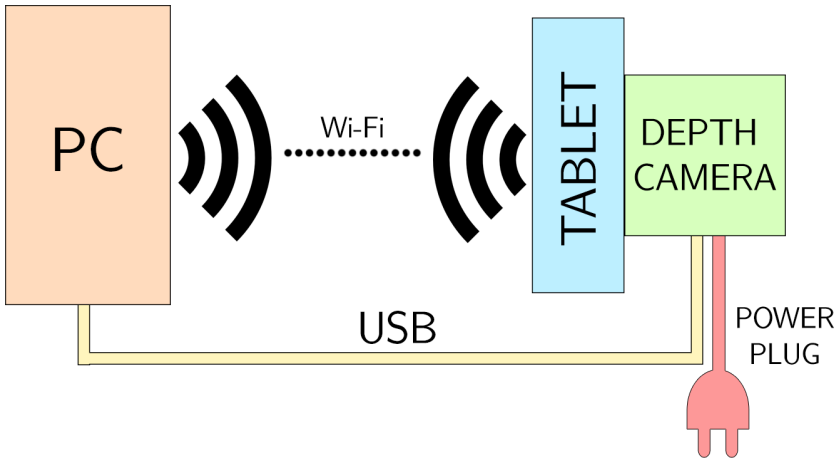


Figure 3.1: Hardware setup.

The general hardware setup is illustrated in Figure 3.1 and consists of a Microsoft Kinect that is attached to the back of an Android tablet. Due to the predetermined platform of the KinectFusion implementation, compatibility issues with Android and high hardware requirements, the main application has to be run on a Windows PC. As a result, the Kinect must not only be connected to a power outlet but also to the Windows PC via USB. During ongoing usage of the system, the display of the Windows PC is being streamed over the network to the Android tablet using *Splashtop Streamer HD* [75], so that the handheld can function as the application's sole display output. In addition, users can interact with the tablet by using touch gestures that are transmitted back to the desktop PC, where they are recognized as mouse input. As a result, the system feels like a native tablet application, even though all the required processing power is being provided by the PC. The only limiting factor is the length of the cable used to connect the Kinect to the PC and power outlet, which can be extended by using USB and power cord extensions.

In possible future iterations of this system, a Microsoft Surface tablet with a high-performing GPU could be used to eradicate the need of two separate devices dedicated to input/output and program execution respectively. Due to the high cost of such a device and this prototype being primarily a proof of concept, the slightly more cumbersome setup is being used. It has, however, no disadvantages in terms of functionality.

3.2 Workflow Overview

The developed application provides users with several tools needed to digitally reconstruct a scene, segment it into distinct objects, process it, manipulate and navigate it in certain ways and then export it for use in other applications. The simplified workflow from a user's perspective is illustrated in Figure 3.2.

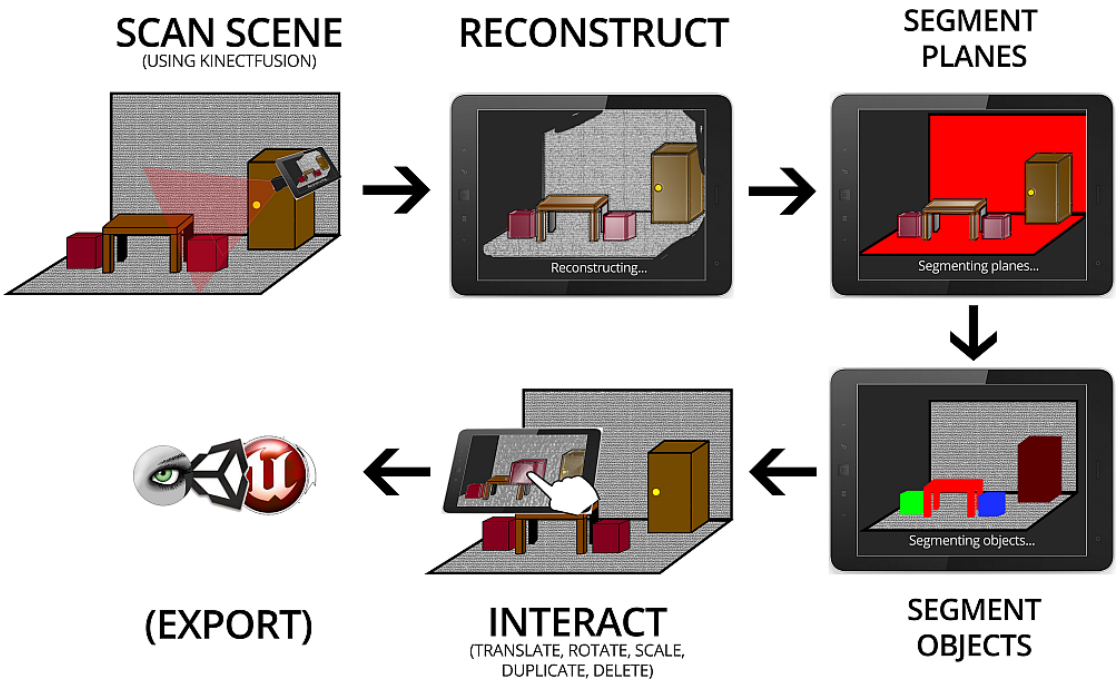


Figure 3.2: Simplified application workflow.

In the preparation step, users can adjust the desired size of the reconstructed scene. The bigger the scene, however, the lower the overall quality of the reconstructed mesh. Section 3.4 will deal with this step of the workflow in more detail.

After changing the parameters to their liking, users can start the scanning process. The environment is scanned by moving around the scene while capturing it from different angles through the tablet with an attached Kinect camera. This step utilizes the KinectFusion algorithm presented in Section 2.2 and therefore allows users to get a real-time preview of the ongoing scene reconstruction at all times. The scanning procedure can be restarted at any given time if the current reconstruction proves to be unsatisfactory. When users are satisfied with the current preview, they can finish the scan and start the automatic scene reconstruction. During this procedure, several processing and cleaning algorithms are performed to improve performance and general usability of the reconstructed model. For more details about the scanning and reconstruction steps, see Section 3.5

After the reconstruction has finished, users are presented with the refined virtual representation of their scene. In a first segmentation step, all immovable planar surfaces (e.g. walls, floor, ceiling) or surfaces on which distinct (movable) objects are placed (e.g. table surfaces), are detected and segmented. This is a necessary preliminary step for the subsequent object segmentation. During this procedure, the algorithm uses plane fitting (see Section 2.3.1) to highlight possible planar surfaces found in the scene. If the highlighted area is too thick, too thin or contains holes, users can adjust certain parameters to improve the suggestion. Each plane has to be confirmed or rejected manually. In

the case of confirmation, the highlighted area is segmented as a distinct (immovable) object and will not be included in the subsequent (movable) object segmentation. If it is rejected, no segmentation is performed. Either way, the algorithm tries to find other planar surfaces and presents them to the user in a similar fashion until no more planes with sufficient size can be found. The process is described in more detail in Section 3.6.1].

In the next step, the application allows users to segment all the parts that have not been classified as static planar surfaces into distinct objects. To this end, users can choose between Euclidean (see Section 2.3.2) or Region Growing Segmentation (see Section 2.3.3) and are able to adjust parameters for either one of them. A preview of the segmentation with the chosen method is presented with the click of a button. It shows the resulting clusters with the currently selected parameters, where each distinctly segmented object in the scene has a different color. If the preview is considered to be good enough, users finish this step, which results in the reconstruction model being segmented according to the chosen options. The object segmentation is discussed in-depth in Section 3.6.2.

After the automated object segmentation has been performed, users have the possibility to refine the resulting clusters. This is especially helpful if not all objects could be separated from one another during the previous step. A semi-transparent plane spanning across the entire reconstruction can be aligned to either one of the three axes and subsequently rotated and translated. The existing mesh segments can then be split into two subclusters along this prepositioned plane. This procedure can be performed as many times as needed to achieve a more satisfying segmentation. More details can be found in Section 3.6.3.

The model has now been segmented into distinct movable objects as well as immovable planes and can be simplified and improved during the processing step. Users can eliminate small objects that were accidentally created during scanning or segmentation. To this end, all connected components that contain less than a specified number of vertices can be removed. If desired, the application can try to fill holes with a provided minimum diameter. The methods used during this step are described in Section 3.7.

After the final processing step has been performed, the application enters the scene navigation and manipulation mode. By utilizing the tracking functionality of KinectFusion without any reconstruction taking place, users can navigate through the reconstructed scene by moving in the real world while carrying the tablet. As the Kinect is attached to the back of the handheld, the display can be used as a window into the virtual world, which means that users can see each part of the virtual reconstruction by pointing the prototype in the direction of its real-world counterpart. In addition, the touchscreen can be used to interact with the reconstructed scene. By performing simple gestures, previously segmented distinct objects can be picked up and carried around by physically moving in the scene. They can even be scaled and rotated along the camera axes and then placed on any other movable or immovable objects (including planes such as the floor or walls). Movable objects can also be duplicated or removed from the scene entirely. For a quick overview of the current process, users can always switch to an orbital camera and assess the situation. Using these techniques, the reconstructed environment can be explored and manipulated in an intuitive way. When users are satisfied with their

manipulated scene, they can export either the whole model or each individual segment for use in other applications such as mesh editors or game engines. This step is described in more detail in Section 3.8.

Users can switch back to previous steps at any time while using the application. This will, however, result in losing the current progress in all subsequent steps. By switching back to the preparation stage, for example, the whole reconstruction process can be started over without having to restart the application.

3.3 User Interface

As stated in Section 3.1.2, users control the prototype exclusively by interacting with a tablet computer. While some interactions only require physical movement around the scene as well as orientational adjustments while carrying the handheld, most require explicit touch input. Since the prototype was implemented as a Microsoft Windows application and is only streamed to the tablet, these particular aspects have to be taken into consideration during designing the interface.

As a result, the graphical user interface (GUI) during the different application stages was primarily designed for tablet displays, which are typically smaller than desktop computer screens. While it was primarily tested at a resolution of 1280x800, as used by the tablet chosen for the prototype (see Section 3.1.1), the application scales well and can be used with a wide variety of resolutions.

3.3.1 Overview

As shown in Figure 3.3, the overall composition of the GUI allows the reconstructed scene to occupy most of the screen. This is particularly important, since too many obtrusive overlays would hinder the goal of the tablet acting as a window into the virtual world. Whenever controls such as buttons and sliders are needed in a particular step, they are generally placed in a panel on the right side of the screen. This allows users to easily interact with them even while holding the tablet with both hands. During the steps requiring two buttons due to binary decisions, such as scanning (see Section 3.5) and plane segmentation (see Section 3.6.1), the controls are placed vertically centered on the left and right side of the screen. These regions are particularly easy to reach for users carrying the tablet, as both thumbs will usually be positioned roughly over these areas. Additionally, most buttons and sliders have a deliberately large scale to allow for comfortable touch input without much frustration due to small controls. During most application stages, the button on the lower right corner of the scene view can be used to reset the orbital camera.

The bar at the top of the screen is constantly visible throughout all stages of the application and visualizes a user's current progress. Users advance through the displayed steps from left to right, with the currently active one being highlighted. Users can jump to any step by selecting the corresponding tab. However, the different procedures require the completion of previous stages to work properly. Therefore, it is primarily designed

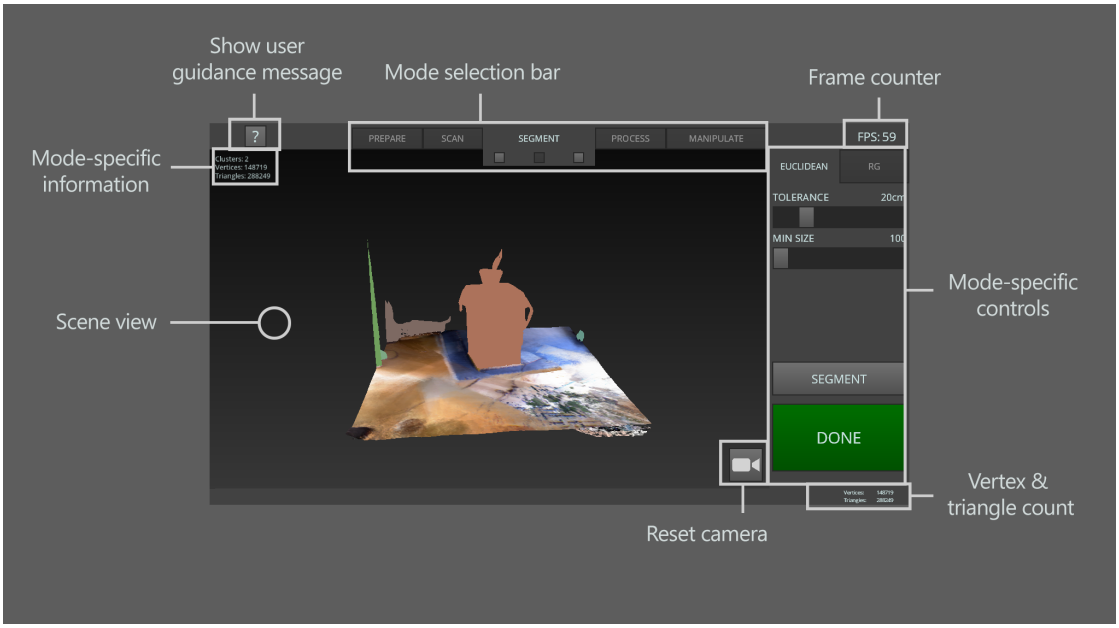


Figure 3.3: Application interface overview.

as a means to go back to previous steps only if users want to revert a specific decision. It is also noteworthy that the progress of the current stage will be lost if users switch to a previous step and finish it. The mode selection bar can also be used to start a new reconstruction by going back to the preparation screen. Only the segmentation stage consists of more than one substep (plane, object and plane cut segmentation), even though it is summarized into a single tab in order to save screen space. However, as shown in Figure 3.3, each of the distinct segmentations is visualized by small squares below the corresponding tab element and can be selected individually.

In addition, the GUI displays specific information about the scene and the current state of the system. The vertex and triangle count of the displayed model is shown in the lower right corner of the screen at all times. This information is also displayed in the upper left corner of the scene view during all segmentation stages, along with the current cluster count. The frames per second are displayed in the upper right. Users should pay particular attention to it during the scanning procedure, where the counter should be as close to 30 as possible to ensure good results. If it is significantly lower, users should consider lowering the overall mesh quality or using better hardware to optimize the scan. Whenever the system has to perform automatic steps without needing manual input, users are continuously informed about the processes that are currently being executed.

3.3.2 User Guidance

There are a total of seven distinct stages in the system, each of which has slightly different controls and a variety of options. In addition, users are required to follow some basic

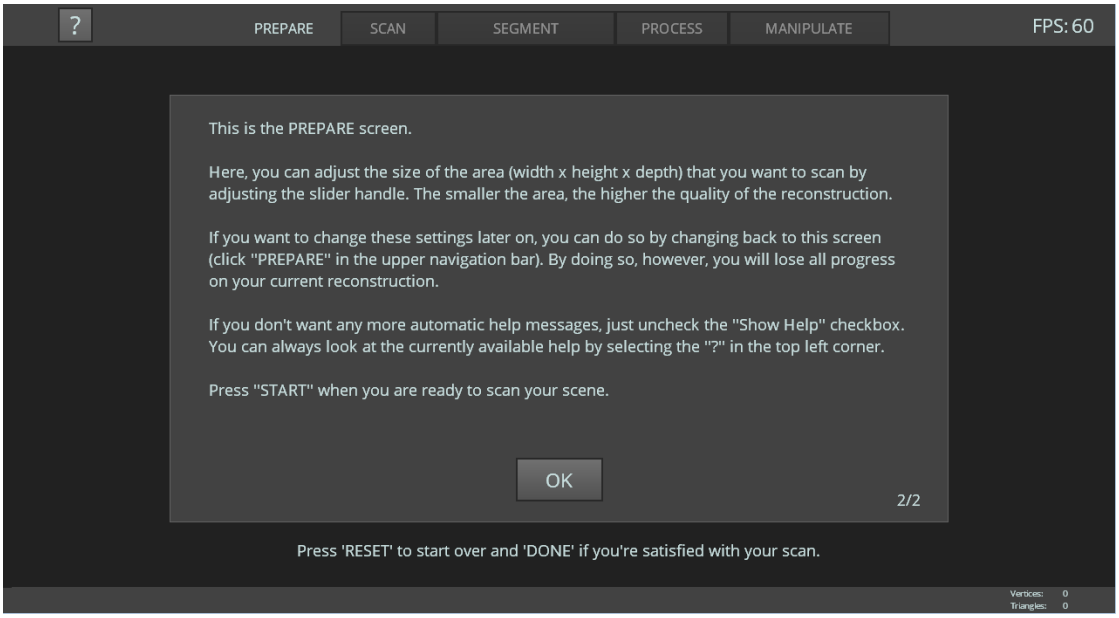


Figure 3.4: Guidance messages can be displayed to lead users through the application stages.

guidelines to achieve satisfactory results. These prerequisites to successfully use the application can be overwhelming, especially to first-time users.

In order to lower this threshold for novices and to provide tips for experienced users, detailed guidance messages can be displayed during each stage of the application. They provide a quick overview of the currently displayed interface elements as well as some usage hints. An example is shown in Figure 3.4.

The messages can be displayed by pressing the small button with the question mark in the upper left corner. This opens the help window for the currently active stage. A tutorial mode can be activated as well by checking the *Show Help* option during the preparation step (see Section 3.4). This leads to an automatic display of guidance messages about important topics at appropriate times, such as right before users advance from one stage to the next. As a result, users will be guided through the whole process of scanning, reconstructing, segmenting and manipulating their specific scene.

3.4 Preparation

The preparation screen is the starting point of the developed application. It enables users to adjust scanning parameters as well as general options. In addition, it prepares users for the scanning procedure by showing hints about optimized physical movement as well as specific interface elements. Figure 3.5 shows the screen during this application step.

First and foremost, users can use the central slider element to adjust the size of the scene to be scanned. This will directly effect the dimensions of the reconstruction volume

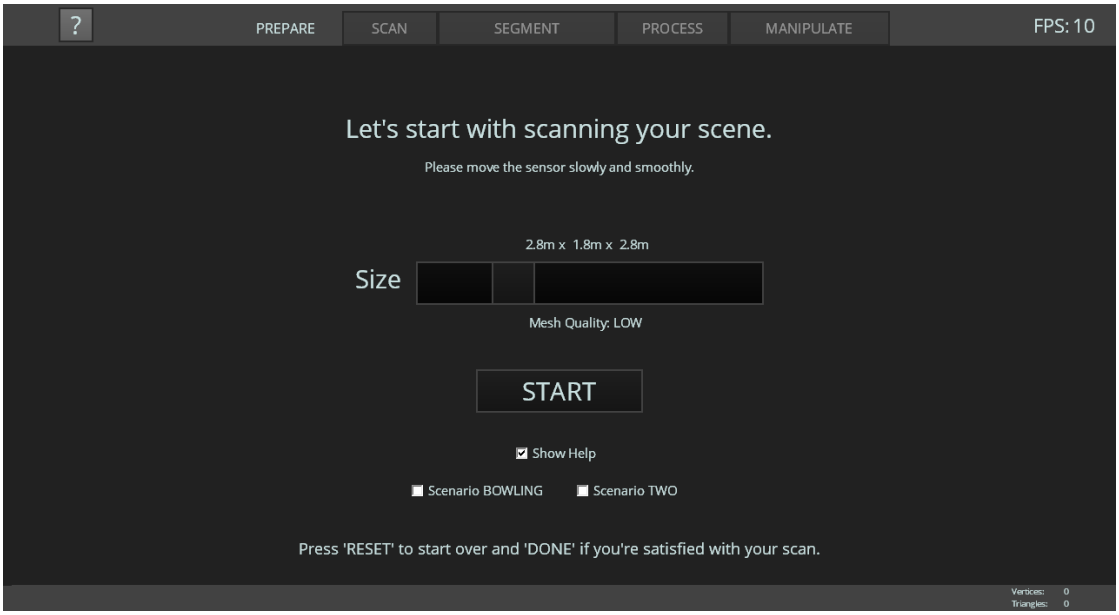


Figure 3.5: Application interface during the preparation step.

during the scanning procedure. Everything outside this predefined area will be completely ignored by the depth camera. The position of the bounding box with the specified size is determined by the position of the Microsoft Kinect at the exact moment the scan starts. The area starts 35 centimeters in front of the Kinect in viewing direction and ends after the chosen depth. The initial camera position also determines the center of the scanning area with the specified height and width. The positioning of this bounding box relative to the handheld device is illustrated in Figure 3.6.

The absolute number of voxels in KinectFusion‘s TSDF reconstruction volume (see Section 2.2.3) is fixed to 512 x 384 x 512 (width x height x depth). Only the number of voxels per meter (VPM), which directly influences the dimensions of the volume, can be changed by adjusting the slider handle. The fixed voxel count has two implications:

1. The proportions of the reconstruction area never change, with the width and depth at exactly the same, and the height at $\frac{3}{4}$ of that size. The slightly reduced height has been chosen as a result of most scenes having more relevant detail in the direction of the x- and z-axis than y-axis, if an ideal starting scanning position with a viewing direction parallel to the ground is being assumed.

2. The specified size of the reconstruction volume is inversely proportional to its resolution. As the desired scanning area increases, the voxel count per meter decreases, which directly affects the quality of the reconstructed mesh. On the other hand, users can choose a very small volume to create a very high quality reconstruction. The following examples illustrate this implication: A user chooses a volume size of $4m$ x $3m$ x $4m$. As a result of the absolute voxels being fixed at

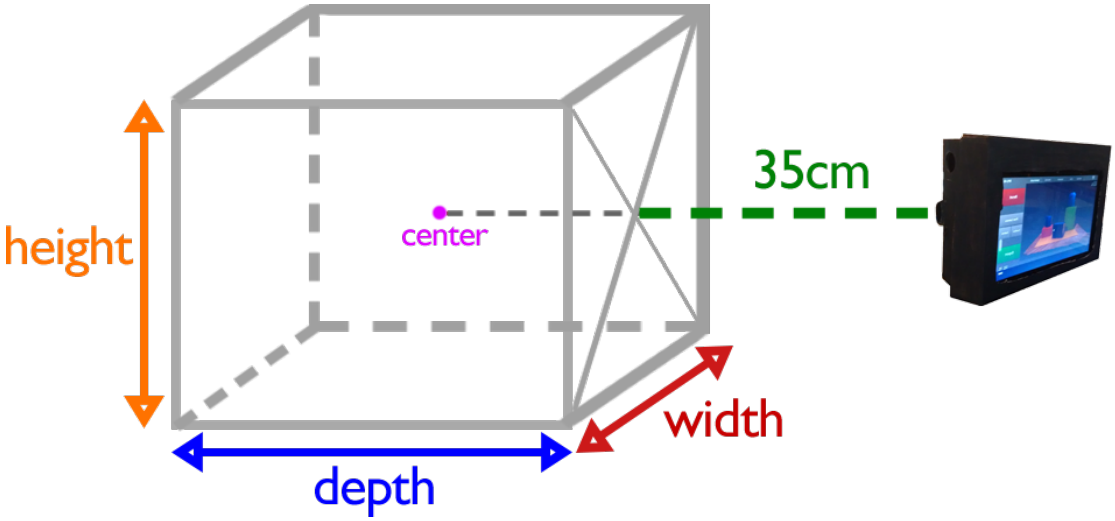


Figure 3.6: Positioning of the reconstruction volume relative to the depth camera.

512 x 384 x 512, the required VPM for this scenario is 128, since $\frac{512}{4} = 128$ in x-, $\frac{384}{3} = 128$ in y- and $\frac{512}{4} = 128$ in z-direction. In the same manner, a volume size of $1m$ x $0.75m$ x $1m$ means a VPM of 512 and therefore a 4-times increase in overall mesh quality.

This system has been chosen to allow adjustments to the volume size according to their specific needs without having a significant drop in performance. A voxel count beyond these fixed values results in either a greatly reduced number of frames per second during the scanning procedure or in premature termination due to the GPU being unable to handle the high memory requirements (see Section 2.2.3 for details). The minimum allowed volume size is $1m$ x $0.75m$ x $1m$ and the maximum $7m$ x $5.2m$ x $7m$. Dimensions beyond the chosen limit would result in very poor reconstruction quality.

A text label right below the slider control gives users a general indication of the mesh quality resulting from their chosen volume size. The three possible values correspond to the following VPM ranges:

- **Low:** $<$ 200 VPM
- **Medium:** 200 - 300 VPM
- **High:** $>$ 300 VPM

Users can also indicate whether or not they wish to see tutorial messages (see Section 3.3.2 for more details) during each application step by (un-)checking the corresponding check box. A click on *START* initiates the scanning procedure.

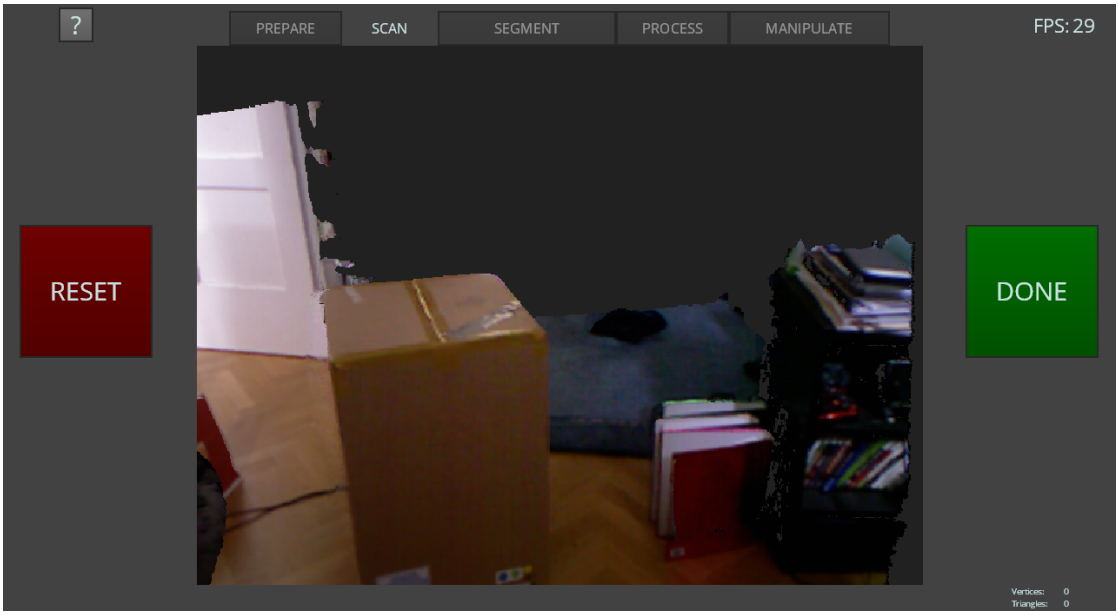


Figure 3.7: Application interface during the scanning step.

3.5 Scanning

During this step, users are able to scan their scene with the parameters specified in the preparation stage. After a successful scan, the scene is digitally reconstructed as well as automatically cleaned using various filters. The user interface during this procedure is shown in Figure 3.7. The necessary steps for scene capture as well as subsequent 3D reconstruction are explained in more detail in the following sections.

3.5.1 Scene Capture

Before the scan starts, a three-second countdown gives users time to get into the position that best aligns the bounding box of the reconstruction volume with the boundaries of the scene to be scanned. Ideally, the tablet with the attached depth camera should be about 50 centimeters away from the scene (or object) with a viewing direction parallel to the floor. If users do not succeed at first or the scanning procedure encounters alignment problems, the process can be restarted by clicking *RESET*.

During the scanning procedure, users have to slowly move around the scene and capture it from as many different angles as possible. A live preview of the reconstructed scene is shown directly on the tablet's screen at all times, which makes it significantly easier to identify remaining holes and gradually fill them by scanning objects from different angles. Due to the Microsoft Kinect being directly attached to the back of the tablet, the reconstruction preview is always from the user's perspective. Some general guidelines should be considered in order to perform a quality scan:

- The camera should be moved slowly and smoothly around the scene. No abrupt changes in the camera's orientation or position should occur, as the alignment algorithm used for reconstruction (see Section 2.2.2) assumes very small movements between each frame to allow for real-time reconstruction.

- A distance of at least 35 centimeters between the Kinect and solid objects should be kept, as everything closer than that can not be recognized and only disrupts the ongoing scan.

- In case of alignment errors due to sudden movement or other emerging problems, the application might fail to continue reconstruction due to losing the user's current position. Realignment can be attempted by capturing the scene from a perspective that has previously been scanned successfully. The algorithm will continue as soon as it is able to rediscover the camera's position.

- Should the scan fail due to unsatisfying results or unrecoverable camera tracking, users are recommended to restart the reconstruction from the beginning.

Since the Microsoft Kinect produces depth images at 30 frames per second, the scanning procedure is highly reliant on working at a frame rate as close to 30 as possible. This allows for a high success rate during point cloud alignment between frames. The specific maximum voxel count in all dimensions, as introduced in Section 3.4, has been chosen to allow a high enough frame rate at any desired volume size while running the application on the PC used during this research project (see Section 3.1.1). On computers with lower hardware specifications, the voxel count might have to be adjusted to allow a high enough frame rate for successful scans. When users are satisfied with their reconstruction, the capturing process can be finished by pressing *DONE*.

3.5.2 Reconstruction

As soon as the scanning procedure is finished, the application takes the acquired reconstruction volume and converts it into a mesh, consisting of vertices and triangles, with accurate surface geometry (see Section 2.2.4 for details).

The resulting mesh generally has a very high vertex density and a moderate amount of noise due to the nature of the surface reconstruction method and Microsoft Kinect's way of recognizing depth information. Since the extremely high amount of vertices and triangles has a direct impact on performance during subsequent stages of the application workflow, all vertices that are closer than 0.5 millimeters are automatically merged in a first filtering step. This reduces the vertex and triangle count by a significant number without having any visually noticeable effect on the mesh's quality. For example, a mesh with 1.000.000 vertices and 100.000 triangles can typically be reduced to about 150.000 vertices and 20.000 triangles.

To further improve the quality and smoothness of the reconstructed geometry, Laplacian smoothing is performed [18]. This filter choses a new position for each vertex based

on its neighboring vertices, which results in a much smoother vertex distribution and surface geometry.

Furthermore, every connected component (i.e. cluster composed only of connected triangles) that consists of less than $\frac{1}{100}$ of the mesh's total amount of vertices is removed. This step greatly reduces noise in the reconstructed model and simplifies subsequent segmentation procedures without removing objects of interest, since they generally have a higher relative vertex count.

After these initial smoothing and noise reduction filters have been applied, the mesh is cleaned up by removing all duplicate, unreferenced or non-manifold vertices and triangles as well as degenerate faces (i.e. faces that consist of less than three vertices). If any vertex is missing or has deficient normal vector information, it is recalculated by using the cross product of vectors between the vertex and neighboring vertices.

All these steps are performed automatically, but can take between a few seconds and one minute, depending on the chosen volume size and the density of objects in the reconstructed scene. As soon as the initial mesh reconstruction is finished, users can inspect their reconstructed mesh with an orbital camera while the remaining filters are being applied.

3.6 Segmentation

The reconstructed and cleaned virtual scene is now ready for segmentation. During this procedure, the initial model is split into submeshes that correspond to distinct objects in the real world. The methods used in this system are primarily designed to segment relatively large items that are placed on a planar surface and have at least some spatial distance between each other. Even though the segmentation of directly adjoining small objects is possible by choosing the right method and parameters as well as by utilizing the refinement tools provided by the prototype, it is not the primary focus and requires significantly more effort.

This part of the application workflow comprises three successive steps: Recognition and segmentation of planar surfaces, automatic clustering of remaining objects and refinement by utilizing a plane cut technique. Each of these steps is explained in detail in the next sections.

3.6.1 Plane Segmentation

In order to properly distinguish between separate objects in a scene, the application first has to recognize static parts of the environment that are not going to be part of any of these objects. Most of the time, these include a ground plane - either the actual floor of a room or the surface of a table -, ceiling and/or walls. As soon as all of these elements are removed from the mesh, components that are not part of the same physical object in reality are disconnected in the virtual reconstruction as well and can be distinguished from one another.

In human-made environments, all the mentioned static segments tend to have a more or less planar geometry. Even more so if only one of their surfaces is visible, as is often the case in models retrieved through 3D scans. As a result, recognizing and segmenting planes in the mesh can be a very powerful preliminary step to subsequent cluster extraction.

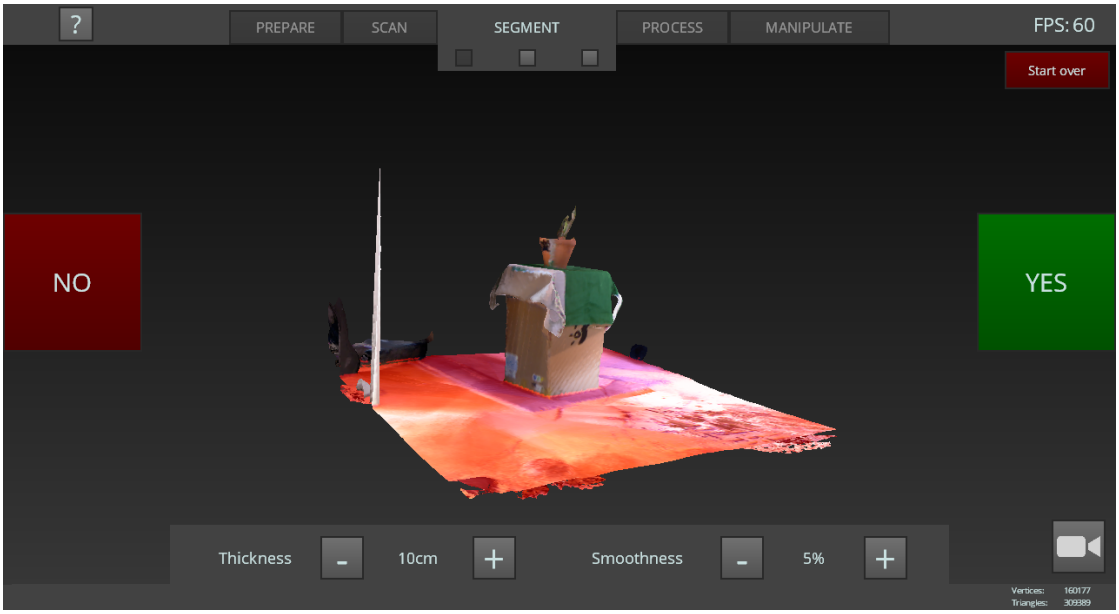


Figure 3.8: Application interface during the plane segmentation step.

Models obtained through 3D scanning tend to differ greatly from one another in terms of amount, variety and size of planar surfaces present. The sought-after planes might also have a different thickness depending on the scale, chosen dimensions and/or remaining noise in the scene. In addition, the final segmentation intended by the user can not always be easily deducted from the scene itself. As an example, the surface of a table can either be a static ground plane or part of the distinct physical object *table*, depending on whether the user needs the table to be segmented and movable as a whole or is just interested in the objects that are placed on its surface. As a result, the proposed system incorporates user input into its plane segmentation method. The user interface during this stage can be seen in Figure 3.8.

The application starts this procedure by utilizing the RANSAC plane fitting method discussed in Section 2.3.1 in order to find the biggest plane parallel to the initial viewing direction of the Microsoft Kinect during the scanning step. Assuming an ideal initial camera pose, as discussed in Section 3.5.1, these requirements lead to the highest possibility of finding the ground plane, which should be present in almost every scanned scene and whose segmentation is of utmost importance to subsequent clustering steps.

After finding all planes parallel to the viewing direction, the system looks for possible fits parallel to the other two axes to search for walls. The whole procedure is then

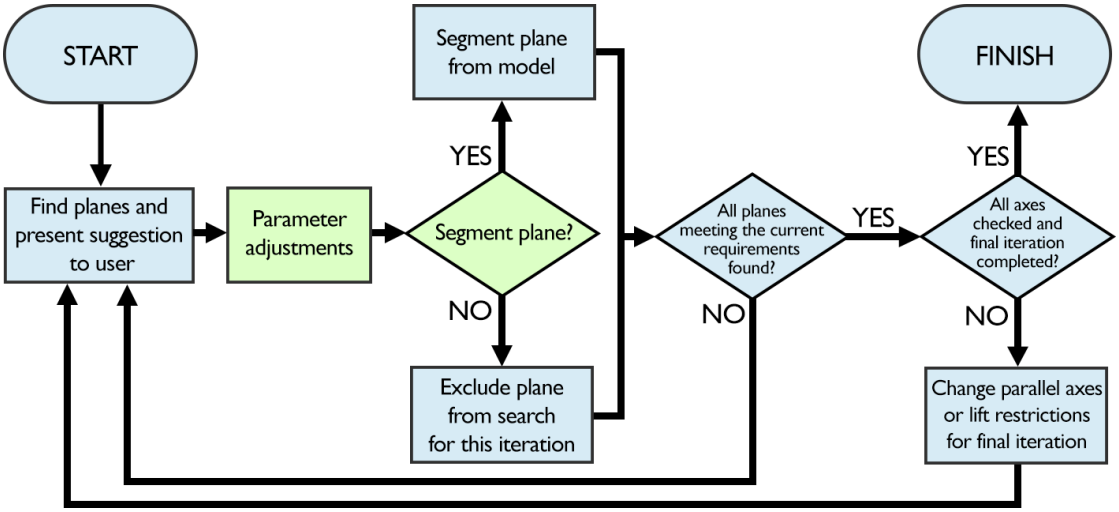


Figure 3.9: Workflow during the plane segmentation stage. Blue steps are performed automatically, while green steps require user input.

repeated without the restriction on plane orientation for all vertices that have not yet been segmented. Only planes that consist of at least $\frac{1}{15}$ of the total amount of vertices are considered. This is due to the application only being interested in the biggest planar surfaces that connect distinct components in the reconstructed model. In addition, users should not be presented with too many false positives during this step.

If a plane is found, it is highlighted in the virtual reconstruction of the scene. This acts as a suggestion to the user, who now has the possibility of inspecting the highlighted area by using the provided orbital camera. If the area is too thick (i.e. falsely includes objects that should not be part of the ground plane) or too thin (i.e. does not incorporate the whole ground plane), users can adjust the *thickness* of the recognized plane. Users can also change the maximum angle allowed between the normal vectors of two adjacent vertices so that they are still considered to be part of the same plane by de- or increasing the *smoothness*. Parameter adjustments result in an updated segmentation preview, which enables users to immediately evaluate their choices.

If a suggested plane should be considered a static part of the scene, users can confirm the segmentation by clicking the green *YES* button. The highlighted area is then classified as a static surface and the application tries to find other planes in the remaining scene, which are then suggested in a similar manner. If the suggestion is rejected by pressing the red *NO* button, the application simply searches for other planes without performing any segmentation. The whole process can be started over at any point by pressing the corresponding dark red *Start over* button in the top right corner. The workflow during this procedure is illustrated in Figure 3.9.

As soon as no more planes that have not yet been rejected or segmented meet the current requirements, the plane segmentation procedure finishes. The result should be a model of the reconstructed scene where each static planar surface has been recognized

as such and can therefore be ignored during the subsequent segmentation of distinct movable objects. It is possible for users to not select any plane during this step. However, this makes it difficult to achieve a satisfactory result in the object segmentation stage, as discussed in the following section.

3.6.2 Object Segmentation

After static planar surfaces have been identified in the previous step, the remaining model can be segmented into distinct clusters. The application uses one of two methods for this procedure, depending on user input. The first choice is Euclidean Cluster Extraction, which is a segmentation method that divides a model into clusters using only the spatial distance between adjacent vertices. The algorithm is described in more detail in Section 2.3.2. The second possible choice is Region Growing Segmentation, which, starting from a set of seed points, assigns neighbors to regions based on their normal vectors and certain smoothness constraints. More information can be found in Section 2.3.3.

As soon as the application switches to this step of the segmentation pipeline, initial clustering is performed using Euclidean Cluster Extraction with default parameters. In most cases, this initial segmentation produces sufficiently good and robust results. Note, however, that this method will only assign objects to different clusters if they are not connected to each other. That means that if static parts of the scene such as the floor have not been recognized and segmented during plane segmentation (see Section 3.6.1) or two or more movable objects are, for example, placed on top of each other, the Euclidean

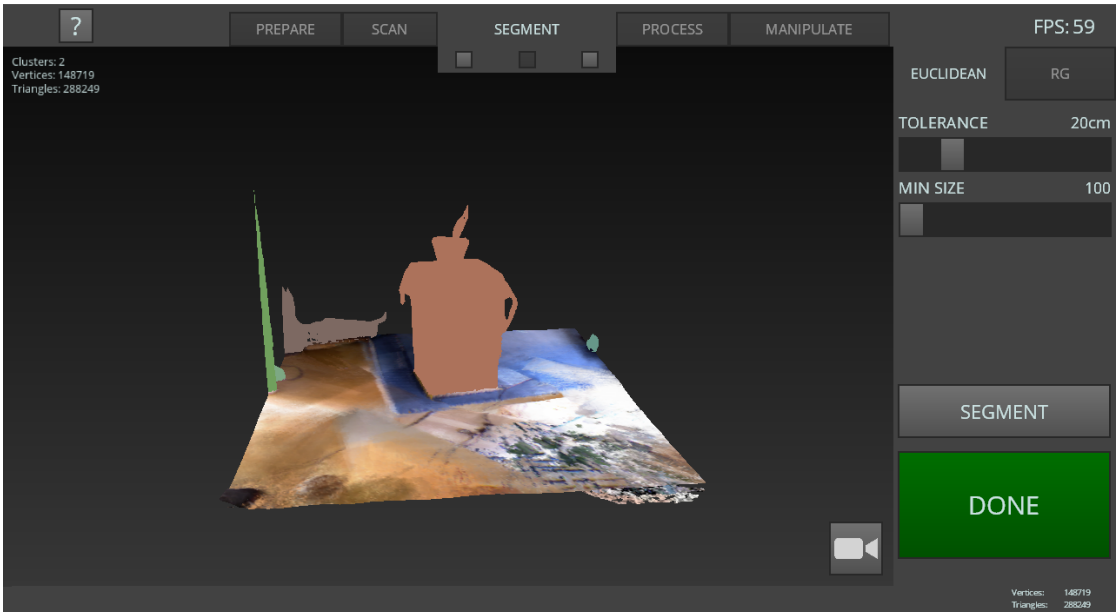


Figure 3.10: Application interface during the object segmentation step.

Segmentation will assign multiple objects to the same cluster. In case of a completely missing plane segmentation, this technique might put the whole model into a single cluster. Many of these problems can be solved during the subsequent refinement step (see Section 3.6.3), but sometimes Region Growing Segmentation with appropriate parameters might be a better choice.

Therefore, and due to varying scale, noise and complexity of different scenes, users have the possibility to choose between the two methods by selecting the corresponding tab. In addition, certain parameters can be adjusted with slider controls in order to perform a segmentation that produces satisfying results for the user's specific reconstruction and requirements. The interface during this step is shown in Figure 3.10. The following parameters can be changed, depending on the selected segmentation method:

Euclidean Cluster Extraction

- **Tolerance:** Two vertices are considered to be in the same cluster only if their Euclidean distance in centimeters is smaller than this value.
- **Min Size:** If a segmented cluster consists of less vertices than specified here, it is discarded.

Region Growing Segmentation

- **Smoothness:** Maximum angle between the normal vectors of the current seed point and a neighboring vertex, so that they are still considered to be in the same cluster.
- **Curvature:** Maximum curvature between the current seed point and a neighboring vertex in the same region, below which the latter will be used as a subsequent seed point.
- **Neighbors:** Number of closest neighbors stored for each vertex in the k-d tree representation constructed and used during the segmentation (see Section 2.3.2 for details about this specific data structure).
- **Min Size:** If a segmented cluster consists of less vertices than specified here, it is discarded.

Right after initial clustering and whenever users hit the *SEGMENT* button, the application presents a preview of the currently selected segmentation method with the chosen parameters. It shows the complete virtual reconstruction, where each resulting cluster has a different color. The colors are chosen in a way that tries to enable proper distinguishability. Static planes that have already been segmented during the previous step as well as submeshes that are too small (according to the specified *Min Size* parameter) keep their natural texture and are not color-marked. Figure 3.10 shows an example of this highlighting technique. In addition, the preview can be explored with an

orbital camera. This allows users to immediately inspect the consequences of different options and evaluate possible results.

When users are satisfied with the chosen segmentation method and parameters, they can confirm them by pressing *DONE*. As a result, the automatic clustering step is finalized and the selected segmentation applied to the reconstructed model. This procedure produces a number of duplicate vertices and triangles along the edges between distinct clusters. Therefore, basic cleaning filters similar to those used during reconstruction (see Section 3.5.2) are applied.

The Euclidean Cluster Extraction should be considered the preferred method in this step. Even though it tends to under-segment the reconstruction (i.e. it assigns multiple objects to the same clusters), the result can be improved by using the subsequent plane cut segmentation described in Section 3.6.3. Region Growing, on the other hand, generally over-segments objects (i.e. one object is divided into more than one cluster), which can not be corrected with the tools provided by this prototype. However, if the preliminary plane segmentation step could not be performed for any reason, Region Growing Segmentation with properly chosen parameters produces better results, as it is not dependent on spatial distance between objects. Region Growing is generally the better choice if the primary goal is the segmentation of individual item surfaces and not for resulting clusters to resemble connected real-world objects.

3.6.3 Plane Cut Segmentation

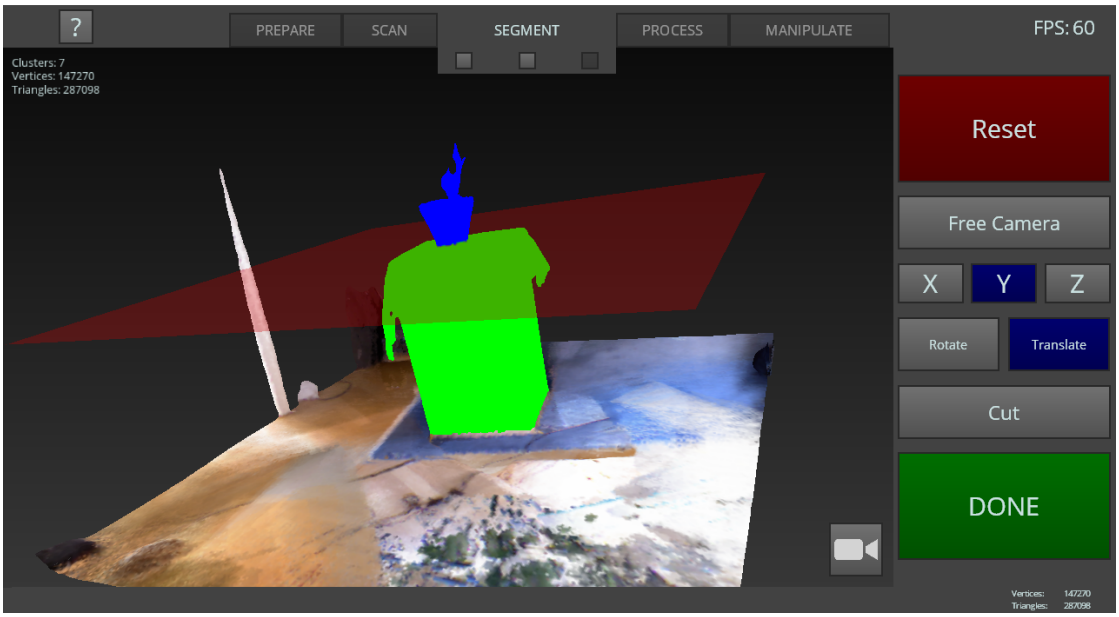


Figure 3.11: Application interface during the plane cut segmentation step.

If the submeshes produced by the automatic object segmentation are under-segmented, as is often the case when using Euclidean Cluster Extraction, users sometimes need

to be able to refine the result by further splitting the segments. For this purpose, the application provides an optional, intermediate step, during which users can cut existing segments into two respective submeshes along a manually placed plane. The interface during this procedure is shown in Figure 3.11.

This technique has been chosen as an additional refinement step due to its high usefulness and good usability, while being relatively easy to implement. It is an optional step and might not be needed in a majority of cases due to the automated object segmentation (see Section 3.6.2) already producing satisfying results. There are a lot of cases where this method can not be used to improve an under-segmented scene. For example, whenever two distinct objects are in the same cluster and can not be properly split along one (or more) planes, this additional segmentation step is not applicable. Despite these shortcomings, the plane cut method has still been included in the prototype, since it provides users with additional options if the primary segmentation algorithms fail to provide sufficiently good results.

The plane used during this step spans across the entire reconstruction volume and is semi-transparent, so that users can see exactly where it is placed in relation to the rest of the model, regardless of the current perspective. If a ground plane, i.e. a plane with sufficient size parallel to the initial viewing direction of the user, has been found during plane segmentation (see Section 3.6.1), the semi-transparent plane is aligned with that surface. If no planar surface has been found, however, it is aligned to be perpendicular to the y-axis of the reconstruction volume. In any case, users have the possibility to realign the semi-transparent plane to be perpendicular to either of the three axes by pressing the corresponding button.

If users click on a specific point on the reconstructed model while the *Translate* mode is selected, the plane will be shifted to that position, while still staying perpendicular to the selected axis. By dragging the plane in a direction, it can be moved along this axis. Clicking on a part of the mesh will also select that specific cluster as the one to be cut. Similar to the preview during the automated object segmentation as described in Section 3.6.2, the selected segment is color-marked, where blue and green colors represent the two submeshes that would be created by cutting along the plane in its current position.

Users can rotate the plane along each of the axes that are not currently selected by changing to the *Rotate* mode and dragging the plane in the chosen direction. The rotation is limited to an angle of 30 degrees, since the controls for this transformation become unintuitive after this point due to rotating in a direction opposite to user input. However, orientations beyond 30 degrees can easily be performed by switching to another perpendicular axis.

Note that the orbital camera used for navigating the scene is disabled during plane transformation, since the drag gesture cannot be used for both camera and rotation as well as translation. Users can, however, switch to *Free Camera* mode, which allows them to change their perspective with the same orbital camera that was used during previous application steps. In this mode, plane repositioning is deactivated. After switching back to transforming the semi-transparent plane, all rotations will be performed relative to the

current perspective of the virtual camera to allow for more intuitive and refined controls. The view can be reset by clicking on the camera button in the lower right corner.

A cut on the selected cluster along the currently positioned plane can be performed by clicking on *Cut*. The application will immediately update the reconstruction model to reflect the new changes. Users can perform as many plane cuts as needed in order to achieve a mesh segmentation according to their requirements. The current progress can be reset by pressing the corresponding button. This will, however, revert the mesh to the state before any cut was performed. The segmentation procedure can be finished by pressing *DONE*.

3.7 Processing

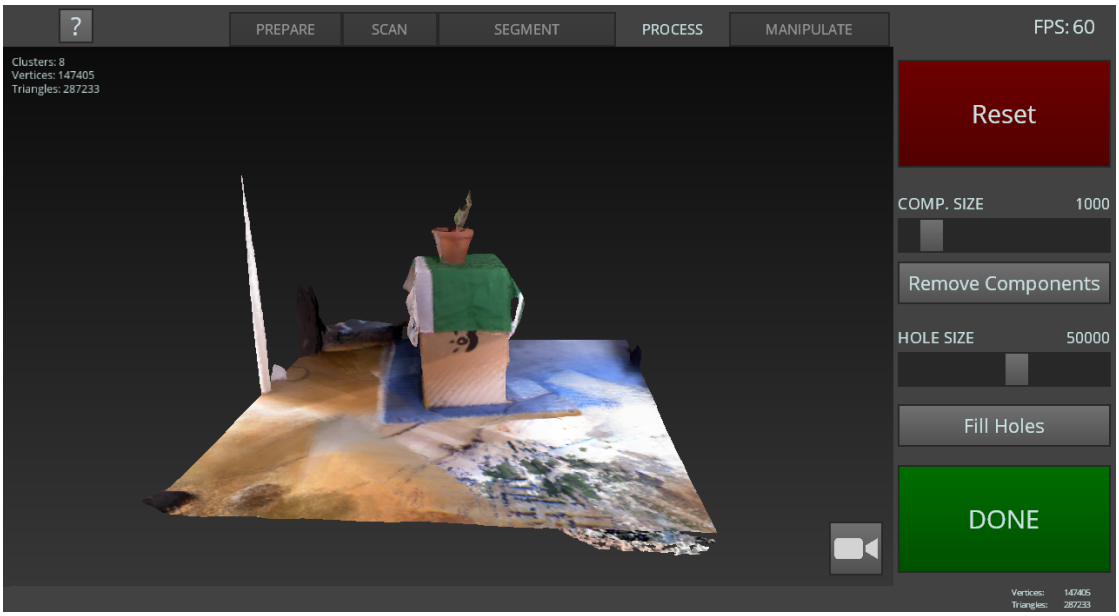


Figure 3.12: Application interface during the processing step.

After successfully completing the previous stages of the prototype, the virtual scene reconstruction should be properly segmented. The model is ready to be cleaned and processed one final time before starting scene manipulation. Even though several cleaning filters are performed automatically during and after each segmentation step, the processing stage allows users to perform two additional methods to potentially improve their mesh with individually chosen parameters. This manual processing step has been provided to allow users to improve or repair certain parts of their reconstruction according to their specific needs. Users can reset their progress during this stage at any time by pressing *Reset*. The two available methods are explained in the following sections. Figure 3.12 shows the user interface during this step.

3.7.1 Small Component Removal

Even though several application stages already remove many unnecessary components automatically, they only do so in order to reduce noise through elimination of outliers. However, the different clustering methods - Region Growing Segmentation (see Section 3.6.2) and plane cut refinement (see Section 3.6.3) in particular - often produce very small clusters that do not resemble any physical object or are not needed in the model for some other reason, but are still too big to be automatically removed. Therefore, users have the possibility to manually remove all connected components that are smaller than a user-provided threshold. This value can be controlled by adjusting a slider control and represents the minimum number of vertices needed for each cluster to remain in the scene. By clicking on *Remove Components*, users can apply the filter to the whole model and the visual representation is updated accordingly.

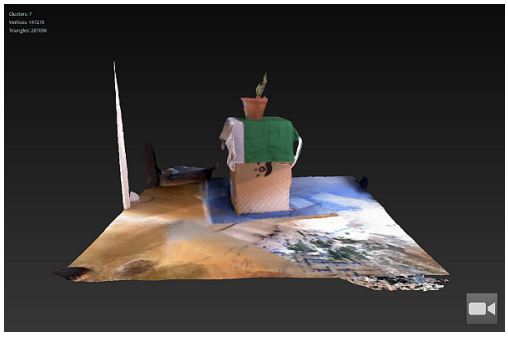
3.7.2 Hole Filling

As discussed in Section 2.2.3, the capturing procedure stores surfaces of the scene by continuously filling a TSDF volume with aligned depth data retrieved from the Microsoft Kinect. The camera can only perceive the depth of the closest surface and does not know anything about occluded objects behind them. Even if users scan the scene from every possible angle, there are certain surfaces of the scene that will remain occluded. Examples include parts of the floor other objects are standing on, the section of the wall directly behind a small cabinet or even the side of an individual object that is directly adjacent to another one.

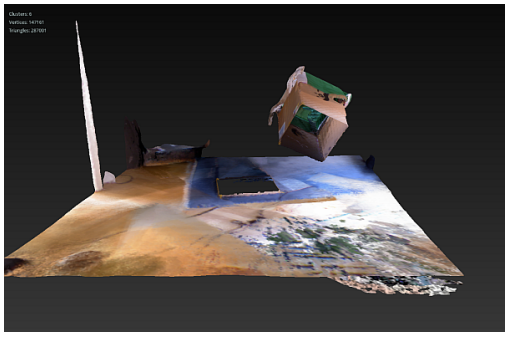
These missing surfaces do not pose a problem to unmodified scanned models, as they could not be seen even if present. They can, however, become an apparent flaw if the reconstruction is segmented into distinct objects that are subsequently being moved around. In that case, the missing surfaces will be visible as holes, as illustrated in Figures 3.13a and 3.13b. To fix a majority of these problems, the application offers the possibility to automatically fill holes in specific parts of the scene. Users can specify the maximum hole size (in number of edges composing the boundary) to be filled by adjusting the designated slider. If the intent is to fill every single gap, users should choose the maximum possible amount. By clicking *Fill Holes*, the filling algorithm is performed either on the currently selected cluster or, in the case of no selection, on the whole mesh.

The algorithm identifies holes by finding boundary edges, i.e. edges that are attached only to a single triangle, that form a closed loop. If the number of edges in the loop is smaller than the provided threshold, the gap is filled by creating new triangles. Their color is determined by interpolating between the colors of edge vertices. Figure 3.13c shows a 3D model after performing the described procedure.

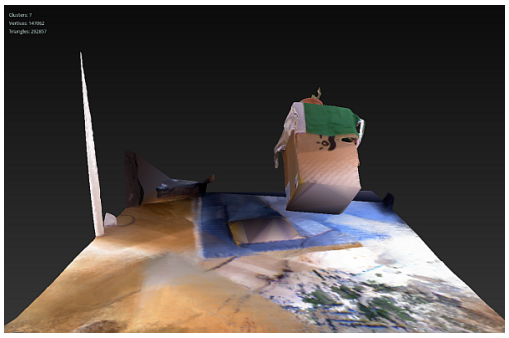
In short, the provided method fills holes by creating a planar surface between boundary edges. Therefore, it only provides good results if the missing surface is planar and does not include complex geometry. Fortunately, most of the holes resulting from scanning with a depth camera are planar. This also includes gaps that appear after segmenting the scene. For example, if an object standing on the floor is segmented, both the resulting



(a) Reconstructed model.



(b) Missing surfaces after segmentation.



(c) Reconstruction after closing gaps.

Figure 3.13: Utilization of provided hole filling method.

hole in the object as well as the one on the floor are planar. Exceptions would be physical objects that are only partly inside the reconstruction volume as well as those with intersecting surfaces. If the hole filling method is applied in these cases, the resulting surface will not be accurate. However, this method is only supposed to be a very simple tool for fixing gaps that were created during the application's segmentation stages and should not be used to substitute complex missing object parts.

3.8 Manipulation

Now that the scene has been successfully reconstructed, segmented into static planes and distinct movable objects as well as processed according to the specific user's needs, it is ready to be manipulated. During this step, users are able to freely move around in the scene by physically moving in the real world while carrying the tablet. They can use the touch display to interact with reconstructed objects and modify them in different ways. The interface during scene manipulation is shown in Figure 3.14. The different options in this phase are explained in more detail in the following sections.

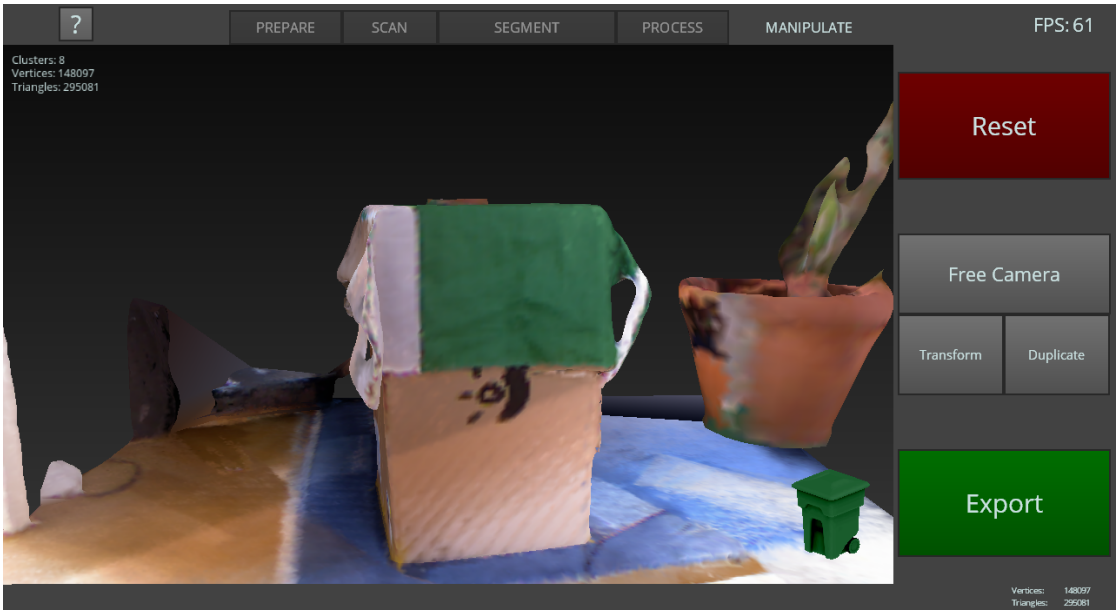


Figure 3.14: Application interface during the manipulation step.

3.8.1 Navigation

With the exception of the scanning step, all previous stages of the application offered scene navigation in the form of an orbital camera, which could be controlled by using different touch gestures on the tablet. Users could translate, rotate and scale the camera's perspective arbitrarily. This resembles the classic approach to scene navigation, as used by many available mesh viewer applications. In the manipulation stage, however, the developed prototype takes a different approach.

The KinectFusion algorithm (see Section 2.2), which is used during the application's scanning part, compares the newly acquired depth data with the already captured reconstruction volume to estimate a user's position and orientation relative to the scene in each frame. Using this estimation, the new depth data is then aligned with and incorporated into a single model. Essentially, this part of the algorithm changes the perspective of a virtual camera to be the same as the user's.

At the current stage of the application, the scan has already been performed. However, the TSDF reconstruction volume (see Section 2.2.3 for more information) that has been filled during the scan is still available, and the Microsoft Kinect attached to the tablet can still capture depth data as well. Furthermore, it can be assumed that the real-world scene has not changed since the scan, because all subsequent modifications were purely virtual. As a result, the tracking part of the KinectFusion algorithm (as described in Section 2.2.2) can be utilized with newly acquired depth data in order to update the virtual camera's perspective to correspond exactly to that of the Microsoft Kinect, without actually adding the new depth data to the already finished reconstruction. This makes it possible for users to freely move around in a scanned scene by physically moving

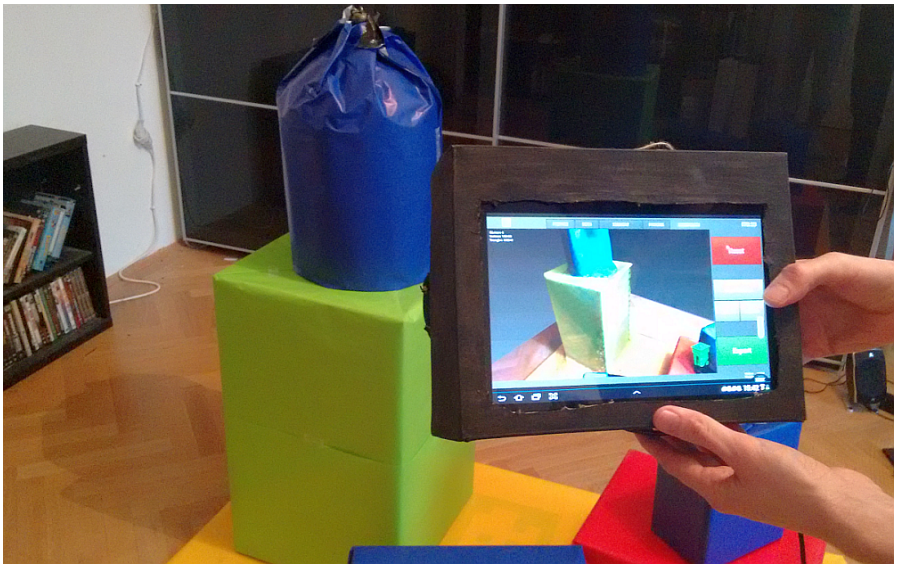


Figure 3.15: The tablet functions as a window into the virtual world.

in the real-world counterpart while holding the tablet with the attached depth camera. The orientation of the camera is updated to that of the Kinect every frame, which allows the tablet display to be used as a window into the virtual world. If users point the depth camera at a specific part of the real scene, they can see the corresponding part of the virtual reconstruction as shown in Figure 3.15.

By utilizing this method, users are presented with a natural and playful way to navigate through the reconstructed scene. They do not have to control an abstract orbital camera and can instead move and look around freely as well as inspect specific parts of the scene by walking to the corresponding section in the real scene and adjusting the display orientation. Moreover, this technique enables intuitive ways to reposition distinct objects by utilizing physical movement, as described in Section 3.8.2.

By using the camera tracking of KinectFusion's ICP implementation, this navigation technique also inherits its potential downsides. Changes to the real scene after a finished reconstruction can lead to tracking failures due to the newly acquired depth data not corresponding to the reconstruction volume obtained during scanning. In addition, for the pose estimation to work, the Kinect needs to recognize some part of the scene that has been reconstructed. As a result, positional and orientational tracking does not work if the camera is pointing, for example, at a ceiling that has not been previously scanned, even if the user is standing right inside the reconstructed scene. This technique also assumes that the obtained data was collected from all possible angles, which in return enables 360-degree navigation. If a scan only captured objects from a single perspective, tracking will not work while inspecting it from different angles during this step.

However, since this prototype is primarily designed to work on reconstructed static scenes that have been scanned with as much detail as possible, and users are assumed to mainly point the tablet in the direction of objects they want to modify, the implemented

navigation technique works well in a majority of cases. Furthermore, since the tracking algorithm operates on already captured scene data, it is not as sensitive to abrupt changes in perspective as it is during the scanning stage described in Section 3.5.1 and will recover from tracking failures without problems.

Users also have the possibility of inspecting their manipulated scene by switching to a free orbital camera with the click of a button. This gives them a current overview and allows them to quickly assess the situation. While doing so, however, interaction with objects as described in Section 3.8.2 is deactivated, since the used techniques are explicitly designed for the default navigation mode utilizing camera tracking.

3.8.2 Interaction

The prototype offers the possibility to interact with the reconstructed scene during this stage in different ways. The used interaction techniques enable users to reposition, duplicate and remove objects, while also providing basic transformation abilities in the form of rotation and scaling. This application stage is designed to offer users a playful and natural approach to restructuring their scene, while still being powerful enough to allow for a wide variety of modifications. All performed modifications can be reverted at any given time by pressing *Reset*. The different ways to manipulate objects are described in the following sections.

Repositioning

Making the repositioning of objects as natural as possible is one of the most important features for intuitive scene manipulation. The primary goal of this interaction technique is to allow users to relocate all segmented movable objects.

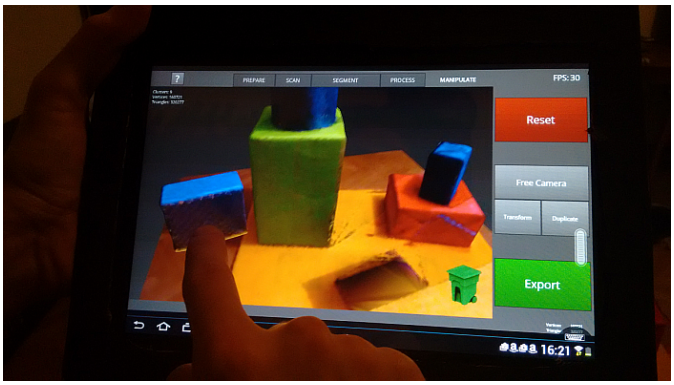


Figure 3.16: Objects can be repositioned with drag-and-drop gestures.

Repositioning is the default interaction mode during this stage and is active whenever a user has not specifically selected any other technique. When a segmented object can be seen from the user's perspective (and is therefore visible on the tablet's display), it can be selected by tapping it on the touchscreen. By pressing and holding, an object can

be picked up, i.e. it is attached to the front of the camera. The usage of this method is shown in Figure 3.16. In conjunction with the provided scene navigation technique, this allows users to carry objects around by dragging their fingers over the display and by physically moving in the real world, since the selected item remains attached to the camera while the perspective changes.

When users drag an object over another surface, including static as well as non-static parts of the scene, it is relocated to the spot currently pointed to. The exact position is determined by the normal vector orientation of the selected spot's triangle as well as the current viewing direction of the camera. For example, if a user picks up a box and drags it over the floor, it is placed on top of the floor. If the box is dragged over a wall or closet while standing in front of it, the box is positioned in front of it as well. This simulates the positioning of objects in the real world and makes it easier to rearrange them in a way that would also make sense in reality. For example, objects can be picked up, carried around, and dropped somewhere, without having to precisely position them so that their bottom edges align with the floor again, since the application automatically takes care of that.

The position of a picked up object is constantly updated as the selected spot under the finger changes, either by dragging or by adjusting the current perspective of the camera. When the finger is lifted, the object is dropped at the specified position. If it is released while being dragged over a part of the screen where no reconstructed objects can be seen, the object's position is reset to its location before pick-up.

Transformation

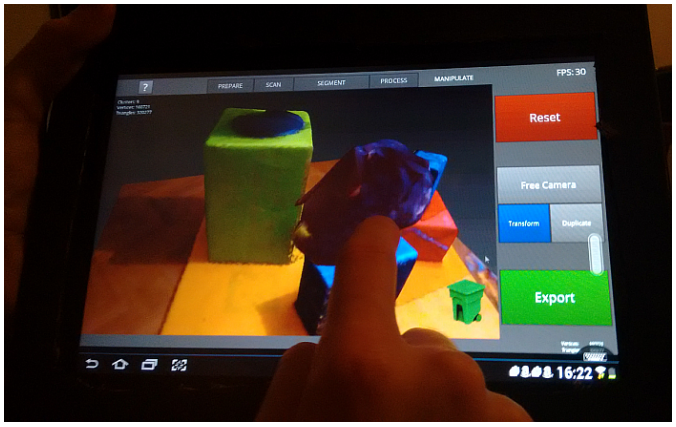


Figure 3.17: Object rotation during transformation mode.

Another important aspect of scene manipulation is the adjustment of an object's orientation and scale. Just as in the other interaction modes, only segmented movable objects can be transformed, identified static planar clusters stay immovable.

After pressing the *Transform* button, users can reorient an object by selecting it on the touchscreen and dragging the finger in the desired direction of rotation. The used

axes are relative to the current camera orientation at all times, i.e. the rotation of the object always follows the touch gesture and is not limited to the object's or world's axes. This method allows for arbitrary (combined) rotations around the x- and y-axis of the camera's viewing direction with a simple drag gesture. Therefore, any desired rotation can be performed by adjusting the current camera perspective accordingly. An example of object rotation is shown in 3.17.

Objects are always rotated in a way that makes sure that they remain on the surface they are attached to. For example, if a non-cubic box placed on the floor is rotated by 90 degrees so that its height, as measured from the ground plane, changes, the system makes sure that the box is still directly attached to the floor's surface. These considerations ensure smooth and intuitive rotational adjustments while still utilizing the system's unique approach to scene navigation presented in Section 3.8.1.

Scaling an object works in a very similar fashion. Again, after selecting the *Transform* option, users can choose the visible object that they wish to scale by selecting it on the touchscreen. Without lifting the finger from the screen, they can now adjust the object's size by performing simple scroll gestures using two fingers. Scrolling up increases the scale, scrolling down decreases it. By releasing the fingers, the transformation is applied.

Duplication

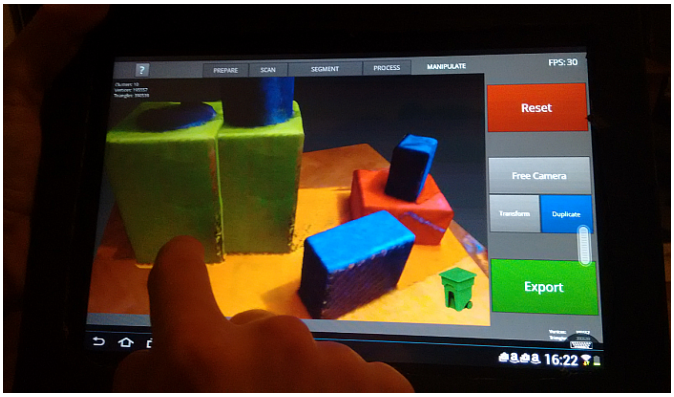


Figure 3.18: During duplication mode, copies can be dragged out of original objects.

In order to give users more possibilities in interacting with their reconstructed scene, the system allows for duplication of segmented objects. As in the other interaction modes, this operation can only be applied to movable clusters, not to static planar surfaces.

After users activate the *Duplication* mode, they can clone an object by selecting the original and dragging a duplicated version out of it. The newly created object is a copy of the original segment in every way and even inherits all performed transformations. It can then be carried around and repositioned in exactly the same way as described in Section 3.8.2. Clones can be independently transformed, repositioned, removed and even copied themselves.

Users can perform duplications as often as they like, even though they have to keep the increasing number of faces and vertices in mind, as this might impact performance when using the exported model in other applications. This interaction technique allows users to create complicated scenes with a great number of similar objects, while only needing to include a single one of them in their actual scan. An example of object duplication can be seen in Figure 3.18.

Deletion

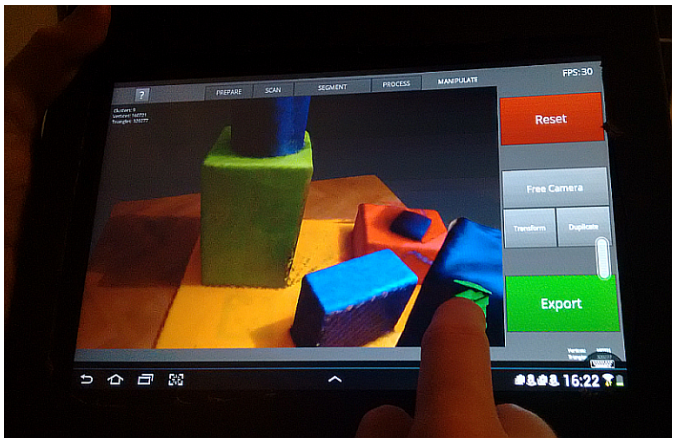


Figure 3.19: Objects can be removed by dragging them over the recycle bin.

If users want to eliminate a specific cluster, for example, if they have created too many duplicates, they can remove segments from the scene completely. To achieve this, the specific object has to be selected and dragged and dropped over the dustbin icon in the lower right corner of the scene view. The usage is illustrated in Figure 3.19.

If the scene manipulation is reset by pressing *Reset*, all removed objects that were part of the original scene are restored to their initial positions.

3.8.3 Export

Finally, as soon as users are satisfied with their reconstruction as well as their performed manipulations, they have the possibility to export the virtual scene for use in other applications, such as mesh viewers, game engines or 3D printing software.

Users can choose between exporting only a specific segment by selecting it or saving the complete reconstruction by not selecting anything. Either way, the *Export* button has to be pressed with the desired selection active, which will open a *Save As* dialog that lets users specify the name of the exported file. They can also choose between OBJ, OFF, PLY or STL file formats, which should cover enough options to accommodate different editing applications.

If the system has detected a ground surface during the plane segmentation step (see Section 3.6.1), it tries to rotate the exported model in a way that aligns the ground

plane with the x-axis. This compensates for tilted reconstructions that resulted from viewing directions at the start of scans that were not perfectly parallel to the ground. With the implemented alignment step, however, it is much easier to reuse the model in other applications as it can be imported with a more standardized orientation.

CHAPTER 4

Implementation

This chapter focuses on the technical implementation of the developed system. First, a short overview of the overall software architecture as well as used frameworks and libraries is given. Then, basic concepts and common methods used in the prototype are explained. Finally, each of the implemented components and their functions are discussed in more detail.

4.1 System Overview

The implemented prototype consists of many intertwined elements with distinct responsibilities. A rough overview of the most important components excluding libraries and other external dependencies is illustrated in Figure 4.1. The main system parts are described as follows:

- **Main Control:** The main entry point and responsible for the application's lifetime. Contains the main window and instantiates the different sub windows, depending on the current stage. The main application loop is also executed in this class. In addition, this component forwards communication from user interface controls and the scanning procedure to the graphics controller.

- **Graphics Control:** This component executes the OpenGL loop in a separate thread, which is responsible for almost all 3D-related operations. This includes rendering of the reconstructed scene, navigation, selection (including manipulation), segmentation and export. The different procedures are highly parallelized and run in several additional threads. Therefore, it is necessary to perform basic concurrency control in order to ensure that concurrent operations produce correct results and are executed in the right order.

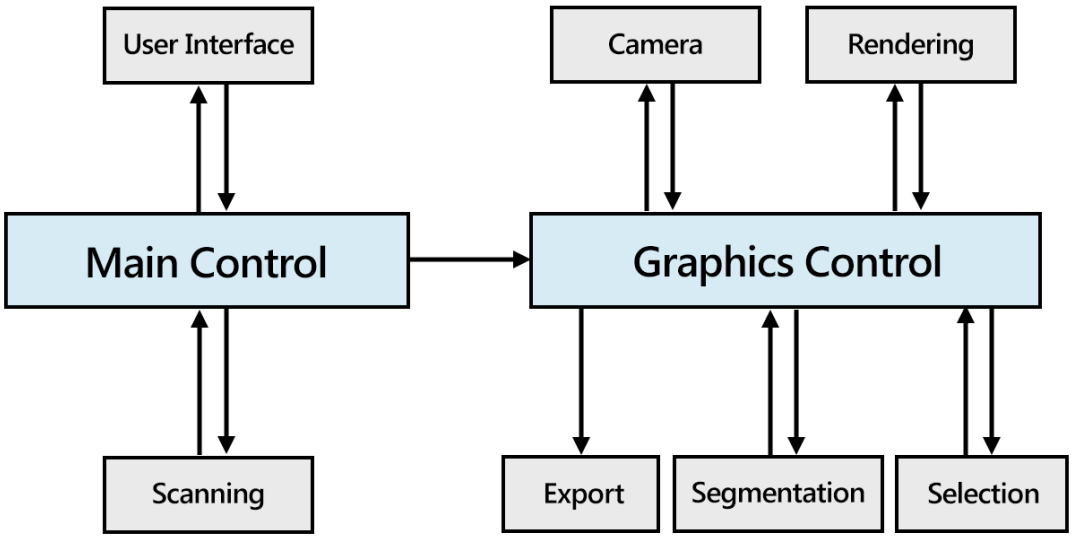


Figure 4.1: Rough system overview excluding libraries.

These two main controllers are responsible for managing a variety of smaller components (or component groups), the most important of which are shortly described as follows:

- **User Interface:** Contains the two-dimensional GUI overlay including controls such as buttons, sliders, labels and similar elements. Listens for user input and forwards it to the main controller class. It is divided into several subcomponents, each of which represents a specific panel on the screen. They are activated and deactivated depending on the currently selected mode, so not all of them are visible and listening for input at any given time.

- **Scanning:** Responsible for communicating with the active capturing device and retrieving the unfiltered reconstructed model. In addition, this component tracks the current position and orientation of the depth camera for use in 3D scene navigation.

- **Rendering:** Receives the current scene data (i.e. the 3D scene model and 2D icons) and renders all required primitives using OpenGL.

- **Camera:** Responsible for scene navigation. Depending on the currently active application stage, the camera can either be orbital or directly coupled to the current orientation and position of the scanning device.

- **Selection:** Handles 3D user interaction, including selection, manipulation and transformation of objects.
- **Segmentation:** Provides several techniques for plane and object clustering.
- **Export:** Used to save the finished scene reconstruction to one or more files in one of several different formats.

A more complete class diagram including most components with their specific inheritance hierarchy as well as most important associations and interactions, is shown in Figure 4.2.

4.2 Dependencies and Foundations

This section provides a quick overview of all libraries and frameworks that were used during this project. Furthermore, basic concepts utilized by two of these frameworks are described in more detail. Note that this is not supposed to be a comprehensive explanation of every function, but instead a short overview of some important recurring concepts that are being used in the application developed during the project. For more detailed information, refer to the official documentations by Microsoft [53] and Khronos [36].

4.2.1 Frameworks and Libraries

The different used libraries and frameworks as well as their specific purpose in this project are shortly described as follows:

- **Win32 API:** C-based framework that enables the development of Windows-based desktop applications. It provides a set of interfaces to several dynamic link libraries (DLL) that can be used to achieve a wide range of features. This prototype uses the Win32 API to display and process the application window and its 2D Graphical User Interface [53].
- **Open Graphics Library (OpenGL)**: Cross-platform API developed by Silicon Graphics that enables interaction with the Graphics Processing Unit (GPU) in order to render 2D and 3D graphics [35].
- **OpenGL Extension Wrangler Library (GLEW):** Helps with the initialization of OpenGL, including the determination of which versions are supported on the current platform [30].
- **OpenGL Mathematics (GLM):** C++ mathematics library for OpenGL software [24].
- **KinectFusionExplorer-D2D/NuiSensorChooser:** These C++ projects are included in the Microsoft Kinect for Windows Developer Toolkit 1.8.0 [48]. They are

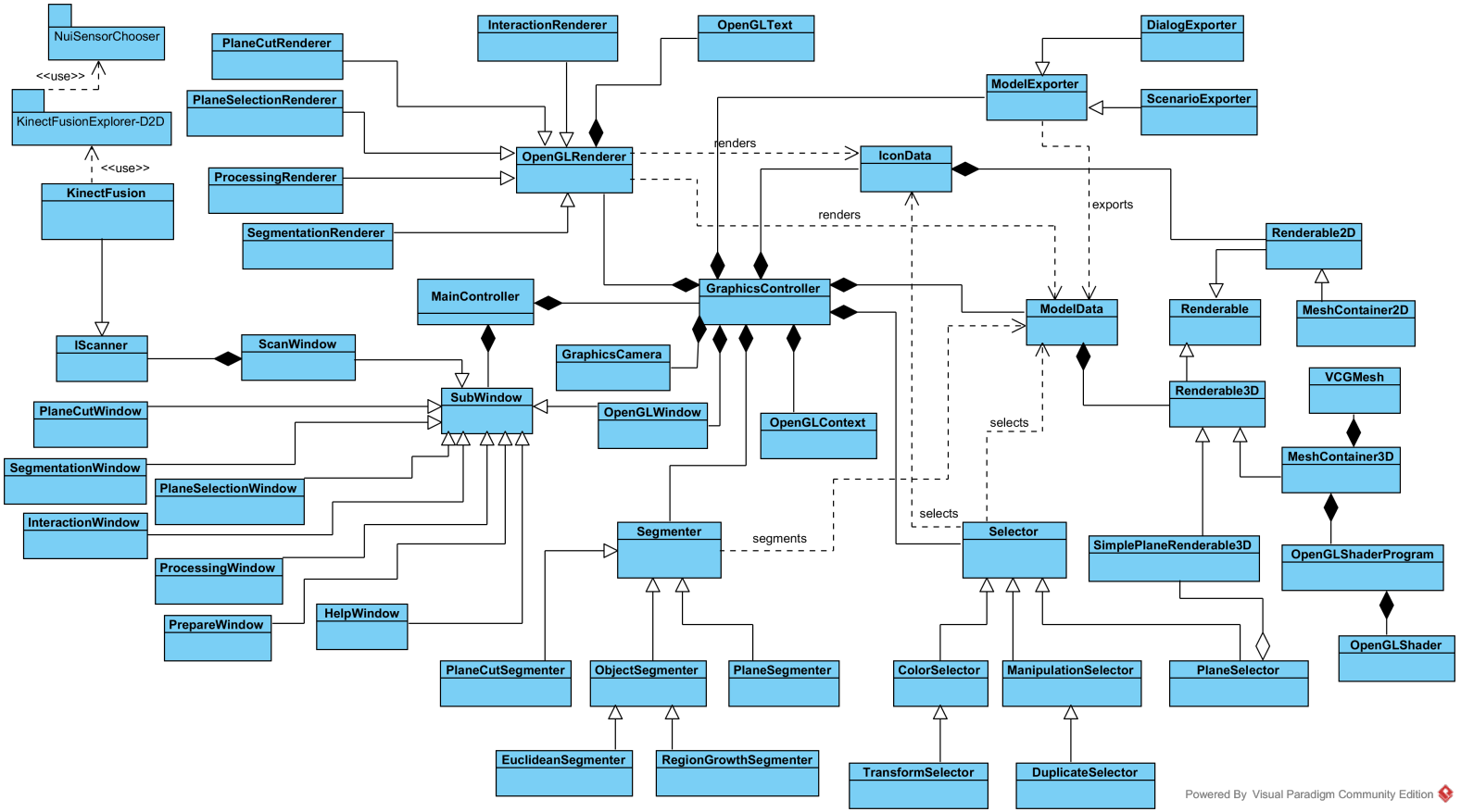


Figure 4.2: Class diagram containing the most important classes and inheritances. Created with Visual Paradigm CE 12.1 [81].

part of an example application capable of utilizing KinectFusion to scan a three-dimensional scene. This prototype uses several modified parts of these projects, including the KinectFusion implementation itself and the communication with the Microsoft Kinect [48].

- **Boost C++ Libraries:** A large compilation of peer-reviewed portable C++ source libraries that can be used for a vast variety of tasks [15]. In this prototype, Boost provides advanced thread management, logging and a robust implementation of smart pointers.
- **Visual and Computer Graphics (VCG) Library:** Open-source C++ library that offers a variety of efficient methods for processing triangular meshes as used in their widely known open-source tool MeshLab [13]. The implemented prototype uses it for several mesh filtering procedures such as vertex merging, hole filling and surface smoothing.
- **Point Cloud Library (PCL):** Open-source library that provides many different algorithms for processing of three-dimensional point clouds and 3D geometry in general [71]. The prototype developed during this thesis uses its implementations of RANSAC Model Fitting (see Section 2.3.1), Euclidean Cluster Extraction (see Section 2.3.2) and Region Growing Segmentation (see Section 2.3.3) during the segmentation stage.
- **FreeType:** Used to read font files and render text on screen [38]. This project utilizes FreeType in conjunction with OpenGL to display informational text on top of the three-dimensional scene.

4.2.2 Win32 API

The *Win32 API* is a set of programming interfaces used to create C and C++ applications for Microsoft Windows [53]. It provides core functionalities such as access to network capabilities, system processes and registry, error handling, thread management and DirectX rendering. The prototype developed during this research project mainly utilizes the *user interface* subsystem of the Win32 API, which provides ways to create windows and different basic controls (i.e. buttons, sliders, labels) and allows proper handling of user input [53]. The following sections describe some relevant Win32 concepts with examples taken from this prototype's implementation.

Window Class

The `Window` class is used to define a reusable type of window and includes information about visual elements such as the background color, application icon and cursor image. In addition, the *window procedure*, which is responsible for handling all relevant events, can be specified (see Section 4.2.2 for more information). Several `Window` classes can be created in an application and different windows can have the same class. The following code snippet includes the initialization used for the main `Window` class in this project:

```
WNDCLASSEX wc = { 0 };
wc.cbSize = sizeof(WNDCLASSEX); //structure size
wc.lpfnWndProc = (WNDPROC)MainWindow::MessageRouter; //window procedure
wc.style = 0; //class styles
wc.hInstance = hInstance; //handle to application instance
wc.hIcon = LoadIcon(NULL, IDI_APPLICATION); //large application icon
wc.hCursor = LoadCursor(NULL, IDC_ARROW); //mouse cursor icon
wc.hbrBackground = backgroundBrush; //background color
wc.lpszMenuName = NULL; //used menu resource
wc.lpszClassName = L"MainWindow"; //class name
wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION); //small application icon

ATOM ClassAtom = RegisterClassExW(&wc);
```

In order to use a `Window` class, it has to be registered with `RegisterClassExW()`. It can then be utilized in the initialization of different windows to automatically apply all defined class-specific properties. For example, the following code is used to create the prototype's main window:

```
HWND hWndMain = CreateWindowExW(WS_EX_CONTROLPARENT, (LPCTSTR)MAKELONG(
    ClassAtom, 0), 0, WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
    WS_CLIPSIBLINGS | WS_CLIPCHILDREN | WS_CAPTION | WS_SYSMENU, 0, 0, 960,
     545, 0,  NULL, hInstance, this);
```

`HWND` variables are handles for windows as well as all other interface elements such as buttons and sliders. The first parameter of `CreateWindowExW` is for extended windows styles, which can be used to change the appearance (e.g. scrollbar or border) of the window. In this case, `WS_EX_CONTROLPARENT` specifies that all child windows should forward their navigation events to this parent window. The next arguments should be a pointer to the class name specified earlier as well as the window title. The fourth parameter specifies different window styles, which define the appearance of elements such as the title bar, as well as clipping behavior of child elements. Coordinates `(0, 0, 960, 545)` specify the x and y positions for the top left corner as well as the width and height of the window. The remaining parameters are for a handle to the parent control (as this is the main window, it does not have any), a handle to the menu, the application instance as well as additional information that will be used during the creation of the window. Child controls such as buttons and sliders are created in a similar manner, but their parent handle has to be set to the control that hosts them instead of `NULL`.

Message Loop

The main Win32 message loop is entered right after initialization and is only exited when quitting the application. The following code snippet shows the main Win32 message loop of this prototype:

```
MSG       msg = { 0 };
while (WM_QUIT != msg.message)
{
```

```
4   while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE) > 0)
5   {
6     TranslateMessage(&msg);
7     DispatchMessage(&msg);
8   }
9
10  /* ...handle other events, update other components... */
11 }
12 QuitApplication();
```

The `PeekMessage()` method is constantly looking for any kind of event (e.g. key strokes, mouse clicks, window resizing and closing). On receipt, it is immediately sent to the window procedure of the control handle it is addressed to by calling `DispatchMessage()`.

The `MSG` structure represents such an event (or a *message*) and is defined as follows:

```
1 typedef struct MSG {
2     HWND        hwnd;
3     UINT        message;
4     WPARAM      wParam;
5     LPARAM      lParam;
6     DWORD       time;
7     POINT       pt;
8 }
```

`hwnd` is the handle to the control the message is addressed to, while `message` is an integer that specifies the type of event. The possible values are defined as constants in the Win32 API for better readability:

```
1 #define WM_CREATE                   0x0001 //window creation
2 #define WM_DESTROY                  0x0002 //window destruction
3 #define WM_MOVE                     0x0003 //window movement
4 #define WM_SIZE                     0x0005 //window resizing
5
6 /* ...more types... */
```

`wParam` and `lParam` are additional parameters that vary from message to message. `time` represents the exact time at which the message was sent, while `pt` contains the current cursor position.

Window Procedure

The window procedure is a method that receives the event messages dispatched from the message loop. Every control has exactly one assigned window procedure. The following is an example taken from the developed application's abstract `SubWindow` class:

```
1 LRESULT CALLBACK SubWindow::SubWindowProc(HWND hWnd, UINT message, WPARAM
    wParam, LPARAM lParam)
2 {
3   switch (message)
```

```
4   {
5   case WM_DESTROY: //on application quit
6     CleanUp();
7     return DefWindowProc(hWnd, message, wParam, lParam);
8     break;
9   case WM_COMMAND: //on button press
10    ProcessUI(wParam, lParam);
11    break;
12  case WM_SIZE: //on window resizing
13    Resize();
14    break;
15  case WM_LBUTTONDOWN: //on left mouse button click (or finger press and
        hold)
16    mouseDown = true;
17    break;
18  case WM_LBUTTONUP: //on left mouse button release (or finger release)
19    mouseDown = false;
20    break;
21  default:
22    return DefWindowProc(hWnd, message, wParam, lParam);
23  }
24 }
```

The `hWnd` parameter represents the handle of the control that a message is addressed to. This is important, since a procedure can (and will, in most cases) handle the events of more than one Win32 control. The remaining parameters contain relevant information about an event, as already explained in Section 4.2.2. Depending on the message type, different actions can be taken. For example, whenever the user presses a button, the `WM_COMMAND` message with the origin control as `lParam` is received, whereupon the appropriate actions are taken in `ProcessUI()`. All events that do not need custom handling can be passed on to `DefWindowProc()`, which is the default Win32 procedure.

4.2.3 OpenGL

As mentioned in Section 4.2.1, the *Open Graphics Library* (OpenGL) is a cross-platform and cross-language API used to interact with a GPU. The API provides a set of functions and constants that can be utilized to render 2D and 3D vector graphics [35]. There are several libraries available that help to create OpenGL windows and try to abstract most of the functions away with wrapper classes [6][60]. However, this prototype uses the API provided by OpenGL directly in order to have more control over the rendering process. Some of the most used basic concepts are briefly explained in the following sections.

Context Creation

An *OpenGL context* essentially stores all state information about a specific instance of OpenGL. In order to be able to use OpenGL at all, a context has to be created. As soon as it is destroyed, the library ceases to function as well. Since OpenGL can not exist without a context, its creation is not part of the OpenGL API and is therefore done

differently on every platform. As this application runs on Microsoft Windows, the Win32 API has to be used for this task.

Every window in Win32 has an associated *device context*, which contains a *pixel format*. It is essentially a structure that specifies the properties of the framebuffer, which contains the currently displayed graphic data. As a result, it has to be set up properly in order for OpenGL to work. The following shows the pixel format for this application's OpenGL context:

```
1 PIXELFORMATDESCRIPTOR pfd;
2 memset(&pfd, 0, sizeof(PIXELFORMATDESCRIPTOR));
3 pfd.nSize = sizeof(PIXELFORMATDESCRIPTOR);
4 pfd.nVersion = 1;
5 pfd.dwFlags = PFD_DOUBLEBUFFER | PFD_SUPPORT_OPENGL | PFD_DRAW_TO_WINDOW;
      //properties of pixel buffer
6 pfd.iPixelType = PFD_TYPE_RGBA; //type of pixel data
7 pfd.cColorBits = 32;
8 pfd.cDepthBits = 32;
9
10 int pixelFormat = ChoosePixelFormat(hDC, &pfd);
11 SetPixelFormat(hDC, pixelFormat, &pfd);
```

Essentially, this structure describes the desired features of an OpenGL pixel format. It has to be subsequently passed to `ChoosePixelFormat()`, which returns the id of the pixel format that most closely resembles it. Finally, it can be applied to the current Win32 context `hDC` with `SetPixelFormat()`.

The context is created by calling `wglCreateContext()`. Before OpenGL can be used, however, the context has to be made current with `wglMakeCurrent()`. One thread can only contain a single current OpenGL context.

This application uses the GLEW library during initialization to determine the platform's supported OpenGL versions, among other things [30]. The context creation is then adjusted accordingly, since older versions require different pixel formats. In order to use this library, a fake OpenGL context has to be set up and destroyed before creating the actual context. This is done largely analogously to the already described procedure and is therefore not covered here.

Vertex Buffer Objects

In order for OpenGL to be able to render a primitive, an array containing all necessary information about its 3D data has to be set up. The following are exemplary vertex and triangle arrays for a triangle mesh representing a simple square on the z-plane:

```
1 float vertices[] = {
2   0.0, 0.0, 0.0, //bottom left vertex (x, y, z)
3   0.0, 1.0, 0.0, //top left vertex
4   1.0, 0.0, 0.0, //bottom right vertex
5   1.0, 1.0, 0.0, //top right vertex
6 }
7
```

```
8 int triangles[] = {
9   0, 1, 2, //triangle 1
10  1, 3, 2  //triangle 2
11 }
```

Subsequently, this data has to be uploaded to the graphics card in order to be rendered. OpenGL uses *Vertex Buffer Objects* (VBO) to store data about the vertices. In this case, a buffer object needs to be generated for each of the two arrays.

```
1 GLuint vbo;
2 glGenBuffers(1, &vbo); //generate vertex buffer object
3 GLuint ibo;
4 glGenBuffers(1, &ibo); //generate index buffer object
```

The variables `vbo` and `ibo` are references to the buffers, which are managed by OpenGL. They have to be bound to a specific type before arrays can be loaded into the buffers:

```
1 glBindBuffer(GL_ARRAY_BUFFER, vbo); //bind vertex buffer object
2 glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), &vertices[0],
    GL_STATIC_DRAW); //set vertex data
3 glBindBuffer(GL_ARRAY_BUFFER, 0); //unbind vbo
4
5 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);  //bind index buffer object
6 glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(triangle), &triangles[0],
    GL_STATIC_DRAW); //set triangle data
7 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0); //unbind ibo
```

Shaders

Shaders in OpenGL are essentially small user-defined programs that are part of the rendering pipeline and operate on the provided vertex data before it is rendered. Shaders are written in the C-style *OpenGL Shading Language* (GLSL) and compiled at runtime. The following shader stages are available, sorted by the order in which they are processed:

- **Vertex Shaders:** Receives and outputs exactly one vertex attribute. Typically performs 3D transformations if necessary. (Mandatory)
- **Tessellation Shaders:** Subdivides sets of vertex data into smaller primitives. (Optional)
- **Geometry Shaders:** Can create new primitive geometries based on the provided vertex data. (Optional)
- **Fragment Shaders:** Calculates the color of every rendered pixel. (Mandatory)
- **Compute Shaders:** Can perform arbitrary tasks. (Optional)

As this application only uses the mandatory shader stages, the following paragraphs exclusively cover the vertex and fragment shaders.

Vertex shaders receive data for each vertex that is stored in the vertex buffer object. In order to keep this example simple, it is assumed that only a 3D position is stored in each vertex. The following is a bare-bones example of such a vertex shader:

```
1 #version 330 //used GLSL version
2
3 in vec3 inPosition; //input vertex position
4
5 void main()
6 {
7   gl_Position = vec4(inPosition, 1.0); //output rendering position
8 }
```

For the sake of simplicity, this shader simply outputs the position that it receives. Note that this would not lead to a correctly rendered cube. In an actual application, the positions would be transformed into 3D camera space by using several provided matrices. The calculated position needs to be assigned to `gl_Position`, which is a global OpenGL variable used in the rendering pipeline.

The fragment shader calculates the color for every single pixel on the screen covered by a primitive. The following shows, once again, a very simple example:

```
1 #version 330 //used GLSL version
2
3 out vec4 outputColor; //output rendering color for one pixel
4
5 void main()
6 {
7   outputColor = vec4(1.0,0.0,0.0,1.0);
8 }
```

To keep this introduction short, this shader simply assigns the color *red* to every processed pixel. It could, for example, also perform additional lighting calculations based on a predefined vertex color.

In order for the shaders to be usable by OpenGL, they have to be compiled first. For that purpose, the source code has to be loaded into a specific buffer. This procedure has to be performed once for each shader.

```
1 GLuint shaderId = glCreateShader(GL_VERTEX_SHADER); //create shader
2 glShaderSource(shaderId, (int)sourceCode.size(), &sourceCode, NULL); //
    attach source code
3 glCompileShader(shaderId); //compile source code
```

Vertex and fragment shaders are part of the same rendering pipeline. As a result, they have to be linked together by attaching them to a newly created *Shader Program*, before they can be used for rendering. This is done in the following way:

```
1 GLuint shaderProgramId = glCreateProgram(); //create shader program
```

```
2 glAttachShader(shaderProgramId, vertexShaderId); //attach vertex shader
3 glAttachShader(shaderProgramId, fragmentShaderId); //attach fragment shader
4 glLinkProgram(shaderProgramId); //link program
```

The method `glLinkProgram()` is used to finalize the program and make it available to OpenGL. The linked shaders can be used during rendering by calling `glUseProgram(shaderProgramId)`. However, only one program can be active at any time.

Users can also specify additional parameters that can be passed to a shader program. These global variables are called *uniforms*, which have to be declared in the shader code as follows:

```
uniform int renderMode;
```

Variables of this type can be used to pass a shader additional data associated with a specific primitive. They can be set right before drawing a set of vertices and typically contain matrices used for 3D projection, current lighting conditions or the active rendering mode.

Vertex Array Objects

The only thing missing is the link between the vertex buffer objects and the shaders, since OpenGL does not know how the vertex data in the buffer is ordered. To this end, *Vertex Array Objects* (VAO) are used. A VAO essentially stores all the links between raw vertex data in a VBO and input attributes of shaders. It is initialized and bound very similarly to VBOs:

```
1 GLuint vao;
2 glGenVertexArrays(1, &vao); //generate vertex array object
3 glBindVertexArray(vao); //bind vertex array object
```

Every input attribute defined in the vertex shader has a unique incrementing id referencing its position. In the example code shown in Section 4.2.3, the only input attribute is `inPosition`, which automatically has the id `0`. A second attribute, such as the vertex color, would have the id `1`, and so on. Now that the VAO is generated and bound, the links between the VBO and the shader attributes can be set up as follows:

```
1 glBindBuffer(GL_ARRAY_BUFER, vbo); //bind vbo to correct type
2 glEnableVertexAttribArray(0); //enable the attribute array for attribute
      with id 0 (inPosition)
3 glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, reinterpret_cast<void
      *>(0)); //set actual link
```

The method `glVertexAttribPointer()` has six parameters. The first one references the id of the shader attribute, while the second and third parameter represent the number of elements this attribute consists of as well as the data type of each element. The fourth parameter decides whether or not the values should be normalized between `-1.0` and `1.0`. The fifth parameter represents the *stride* in the input array, i.e. the

number of bytes between each attribute in the array. The last parameter specifies how many bytes are in front of the first attribute of this type in the array. The last two parameters are particularly important in case of more than one shader attribute being stored in the vertex array.

Drawing

After the vertex buffer objects, shader programs and vertex array objects have been set up, OpenGL is ready to draw the 3D scene onto the screen:

```
1 //clear screen
2 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
3
4 //bind and use appropriate objects
5 glUseProgram(shaderProgramId);
6 glBindVertexArray(vao);
7 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
8
9 //draw triangles
10 glDrawElements(GL_TRIANGLES, triangles.size(), GL_UNSIGNED_INT, (void*)0);
11
12 //unbind/free used buffers and shaders
13 glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
14 glBindVertexArray(0);
15 glUseProgram(0);
```

Since the graphics card already has the vertex data in memory, the primitive can be drawn by calling the `glDrawElements()` function every frame, as long as the corresponding shader program and vertex buffer objects are active. In order to allow other 3D models with different shader programs to be drawn as well, all used buffers and programs should be unbound after drawing. Several additional options for OpenGL rendering (e.g. depth tests and culling) can be set simultaneously. For example, calling `glEnable(GL_DEPTH_TEST)` before making the draw call will enable depth tests. It can be disabled again by using `glDisable(GL_DEPTH_TEST)`.

4.3 Controller

The implemented system is mainly controlled by two particularly important classes. `MainController` contains the starting point of the application and is responsible for its life cycle, while also managing state changes, the 2D user interface and communication between UI, KinectFusion and the 3D engine. `GraphicsController`, on the other hand, is responsible for all 3D-related functionality aside from 3D scanning. This includes rendering, navigation, selection and segmentation. Both classes and their primary responsibilities are described in more detail in the following sections.

4.3.1 Main Controller

The `MainController` class is the central hub and responsible for a wide variety of components. It manages the application's user interface and handles mode changes as well as communication between core system parts. Most of the methods in this class are very straight forward and are either used to update specific user interface elements directly or initiate commands in `GraphicsController` based on user input, all of which are discussed in their own sections. The main components of the Win32 main messaging loop, which is initialized during startup, are explained in a generic way in Section 4.2.2. Therefore, this section only provides a quick overview of the initialization at startup and discusses the implemented methods and components used for managing the current application state.

Initialization

The `InitApplication()` method is the application's entry point, during which all of the main components are initialized. It begins by creating and initializing the `StyleSheet` class, which stores most layout-related information, such as text formatting and button gradients, in a globally accessible place. It is explained in more detail in Section 4.4.2. Subsequently, the main Win32 window and all its constantly visible child interface elements, such as the mode selection bar, the help button and text labels with information about the current scene, are initialized with appropriate parameters. More details about the elements and procedures used for Win32 applications can be found in Section 4.2.2. In addition, the method initializes and populates the custom wrapper classes `GUIContainer`, each containing a logically connected group of interface elements. They simplify UI management by allowing controller classes to perform certain operations (e.g. visibility changes) on all elements in a group simultaneously. See Section 4.4.2 for more information about these wrapper classes.

The method subsequently fills data structures used for state management, which is discussed in more detail in Section 4.3.1. Afterwards, `GraphicsController` is initialized in an additional thread, which will be responsible for 3D graphic operations (see Section 4.3.2). The controller then initiates the initial application stage (*Prepare*) and starts the main messaging loop, which constantly updates all currently active user interface elements and forwards messages and commands to the `GraphicsController`.

State Management

`WindowState` is an enumeration class, consisting of constants representing the distinct application stages. It is declared as follows:

```
1 enum class WindowState {
2   Prepare = 0,
3   Scan = 1,
4   PlaneSelection = 2,
5   Segmentation = 3,
6   PlaneCut = 4,
```

```
7   Processing = 5,
8   Interaction = 6
9 };
```

Elements of the type `SubWindow` represent entire UI panels with own Win32 message loops that can contain a variety of child interface controls (see Section 4.4.1 for more details). During application startup, `MainController` initializes a `SubWindow` for every distinct `WindowState` and stores them as `(WindowState,SubWindow)` pairs in an unordered map as follows:

```
1 //map initialization
2 unordered_map<WindowState, unique_ptr<SubWindow>> subWindowMap;
3
4 //assign object that inherits from SubWindow
5 subWindowMap[WindowState::Prepare] = unique_ptr<PrepareWindow>(new
      PrepareWindow());
6
7 //initialize the subwindow with appropriate parameters
8 subWindowMap[WindowState::Prepare]->Initialize(/* parameters */);
9
10 /* ...initializations of other SubWindows... */
```

This allows for efficient state management, since the controller can quickly retrieve and activate the appropriate `SubWindow` as well as deactivate all unrelated ones by accessing the unordered map with the current `WindowState` as index key, whenever the currently selected application stage changes. The `ChangeState()` method takes care of this procedure:

```
1 void MainController::ChangeState(WindowState _state)
2 {
3   //hide the previously active ui panel
4   if (currentWindow != subWindowMap.end())
5     currentWindow->second->Hide();
6
7   //retrieve the UI panel corresponding to the selected WindowState and
        make it active
8   currentState = _state;
9   currentWindow = subWindowMap.find(currentState);
10  currentWindow->second->Show();
11
12  /* ...other commands that happen on state changes... */
13 }
```

In the main messaging loop, the events of the currently active `SubWindow` are handled as follows:

```
currentWindow->second->HandleEvents(*this);
```

An instance of `MainController` itself is passed to the currently active `SubWindow` every frame. This allows the individual UI panels to send possible user input to the

MainController, which can either forward it to GraphicsController or process it directly.

4.3.2 Graphics Controller

The GraphicsController class manages all application aspects related to 3D graphics and interaction, i.e. navigation, rendering, segmentation, selection and model export. Even though it is initialized by the MainController, it executes all operations in its own thread, which includes a separate Win32 message loop (the basics of which are described in Section 4.2.2) in order to handle events for the UI panel hosting the OpenGL context. Many methods in GraphicsController are simply sending commands to graphics-related components, sometimes in a dedicated thread. Since most of them are explained in their own sections, only the general structure, main loop and event handling techniques are covered in this section.

Overview

As soon as GraphicsController is initialized, it immediately sets up the OpenGL context in its own dedicated thread, as described in Section 4.2.3. If this operation is successful, it continues to initialize all the tools needed for 3D rendering, segmentation and selection. Afterwards, the class begins executing its main loop, which will only be exited once the application is terminated. Note that context creation as well as all subsequent OpenGL operations, which are used by almost every 3D aspect of the application, have to be executed in the same thread due to restrictions of OpenGL.

The main loop contains an additional Win32 message loop, since the controller class is responsible for the Win32 window that contains the OpenGL context. In addition, the loop is responsible for updating the 3D scene and related elements every frame as follows:

```
1 HandleEvents();
2 if (openGLWindow.IsVisible())
3 {
4   openGLWindow.HandleEvents(*this); //window-specific events
5   UpdateFrame(); //rendering
6   HandleInput(); //user input
7   CountFPS(); //update fps counter
8   UpdateSceneInformation(); //update vertex and triangle labels
9 }
```

The main loop performs all graphics-related tasks whenever the OpenGL window is visible. UpdateFrame() renders the 3D scene corresponding to the current application stage and updates the virtual camera. 2D elements such as text overlays and icons (e.g. the trash bin) are also being drawn in this function. HandleInput() continuously checks for user input and activates appropriate 3D selection techniques if necessary. CountFPS() counts the frames that are processed every second and updates a corresponding text label. Information about the scene, such as vertex and triangle counts, is gathered and updated by UpdateSceneInformation(). Events are handled every frame, regardless

of whether or not the scene is actually being rendered at that moment. More details on this aspect can be found in the following section.

Event Handling

Even though `GraphicsController` is responsible for almost all 3D-related operations, many of these methods are called from other threads, since most user input regarding 3D interaction is being registered in the Win32 message loop of `MainController`.

In order to make this possible despite OpenGL's thread restrictions and to reduce other concurrency issues, an event system has been implemented. `GraphicsController` exposes the method `PushEvent()`, which allows the class responsible for its initialization to request modifications by passing an object of the type `GraphicsControlEvent`. The controller class then adds this specific request to an event queue. Every frame, `HandleEvents()` checks the queue and processes all pending requests. As explained before, events are handled regardless of whether or not 3D content is currently being rendered. This allows the application to prepare certain time-consuming 3D operations, such as the initial processing of 3D reconstructions, while users are still finishing other tasks (e.g. reading guidance messages). As a result, the perceived downtime during application usage is reduced. In addition, this ensures that sensitive data structures, such as 3D scene data, is not corrupted due to concurrency issues.

4.4 2D User Interface

As mentioned in Section 4.2.1, the 2D user interface is implemented using the Win32 API [53]. This section describes the different UI panels for each application stage as well as helper classes used for style and layout management.

4.4.1 UI Panels

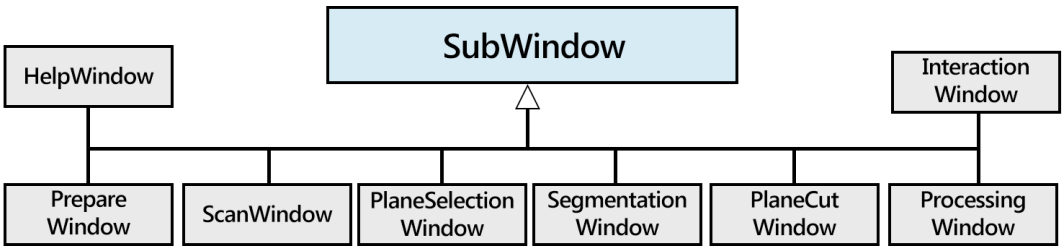


Figure 4.3: Class hierarchy for 2D interface panels.

The main Win32 window, which resides in `MainController`, consists of many different child control elements. Depending on the currently active application stage, only a very small subset of them is visible at a given time. In order to better manage the different mode interfaces, they are split into distinct UI panel classes that inherit from the abstract type `SubWindow`. Each stage has a dedicated panel responsible for

all of its specific interface elements. Furthermore, `HelpWindow` is used to display and manage all guidance messages on top of other panels if necessary, while `OpenGLWindow` provides the canvas for OpenGL rendering.

Since almost all used elements related to Win32 interfaces, such as window classes and procedures, handles and event messages, are already explained in Section 4.2.2, this section will not discuss them in further detail. UI panels can be initialized by calling the following function:

```
virtual void Initialize(HWND _parentHandle, HINSTANCE _hInstance, float
    _marginTop, float _marginBottom, float _marginRight, float _marginLeft,
     std::wstring _className, ColorInt _backgroundColor);
```

The first and second parameter represent the handle to the Win32 parent window and application instance respectively. The next four parameters specify the panel's margins as percentages of the parent control's height and width. This way, the positioning of the panel is always relative to its parent, which allows for proper rearrangement whenever users resize the application. The last two parameters are the name of the window class to use and the specific panel's background color.

As every single UI panel contains its own Win32 window procedure, a distinct window class has to be created on each initialization, before all child controls as well as the panel itself can be created with `CreateWindowEx()`. The window procedure `SubWindowProc()` is responsible for handling Win32 messages addressed to the respective panel (see Section 4.2.2 for a small code snippet). The procedure receives and handles events such as button clicks, window resizing, mouse movement and the customized drawing of specific controls. It can be extended or overwritten by child controls to handle events that are only important in a specific application stage. Whenever a button is pressed, the procedure receives a `WM_COMMAND` message, whereupon `ProcessUI()` is executed with the pressed button name and state as parameters. The concrete implementation of `ProcessUI()` depends on the specific subclass of `SubWindow`, since each of them consists of different controls with distinct purposes. The panels receive their events from the Win32 message loops in the threads they have been initialized on. In this specific application, only `OpenGLWindow` is receiving messages from the message loop in `GraphicsController`, all other panels are initialized and managed by `MainController`.

Elements of type `SubWindow` offer some basic UI functionalities such as `Show()` and `Hide()`, which can be used to control a panel's visibility. `Resize()` changes the size of the panel as well as all child elements depending on the current size of the parent control. Some panels have additional, stage-specific responsibilities. For example, `ScanWindow` handles the communication with the 3D scanning interface, while `PrepareWindow` initiates and displays a countdown whenever users start a scan.

UI panels do not hold any reference to the class that initialized them as member variables in order to avoid unnecessary dependencies. However, if a user presses a button, it is usually expected to have an effect on a variety of components in the application, such as `MainController` or `GraphicsController`. As the UI panels handle the events associated to these controls, but do not have any way to access the controller

classes and their members, they instead push an event into a custom event queue. These events are of the type `SubWindowEvent`, which holds an `enum` variable defining the type of the event. The following is the implementation of the base class:

```
1 class SubWindowEvent {
2   public:
3     enum Type {
4       StateChange, //switch to different application stage
5       Last //helper value to enable inheritance
6     };
7   };
```

The `enum` elements listed are standard events, which can be raised by every element of the type `SubWindow`. As `enum` types cannot be inherited from directly in C++, it has been encapsulated in its own class. This allows child classes of `SubWindow` to extend the event types as follows:

```
1 class InteractionWindowEvent : public SubWindowEvent{
2   public:
3     enum Type {
4       ChangeCameraMode = SubWindowEvent::Last,
5       ChangeManipulationMode,
6       Reset,
7       ExportModel,
8       Last
9     };
10  };
```

Whenever a panel is active and visible, the controller responsible for its creation calls `HandleEvents()` with a self-reference as parameter. The specific panel can then call methods on the provided instance, depending on the elements in its event queue.

4.4.2 Style and Layout Management

In order to make it easier to manage the application's appearance, a variety of different classes is used. They wrap cumbersome Win32 functions and provide a centralized way of storing layout parameters. The following sections give a quick overview of the most important classes used for these tasks.

Button Layout

Customizing the appearance of buttons in Win32 applications is a very cumbersome process. The window event message (see Section 4.2.2) `WM_DRAWITEM`, which is responsible for drawing UI controls on the screen, has to be custom-handled. For example, different background or text colors of interface elements can not be changed without manually performing the entire drawing process. The corresponding (rectangular or circular) region on the screen has to be filled, different brushes have to be instantiated and applied to

certain button elements. In addition, text has to be placed and colored depending on the current state of the button.

This application has a variety of different buttons, many of which share a majority of layout parameters such as background and text color. In order to group the different button styles and handle Win32 control drawing in a more generic way, the `ButtonLayout` class has been implemented. Each element of this class represents exactly one type of button layout. However, different controls can share the same appearance. Button layouts are defined by their `ButtonLayoutParams`, a structure containing the following information:

```
1 struct ButtonLayoutParams
2 {
3   int fontSize; //size of button text
4   ColorInt backgroundColor; //background color
5   Gradient activeGradient, inactiveGradient, pressedGradient; //gradient
        brushes for different states
6   ColorInt activeTextColor, inactiveTextColor; //text colors for different
        states
7   ColorInt activePenColor, inactivePenColor, pressedPenColor; //border
        colors for different states
8   int edgeRounding; //amount of edge rounding
9 };
```

Note that the `backgroundColor` variable represents the color of the UI element *behind* the button, as Win32 can not display transparency properly. Every UI panel in the developed application contains the following unordered map:

```
std::unordered_map<HWND, ButtonLayout> buttonLayoutMap;
```

Each button control handle is assigned exactly one `ButtonLayout` on initialization, which determines its appearance and manages its customized rendering. In the window procedure of a panel, the `WM_DRAWITEM` message for a control can be handled as follows:

```
1 LRESULT CALLBACK SubWindow::SubWindowProc(HWND hWnd, UINT message, WPARAM
    wParam, LPARAM lParam)
2 {
3   /* ...handle other events... */
4   case WM_DRAWITEM:
5   {
6     HWND controlHandle = ((LPDRAWITEMSTRUCT)lParam)->hwndItem; // get
        button handle
7     buttonLayoutMap[controlHandle].Draw(lParam);
8   }
9   /* ...handle other events... */
10 }
```

The `Draw()` method receives the `lParam` variable as a parameter, since it contains the information about the current button state (i.e. pressed, unpressed, inactive). The button is then drawn using the layout parameters specified for the active state.

Style Sheet

The `StyleSheet` class is a singleton, i.e. a globally accessible class of which only a single instance can exist at any time. It provides centralized access to layout parameters that are being used in a wide variety of controls. This includes font size, weight, style and color as well as background brushes for different types of interface elements. It also initializes `ButtonLayout` instances for every distinct button appearance type and stores them in a map. As a result, whenever UI panels initialize button controls and want them to have a specific appearance, they can do so as follows:

```
1 HWND button = CreateWindowEx(/* ...parameters... */); //create button
2 buttonLayoutMap[buttonExport].SetLayoutParams(StyleSheet::GetInstance()->
      GetButtonLayoutParams(ButtonLayoutType::Green)); //assign layout
```

GUI Container

The `GUIContainer` class is a simple wrapper class for an arbitrary number of Win32 control handles. It provides an easy way to simultaneously call commonly used Win32 functions on a number of connected control handles (`HWND`). Examples include `Show()` and `Hide()`, `Enable()` and `Disable()`, as well as `IsCursorInHandle()`, which determines whether or not the mouse cursor is currently hovering over any one of the handles in this `GUIContainer`.

4.5 Rendering

This section describes the classes and data structures used to represent the different 2D and 3D objects in the scene. Text and shader implementations are discussed as well, before explaining the components responsible for actual 3D rendering.

4.5.1 Renderables

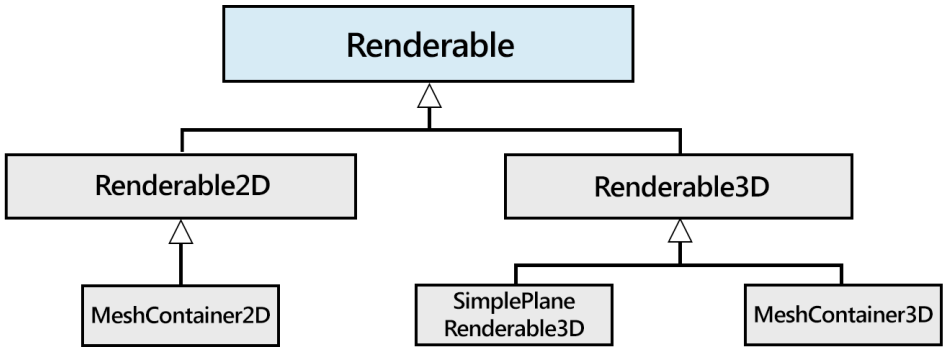


Figure 4.4: Class hierarchy for renderable objects.

Every instance of the type `Renderable` or one of its subtypes represents exactly one self-contained 3D object that can be rendered onto the screen. Every `Renderable` contains an array of vertices and sometimes a list of triangle indices. Each vertex is an object of the custom structure `Vertex`, while indices are represented by `Triangle`, both of which are defined as follows:

```
1  struct Vertex
2  {
3    float x, y, z; //3D position
4    float r, g, b; //vertex color
5    float normal_x, normal_y, normal_z; //normal vector
6  };
7
8  struct Triangle
9  {
10   GLuint v1, v2, v3; //indices of triangle vertices
11 };
```

Each `Renderable` contains its own OpenGL Vertex Buffer Objects (VBOs) as well as Vertex Array Object (VAO), which are created on initialization with the provided vertex and triangle arrays (see Section 4.2.3 for details). There are similar buffers for bounding boxes in each `Renderable`, which contain a uniformly colored, simplified version of the original mesh that can be used for 3D selection (see Section 4.8). Every mesh has an assigned `OpenGLShaderProgram`, which is explained in Section 4.2.3. By calling `Draw()` with the appropriate parameters, the `Renderable` utilizes OpenGL functions to render itself onto the screen. Depending on the availability of a triangle array, the `Renderable` is either drawn by utilizing `glDrawElements` with indices or `glDrawArrays` with the vertex data directly. `DrawForColorPicking()`, on the other hand, renders the bounding box of the mesh.

`Renderable` is an abstract base class with two abstract child classes `Renderable2D` and `Renderable3D`.

Renderable2D

`Renderable2D` represents models that are orthogonally projected onto the screen and therefore do not need to be transformed whenever the camera perspective changes. Instead, their scale and position depend entirely on the current viewport size. For this reason, the `Draw()` function of `Renderable2D` expects only the current OpenGL viewport width and height as parameters. The concrete implementation of this abstract class is `MeshContainer2D`, which provides additional functions for loading a 2D mesh from a file. Objects of this type are typically 2D interface elements, such as the trash bin in the manipulation stage (see Section 3.8).

Renderable3D

`Renderable3D` are three-dimensional objects that are positioned directly in the scene. They have to be projected into 3D space using a special transformation matrix, consisting

of the multiplied model, view and projection matrices.

The *model matrix* transforms coordinates from model to world space. It is composed of the multiplied translation, rotation and scaling matrix of a specific mesh and is therefore saved in each `Renderable3D` separately. The class also exposes a variety of methods to enable different 3D transformations: `SetTranslation()` changes the position of a model, while `SetScale()` can in- or decrease the current size. `AddRotation()`, on the other hand, can be used to add a rotation around a specified axis. Rotations are stored as quaternions to improve performance and avoid possible problems with Euler angles such as gimbal lock [33]. The model matrix is calculated for every frame with `CalculateModelMatrix()` by multiplying the mentioned transformation components.

The *view matrix* contains information about the current camera pose and transforms coordinates from world to camera space. It is calculated by the `GraphicsCamera` component, which is described in Section 4.6. The *projection matrix* transforms the vertices to 2D screen coordinates and depends on the field of view and clipping plane of the chosen camera. The current aspect ratio of the viewport influences the components of this matrix as well. Both the view and the projection matrix are expected as parameters in the `Draw()` method of `Renderable3D`. In addition, the class exposes the method `HighlightTrianglesWithColor()`, which highlights specific triangles of the 3D object by temporarily changing the corresponding vertex colors.

`SimplePlaneRenderable3D` is one concrete implementation of `Renderable3D`, which represents a simple three-dimensional plane. It contains a `PlaneParameters` structure, which consists of the plane normal as well as the plane's distance from the world origin.

`MeshContainer3D` also implements `Renderable3D` and represents an arbitrary three-dimensional triangle mesh. In addition to all inherited functionality, it provides methods for mesh processing, which use the functions provided by the VCG library (see Section 4.2.1) [13]. A `VCGMesh` object is needed for this purpose, since it is the internal mesh representation used in the library. It embodies the same 3D model represented by `MeshContainer3D` itself, but can be used to call the processing methods provided by the VCG library. `CopyInternalToVisibleData()` and `CopyVisibleToInternalData()` are used to synchronize the two representations before and after filters are applied. The following methods in `MeshContainer3D`, which themselves call functions provided by the VCG library, can be used to process the represented mesh:

- **`Clean()`:** Removes duplicate and unreferenced vertices as well as duplicate, degenerate or zero-area faces and ensures that all vertex normals are correct.
- **`FillHoles()`:** Searches for and subsequently closes holes in the mesh by using the VCG methods `EarCuttingIntersectionFill()` and `EarCuttingFill()`.
- **`LaplacianSmooth()`:** Performs Laplacian smoothing on the mesh, which adjusts the positions of vertices based on their neighbors and smooths their overall distribution. Uses VCG's `VertexCoordLaplacian()` method.

- **`MergeCloseVertices()`:** Merges all vertices that are closer than a specified parameter by using the VCG method `MergeCloseVertex()`.
- **`RemoveNonManifoldFaces()`:** Removes all non-manifold faces of a mesh, since they would cause problems during several other procedures.
- **`RemoveSmallComponents()`:** All connected components in a mesh that contain less than a specified number of vertices are removed by utilizing the VCG method `RemoveSmallConnectedComponentsSize()`.
- **`UnsharpColor()`:** Smooths the vertex color distribution of a mesh by using VCG's `VertexColorLaplacian()`.

The method `LoadFromFile()` can be used to fill the vertex and triangle arrays as well as the `VCGMesh` structure with data provided by a saved mesh file by utilizing the `ImporterPLY` class of the VCG library. On loading a mesh, `MeshContainer3D` automatically performs several cleaning and simplification filters. This includes merging all vertices closer than 0.5mm, removing components that contain less than $\frac{1}{100}$ of the mesh's total vertex count and performing Laplacian smoothing. These steps ensure that the application's performance does not decrease because of an unnecessary high number of vertices. In possible future releases, these simplifications could be made optional to allow more control over the resulting mesh quality.

4.5.2 Scene Data

An instance of the type `ModelData` contains an array filled with all `MeshContainer3D` objects of a single three-dimensional scene. The class provides easy ways to perform specific operations on every triangle mesh in a scene with a single function. Each application stage has a corresponding `ModelData` instance, all of which are managed by `GraphicsController`. Storing the scenes of all steps enables users to return to previous modes while undoing all changes applied in subsequent stages. Whenever the current mode changes, `CopyFrom()` is used to copy model data from one stage to the next.

The class offers a variety of methods, most of which call a specific function on either every or one `MeshContainer3D` object in the scene by providing the corresponding index. For example, the `FillHoles()` function can either be called with an index parameter to only fill holes of a specific submesh or without it to perform the operation on all clusters in the scene. `Draw()` is called by `OpenGLRenderer` instances (see Section 4.5.5) and draws every `MeshContainer3D` on the screen. Several helping functions such as `GetCenterPoint()`, which returns the center position of the entire scene, and `GetBasePoint()`, which provides the bottom center position, are available as well. `DuplicateMeshAndGetItsIndex()` is responsible for duplicating a `MeshContainer3D` with all its properties.

The class `IconData` offers similar functionalities for `MeshContainer2D` instances and is therefore representing all 2D icons in the scene. Its `Draw()` method, for example,

renders all 2D objects to the screen. Each icon consists of one default `MeshContainer2D` and an additional one that is only shown as users hover over it.

4.5.3 Text

The `OpenGLText` class is used to render two-dimensional text onto the OpenGL viewport. It utilizes the FreeType Library to access a provided font file as a bitmap and subsequently convert it to a texture that can be rendered by OpenGL [38]. The class uses its own `OpenGLShaderProgram` consisting of custom shaders (see Section 4.5.4 for more details).

`Initialize()` generates the necessary buffers, creates the shader program and loads the font file, whose name has to be provided as a parameter. Before rendering any text, the `PrepareForRender()` function has to be called, as it adjusts the OpenGL options accordingly and binds the necessary buffers and textures. The following method can be used to display a string:

```
1 void OpenGLText::RenderText(
2   wstring text, //text to be rendered
3   int pixelSize, //text size in pixels
4   float x, //horizontal text position, -1 (left) to 1 (right)
5   float y, //vertical text position, -1 (top) to 1 (bottom)
6   float sx, //current viewport width divided by two, used for scaling
7   float sy //current viewport height divided by two, used for scaling
8 );
```

4.5.4 Shaders

`OpenGLShader` is essentially a wrapper class for shader objects used by OpenGL, the basics of which are explained in Section 4.2.3. Each `OpenGLShader` object represents exactly one shader of any type. In this project, the shader code is saved in separate files with the extensions `.vert` and `.frag`, representing vertex and fragment shaders respectively. The `LoadShader()` method can be used to load and compile a shader by providing the file name and shader type. The code for the standard vertex shader used by all 3D models in the scene is as follows:

```
1 #version 330 //used GLSL version
2
3 layout (location = 0) in vec4 inPosition; //input vertex position
4 layout (location = 1) in vec4 color; //input vertex color
5 layout (location = 2) in vec4 inNormal; //input vertex normal
6
7 smooth out vec4 theColor;
8
9 uniform struct Matrices { //3D transformation matrix
10   mat4 projectionMatrix;
11   mat4 modelMatrix;
12   mat4 viewMatrix;
13 } matrices;
```

```
14
15 void main()
16 {
17   vec4 vEyeSpacePosVertex = matrices.viewMatrix*matrices.modelMatrix*
         inPosition;
18   gl_Position = matrices.projectionMatrix*vEyeSpacePosVertex; //3D
         transformed rendering position
19   theColor = color; //color passed to fragment shader
20 }
```

It receives the 3D position, color and normal for each `Vertex`, and calculates the new position by multiplying it with the provided 3D transformation matrix. The unchanged color is passed on to the fragment shader, which is defined as follows:

```
1 #version 330 //used GLSL version
2
3 smooth in vec4 theColor; //color received from vertex shader
4
5 uniform vec4 pickColor; //highlight color used when picked up
6 uniform bool colorPicking = false; //color picking active
7 uniform bool highlight = false; //currently highlighted
8 uniform float alpha;
9
10 out vec4 outputColor;
11
12 void main()
13 {
14   if (colorPicking)
15     outputColor = pickColor;
16   else if (highlight)
17     outputColor = theColor + vec4(0.1,0.0,0.0,0.0); //add highlight color
18   else
19   {
20     outputColor = theColor;
21     outputColor.w = alpha;
22   }
23 }
```

In case of standard rendering, the fragment shader simply outputs the unchanged vertex color. If the uniform `highlight` indicates that the current vertex belongs to a selected model, it returns the specified highlight color. By adjusting `alpha`, the opacity of the mesh can be changed. The uniform `colorPicking` is used during 3D selection, which is explained in Section 4.8. The developed system contains two additional sets of fragment and vertex shaders, one of which is used for font rendering, while the other one is used for orthographically projected 2D icons.

The `OpenGLShaderProgram` class is a wrapper for OpenGL shader programs and consists of one or more `OpenGLShader` objects. Shaders can be added by using `AddShaderToProgram()`. The program can consist of any number of shaders, which have to be linked by calling `LinkProgram()`. Several overloaded `SetUniform()` methods are provided that can be used to set shader uniforms. Each renderable object

in the scene contains its own `OpenGLShaderProgram`, which can be used by calling `UseProgram()` before and `UnUseProgram()` after rendering.

4.5.5 Renderer

The different subtypes of `OpenGLRenderer` are used to handle the entire rendering process for a specific application stage. An instance of every subtype is instantiated and subsequently managed by `GraphicsController`. The `Initialize()` function initializes all required OpenGL parameters (e.g. enabling depth tests and defining the depth range) and initializes the `OpenGLText` object, which is shared by all instances of `OpenGLRenderer` and used to render text. In addition, several static overlays are instantiated as `MeshContainer2D`. These semi-transparent backgrounds are used to enhance the visibility of status messages.

Whenever the system wants to render the current scene, `GraphicsController` calls the `Render()` function belonging to the renderer corresponding to the currently active application stage once every frame. The three required parameters are a reference to the controller itself, the 3D scene data as well as the 2D icon data. The method is implemented as follows:

```
1 void OpenGLRenderer::Render(GraphicsControl& _glControl, ModelData&
      _modelData, IconData& _iconData)
2 {
3   PrepareRender(_glControl); //prepare viewport (clear buffers, set
        background color and other parameters)
4
5   _modelData.Draw(_glControl.GetProjectionMatrix(), _glControl.
        GetViewMatrix()); //draw 3D models
6   _iconData.Draw(_glControl.GetViewportWidth(), _glControl.
        GetViewportHeight()); //draw 2D icons
7
8   SubRender(_glControl, _modelData, _iconData); //stage-specific rendering
9
10  FinishRender(_glControl); //swap buffers for actual rendering
11 }
```

The specific implementation of `SubRender()` depends on the active subtype of `OpenGLRenderer`. Examples of mode-specific rendering logic include drawing the cut-plane in `PlaneCutRenderer` and text overlays in `PlaneSelectionRenderer`. Whenever the application is busy processing data internally, `ShowStatusOverlay()` renders the current system status.

4.6 Navigation

As explained in Section 3.8.1, two navigation modes are available. One uses an orbital camera, the other uses physical position and orientation of the Kinect.

In general, a camera in three-dimensional space is represented by the projection and view matrix. In combination with the model matrix, which contains information

about a specific model's local position and orientation, these matrices compose the 3D transformation matrix used to transform every single triangle mesh into three-dimensional space. The projection matrix contains information about the camera frustum, i.e. field of view, aspect ratio, near and far plane. Since these properties do not change between different navigation modes, only the view matrix needs to be adjusted in order to switch between an orbital camera and navigation through physical reorientation. The three transformation matrices are already explained in Section 4.5.1. The following sections cover some important details about the implementation of both the orbital and sensor camera.

4.6.1 Orbital Camera

`GraphicsCamera` is the class responsible for orbital navigation. Most matrices and vectors in this component are stored as `glm` data types, since many methods utilize functions provided by the OpenGL Mathematics (GLM) library [24]. The most important member variables in this class are as follows:

```
1 glm::vec3 camPosition; //current 3D position
2 glm::vec3 camLookAt; //target point in 3D space
3 glm::vec3 camUpDirection; //up vector
4
5 float zoomFactor = 1.0f; //current zoom factor, with 1 being the default
      (100%)
6 float zoomStep = 0.2f; //zoom adjustment step size
7
8 float orbitXStep = 0.2f; //horizontal rotation step size
9 float orbitYStep = 0.15f; //vertical rotation step size
```

The `GetViewMatrix()` function returns the current view matrix and is called by `GraphicsController` every frame the orbital camera is active. It utilizes the method `glm::lookAt()` in order to calculate the view matrix with the current camera position, target point and up vector as parameters. The following is a shortened version of `GetViewMatrix()`:

```
1 glm::mat4 GraphicsCamera::GetViewMatrix()
2 {
3   if(/* ...scroll gesture detected... */)
4     Zoom(scroll_direction);
5   if(/* ...drag gesture detected... */)
6     Orbit(orbit_amount);
7
8   return glm::lookAt(CalculateCameraPosition(), CalculateCameraLookAt(),
        camUpDirection); //calculate view matrix
9 }
```

`Zoom()` increases or decreases the current zoom factor depending on the scroll direction. `Orbit()` rotates the camera pose whenever a drag gesture is detected:

```
1 glm::mat4 GraphicsCamera::Orbit()
```

```
2 {
3   float cursorOffsetY = (previousCursorPosition.y - currentCursorPosition.y
        ) * orbitYStep; //vertical cursor movement since last frame
4   float cursorOffsetX = (previousCursorPosition.x - currentCursorPosition.x
        ) * orbitXStep; //horizontal cursor movement since last frame
5
6   glm::vec4 camRight = glm::cross(camUpDirection, camDirection); //camera
        right vector
7
8   glm::mat4 yRotation = glm::rotate(cursorOffsetY, camRight); //calculate
        vertical rotation matrix
9   glm::mat4 xRotation = glm::rotate(cursorOffsetX, camUpDirection); //
        calculate horizontal rotation matrix
10  camPosition = (yRotation * (camPosition - camLookAt)) + camLookAt; //
        apply vertical rotation around lookAt location to camera position
11  camPosition = (xRotation * (camPosition - camLookAt)) + camLookAt; //
        apply horizontal rotation
12
13  camUpDirection *= yRotation; //rotate up vector
14  camUpDirection *= xRotation; //rotate up vector
15 }
```

Note that for simplification purposes, several straightforward conversions between different vector types have been omitted from the snippet. Finally, the camera's actual position and view point, which are used for `glm::lookAt()`, are determined with the help of all previously calculated parameters:

```
1 glm::vec3 GraphicsCamera::CalculateCameraPosition()
2 {
3   return currentCamPosition + (camDirection * zoomFactor) + (camRight *
        completeStrafeX) + (camUpDirection * completeStrafeY);
4 }
5
6 glm::vec3 GraphicsCamera::CalculateCameraLookAt()
7 {
8   return currentCamLookAt + (camRight * completeStrafeX) + (camUpDirection
        * completeStrafeY);
9 }
```

4.6.2 Sensor Camera

As explained in Section 2.2.2, the tracking step of the KinectFusion implementation calculates a camera pose and corresponding view matrix for each frame. This matrix can be used directly for navigation during the scanning stage, where the reconstruction is being displayed with Direct3D. The scanner still performs Kinect tracking in the background during scene manipulation, without actually adding more vertices to the reconstruction volume. This enables the stage to utilize the sensor view matrix for navigation as well.

However, the view matrix calculated by KinectFusion is in a left-handed coordinate system as used by Direct3D, whereas OpenGL uses a right-handed coordinate system. Therefore, the matrix provided by KinectFusion has to be converted prior to using it in other parts of the application. `GetTransformedCameraMatrix()` in the `KinectFusion` class takes care of this conversion and returns a view matrix representing the current sensor pose that is usable by OpenGL.

4.7 Scanning

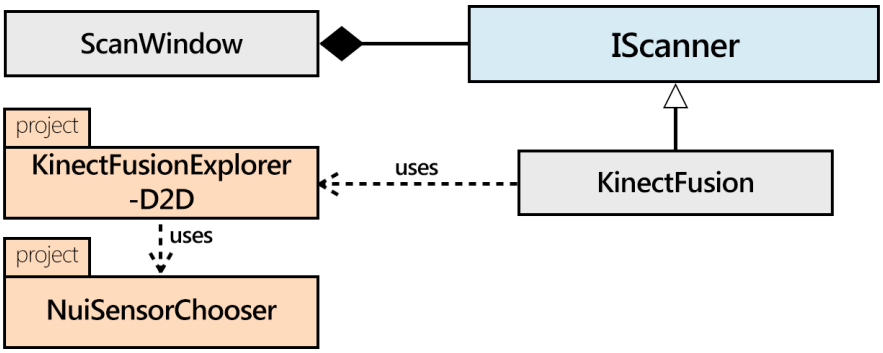


Figure 4.5: Components used for scanning.

The implemented prototype uses two external projects that are part of a KinectFusion example application from the Microsoft Kinect for Windows Developer Toolkit 1.8 [48]. The project *NuiSensorChooser* handles the recognition and initialization of the Kinect camera and has not been modified aside from removing unnecessary components.

KinectFusionExplorer-D2D, on the other hand, provides a complete implementation of the KinectFusion algorithm as described in Section 2.2. Its `KinectFusionParams` header file contains all relevant parameters for the reconstruction procedure, which have been adjusted slightly to satisfy the specific needs of the implemented application and to improve the overall performance. The resolution of the reconstruction volume can also be adjusted in this file by changing the voxels per meter as well as the voxel count in all dimensions. The impact of different parameters and the specific values used in this prototype are explained in Section 3.4.

The `KinectFusionProcessor` class in `KinectFusionExplorer-D2D` contains the methods responsible for the reconstruction algorithm. Aside from a few concurrency improvements and adding a method that returns the current camera matrix for navigation purposes, no major modifications have been made to this class. For more information about these projects, readers are referred to the official documentation by Microsoft [51].

The `ScanWindow` class (see Section 4.4.1) is responsible for the initialization and subsequent use of the 3D reconstruction procedure. It communicates with an implementation of the `IScanner` interface, which is deliberately kept very abstract in order to allow the use of different sensors or reconstruction algorithms in possible future releases.

The implementation used in this prototype is `KinectFusion`, which handles the entire communication to the two external projects mentioned before. It is also responsible for rendering the current reconstruction as well as retrieving the resulting triangles and vertices in appropriate formats for further use.

`Initialize()` has four `HWND` parameters: The first one is the handle to the parent window, while the other three are used to display depth information, the reconstruction volume itself and tracking residuals respectively. The method initializes the interface elements and prepares them for rendering, while also starting the reconstruction loop in `KinectFusionProcessor` in its own thread. `HandleCompletedFrame()` fetches the current reconstruction volume from the processor and renders it to corresponding interface elements in every single frame. In addition, the reconstruction parameters are updated whenever changes are made either by users or the application itself. The methods `PauseRendering()` and `ResumeRendering()` can be used to pause or resume the rendering of the reconstruction, without actually stopping the KinectFusion algorithm. This is used during application stages that utilize the physical orientation of the Kinect sensor for navigation, even though a modified virtual scene is being rendered. `PauseProcessing()`, on the other hand, can stop the reconstruction entirely, while `ResetScan()` restarts the scanning procedure. `FinishMeshSave()` converts the current reconstruction volume to vectors of `Triangle` and `Vertex` elements (see Section 4.5), which can then be obtained and used to construct a triangle mesh for further use in subsequent application stages.

4.8 Selection

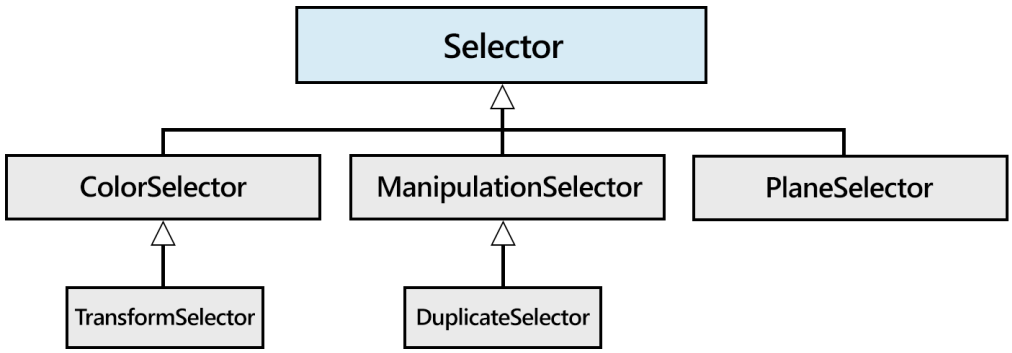


Figure 4.6: Class hierarchy of selection classes.

`Selector` provides abstract methods and variables used for object selection and scene interaction. The `HandleSelection()` method identifies the current state of the left mouse button (or the user's touch input) and forwards the current scene data to one of the abstract methods `HandleLeftMouseClick()`, `HandleLeftMouseDown()` or `HandleLeftMouseRelease()`. These functions are implemented in various child classes, each of which represents a specific type of interaction, such as transformation,

duplication or simple color highlighting. The `Selector` base class also provides an implementation of *color picking* and *raycasting*, both of which are techniques used for object selection. This section explains these methods as well as the most important child classes in more detail.

4.8.1 Color Picking

The basic idea behind color picking is to assign each pickable object in the scene a unique color that does not change during its lifetime. Since the prototype utilizes a 24-bit color scheme for rendering, there are 2^{24} or about 16.8 million distinct values available. Whenever a user selects anything inside the viewport, the application renders every cluster with its unique color. By utilizing OpenGL's `glReadPixels()` method, the color value of the pixel under the current cursor position can be obtained. Due to each color code being unique, this makes it possible to identify selected objects. The framebuffer is immediately cleared after selection and therefore never swapped, which means that users will not notice this procedure at all.

In this prototype, color codes are simply incrementing integer values assigned whenever a `MeshContainer` is created and added to a `ModelData` object (see Section 4.5 for details). The `ColorCoder` namespace provides static helper methods for conversion between integer codes and RGB colors for object picking:

```
1  namespace ColorCoder
2  {
3    static int ColorToInt(int r, int g, int b)
4    {
5      return (r) | (g << 8) | (b << 16);
6    }
7
8    static glm::vec4 IntToColor(int index)
9    {
10     int r = index & 0xFF;
11     int g = (index >> 8) & 0xFF;
12     int b = (index >> 16) & 0xFF;
13
14     return glm::vec4(float(r) / 255.0f, float(g) / 255.0f, float(b) / 255.0
           f, 1.0f); //divide by 255 for OpenGL
15   }
16 }
```

The methods utilize standard bit manipulation in order to convert each integer to a unique color with red, green and blue components, and the other way around. Note that OpenGL represents colors as floating point values between 0 and 1, so each component has to be divided by 255.

Whenever the user clicks on a pixel inside the OpenGL viewport, the integer color code of the object directly below the cursor can be received by calling the `GetIndexOfMeshUnderCursor()` method of `Selector`. During this procedure, every object is rendered with its unique color by setting the respective shader uniforms

`pickColor` and `colorPicking` accordingly (see Section 4.5.4 for details about these data types). The `GetColorCodeUnderCursor()` method subsequently reads the pixel color under the cursor and converts it to an integer code as follows:

```
1  int Selector::GetColorCodeUnderCursor(HWND _windowHandle)
2  {
3    //retrieve cursor coordinates
4    POINT cursorPos;
5    GetCursorPos(&cursorPos);
6    ScreenToClient(_windowHandle, &cursorPos);
7
8    //OpenGL uses inverted y screen coordinates, so they need to be adjusted
9    RECT rect;
10   GetClientRect(_windowHandle, &rect);
11   cursorPos.y = rect.bottom - cursorPos.y;
12
13   //read color under cursor
14   BYTE colorByte[4];
15   glReadPixels(cursorPos.x, cursorPos.y, 1, 1, GL_RGB, GL_UNSIGNED_BYTE,
         colorByte);
16
17   //convert color to integer code
18   int output = ColorCoder::ColorToInt(colorByte[0], colorByte[1], colorByte
         [2]);
19
20   return output;
21 }
```

The returned color code identifies the selected object. If the code corresponds to the viewport's current background color, no selection is performed.

4.8.2 Raycasting

While color picking is useful to identify the object under the mouse cursor, some operations need to know the exact 3D coordinates of a selected point in the reconstructed scene. To this end, the application projects a ray into the scene, starting from the current cursor position on the camera's near plane and going all the way to the projected end point on the far plane. If it intersects with any triangle of the reconstructed 3D model, the exact hit point coordinates can be obtained. In case of multiple intersections, only the one closest to the current camera position is chosen as the intended selection.

The `GetRayCastFromCursor()` method of `Selector` is responsible for the projection of the ray. It uses `glm::unProject()` with the cursor position, view and projection matrix as well as the current size of the OpenGL viewport as parameters in order to obtain the ray's start and ending point. To increase the performance of user selection, the developed prototype utilizes raycasting in conjunction with color picking. First, the selected object is determined by using color picking as described in Section 4.8.1. Afterwards, instead of inspecting the whole scene, only the triangles belonging to the identified object are checked for intersections to obtain the exact hit point coordinates.

The method `GetHitPoint()` with the projected ray as input parameter is used for this purpose.

4.8.3 Duplication and Manipulation

During the manipulation mode, users are able to move objects around and place them in another location. In addition, they can drag and drop items over the trash bin to remove them from the scene entirely. These interactions are implemented in `ManipulationSelector`, which inherits from the abstract `Selector` base class. `HandleLeftMouseClick()` retrieves the currently selected object index through color picking. The concrete implementation of `HandleLeftMouseDown()` in a slightly shortened version is as follows:

```
1  void ManipulationSelector::HandleLeftMouseDown(int selectedIndex, ModelData
       & _modelData, /* ... */)
2  {
3    if (selectedIndex != -1) //if any object is selected
4    {
5      int indexOfMeshUnderCursor = GetIndexOfMeshUnderCursor(/* ... */);
6      Ray cursorToFarPlaneRay = GetRayCastFromCursor(/* ... */);
7
8      if (indexOfMeshUnderCursor != -1) //if any object is under the cursor
9      {
10       Vertex hitPoint = _modelData.GetHitpoint(indexOfMeshUnderCursor,
             cursorToFarPlaneRay); //retrieve hit point through raycasting
11       std::vector<int> orientation = GetVertexOrientationFromRayPerspective
             (hitPoint, cursorToFarPlaneRay); //calculate surface orientation
12       _modelData.TranslateMeshToPoint(_selectedIndex, hitPoint, orientation
             ); //translate selected mesh
13
14     }
15     else
16     {
17       _modelData.TranslateMeshToCursorRay(_selectedIndex,
             cursorToFarPlaneRay, 5); //attach selected object to cursor, in
             front of the camera
18     }
19   }
20 }
```

Whenever a user has picked up an object, this method continuously checks if the dragging finger is over another, not currently selected cluster. If no suitable surface is detected, the carried object is attached to the cursor just in front of the camera. Otherwise, raycasting is used to calculate the exact hit point. Whenever an object is attached to another cluster, it is positioned in a way that avoids triangle intersections entirely. This means that every single vertex of the repositioned object ends up on the same side of the surface they are placed upon. For example, if an item is repositioned on the floor, it should be placed on top of that surface in its entirety. For this purpose, `GetVertexOrientationFromRayPerspective()` determines the orientation of the

calculated hit point relative to the current camera perspective. Using this information, `TranslateMeshToPoint()` translates the selected mesh to the hit point.

`HandleLeftMouseRelease()` drops the currently selected mesh in its current location or resets its position if it is released over an empty part of the scene. Whenever a mesh is dropped while dragging it over the trash bin icon, it is removed.

The `DuplicateSelector` inherits from `ManipulationSelector` and handles object duplication. The component clones items on pickup and performs all subsequent manipulations on the created copy instead of the original.

4.8.4 Highlighting and Transformation

The `ColorSelector` class inherits from `Selector` and highlights the currently selected object by temporarily adding red to their original surface color. This technique is used during the application's processing stage.

`TransformSelector` inherits from `ColorSelector` and adds the possibility of object rotation and scaling. `HandleLeftMouseDown()` checks whether or not any item has been selected and subsequently performs transformations depending on user input. Scaling is handled by `HandleScale()`, which adjusts the size of an object whenever scroll input is detected. `HandleRotation()` is responsible for rotating objects with drag gestures. Rotations are always relative to the current camera perspective. The code responsible for object rotation is as follows:

```
1  void TransformSelector::HandleRotation(GraphicsControl& _glControl,
       ModelData& _modelData, int _selectedIndex)
2  {
3    //get current cursor position
4    POINT pCur;
5    GetCursorPos(&pCur);
6    float offSetX = (float)((pCur.y - oldPosY) * 0.2f);
7    float offSetY = (float)((pCur.x - oldPosX) * 0.2f);
8
9    if (firstClick) //only if an old cursor position is available
10   {
11     //use camera view matrix as rotational axes
12     glm::mat4 cameraMatrix = _glControl.GetViewMatrix();
13     glm::vec3 horizontalRotation = glm::vec3(cameraMatrix[0][0],
           cameraMatrix[0][1], cameraMatrix[0][2]);
14     glm::vec3 verticalRotation = glm::vec3(cameraMatrix[1][0], cameraMatrix
           [1][1], cameraMatrix[1][2]);
15
16     _modelData.RotateMeshAroundAxis(_selectedIndex, offSetX,
           horizontalRotation); //horizontal rotation
17     _modelData.RotateMeshAroundAxis(_selectedIndex, offSetY,
           verticalRotation); //vertical rotation
18   }
19
20   //store cursor position for next frame
21   oldPosX = pCur.x;
22   oldPosY = pCur.y;
```

```
23    firstClick = true;
24 }
```

The application ensures that the transformed object always stays attached to the surface it is currently placed upon, regardless of scale or orientation. This is achieved by using appropriate pivot points for scaling as well as automatic repositioning after rotation. For example, if users transform a chair placed on the floor of a room, it will always remain attached to the ground plane.

4.8.5 Plane Cut Interaction

`PlaneSelector` is another direct child class of `Selector` and is responsible for user interaction during the plane cut segmentation stage. `HandleLeftMouseClick()` retrieves the index and exact hit point of a selected mesh through color picking and raycasting and subsequently translates the cut plane to the calculated position. The plane's surface is always parallel to the currently active axis, which can be manually selected by the user. `HandleLeftMouseDown()` translates or rotates the plane depending on the selected transformation mode. Both of these transformations are performed based on current mouse or finger movement. Similarly to `TransformSelector` (see Section 4.8.4), rotations are always relative to the current camera perspective. Orientation adjustments are limited to 30 degrees in all directions. Whenever the plane has been repositioned, the `PlaneCutPreview()` method of `ModelData` visualizes the proposed segmentation using the calculated parameters of the cut plane.

4.9 Segmentation

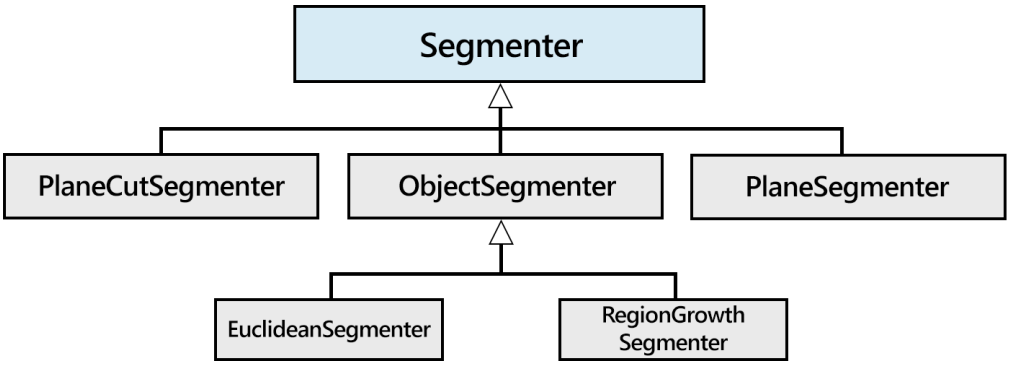


Figure 4.7: Class hierarchy of segmentation classes.

3D segmentation techniques are implemented in the abstract `Segmenter` component as well as the classes that inherit from it. Each concrete implementation has to be instantiated with a data structure that contains all custom parameters necessary for the specific segmentation technique. These options can be changed after initialization by

calling `SetSegmentationParams()` with an updated structure of the required type. `UpdateSegmentation()` takes a 3D model as an input parameter and segments it according to the chosen parameters. The resulting clusters can then be retrieved by calling `FinishSegmentation()`. The different segmentation techniques are implemented in their own `Segmenter` child classes. Every one of them provides a concrete implementation of `Segment()`, which contains the segmentation algorithm, as well as other technique-specific functions. This section covers each of the three clustering implementations in more detail. Since the developed system utilizes the Point Cloud Library (PCL) [71] for 3D segmentation, several manual conversions between data structures have to be performed, which are briefly explained as well.

4.9.1 Point Cloud Conversion

The segmentation techniques provided by the PCL operate on point clouds, while the developed prototype utilizes triangle meshes as data structures for 3D models. As a result, the data has to be converted before segmentation. `Segmenter` provides the `ConvertToPointCloud()` method for this purpose, which takes a `MeshContainer` object and converts it to a `PointCloud`.

The conversion to a point cloud is very straightforward: Every single distinct vertex in the triangle mesh simply has to be added to a new `PointCloud` object. However, since the individual cluster clouds have to be converted back to a `MeshContainer` after segmentation for subsequent use in the application, the mesh triangles have to be conserved as well. For this purpose, the original index of each vertex is stored in an unordered map with the `Vertex` structure as key. The required hash function has been implemented by utilizing the `hash_value()` and `hash_combine()` methods provided by the Boost Library [15]. It calculates a distinct index for each vertex using only its 3D coordinates. As a result, since the position does not change during segmentation, the original vertex index can be retrieved after finishing the clustering procedure. A second map is used to store triangle information for every vertex. Each entry consists of the original index as key and a list of triangles that contain the corresponding vertex. By using these data structures, the application is able to reconstruct original triangles in each resulting cluster as described in Section 4.9.5.

4.9.2 Plane Segmentation

The `PlaneSegmenter` class provides methods and variables needed for Plane Model Segmentation using RANSAC, which is explained in detail in Section 2.3.1. Objects of this type are initialized with `PlaneSegmentationParams`, which is a structure that contains user-chosen parameters needed for this segmentation (i.e. plane smoothness and thickness). As mentioned in Section 3.6.1, thickness determines how close a point must be to the resulting plane in order to be considered a part of it. Smoothness represents the maximum angle allowed between the normal vectors of two vertex neighbors in the plane.

The prototype uses the RANSAC implementation provided by the Point Cloud Library [63]. The `Segment()` method is the core part of the segmentation procedure and tries to find the biggest planar surface and its parameters in the remaining point cloud according to certain constraints. First, it detects the biggest plane parallel to the world's x-axis. If no surface that contains more than $\frac{1}{15}$ of the whole point cloud is found, the z- and the y-axis are checked. If no plane can be found after these iterations, a final check without parallelism constraints is performed. This sequencing increases the likelihood of identifying the floor, ceiling or walls. Algorithm 4.1 shows the individual steps of this procedure in more detail.

Algorithm 4.1: Algorithm used in `PlaneSegmenter::Segment()`.

Input: Point cloud C, segmentation parameters S
Output: Plane indices I, plane coefficients P

```
1  axis a = x;
2  for i = 0 to 3 do
3      while true do
4          if i < 3 then
5              Find biggest plane in C parallel to a with RANSAC;
6          else
7              Find biggest plane in C with RANSAC;
8          end
9          Create I_temp with plane indices;
10         Create P_temp with plane coefficients;
11         if size of I_temp <= (size of C)/15 then
12             break;
13         end
14         Add I_temp to I;
15         Add P_temp to P;
16     end
17     Set a to next axis (x → z → y);
18 end
```

If a planar surface is found, the inlier vertices are highlighted in the rendered scene with `UpdateHighlights()` and proposed to the user. If the segmentation is confirmed, `ConfirmLastSegment()` adds all vertices that are part of the found plane to a cluster. Afterwards, `Segment()` is called again for the remaining point cloud. If the proposal is rejected, `RejectLastSegment()` calls `Segment()` without creating a new cluster. Instead, specific constraints are used to make sure that the rejected surface is not proposed again. If no more planes can be found, the procedure terminates and the reconstructed scene is updated by calling `FinishSegmentation()`.

Note that the prototype assumes the first detected surface to be the ground plane, which is then used to align the cut plane during the segmentation step described in Section 4.9.4. This assumption can only be made due to the recommended starting

position at the beginning of the scanning stage (see Section 3.4). This increases the likelihood of the floor being nearly parallel to the world's x-axis. However, in order to be more flexible, the ground plane detection could be improved in future releases.

4.9.3 Object Segmentation

The different object clustering techniques are implemented in `ObjectSegmenter` and its child classes. The base class provides a concrete implementation of the method `UpdateHighlights()`, which is used to let users preview their segmentation before actually applying it to the scene. This is achieved by assigning a unique color to each of the resulting clusters in the rendered scene. Users can therefore see the effect of their parameter choices and decide whether or not further adjustments are necessary. On confirmation, `FinishSegmentation()` performs the chosen segmentation and the application continues to the next stage.

The procedures provided by `ObjectSegmenter` are only performed on parts of the scene that have not been recognized as planar surfaces in the previous segmentation step described in Section 4.9.2. The different clustering techniques are implemented in two classes that inherit from `ObjectSegmenter`, both of which are explained in the following sections.

Euclidean Segmentation

One of the two available object segmentation methods is Euclidean Cluster Extraction, which segments a three-dimensional scene based on the distance between neighboring vertices. The general approach and algorithm of this technique is explained in Section 2.3.2. `EuclideanSegmenter` is the component containing the implementation of this clustering method. Each instance of this type has to be instantiated with `EuclideanSegmentationParams`. This structure contains the two user-defined parameters used for segmentation, *cluster tolerance* and *minimum cluster size*, the effects of which are explained in Section 3.6.2.

The developed application utilizes the Euclidean Cluster Extraction implementation provided by the Point Cloud Library [62]. The `Segment()` method is responsible for initializing and performing the actual segmentation:

```
1  bool EuclideanSegmenter::Segment()
2  {
3    pcl::EuclideanClusterExtraction<pcl::PointXYZRGB> ec;
4
5    //set segmentation parameters (EuclideanSegmentationParams)
6    ec.setClusterTolerance(segmentationParameters.clusterTolerance);
7    ec.setMinClusterSize(segmentationParameters.minComponentSize);
8
9    //execute segmentation on point cloud, extract cluster indices
10   ec.setInputCloud(mainCloud);
11   ec.extract(segmentationClusterIndices);
12
13   if (segmentationClusterIndices.size() == 0)
```

```
14      return false;
15
16    return true;
17  }
```

The `extract()` method performs the extraction and returns the vertex indices for each detected cluster. Euclidean segmentation is the default clustering technique and tends to yield the most consistent results for three-dimensional reconstructed scenes.

Region Growing Segmentation

`RegionGrowthSegmenter` contains the implementation of the Region Growing Segmentation method. This technique starts from a set of selected seed points and continuously adds neighboring vertices to the same clusters based on certain criteria. The generalized approach and algorithm is explained in Section 2.3.3.

This prototype utilizes the implementation provided by the Point Cloud Library [64]. In this version, the algorithm starts by selecting a seed point with the lowest curvature value. Then, the algorithm finds the angles between the current seed point's normal and the normals of neighboring vertices and adds neighbors to the cluster if they are below a specified threshold. Afterwards, if any neighbor has a curvature value that is below a certain value, it is added to the collection of current seed points. The cluster grows along these seeds until there are no more neighbors left that have a low enough curvature. Then, the algorithm starts again by selecting a new seed from the remaining vertices and grows the next region in an analogous fashion. This continues until every point in the cloud has been assigned to a cluster.

Instances of the type `RegionGrowthSegmenter` have to be constructed with the structure `RegionGrowthSegmentationParams`. It contains all user-chosen parameters necessary for the segmentation, including the mentioned curvature and angular thresholds. The different options are explained in Section 3.6.2. The `Segment()` method, which is responsible for initializing and performing the segmentation technique, is implemented as follows:

```
1  bool RegionGrowthSegmenter::Segment()
2  {
3    //initialize k-d tree structure
4    pcl::search::Search<pcl::PointXYZRGBNormal>::Ptr tree = boost::shared_ptr
         <pcl::search::Search<pcl::PointXYZRGBNormal> >(new pcl::search::
         KdTree<pcl::PointXYZRGBNormal>);
5
6    pcl::PointCloud<pcl::Normal>::Ptr temporaryNormalCloud(new pcl::
         PointCloud<pcl::Normal>);
7
8    /* ...initialization of cloud containing vertex normals... */
9
10   pcl::RegionGrowing<pcl::PointXYZRGBNormal, pcl::Normal> reg;
11
12   //choose k-d tree structure for neighborhood search
13   reg.setSearchMethod(tree);
```

```
14
15   //set segmentation parameters
16   reg.setMinClusterSize(segmentationParameters.minComponentSize);
17   reg.setNumberOfNeighbours(segmentationParameters.numberOfNeighbors);
18   reg.setCurvatureThreshold((segmentationParameters.curvatureThreshold /
         10.0));
19   reg.setSmoothnessThreshold((segmentationParameters.smoothnessThreshold /
         10.0) / 180.0 * M_PI); //convert to radians
20
21   reg.setInputCloud(mainCloud);
22   reg.setInputNormals(temporaryNormalCloud);
23   reg.extract(segmentationClusterIndices);
24
25   if (segmentationClusterIndices.size() == 0)
26     return false;
27
28   return true;
29 }
```

In order to search for nearest neighbors of seed points, a k-d tree data structure, as explained in Section 2.3.2, is used during segmentation. The method `extract()` performs the cluster extraction and returns the vertex indices for each resulting region.

4.9.4 Plane Cut Segmentation

This segmentation technique cuts an object in two along a prepositioned plane. The two resulting clusters can then be subdivided further to continuously improve the segmentation. This optional tool is especially useful in order to split distinct objects that have been wrongly segmented into the same region by a prior Euclidean Cluster Extraction due to being too close to each other. However, it can also be used as a main segmentation tool if the objects that should be divided can be split along a plane and do not overlap each other.

The technique is implemented in `PlaneCutSegmenter`, the instances of which require a `PlaneCutSegmentationParams` object. This structure contains the parameters of the manually positioned cut plane, i.e. a point on the surface and the plane normal. `Segment()` performs the actual segmentation by using the following algorithm:

Essentially, the method utilizes the plane equation in order to determine the position of each vertex relative to the cut plane, i.e. whether it is on one side or the other [83]. The point indices are then assigned to the corresponding cluster. `UpdateHighlights()` assigns each of the two resulting clusters a unique color in order to provide an appropriate segmentation preview.

4.9.5 Triangle Mesh Conversion

After a successful segmentation, `EstimateIndices()` constructs the individual point clouds for each cluster with the index lists provided by the `Segment()` method. In

Algorithm 4.2: Plane Cut Segmentation.

Input: Point cloud C, cut plane parameters P
Output: Segmented clusters S_1, S_2

```
foreach point p_i ∈ C do
    dot = p_i.x * P.x + p_i.y * P.y + p_i.z * P.z − P.d;
    if dot > 0 then
        Add p_i to S_1;
    else
        Add p_i to S_2;
    end
end
```

addition, the original vertex indices are retrieved from the map constructed during point cloud conversion, as described in Section 4.9.1.

In order for the rest of the application to use the segmented cluster data, the point clouds have to be converted back to triangle meshes in `ConvertToMesh()`. The vertices can be converted without any modification. Faces, however, have to be recreated using the original vertex indices and the stored triangle map, which contains information about all the faces connected to a specific vertex (see Section 4.9.1).

The procedure loops through every vertex of a cluster and retrieves all faces it used to be a part of in the unsegmented mesh. It then looks for all the vertices these triangles consisted of in the current cluster. If all vertices of a triangle are found, the face can be reconstructed without any modifications to the vertex data. However, some triangles in the original mesh might have been split up during the segmentation procedure. In these cases, the missing vertices are duplicated and subsequently added to each of the two clusters in order to recreate the original faces. Although this does lead to redundant vertex data, this approach is preferred as it ensures the preservation of the entire mesh surface. The simplified procedure for the implemented triangle reconstruction is shown in Algorithm 4.3.

4.10 Export

The export of 3D models into different file formats is implemented in the abstract `ModelExporter` class. `SaveMeshToFile()` utilizes the VCG library to save a single mesh to a specific location in OBJ, OFF, PLY or STL format. The virtual method `Export()` takes a `ModelData` object as input parameter, which represents a three-dimensional scene and can contain more than one triangle mesh (see Section 4.5.2). This method is intended to be used for actual model export and therefore needs to be implemented in classes that inherit from `ModelExporter`.

The child class `DialogExporter` saves either the currently selected submesh or the entire reconstructed scene to a single file. `GetFileNameFromSaveFileDialog()`

Algorithm 4.3: Triangle reconstruction after segmentation for a single cluster.

Input: Cluster vertices V, original vertex cluster indices I, triangle map M
Output: Cluster vertices V, cluster triangles T

```
1  Retrieve original vertex indices i of V from I;
2  foreach i_i ∈ i do
3      Retrieve triangles T_temp from M by using i_i as index;
4      foreach t_i ∈ T_temp do
5          Create triangle t;
6          foreach vertex index c_i ∈ t_i do
7              if c_i ∉ i then
8                  Add missing vertex to V;
9              else
10                 Remove c_i from i;
11             end
12             Add c_i to t;
13         end
14         Add t to T;
15     end
16 end
```

opens a Win32 file save dialog in order to determine the desired storage location and format. This exporting option is used by default during normal application use.

`ScenarioExporter`, on the other hand, saves every single segmented cluster of the scene to a different file. The storage location and format cannot be changed. This class is used to export the individual objects of the scene for further use in other applications during the user study described in Chapter 5.

If a ground plane has been identified during plane segmentation (see Section 4.9.2), the exported model is reoriented in a way that aligns the surface normal of the ground plane with the world's y-axis. By doing so, the scene is moved into an upright position, which simplifies subsequent placement and modifications in other applications.

CHAPTER 5

Evaluation

In order to evaluate the usability and functionality of the developed system, a small user study was conducted. This chapter describes the setup and procedure of the different test scenarios as well as methods used to collect data during the evaluation. Afterwards, the obtained results are presented and discussed.

5.1 Participants

The system was evaluated with five participants between 23 and 27 years of age. Since the developed application is primarily intended for people who are able to use the reconstructed scene in other applications, all users had to be proficient with computers and mobile applications. None of the participants had used the developed system before. All users were reasonably experienced in interacting with virtual three-dimensional scenes, with three users having a background in computer science and/or game development. A pilot study with one additional participant was conducted prior to the main evaluation in order to identify possible problems regarding the test setup, scenarios and the application interface.

5.2 Procedure

At the beginning of the study, participants received information about the general purpose and handling of the developed system. They were also informed about the methods used for data collection and the overall procedure of the study.

Afterwards, the participants were asked to complete two different scenarios, each of which required them to use the developed system in order to complete a specified task. The application displayed user guidance messages during different stages that were designed to provide all instructions necessary to successfully complete the scenarios. In order to evaluate the effectiveness of these messages and to more thoroughly test the

intuitiveness of the system, participants did not receive additional instructions on how to use the application beforehand.

Both of the prepared scenarios were focused on different aspects of the system, even though they required similar steps in order to be completed. Users were expected to be able to finish the second task quicker than the first one due to an increased familiarity with the interface and most of the required actions.

The participants were observed during their interaction with the system in order to identify emerging problems. After completing both tasks, participants were given a questionnaire containing a standardized usability test as well as several additional questions about their interaction with the developed application.

5.3 Test Scenarios

Two test scenarios were prepared for the user study. Each of them required participants to scan, reconstruct and manipulate a small prearranged scene. Users received instructions on the expected outcome along with some useful hints on working with the required application tools.

The entire study was conducted at the same location to ensure equal conditions for every participant. The used area was well-lit and had enough space to move around during scanning and scene manipulation. Since the prototype requires a high-end graphics card, the system was connected to an appropriately equipped computer at all times (see Section 3.1.1 for the exact specifications). Even though both scenarios were focused on the same highlighted area, they had different item arrangements. Non-relevant objects in the immediate vicinity of the prepared scene were removed, if possible, in order to make the reconstruction procedure easier.

Most of the objects used during the scenarios were single-colored boxes or cylinders. This increased their visual distinguishability and made it easier to assess the quality of user-performed segmentations after finishing the scenarios.

5.3.1 Bowling

In the first scenario, participants were asked to transform the prepared scene (see Figure 5.1) into a virtual pin area of a bowling lane, consisting of around ten pins placed in a custom formation. To this end, the highlighted yellow area and all the objects on it had to be scanned and digitally reconstructed first. Then, the scene had to be segmented in order to separate distinct objects from the floor. The red box and blue cylinder were deliberately unnecessary parts of the scene that had to be digitally removed by the participant. Holes left open during the scanning procedure had to be closed in the processing stage. Finally, users were asked to use the prototype's scene manipulation tools to produce an arbitrary number of duplicates of the pin and place them in the virtual scene in a way that resembles a setup used in bowling. Participants were encouraged to make their finished scene recognizable by scaling pins differently or by placing them in a custom formation.

Figure 5.1: Setup for the first test scenario of the user study.

After successfully exporting the modified scene, participants had the opportunity to play a game of bowling in a small application developed with Unity [80] (see Figure 5.2). The reconstructed and manipulated pin formation was dynamically imported at the beginning of the game. The ground plane was removed automatically, while the pins were placed in a bowling alley and received basic physical behavior. Afterwards, test subjects were able to throw the bowling ball by using touch gestures in order to hit as many pins as possible. This game was primarily a reward for completing the first

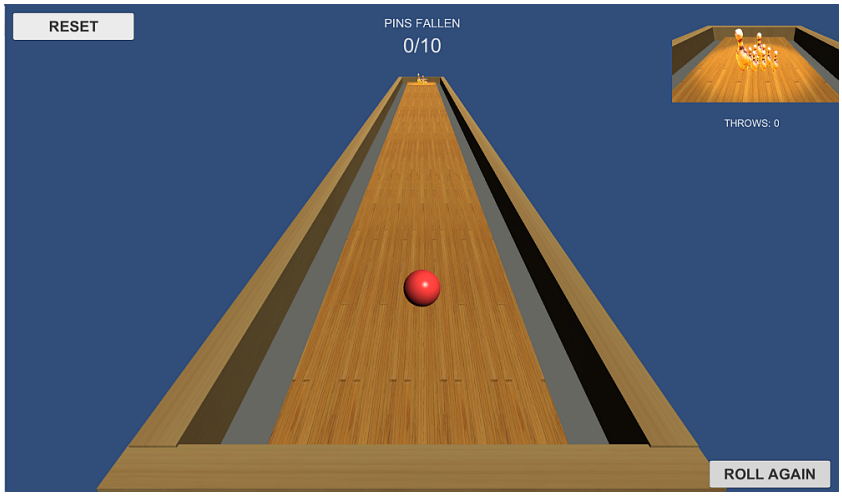


Figure 5.2: Bowling game during first test scenario. The pin setup that was prepared by the participant during the first phase was dynamically imported and integrated into the game.

scenario. Its outcome was therefore not relevant to the results of the study. However, the small application showcased the possibilities of using scanned and reconstructed scenes in other virtual environments.

The scenario required participants to use most of the prototype's functions without having to deal with a cluttered scene that might be difficult to segment or manipulate. The primary focus was to observe users' first interaction with the system and potential difficulties. Since the scene composition did not require plane cut segmentation, the respective application stage was automatically skipped.

5.3.2 Stacked Boxes

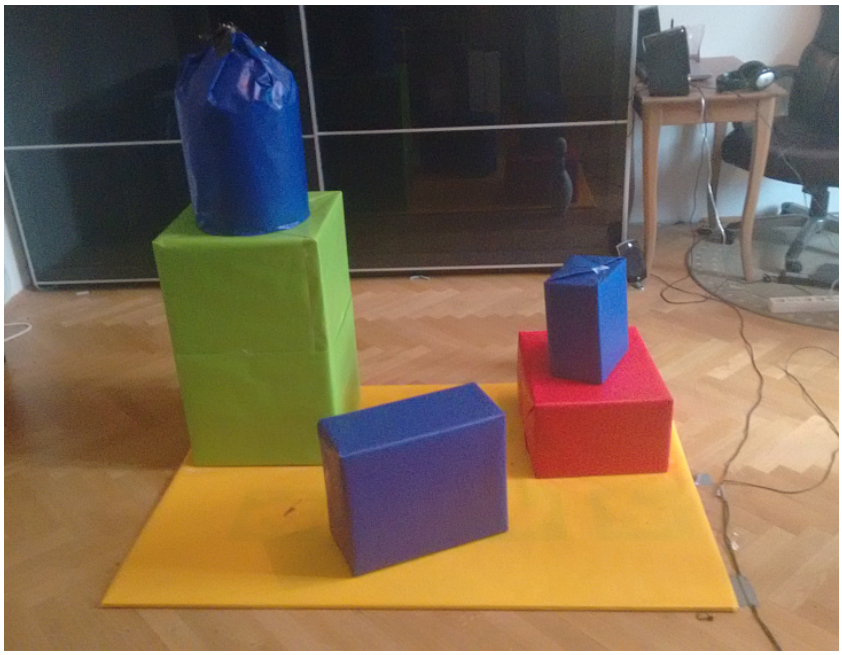


Figure 5.3: Setup for the second test scenario of the user study.

The second scenario was primarily focused on testing the effectiveness and usability of the segmentation tools provided by the developed system. Once again, users were asked to scan and reconstruct the highlighted yellow area and all the objects on it. The scene featured differently colored boxes and one cylinder that had to be segmented after reconstruction (see Figure 5.3).

In contrast to the first scenario, some of the objects were stacked in order to provide an additional level of difficulty. Users had to utilize the plane cut tools to properly separate some of the items from each other. Even though subsequent editing of the virtual reconstruction was not the primary task of this scenario, participants were still encouraged to try out different manipulation tools after performing a successful segmentation.

Finally, the segmented and manipulated scene had to be exported in order to finish the scenario. The application automatically created a model for the entire modified scene as well as for every individual object, which made subsequent visual assessment of the segmentation quality easier. Even though the second scenario was slightly more complicated than the first, participants were expected to finish it quicker due to an increased familiarity with most of the tools.

5.4 Data Collection

In addition to visually observing participants during the test scenarios, their interaction with the system was documented with pictures, screen captures and an application log. Further problems or questions that emerged during testing were documented manually. Questionnaires were used to gather feedback about specific aspects of the interaction with the developed prototype.

5.4.1 Screen Capture

The free and open-source *Open Broadcaster Software* (OBS) was used to record the screen during the two test scenarios in the user study [57]. The resulting videos helped the evaluation process by providing additional information in case of emerging problems.

5.4.2 Application Log

By utilizing the Boost Library (see Section 4.2.1), almost all events and interactions during application usage are written to a log file. Each entry is comprised of the current time stamp, application stage and a detailed event description. During the user study, a log was created for each completed scenario. These files can be helpful if emerging problems have to be analyzed more thoroughly, because the exact sequence of specific performed interactions can be reconstructed.

5.4.3 Exported Model

As the final step in each test scenario, participants were asked to export their modified scene as a 3D model. Upon pressing the respective button, the application automatically saves the entire scene as well as every individual object to different .obj files. These models can be qualitatively analyzed to assess how well participants were able to follow the instructions of each scenario and whether or not the results differ substantially from the expected output.

5.4.4 Questionnaire

The questionnaire handed out to participants after completing both scenarios consists of nine short sections: One for general questions, one for each application stage and one concerned with the hardware setup.

The general section starts with a ten-question *System Usability Scale* (SUS), a standardized usability test developed by John Brooke [10]. It has been used in a wide range of studies and has become a standard for usability evaluation [4]. It is composed of ten items concerned with a user's subjective assessment about the system, each of which has the form of a statement:

- Q1 I think that I would like to use this system frequently
- Q2 I found the system unnecessarily complex
- Q3 I thought the system was easy to use
- Q4 I think that I would need the support of a technical person to be able to use this system
- Q5 I found the various functions in this system were well integrated
- Q6 I thought there was too much inconsistency in this system
- Q7 I would imagine that most people would learn to use this system very quickly
- Q8 I found the system very cumbersome to use
- Q9 I felt very confident using the system
- Q10 I needed to learn a lot of things before I could get going with the system

For each statement, participants have to indicate the degree of agreement on a Likert scale, ranging from one (*strongly disagree*) to five (*strongly agree*). In order to avoid acquiescence bias, the ordering of items is alternated such that a positive statement is always followed by a negative one. With the results of these assessments, a final SUS score ranging from 0 to 100 can be derived, where a higher value indicates better system usability.

The general section contained six additional items specific to the developed system. These statements followed the ordering and rating principles of the System Usability Scale:

- Q11 I found the system to be visually appealing
- Q12 I found many interface elements and descriptions to be misleading
- Q13 I thought the help messages made it easier to understand and use the system
- Q14 I found it very difficult to produce good results with the provided tools
- Q15 I think that there are many practical applications for the system
- Q16 I personally would not find much use for the system

The remaining sections featured similar, system-specific items about individual aspects of the different application stages and the used hardware setup. Furthermore, participants had the opportunity to leave additional comments about aspects they particularly liked or disliked at the end of every section. The entire questionnaire is attached in Appendix A.

5.5 Pilot Study

Prior to the main evaluation, a short pilot study with one additional participant was conducted. During this test run, several minor technical issues were identified and subsequently fixed. In addition, the following changes to the evaluation setup and procedure were made based on the feedback gathered during the pilot study:

Figure 5.4: Evolution of the prototype.

- The participant reported difficulties holding the prototype while simultaneously interacting with the touch screen. Even though the device was reasonably heavy, it did not have handles or any other convenient way to carry it. Moreover, the system setup was not particularly visually appealing. As a result, the prototype was improved by placing both the depth camera as well as the tablet inside a painted cardboard box with appropriate cut-outs. Holes on the side of the box should make it easier to carry the box with one or two hands. Figure 5.4 depicts the evolution of the prototype.

- The user guidance messages displayed during the test scenarios were perceived as too long and complicated. As a result, almost the entire text was rephrased and shortened in order to focus on essential information.

- After completing the questionnaire, the participant had additional comments about several specific parts of the application, but did not find a suitable field on the form. As a result of this and in order to gather additional feedback about the system, the questionnaire was significantly extended. It originally only contained one page consisting of a System Usability Scale and general questions about the system. After the change, the questionnaire featured an additional section for each individual application stage as well as one page about the hardware setup.

5.6 Results and Discussion

This section presents the information gathered during the user study through observation, questionnaires and analysis of segmented 3D reconstructions created by participants. Furthermore, the obtained results are discussed and analyzed in detail in order to evaluate the prototype's usability and effectiveness as well as identify potential problems and areas of improvement.

5.6.1 Observations

All participants were able to finish both test scenarios without major hindrances. Since they received only limited instructions before approaching the tasks, they had to rely on the usefulness of user guidance messages as well as an intuitive interface during the individual application stages in order to correctly use the system. Participants were able to finish most of the necessary steps without having to ask for help. Only a few parts required additional hints in order to be completed. Problems that occurred during the evaluation as well as some of the most noteworthy observations are discussed in the following sections.

Initial Scan Position

Figure 5.5: Participants trying to find a perfect starting position for scanning.

Four out of five participants had at least some problems trying to figure out the correct starting position for the scanning procedure (as explained in Section 3.4). It seemed to be difficult to intuitively recognize whether or not the resulting bounding box contained the entire highlighted area. This lead to participants having to reset a few times and needing additional positioning hints before being able to find a reasonably good starting pose. This initial step of the scanning procedure should be improved by visualizing the bounding box at the beginning of the scan.

Furthermore, guidance messages specified that aligning the viewing direction of the camera with the floor would make subsequent segmentation easier. Even though participants seemed to find aligning the tablet in such a way somewhat cumbersome, all of them managed to do it correctly (see Figure 5.5). It should be noted that users are allowed to ignore this alignment constraint in the current implementation, even though doing so increases the time it takes the plane selection algorithm to find correct surfaces. The system could be improved such that camera alignment does not influence the efficiency and reliability of the plane segmentation procedure in any way. Essentially, users should be able to start their scan from any desired viewing direction without having to worry about subsequent segmentation.

Holes During Scan

Two participants unintentionally created big holes in their reconstruction during the second scenario by not thoroughly scanning objects in the scene from every angle. They stated that they were unable to spot these holes during the capturing process. As a result of the size and form of the resulting gaps, participants were unable to sufficiently close them during the subsequent processing stage, since the used algorithm is primarily suited for filling small planar holes. In order to improve the ability of users to detect missing surfaces during the scan, a more noticeable background color could be used, since gaps in dark gray seem to be difficult to spot. In a future version, holes could be automatically detected and highlighted even while capturing the scene. An improvement to the hole filling algorithm is also a possibility, even though leading users to a thorough scan should be prioritized as this typically produces the best results.

Prototype Handling

In general, participants seemed very comfortable while using the prototype. Even holding the device while simultaneously interacting with the touch screen did not result in any major problems. The changes to the prototype casing as a result of the pilot study (see Section 5.5) seemed to have sufficiently improved the handling of the system. However, the wiring of the device appeared to be a major nuisance, since it significantly restricted its mobility. During the scanning part of the scenarios, cables had to be kept out of the camera's field of view manually by the test supervisor. Additionally, since the application was running on the desktop computer and streamed to the tablet, some standard touch gestures (e.g. pinch, zoom) could not be properly mapped to the PC and had to be replaced by scrolling. It was apparent that this was not entirely intuitive for most participants. However, these circumstances are the result of the device being a prototype. Future versions could be run directly on a Microsoft Surface tablet [52] to improve gesture controls. In addition, a depth camera without the need for an external power supply could be used to remove the necessity of wires completely.

Unintentional Duplication

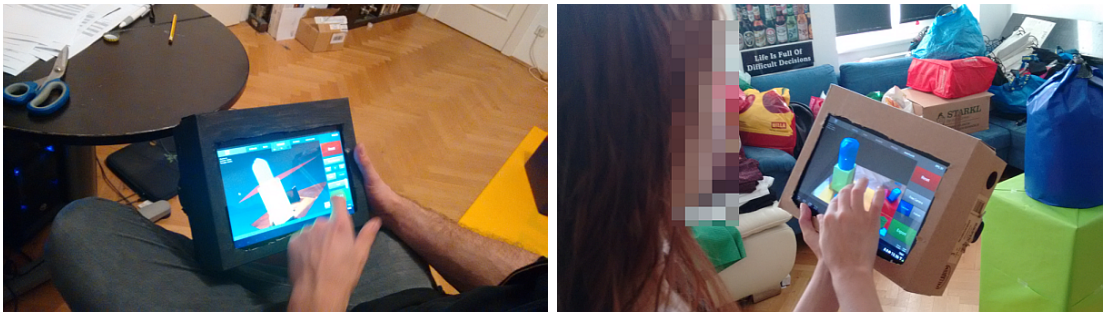
Two participants unintentionally produced multiple bowling pin duplicates in the same location during the first test scenario. An analysis of the application log in conjunction with screen captures revealed that this was due to problems with the display streaming software, which wrongly detected some drag and drop gestures as clicks and immediate releases. Instead of duplicates being dragged out of objects, they were created and immediately placed in their initial position, i.e. directly on top of the original item. As a result, participants did not notice that they already cloned the item and therefore created more duplicates until they were able to move it around. This led to an unnecessary high number of vertices in the scene and could cause problems when using the individual objects with physics in other applications, such as the bowling game in the first test scenario.

Even though this issue is a direct result of the prototype-like nature of the device, it still highlights a potential problem with the visualization of duplicates. The application should offer some way of seeing how many object replicas exist at any given time. Shifting the initial position of duplicates slightly could be another solution. This would make clones visible even if they are not manually repositioned.

Tracking Performance

Even though the entire study was conducted using the same hardware, one participant experienced some performance issues due to a problem with the screen capturing software, which reduced the frame rate to about 15 instead of 30-60 frames per second. Unfortunately, this issue was only noticed and resolved half way through the study. Even though the low frame rate did not affect all of the system's functionality, the tracking algorithm used during scanning and manipulation was significantly impaired and failed to estimate the position and orientation of the participant several times. Even though this particular issue is entirely caused by external software and therefore not relevant for the evaluation of this system's usability, it emphasized the need for a stable frame rate and adequate hardware. The tracking part of the ICP algorithm used by KinectFusion (see Section 2.2.2) requires at least 20-30 frames per second in order to guarantee smooth tracking of the camera pose.

User Experience



(a) Plane cut segmentation. (b) Scene manipulation (pilot study).

Figure 5.6: Participants interacting with the system during different application stages.

With the assistance of displayed guidance messages, participants seemed to be able to quickly understand and use the basic functions of the system without major problems (see Figure 5.6). In some cases, users had to try out tools before being completely sure about their purpose. This is, however, expected and even encouraged by providing the option of undoing accidental changes in every stage. The unique interaction and navigation through physical movement in the manipulation stage was not immediately apparent to three participants. They seemed to expect an orbital camera similar to those in the previous stages and therefore tried to navigate using touch gestures. It took some time

for them to notice that moving while looking at the scene changes the virtual perspective. After recognizing this, however, all participants seemed to intuitively understand this unusual method of scene interaction. Moreover, users gave the impression of enjoying the fact of the tablet display acting as a window into the virtual world. Participants also seemed to appreciate the inclusion of the small bowling game at the end of the first scenario. Dynamically utilizing their previously reconstructed and modified scene seemed to have been a successful way of rewarding their work as well as showcasing some of the possibilities of the developed system.

5.6.2 Questionnaire Analysis

Participants were asked to fill out a questionnaire after finishing both test scenarios. This section presents some of the most interesting findings and discusses their possible implications.

System Usability Scale

Interpreting scores of System Usability Scales can be difficult, since they can only be evaluated in a meaningful way when compared to the scores of other systems. Bangor et al. [3] added an adjective rating scale based on the analysis of a large number of usability tests. It assigns each range of scores a corresponding adjective as follows: *Worst Imaginable* (< 25), *Poor* (25 - 38), *Ok* (38 - 52), *Good* (52 - 73.5), *Excellent* (73.5 - 85) and *Best Imaginable* (85 - 100). The average SUS score is 70, which is also considered the minimum value needed in order to have acceptable usability [3].

Participant	**SUS Score**	**Learnability**	**Usability**
P-1	82.5	100.0	78.1
P-2	87.5	100.0	84.4
P-3	82.5	100.0	78.1
P-4	60.0	75.0	56.3
P-5	77.5	75.0	78.1
Mean	**78.0**	**90.0**	**75.0**
Std Dev	10.7	13.7	10.8

Table 5.1: Individual and mean System Usability Scale scores (including standard deviation) for the developed system. In addition, the table includes the results of the two-factor analysis for usability and learnability as suggested by Lewis et al. [39].

The system developed during this thesis scored 78 out of 100 points (standard deviation: 10.7), which indicates above-average, *excellent* usability. Lewis et al. [39] suggested splitting the scores into the two factors usability and learnability. According to this, the developed system scored 75 out of 100 for usability (standard deviation: 13.7) and 90 out of 100 for learnability (standard deviation: 10.8). The exact results for each participant are shown in Table 5.1.

It is noteworthy that the participant who gave the lowest SUS score (P-4) is also the one who experienced significant frame rate drops due to technical issues as explained in Section 5.6.1. As a result, the system had considerable problems correctly tracking his position and orientation, which very likely impacted the user experience in a negative way.

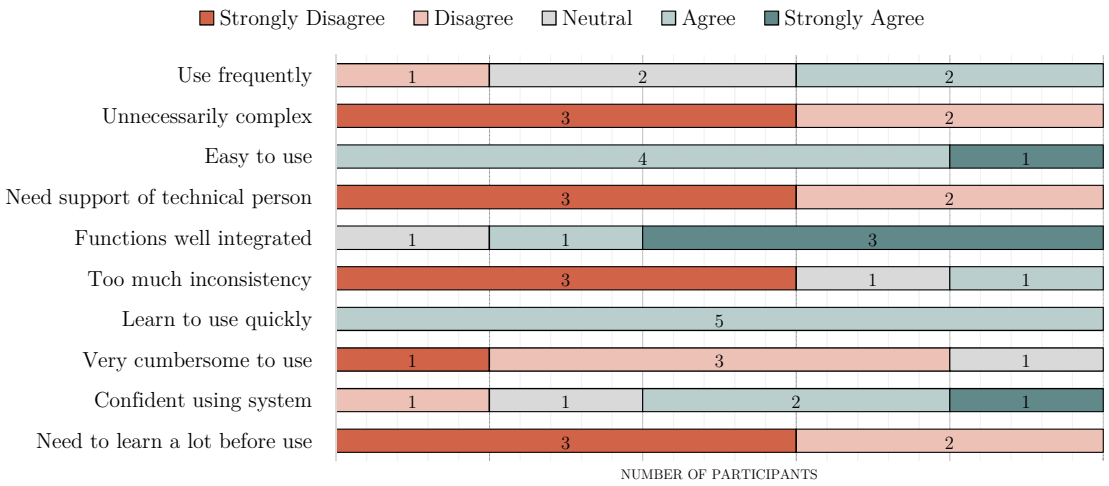


Figure 5.7: Individual ratings for each statement of the System Usability Scale.

Figure 5.7 shows the ratings for each individual statement of the System Usability Scale. Participants seemed to find the developed prototype easy to learn and subsequently use. The participant who experienced technical issues did not feel entirely confident while using the system and found the prototype somewhat inconsistent. The results also indicate that participants were unsure whether or not they would use the system frequently, although there was a slight positive tendency with two users agreeing, one disagreeing and two staying neutral on the subject.

Interface and Practical Application

Figure 5.8 shows the results of all questions related to the general user interface and practical application of the system. The findings indicate that a majority of participants found the system visually appealing. Even though there are many stages with a variety of different interface elements throughout the developed application, no participant found any part of the UI misleading or unclear. In addition, the test subjects did not find it difficult to produce good results with the provided tools. However, the displayed user guidance messages seem to be in need of improvement, with only one participant thinking that they were truly useful. The questionnaire also explored whether or not users would find use for the system even outside the user study. The results indicate that a majority of test subjects thought that there are many practical applications for the developed application and that they would use the system themselves.

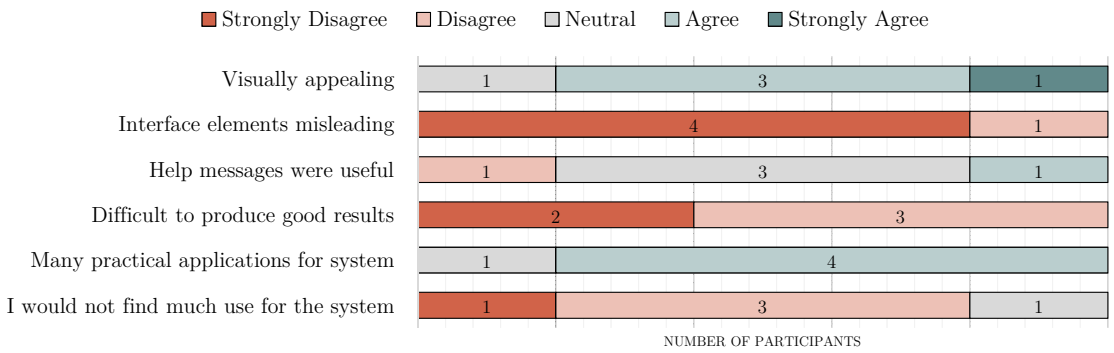


Figure 5.8: Individual ratings for each statement concerned with the interface or practical application.

Participants also had the opportunity of leaving comments about general aspects of the system they particularly liked or disliked. Four participants explicitly stated how intuitive and well designed the user interface was and two particularly liked the way of manipulating the scene. One user disliked the tracking and interaction problems he had due to technical issues (frame rate drops) while using the system.

Preparation

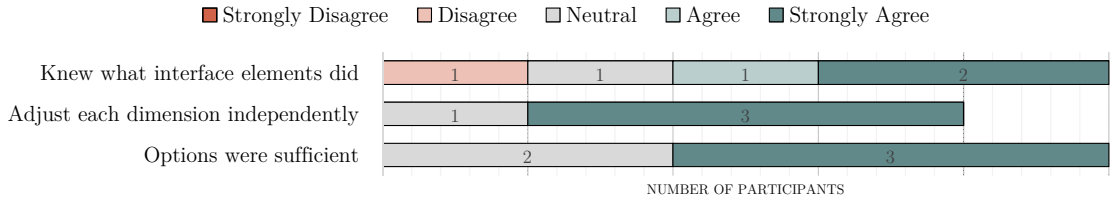


Figure 5.9: Individual ratings for selected statements relating to the preparation stage.

The results of the questionnaire section about the preparation stage are shown in Figure 5.9. Even though the interface during this step made sense to the majority of participants, one of them was not entirely sure what each of the controls on the screen was for. He noted that he did not really know the implications of selecting different values with the size slider. Another participant stated that she wanted more information on how exactly the chosen dimensions influenced the displayed mesh quality. These statements indicate that better labeling is required and future releases could feature additional informations in order to indicate the mesh quality.

A majority of participants would have liked to be able to adjust each of the dimensions of the reconstruction volume separately (one user abstained). One test subject stated in an optional comment that he would have preferred to change the size of the volume while already seeing the image of the depth camera. This would make it easier to correctly

set the dimensions without having to go back and forth between the preparation and scanning screen.

Despite these remarks, no participant found the options during this stage to be completely insufficient. In addition, three of them specifically liked the simplicity of the user interface, as stated in the respective comment sections.

Scanning

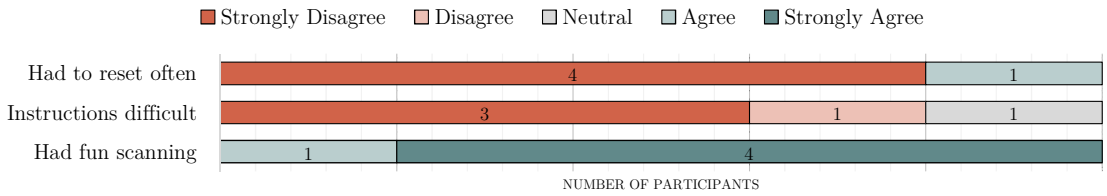


Figure 5.10: Individual ratings for selected statements relating to the scanning stage.

According to the results of the scanning section in the post-study questionnaire (shown in Figure 5.10), the majority of participants did not experience any problems while scanning their scene. Only one user had to reset often due to reconstruction errors. No one found the instructions to be too difficult to follow. More importantly, all participants had fun digitally capturing real objects.

However, according to the corresponding comment section, all participants found the device's wiring very cumbersome during scanning, since it had to be manually kept out of the camera's view at all times. This reinforces the intent of achieving a completely wireless solution in future releases.

Plane Selection

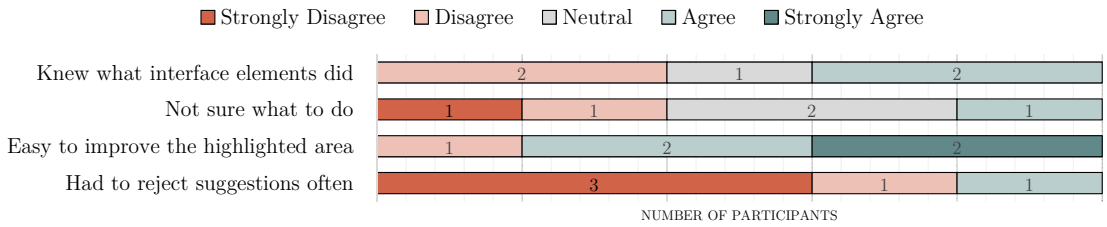


Figure 5.11: Individual ratings for selected statements relating to the plane selection and segmentation stage.

The results of the questionnaire section about the plane segmentation stage are shown in Figure 5.11. Two out of five participants did not know the exact purpose of each interface element. One of those two also stated that he did not know what he was supposed to do at the beginning of this application stage. Another participant stated

that she did not know the difference between plane smoothness and thickness and had the feeling that she could achieve the same result with different settings. These findings indicate that the instructions and labels used for this stage should be improved.

However, the majority of participants found it easy to improve the highlighted area by adjusting the parameters and did not have to reject many suggestions before getting satisfying results. This indicates that, as soon as users are able to understand the different features of this stage, they have no difficulties getting a good outcome.

Object Segmentation

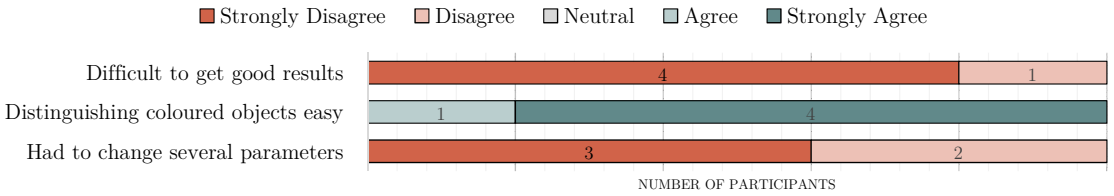


Figure 5.12: Individual ratings for selected statements concerned with the object segmentation stage.

The findings of the questionnaire section regarding the object segmentation procedure show that participants had no noteworthy problems during this stage (see Figure 5.12). No test subject had any difficulties getting a good segmentation result or had to change many parameters in order to do so. Even though no participant found it particularly difficult to distinguish between the differently colored segmentation clusters, one of them noted that the used colors were too similar in some cases. For future releases, the distinguishability of resulting segments should be improved by enhancing the algorithm responsible for color assignment.

It should be noted that participants only had to use Euclidean Clustering for both test scenarios. Region Growing Segmentation is arguably more difficult to understand and choosing the right parameters for a satisfying outcome is typically more time-consuming. The answers to the statements in this section might have been different if users had to use both segmentation methods.

Plane Cut

The results of the questionnaire section dealing with the plane cut segmentation application stage are shown in Figure 5.13. Participants did not have difficulties positioning or rotating the cut plane. One of them, however, stated that she would have preferred to see the current rotation axis while adjusting the plane orientation. This could be a useful improvement for future releases. One test subject did not have to use plane rotation and therefore abstained. Overall, all participants were able to achieve satisfying results without noteworthy problems.

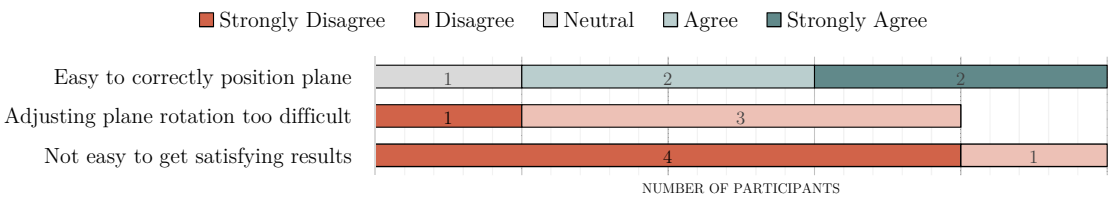


Figure 5.13: Individual ratings for selected statements relating to the plane cut stage.

Processing

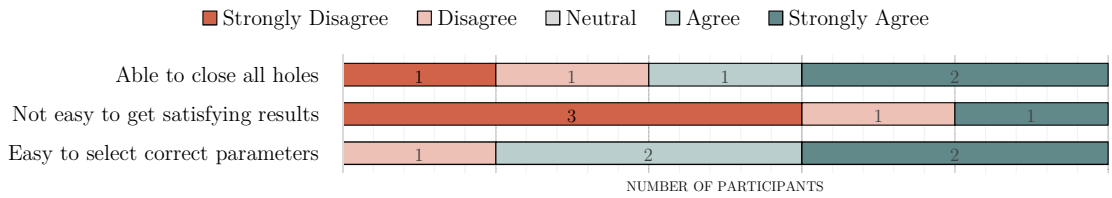


Figure 5.14: Individual ratings for selected statements relating to the processing stage.

The statement ratings of the questionnaire's processing section shown in Figure 5.14 reflect the observations discussed in Section 5.6.1. Two participants had large holes in their reconstruction due to not scanning certain objects from every angle. Since the hole-filling algorithm is only effective for small gaps, these users were not able to close them completely. One of the two participants had more holes remaining than the other and coherently reported difficulties in getting satisfying results as well as selecting the correct parameters. Even though these occurrences indicate that a better hole-filling algorithm would be helpful, the majority of participants had no problems during this stage due to a thoroughly scanned scene. Therefore, leading users to flawless scans should be prioritized over enhancing post-processing of reconstructions. Possible improvements are already discussed in Section 5.6.1

Manipulation and Navigation

Figure 5.15 shows the results of the questionnaire section concerned with interaction and navigation during the scene manipulation stage. As discussed in Section 5.6.1, one participant had significant frame rate drops due to technical difficulties with screen capturing during this stage, which decreased the system's tracking quality and influenced the user experience. As a result, this participant had problems figuring out how to navigate the scene and getting satisfying results. The same user also reported several tracking failures. Other participants had significantly less problems with correct pose estimation and therefore a smoother experience while navigating the scene through physical movement.

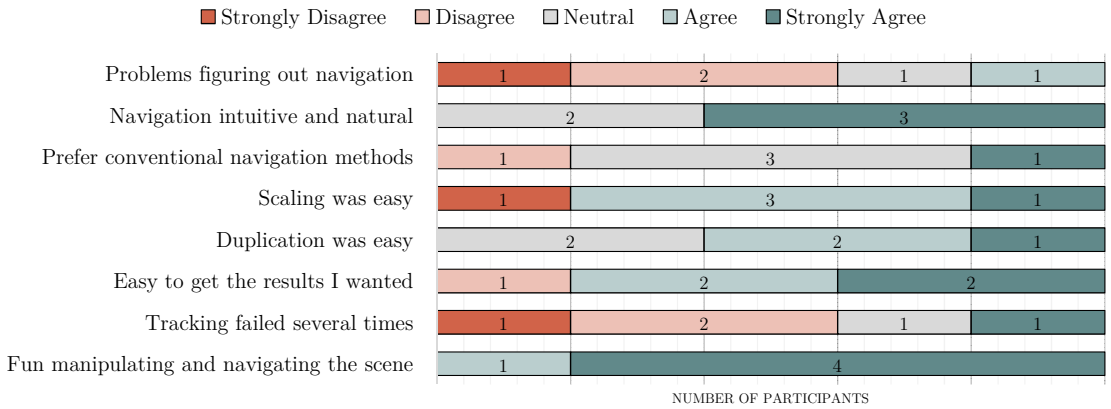


Figure 5.15: Individual ratings for selected statements relating to interaction and navigation during the manipulation stage.

Scaling objects to different sizes posed no problem for four participants, the remaining one reported significant issues regarding the recognition of corresponding touch gestures. This was likely because of mapping problems due to the application being streamed to the tablet instead of running natively. Three participants found duplication easy, while two gave the respective statement a neutral rating. The optional comments revealed that this was caused by occasional failure of the feature due to incorrectly recognizing the respective gesture, which resulted in accidental clone creation as observed in Section 5.6.1. As already stated, this feature needs to be improved, either by making the duplication less prone to errors or by enhancing the visualization or positioning of cloned objects.

In the current prototype, users can inspect their edited scene with an orbital camera at any time during the manipulation stage. However, objects can only be rearranged while navigating with physical movement and reorientation. When asked whether or not they preferred to use conventional navigation methods for editing their scene, the participant with technical issues strongly agreed, one disagreed and three stayed neutral on the matter. Even though this does not indicate a clear preference, it might be helpful to offer some alternative form of navigation for scene manipulation. One participant stated in the comment section that she would have liked to be able to temporarily switch to a bird's eye view for more precise object placement. This could be added as an additional feature in future releases to increase the usefulness of provided scene manipulation tools.

Nevertheless, three out of five participants found navigating the scene by using physical movement very intuitive and natural, with two staying neutral on this matter. Three test subjects explicitly stated how much they enjoyed being able to use the tablet as a window into the virtual world while carrying objects around. Moreover, they noted that this made small adjustments to the camera's position and orientation easier and more intuitive. All five participants stated that they had fun while manipulating and maneuvering the scene during this stage.

These findings indicate that this unusual form of navigating a virtual world resonates with users, since it seems to be very easy to grasp and subsequently use. The ability to

carry objects around and look at the scene from different, natural perspectives appears to be highly appreciated. However, whenever precise object placement was necessary, participants seemed to prefer more conventional forms of navigation, such as a camera in bird's eye view. Offering the ability to switch between both options might be a good solution for future releases.

Hardware Setup

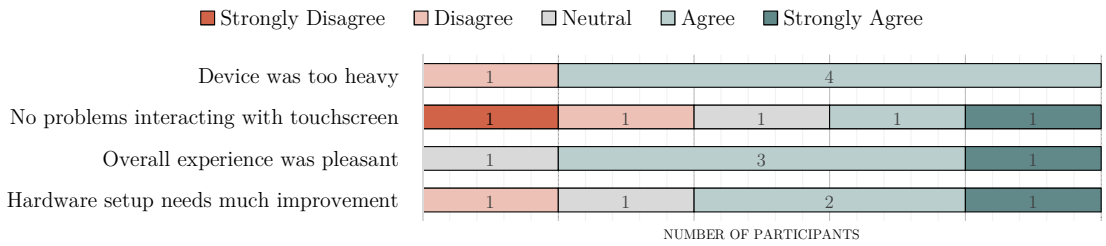


Figure 5.16: Individual ratings for selected statements relating to the used hardware setup.

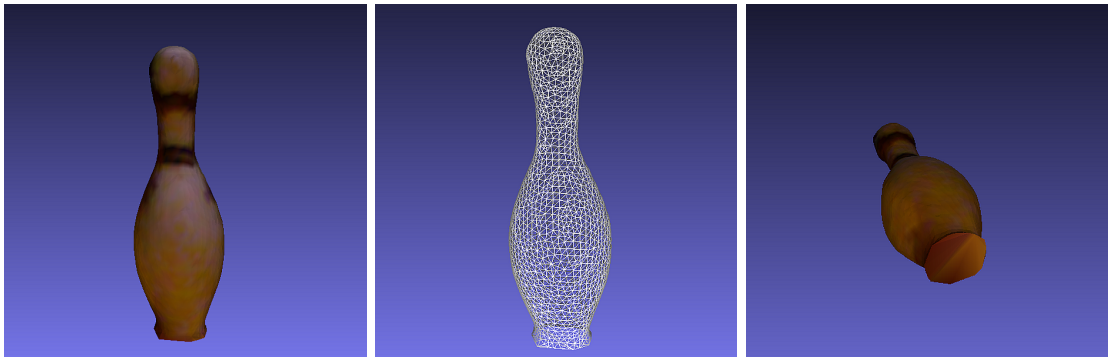
In the last section of the questionnaire, participants were asked about their opinion on the current device prototype. The results are shown in Figure 5.16. Four out of five participants found the device too heavy. Two had problems interacting with the touch screen due to wrong gesture recognition as a result of the application being streamed to the tablet and not running natively. Three participants thought that the hardware setup needed much improvement. The wiring of the device was a nuisance for all test subjects, since it made the interaction much more cumbersome. However, four out of five participants had an overall pleasant experience while using the prototype, with one staying neutral on this matter.

These results are not surprising due to the proof-of-concept nature of the developed prototype. By running future releases natively on, for example, a Microsoft Surface tablet [52], problems with touch gesture recognition and mapping could be almost entirely avoided. In addition, by using a smaller depth camera without the need for an external power supply, the weight could be drastically reduced while simultaneously getting rid of the cumbersome wiring.

5.6.3 Created Reconstructions

Participants produced a number of virtual reconstructions during the course of the two test scenarios. This section takes a look at some examples and analyzes them in order to assess the effectiveness of the developed tools.

Figure 5.17 shows the virtual reconstruction of a bowling pin that was created as a result of one participant completing the first test scenario, the details of which are explained in Section 5.3.1. The reconstructions created by other participants are very

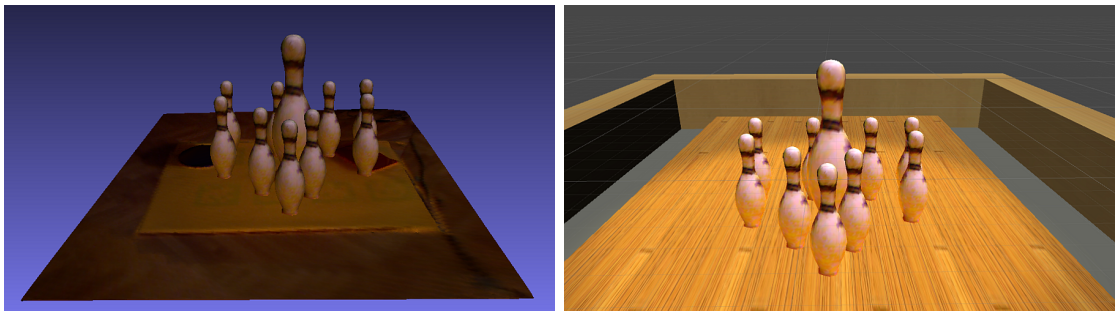


(a) Colored (upright) (b) Wireframe (upright) (c) Colored (lying down)

Figure 5.17: Virtual reconstruction of a bowling pin that was created by one participant during the first test scenario.

similar in terms of size and quality. Despite the blurriness of colors due to the small resolution of the used depth camera, the quality of the shown model is reasonably high. It consists of 2792 vertices and 5381 faces and uses per-vertex coloring. Note that the KinectFusion algorithm tends to produce reconstructions with an even higher point density. The developed system processes and cleans these reconstructions to reduce the number of vertices while distributing them more evenly.

During segmentation, the non-visible touching surfaces of two separated objects result in holes in the reconstruction. Looking at Figure 5.17c, it can be seen that the participant was able to close the gap at the bottom side of the bowling pin by using the provided hole-filling tools.



(a) Reconstruction (b) Game

Figure 5.18: Custom bowling setup scene that was created by one participant at the end of the first test scenario as well as its usage in a small demo application.

Figure 5.18a shows the entire custom bowling setup exported by a participant after completing the final stage of the first scenario. The test subject used the manipulation tools to create nine duplicates of the reconstructed pin. They were then arranged in a custom formation and scaled differently in order to make the result recognizable.

Furthermore, participants were asked to remove all objects but the pin and the ground plane from the reconstructed scene. Figure 5.18b shows the virtual setup after being loaded into a small bowling game that participants were able to play after finishing the scenario. The demo application dynamically imported all objects into the illuminated scene, automatically removed the ground plane and gave the pins basic physical properties.

In the second test scenario, participants had to scan a scene containing colored boxes and cylinders, some of which were stacked on top of each other. As described in Section 5.3.2, the goal was to segment the reconstructed scene into distinct objects by using the tools provided by the developed prototype. Figure 5.19 shows the virtual scene as reconstructed by one of the participants. Some objects in that model contain small miscolored patches, such as the blue blur on the front surface of the green box and the yellow rims on the blue crate in front of it. These occur due to some pixels of Kinect's RGB camera being matched to the wrong vertices in the reconstruction volume when participants make rapid movements during scanning. As a result, a few triangles on a surface might accidentally receive the color information of an object that was directly in front or behind it during scene capture. These flaws have to be fixed manually after reconstruction. Considering, however, that textures have to be refined for high quality scenes anyway due to their low resolution, this is not particularly troublesome. Furthermore, a higher fidelity and frame rate of the depth and RGB camera would decrease the amount of incorrectly assigned vertex colors. The mesh quality is, apart from these small flaws, generally high. The reconstructed scene consists of 136.877 vertices and 162.878 triangles.

As discussed in Section 3.6.2, the implemented automatic object segmentation with Euclidean Clustering is unable to separate directly adjoining, non-planar items. Therefore,

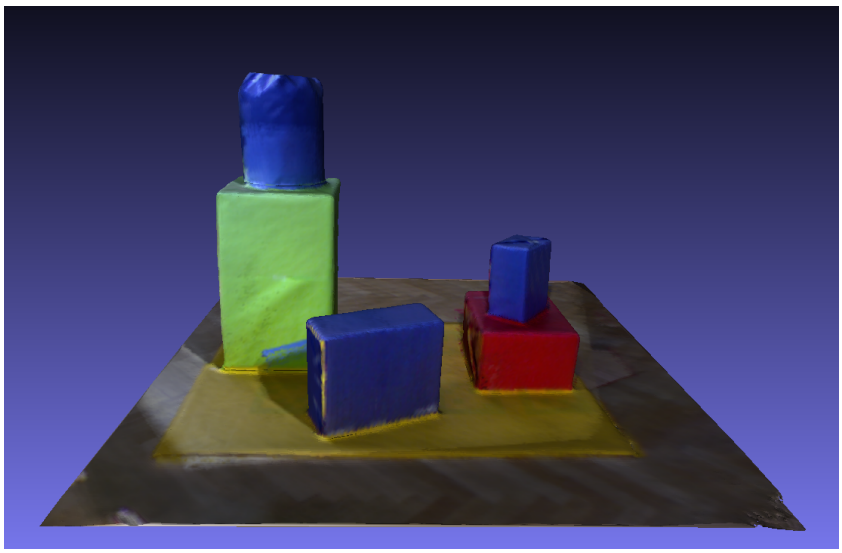
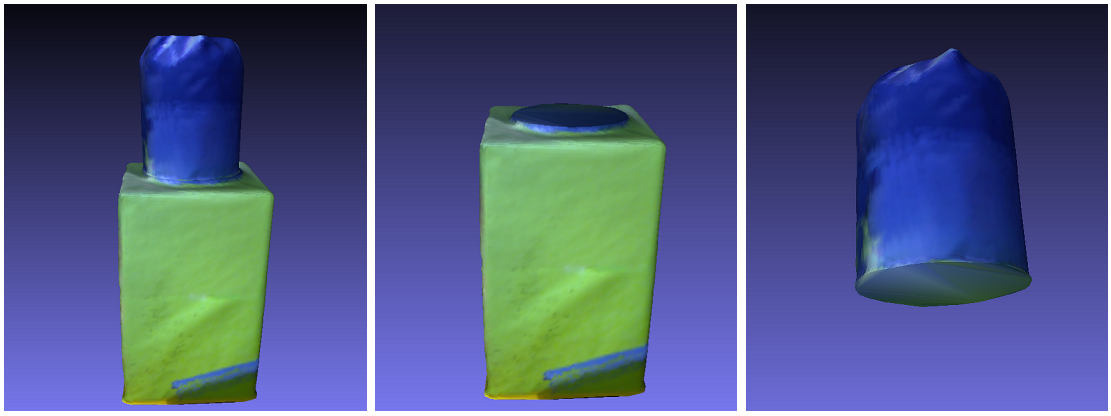


Figure 5.19: Virtual reconstruction of the scene shown in Figure 5.3 that was created by a participant during the second test scenario.



(a) Connected Objects (b) Segmented First Object (c) Segmented Second Object

Figure 5.20: Two connected object reconstructions before and after plane cut segmentation as produced by a participant during the user study.

participants had to use the plane cut tools provided by the developed system in order to segment stacked objects. The result of this procedure, as performed by one user during the second test scenario, are shown in Figure 5.20. Users were able to separate stacked objects after they took some time carefully adjusting the alignment of the cut plane. In addition, resulting gaps could be closed with the provided hole filling tools without any problems. However, as shown in Figure 5.20b, a small part of the upper object is still attached to the lower one. Other participants produced very similar results. It seems to have been difficult to align the cut plane precisely in between two objects, even if the two adjoining surfaces were entirely planar. These findings indicate that even though the current plane cut segmentation tools can be used for rudimentary separation, more sophisticated tools are needed in order to cleanly segment two directly adjoining objects. A possible improvement would be to use curvature information of vertices within a certain distance of the cut plane in order to find the exact transition between two objects.

CHAPTER 6

Conclusion and Future Work

In this research project, a system capable of virtual reconstruction and manipulation of 3D scenes has been designed and implemented. The device enables users to create 3D scans quickly and intuitively. The resulting reconstructions can then be clustered into distinct static surfaces and movable objects by utilizing a combination of automatic and manual segmentation techniques. Users are able to clean the mesh and close unwanted holes in the subsequent processing step. The reconstructed scene can then be navigated by using the tablet screen as a window into the virtual world. Users can inspect the reconstruction from different perspectives by physically moving and reorienting the device in the real world. Through touch screen interaction, previously segmented objects can be repositioned, duplicated, transformed and removed. When users are satisfied with their edited scene, it can be exported to one of several common file formats. The system provides the ability to save the whole scene as well as individual segmented objects. The exported models can then be used in other applications, such as game engines or advanced mesh editors.

The prototype consists of a tablet computer and a Microsoft Kinect. The KinectFusion algorithm is used for 3D scanning and reconstruction as well as user tracking during the scene manipulation stage. Using modern OpenGL, a basic graphics pipeline has been developed to enable efficient 3D rendering. A variety of libraries, such as the Point Cloud Library and VCG Library, has been used in conjunction with own implementations to provide segmentation and processing capabilities. The various functions of the system are very reliant on a stable frame rate in order to run smoothly, which results in relatively high hardware requirements. Due to the unavailability of a tablet with sufficient specifications, the prototype application is running on a desktop PC and streamed to the screen of the handheld device. However, since the tablet functions as a user's primary interaction device, the application interface was designed specifically for mobile screens. In order to improve the device's usability, the display as well as the depth camera have been housed inside a cardboard casing with appropriate cut-outs for better handling.

The effectiveness and usability of the prototype was evaluated during a small user study. Participants had to complete two test scenarios, each of which required them to use the developed system in order to complete a specified task. The first scenario was primarily concerned with the ease-of-use and intuitivity of the system as a whole, while the second one focused more on segmentation tools and the quality of resulting models. After completing these tasks, participants were asked to fill out a questionnaire consisting of a System Usability Scale and a variety of additional questions about the prototype.

The evaluation showed that the system has good usability and can be learned easily. Users are able to digitally reconstruct and segment a three-dimensional scene in a relatively short time and without major technical hurdles. Moreover, the findings indicate that navigating the scene through physical movement feels highly intuitive and natural. In addition, the 3D models created by participants were of reasonably high quality, considering the low resolution of the Microsoft Kinect. Furthermore, a small game was developed for the participants of the user study that dynamically imported and used their segmented reconstructions. This application showcased how models created with the developed prototype could be used in other applications without much effort.

The evaluation also revealed a few issues and several possible areas of improvement for future releases, the most important of which are listed as follows:

Bounding Box Visualization
Participants found it difficult to find a good initial position for the scanning procedure, since they were unable to see the bounding box before starting their scene capture. Even though the scan can be reset at any time, it is still a minor obstacle that has to be overcome by trial and error. A future release should visualize the bounding box with the chosen dimensions either before or at the beginning of the scene capture.

Explanations and Labeling
Since some of the used techniques and algorithms are reasonably complex, the system provides a variety of guidance messages during the different application stages. Even though they seemed to be able to lead users to satisfying results, a few participants still revealed difficulties understanding the exact purpose of certain procedures. It seems that better interface labeling and optional, more detailed explanations could go a long way towards improving the overall experience.

Camera Alignment for Plane Segmentation
Users are recommended to align the viewing direction of the camera with the floor surface at the beginning of the capturing process. This is due to the currently implemented plane segmentation technique assuming major planes (i.e. walls, floor and ceiling) to be parallel or perpendicular to this direction in order to speed up surface recognition. Users seemed to find this additional constraint somewhat cumbersome and had a few issues finding the correct pose. The system could be improved such that the camera alignment does not affect the efficiency and reliability of the plane segmentation tools in any way.

Reconstruction Holes
Some users had problems with relatively big gaps that were accidentally left open during scene capture. Since the algorithm used for hole filling is primarily suited for small planar gaps, these flaws could not be completely corrected. While improvements to the hole filling algorithm are a possibility, leading users to thorough scans should be prioritized. Therefore, a more noticeable background color during scene capture as well as possible automatic gap highlighting could help users in this regard.

Alternative Manipulation Views
During scene editing, certain camera perspectives (e.g. a bird's eye view) are difficult to achieve since users would have to physically move the tablet into a position where they are typically unable to simultaneously interact with the touch screen. These views are, however, helpful for certain, very precise object manipulations. Future versions could feature the option of temporarily switching to a fixed bird's eye view while still being able to use the manipulation tools.

Improved Plane Cut Segmentation
The study revealed that the provided plane cut tools alone are not sufficient to cleanly separate two stacked items that could not be automatically segmented with Euclidean clustering. The existing procedure could be improved by using curvature information of vertices within a certain distance of the cut plane in order to find the exact transition between objects or by using more sophisticated segmentation techniques.

Native Execution on Tablet
The application is streamed from a desktop to a tablet computer since the hardware requirements are too high for the currently used handheld device. This results in several issues with correct recognition of touch gestures such as pinch and zoom, since they are converted to mouse input after being sent to the PC. By using a Windows tablet with sufficient hardware, these problems could be solved completely. In addition, the system would not rely on an additional stationary computer for processing.

Better Depth Camera
Using a depth camera with higher resolution and overall better specifications, such as the Kinect for Xbox One, would increase the stability of the tracking algorithm as well as the overall quality of produced 3D models. By utilizing a depth sensor without the need of an external power supply, such as the Structure Sensor, in combination with a modern Windows tablet, the need for external wires could be completely eradicated. Since cables seemed to be a major hindrance during the user study, this would significantly improve the overall user experience.

Despite these limitations, the research project showed that utilizing depth camera tracking for navigation introduces new and exciting possibilities for interaction with 3D

reconstructed environments. Editing scenes by using the tablet as a window into the virtual world more closely resembles real-world object manipulations. Besides offering intuitive tools for 3D scanning and segmentation, the developed system could be used to see how modifications would visually affect a specific scene without having to actually make changes in the physical world. Future applications could also include simulations of real-world scenarios in reconstructed virtual environments for entertainment or training purposes.

APPENDIX A

Questionnaire

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

Questionnaire

Instructions: For each of the following statements, mark one box that best describes your reactions to the used system.

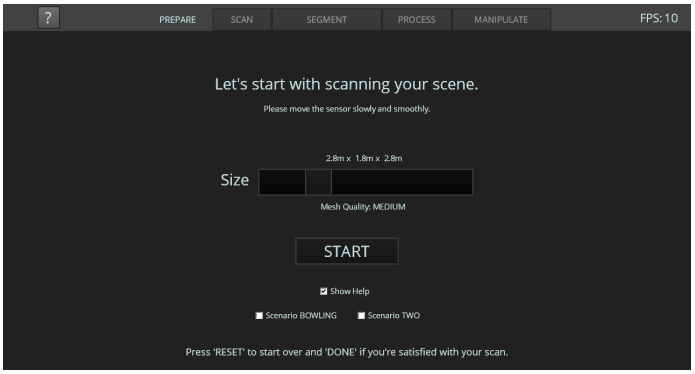
#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I think that I would like to use this system frequently	☐	☐	☐	☐	☐
2	I found the system unnecessarily complex	☐	☐	☐	☐	☐
3	I thought the system was easy to use	☐	☐	☐	☐	☐
4	I think that I would need the support of a technical person to be able to use this system	☐	☐	☐	☐	☐
5	I found the various functions in this system were well integrated	☐	☐	☐	☐	☐
6	I thought there was too much inconsistency in this system	☐	☐	☐	☐	☐
7	I would imagine that most people would learn to use this system very quickly	☐	☐	☐	☐	☐
8	I found the system very cumbersome to use	☐	☐	☐	☐	☐
9	I felt very confident using the system	☐	☐	☐	☐	☐
10	I needed to learn a lot of things before I could get going with the system	☐	☐	☐	☐	☐
11	I found the system to be visually appealing	☐	☐	☐	☐	☐
12	I found many interface elements and descriptions to be misleading	☐	☐	☐	☐	☐
13	I thought the help messages made it easier to understand and use the system	☐	☐	☐	☐	☐
14	I found it very difficult to produce good results with the provided tools	☐	☐	☐	☐	☐
15	I think that there are many practical applications for the system	☐	☐	☐	☐	☐
16	I personally would not find much use for the system	☐	☐	☐	☐	☐

(Optional) Specific parts/aspects of the system I particularly liked:

(Optional) Specific parts/aspects of the system I did not like:

(Optional) Other comments:

PREPARE



Instructions: For each of the following statements, mark one box that best describes your reactions to the PREPARE stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I would have liked to be able to adjust each one of the three space dimensions (height, width, depth) independently	☐	☐	☐	☐	☐
5	I felt like the options before starting my scan were sufficient	☐	☐	☐	☐	☐

(Optional) I found the following tasks in the PREPARE stage too difficult or cumbersome (*Explain why*):

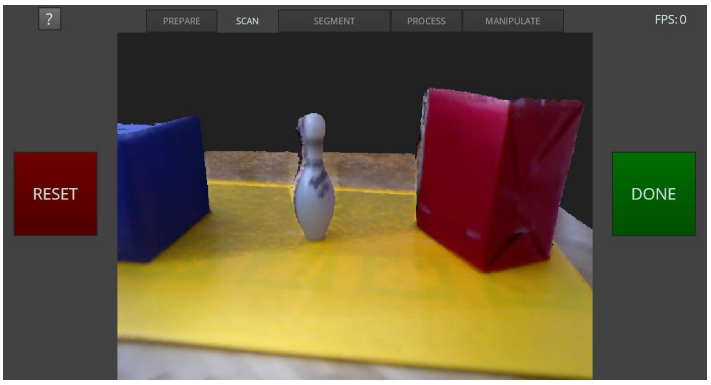
(Optional) Specific parts/aspects of the PREPARE stage I did not like:

(Optional) Specific parts/aspects of the PREPARE stage I particularly liked:

(Optional) Other comments:

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

SCAN



Instructions: For each of the following statements, mark one box that best describes your reactions to the SCAN stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I had to reset often due to scanning errors	☐	☐	☐	☐	☐
5	I liked the placement of the RESET and DONE buttons	☐	☐	☐	☐	☐
6	I found correctly following the instructions in this stage very difficult	☐	☐	☐	☐	☐
7	I had fun scanning the scene	☐	☐	☐	☐	☐

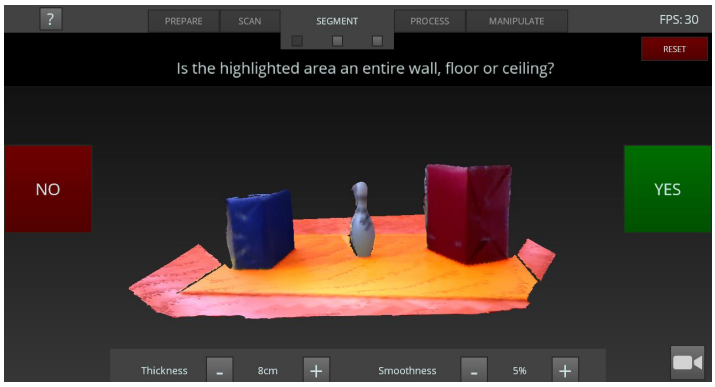
(Optional) I found the following tasks in the SCAN stage too difficult or cumbersome (*Explain why*):

(Optional) Specific parts/aspects of the SCAN stage I did not like:

(Optional) Specific parts/aspects of the SCAN stage I particularly liked:

(Optional) Other comments:

PLANE SEGMENTATION



Instructions: For each of the following statements, mark one box that best describes your reactions to the PLANE SEGMENTATION stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I was not sure what I was supposed to do at the beginning of this stage	☐	☐	☐	☐	☐
4	It was easy to improve the highlighted area by adjusting the thickness and smoothness	☐	☐	☐	☐	☐
5	I had to reject multiple suggestions before a correct plane was highlighted	☐	☐	☐	☐	☐
6	I liked the placement of the different interface elements	☐	☐	☐	☐	☐

(Optional) I found the following tasks in the PLANE SEGMENTATION stage too difficult or cumbersome (*Explain why*):

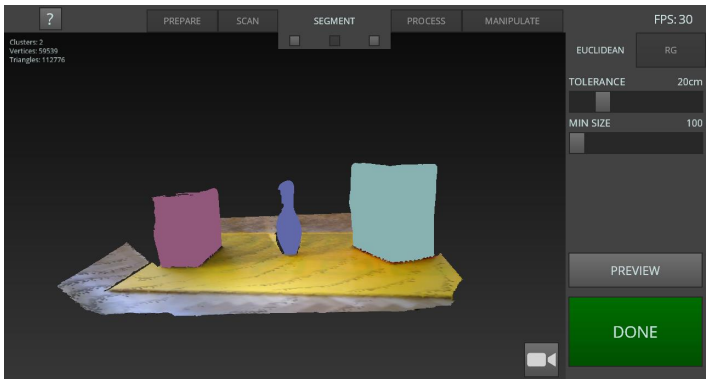
(Optional) Specific parts/aspects of the PLANE SEGMENTATION stage I did not like:

(Optional) Specific parts/aspects of the PLANE SEGMENTATION stage I particularly liked:

(Optional) Other comments:

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

OBJECT SEGMENTATION



Instructions: For each of the following statements, mark <u>one</u> box that best describes your reactions to the OBJECT SEGMENTATION stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	It was difficult to get good results with the provided segmentation methods	☐	☐	☐	☐	☐
5	I could easily distinguish the differently coloured objects in the scene	☐	☐	☐	☐	☐
6	I needed to change several parameters in order to get a good result	☐	☐	☐	☐	☐

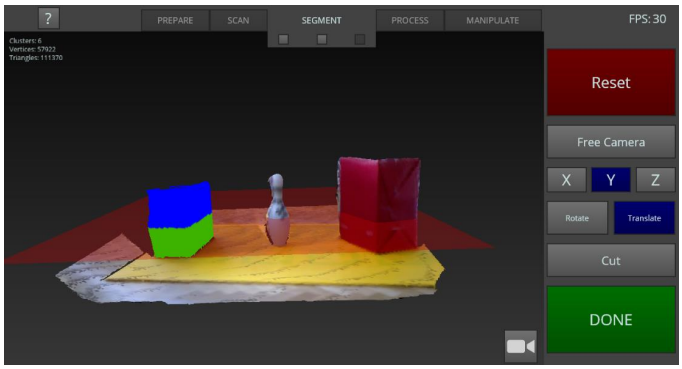
(Optional) I found the following tasks in the OBJECT SEGMENTATION stage too difficult or cumbersome (*Explain why*):

(Optional) Specific parts/aspects of the OBJECT SEGMENTATION stage I did not like:

(Optional) Specific parts/aspects of the OBJECT SEGMENTATION stage I particularly liked:

(Optional) Other comments:

PLANE CUT SEGMENTATION



Instructions: For each of the following statements, mark <u>one</u> box that best describes your reactions to the PLANE CUT SEGMENTATION stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I feel like the plane cut segmentation step is unnecessary	☐	☐	☐	☐	☐
5	I found it easy to correctly position the cut plane	☐	☐	☐	☐	☐
6	I thought that adjusting the plane rotation was too difficult	☐	☐	☐	☐	☐
7	The blue and green colours before cutting helped me get a good segmentation	☐	☐	☐	☐	☐
8	I had to reset a few times before I got satisfying results	☐	☐	☐	☐	☐

(Optional) I found the following tasks in the PLANE CUT SEGMENTATION stage too difficult or cumbersome (*Explain why*):

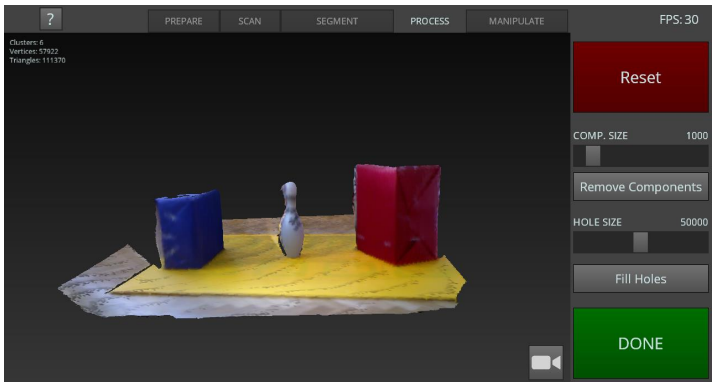
(Optional) Specific parts/aspects of the PLANE CUT SEGMENTATION stage I did not like:

(Optional) Specific parts/aspects of the PLANE CUT SEGMENTATION stage I particularly liked:

(Optional) Other comments:

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

PROCESS



Instructions: For each of the following statements, mark one box that best describes your reactions to the PROCESS stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I was able to close all the holes I wanted to close	☐	☐	☐	☐	☐
5	I had to reset a few times before I got satisfying results	☐				☐
6	I had no difficulties selecting the correct parameters	☐	☐	☐	☐	☐

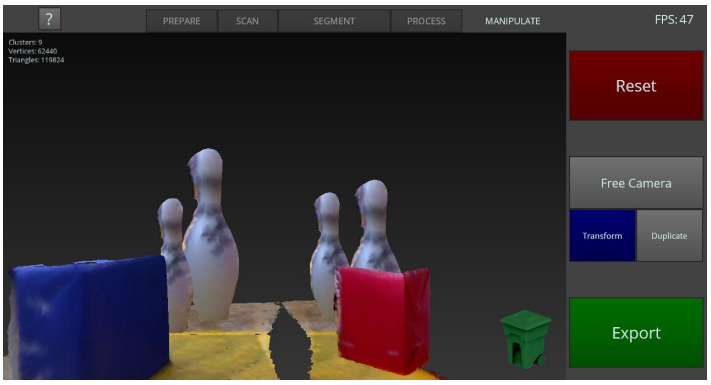
(Optional) I found the following tasks in the PROCESS stage too difficult or cumbersome (*Explain why*):

(Optional) Specific parts/aspects of the PROCESS stage I did not like:

(Optional) Specific parts/aspects of the PROCESS stage I particularly liked:

(Optional) Other comments:

MANIPULATE



Instructions: For each of the following statements, mark <u>one</u> box that best describes your reactions to the MANIPULATE stage of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I knew exactly what each of the different interface elements did	☐	☐	☐	☐	☐
2	I found the screen to be unnecessarily cluttered	☐	☐	☐	☐	☐
3	I found this stage to be visually appealing	☐	☐	☐	☐	☐
4	I had problems figuring out how to navigate the scene	☐	☐	☐	☐	☐
5	I thought that navigating the scene with physical movement felt intuitive and natural	☐	☐	☐	☐	☐
6	I would have preferred other navigation methods (e.g. orbital cameras via keyboard and mouse)	☐	☐	☐	☐	☐
7	I thought that scaling an object to the desired size was easy	☐	☐	☐	☐	☐
8	I felt like it was difficult to adjust an object's rotation the way I wanted	☐	☐	☐	☐	☐
9	I thought that duplicating objects was intuitive and easy	☐	☐	☐	☐	☐
10	I did not like the way objects are deleted (drag and drop in the trash bin)	☐	☐	☐	☐	☐
11	I found it easy to get the results I wanted	☐	☐	☐	☐	☐
12	The application failed to track my position and orientation correctly several times during this stage	☐	☐	☐	☐	☐
13	I had fun manipulating and navigating the scene in this stage	☐	☐	☐	☐	☐

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

(Optional) I found the following tasks in the MANIPULATE stage too difficult or cumbersome (*Explain why*):

(Optional) Specific parts/aspects of the MANIPULATE stage I did not like:

(Optional) Specific parts/aspects of the MANIPULATE stage I particularly liked:

(Optional) Other comments:

This questionnaire is based on the System Usability Scale (SUS), which was developed by John Brooke while working at Digital Equipment Corporation. © Digital Equipment Corporation, 1986.

Participant ID: _____ A Tool for Interactive Segmentation and Manipulation Of 3D Reconstructed Scenes Date: ____/____/____

PROTOTYPE SETUP

Instructions: For each of the following statements, mark one box that best describes your reactions to the hardware setup of the used system. If you haven't used a mentioned aspect of the system, just skip to the next row.

#		1 – Strongly Disagree	2	3	4	5 – Strongly Agree
1	I felt like the device was too heavy	☐	☐	☐	☐	☐
2	I had no problems interacting with the touchscreen	☐	☐	☐	☐	☐
3	I had troubles simultaneously holding the prototype and interacting with the touchpad	☐	☐	☐	☐	☐
4	The overall experience while using the device was pleasant	☐	☐	☐	☐	☐
5	I feel like the hardware setup needs much improvement	☐	☐	☐	☐	☐

(Optional) I found the following aspects of the setup too difficult or cumbersome (*Explain why*):

(Optional) Specific parts/aspects of the hardware setup I did not like:

(Optional) Specific parts/aspects of the hardware setup I particularly liked:

(Optional) Other comments:

List of Figures

2.1 The Microsoft Kinect and its main components. Modified after Zhang [85]. . 5
2.2 IR dot pattern projected onto a scene. Retrieved from [23]. 6
2.3 KinectFusion pipeline. Retrieved from Newcombe et al. [31]. 7
2.4 A multi-resolution pyramid with three levels. Modified after Pirovano [44]. . 8
2.5 One slice of a 3D TSDF volume representation, where each square represents a voxel with the corresponding TSDF value inside. Positive values are in front of the surface, while negative values are behind it, with the surface itself being at the zero-crossing. Retrieved from Zhaoyang et al. [87]. 9
2.6 RANSAC fitting example for two-dimensional data. 11
2.7 K-d tree representation for three-dimensional points. Each node represents a single vertex. The bold value indicates the dimension used for splitting the child trees. Modified after Cibils [12]. 13
2.8 Euclidean Cluster Extraction example. 14
2.9 Region Growing Segmentation example using smoothness and curvature constraints. Red points have been rejected, other colors represent different resulting regions. Retrieved from [64]. 15
2.10 Result of a Min-Cut Based Segmentation. Black pixels represent the foreground cluster (a street light). Retrieved from Golovinskiy et al. [26]. 16
2.11 Segmentation using an interactively placed plane. The model is cut along the plane, with blue indicating the submesh above the plane and green the one below it. 17
2.12 Segmentation techniques based on interactive user input. Figure 2.12a retrieved from Fan et al. [17], Figure 2.12b retrieved from Zheng et al. [88]. . . 17
2.13 Interface after a finished scan with the free version of the ReconstructMe software [66]. 19
2.14 Skanect interface during the processing step [58]. 20

3.1 Hardware setup. 23
3.2 Simplified application workflow. 24
3.3 Application interface overview. 27
3.4 Guidance messages can be displayed to lead users through the application stages. 28
3.5 Application interface during the preparation step. 29

3.6 Positioning of the reconstruction volume relative to the depth camera. 30
3.7 Application interface during the scanning step. 31
3.8 Application interface during the plane segmentation step. 34
3.9 Workflow during the plane segmentation stage. Blue steps are performed automatically, while green steps require user input. 35
3.10 Application interface during the object segmentation step. 36
3.11 Application interface during the plane cut segmentation step. 38
3.12 Application interface during the processing step. 40
3.13 Utilization of provided hole filling method. 42
3.14 Application interface during the manipulation step. 43
3.15 The tablet functions as a window into the virtual world. 44
3.16 Objects can be repositioned with drag-and-drop gestures. 45
3.17 Object rotation during transformation mode. 46
3.18 During duplication mode, copies can be dragged out of original objects. . . . 47
3.19 Objects can be removed by dragging them over the recycle bin. 48

4.1 Rough system overview excluding libraries. 52
4.2 Class diagram containing the most important classes and inheritances. Created with Visual Paradigm CE 12.1 [81]. 54
4.3 Class hierarchy for 2D interface panels. 67
4.4 Class hierarchy for renderable objects. 71
4.5 Components used for scanning. 80
4.6 Class hierarchy of selection classes. 81
4.7 Class hierarchy of segmentation classes. 86

5.1 Setup for the first test scenario of the user study. 97
5.2 Bowling game during first test scenario. The pin setup that was prepared by the participant during the first phase was dynamically imported and integrated into the game. 97
5.3 Setup for the second test scenario of the user study. 98
5.4 Evolution of the prototype. 101
5.5 Participants trying to find a perfect starting position for scanning. 102
5.6 Participants interacting with the system during different application stages. . 104
5.7 Individual ratings for each statement of the System Usability Scale. 106
5.8 Individual ratings for each statement concerned with the interface or practical application. 107
5.9 Individual ratings for selected statements relating to the preparation stage. . 107
5.10 Individual ratings for selected statements relating to the scanning stage. . . . 108
5.11 Individual ratings for selected statements relating to the plane selection and segmentation stage. 108
5.12 Individual ratings for selected statements concerned with the object segmentation stage. 109
5.13 Individual ratings for selected statements relating to the plane cut stage. . . 110
5.14 Individual ratings for selected statements relating to the processing stage. . . 110

5.15 Individual ratings for selected statements relating to interaction and navigation during the manipulation stage. 111
5.16 Individual ratings for selected statements relating to the used hardware setup. 112
5.17 Virtual reconstruction of a bowling pin that was created by one participant during the first test scenario. 113
5.18 Custom bowling setup scene that was created by one participant at the end of the first test scenario as well as its usage in a small demo application. . . . 113
5.19 Virtual reconstruction of the scene shown in Figure 5.3 that was created by a participant during the second test scenario. 114
5.20 Two connected object reconstructions before and after plane cut segmentation as produced by a participant during the user study. 115

List of Algorithms

2.1 Plane Model Segmentation using RANSAC. 12
2.2 Euclidean Cluster Extraction. 13
2.3 Region Growing Segmentation. 15

4.1 Algorithm used in `PlaneSegmenter::Segment()`. 88
4.2 Plane Cut Segmentation. 92
4.3 Triangle reconstruction after segmentation for a single cluster. 93

Bibliography

[1] M. Attene, B. Falcidieno, and M. Spagnuolo. Hierarchical mesh segmentation based on fitting primitives. *The Visual Computer*, 22(3):181–193, 2006.

[2] M. Attene, S. Katz, M. Mortara, G. Patane, M. Spagnuolo, and A. Tal. Mesh Segmentation - A Comparative Study. In *2006. IEEE International Conference on Shape Modeling and Applications*, page 7, June 2006.

[3] A. Bangor, P. Kortum, and J. Miller. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *Journal of Usability Studies*, 4(3):114–123, 2009.

[4] A. Bangor, P. T. Kortum, and J. T. Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.

[5] H. Benhabiles. *3D-mesh segmentation: automatic evaluation and a new learning-based method*. PhD thesis, Université des Sciences et Technologie de Lille-Lille I, 2011.

[6] C. Berglund and M. Geelnard. GLFW. `http://www.glfw.org/`. Online; Accessed: 2016-01-15.

[7] P. Besl and R. Jain. Segmentation through variable-order surface fitting. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 10(2):167–192, March 1988.

[8] P. J. Besl and N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14(2):239–256, February 1992.

[9] D. A. Bowman and L. F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 35–ff. ACM, 1997.

[10] J. Brooke. SUS: A quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[11] X. Chen, A. Golovinskiy, and T. Funkhouser. A Benchmark for 3D Mesh Segmentation. *ACM Transactions on Graphics (TOG)*, 28(3):73, August 2009.

[12] C. Cibils. Stanford University Assignment 3: KDTree. `http://web.stanford.edu/class/cs106l/handouts/assignment-3-kdtree.pdf`. Online; Accessed: 2016-01-15.

[13] P. Cignoni and F. Ganovelli. Visualization and Computer Graphics Library (VCG). `http://vcg.isti.cnr.it/vcglib/`. Online; Accessed: 2016-01-15.

[14] Darek Hoiem. How the Kinect Works. `https://courses.engr.illinois.edu/cs498dh/fa2011/lectures/Lecture%2025%20-%20How%20the%20Kinect%20Works%20-%20CP%20Fall%202011.pdf`. Online; Accessed: 2016-01-15.

[15] B. Dawes, D. Abrahams, and R. Rivera. Boost C++ Libraries. `http://www.boost.org/`. Online; Accessed: 2016-01-15.

[16] D. Eberly. Clipping a mesh against a plane. *Geometric Tools*, 2008.

[17] L. Fan, L. Lic, and K. Liu. Paint Mesh Cutting. *Computer Graphics Forum*, 30(2):603–612, 2011.

[18] D. Field. Laplacian smoothing and Delaunay triangulations. *Communications in Applied Numerical Methods*, 4(6):709–712, 1988.

[19] M. A. Fischler and R. C. Bolles. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Communications of the ACM*, 24(6):381–395, June 1981.

[20] J. Francone and L. Nigay. Using the User's Point of View for Interaction on Mobile Devices. In *23rd French Speaking Conference on Human-Computer Interaction*, page 4. ACM, 2011.

[21] B. Freedman, A. Shpunt, M. Machline, and Y. Arieli. Depth mapping using projected patterns, October 2008. US Patent App. 11/899,542.

[22] D. Fritz, A. Mossel, and H. Kaufmann. Markerless 3D Interaction in an Unconstrained Handheld mixed Reality Setup. *The International Journal of Virtual Reality*, 15(1):25–34, 2015. invited.

[23] Futurepicture. Kinect Hacking 105. `http://www.futurepicture.org/?p=129`. Online; Accessed: 2016-01-15.

[24] G-Truc Creation. OpenGL Mathematics (GLM). `http://glm.g-truc.net/`. Online; Accessed: 2016-01-15.

[25] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical Face Clustering on Polygonal Surfaces. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 49–58. ACM, 2001.

[26] A. Golovinskiy and T. Funkhouser. Min-Cut Based Segmentation of Point Clouds. In *2009 IEEE 12th International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 39–46, September 2009.

[27] L. C. Goron, L. Tamas, I. Reti, and G. Lazea. 3D laser scanning system and 3D segmentation of urban scenes. In *2010 IEEE International Conference on Automation Quality and Testing Robotics (AQTR)*, volume 1, pages 1–5, May 2010.

[28] D. D. Hearn and M. P. Baker. *Computer Graphics with OpenGL*. Prentice Hall Professional Technical Reference, 2003.

[29] O. Hilliges, D. Kim, S. Izadi, M. Weiss, and A. Wilson. HoloDesk: Direct 3D Interactions with a Situated See-through Display. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2421–2430. ACM, 2012.

[30] M. Ikits and M. Magallon. OpenGL Extension Wrangler Library (GLEW). `http://glew.sourceforge.net/`. Online; Accessed: 2016-01-15.

[31] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 559–568, New York, NY, USA, 2011.

[32] J. Jankowski and M. Hachet. A Survey of Interaction Techniques for Interactive 3D Environments. In *Eurographics 2013 - STAR*, May 2013.

[33] E. M. Jones and P. Fjeld. Gimbal Angles, Gimbal Lock, and a Fourth Gimbal for Christmas. `http://www.hq.nasa.gov/alsj/gimbals.html`. Online; Accessed: 2016-01-15.

[34] D. Kaushik, R. Jain, et al. Natural User Interfaces: Trend in Virtual Interaction. *arXiv preprint arXiv:1405.0101*, 2014.

[35] Khronos Group. OpenGL. `https://www.opengl.org/`. Online; Accessed: 2016-01-15.

[36] Khronos Group. OpenGL 4.5 Reference Page. `https://www.opengl.org/sdk/docs/man/`. Online; Accessed: 2016-01-15.

[37] D. Lee and J.-I. Hwang. Design and evaluation of smart phone-based 3D interaction for large display. In *2015 IEEE International Conference on Consumer Electronics (ICCE)*, pages 657–658, January 2015.

[38] W. Lemberg, D. Turner, and R. Wilhelm. The FreeType Project. http://www.freetype.org/. Online; Accessed: 2016-01-15.

[39] J. R. Lewis and J. Sauro. The Factor Structure of the System Usability Scale. In *Human Centered Design*, pages 94–103, 2009.

[40] S. Lloyd. Least Squares Quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[41] A. Martinet, G. Casiez, and L. Grisoni. The design and evaluation of 3D positioning techniques for multi-touch displays. In *3D User Interfaces (3DUI), 2010 IEEE Symposium on*, pages 115–118, March 2010.

[42] G. P. Meyer and M. N. Do. 3D GrabCut: Interactive Foreground Extraction for Reconstructed 3D Scenes. In *Eurographics Workshop on 3D Object Retrieval*. The Eurographics Association, 2015.

[43] A. R. Miandarhoie and K. Khalili. Iterative Mesh Segmentation Using Approximated Voronoi Diagram. *Procedia Technology*, 19(0):106–111, 2015.

[44] Michele Pirovano. Kinfu – an open source implementation of Kinect Fusion + case study: implementing a 3D scanner with PCL. http://www.michelepirovano.com/pdf/3d-scanning-pcl.pdf. Online; Accessed: 2016-01-15.

[45] Microsoft. Depth Space Range. http://msdn.microsoft.com/enus/library/hh973078.aspx#Depth_Ranges. Online; Accessed: 2016-01-15.

[46] Microsoft. Depth Stream. https://msdn.microsoft.com/en-us/library/jj131028.aspx. Online; Accessed: 2016-01-15.

[47] Microsoft. DepthImageFormat Enumeration. http://msdn.microsoft.com/en-us/library/microsoft.kinect.depthimageformat.aspx. Online; Accessed: 2016-01-15.

[48] Microsoft. Kinect for Windows SDK. https://www.microsoft.com/en-us/kinectforwindows/. Online; Accessed: 2016-01-15.

[49] Microsoft. Kinect for Windows Sensor Components and Specifications. https://msdn.microsoft.com/en-us/library/jj131033.aspx. Online; Accessed: 2016-01-15.

[50] Microsoft. Kinect for Xbox One. http://www.microsoftstore.com/store/msusa/en_US/pdp/Kinect-for-Xbox-One/productID.307499400. Online; Accessed: 2016-01-15.

[51] Microsoft. Kinect Fusion Explorer D2D C++ Sample. https://msdn.microsoft.com/en-us/library/dn188697.aspx. Online; Accessed: 2016-01-15.

[52] Microsoft. Microsoft Surface Devices. https://www.microsoft.com/surface/en-us. Online; Accessed: 2016-01-15.

[53] Microsoft. Windows API Index. https://msdn.microsoft.com/en-us/library/windows/desktop/ff818516(v=vs.85).aspx. Online; Accessed: 2016-01-15.

[54] A. Mossel, B. Venditti, and H. Kaufmann. 3DTouch and HOMER-S: Intuitive Manipulation Techniques for One-Handed Handheld Augmented Reality. In *Proceedings of the 15th International Conference of Virtual Technologies (VRIC'13)*. ACM, 2013.

[55] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohi, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE international symposium on Mixed and augmented reality (ISMAR)*, pages 127–136. IEEE, 2011.

[56] A. Nguyen and B. Le. 3D point cloud segmentation: A survey. In *2013 6th IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pages 225–230, November 2013.

[57] OBS Project. Open Broadcaster Software. https://obsproject.com/. Online; Accessed: 2016-01-15.

[58] Occipital. Skanect. http://skanect.occipital.com/. Online; Accessed: 2016-01-15.

[59] Occipital. Structure Sensor. http://structure.io/. Online; Accessed: 2016-01-15.

[60] P. W. Olszta, A. Umbach, S. Baker, J. F. Fay, J. Tsiombikas, and D. C. Niehorster. FreeGLUT. http://freeglut.sourceforge.net/. Online; Accessed: 2016-01-15.

[61] J. M. Palacios, C. Sagüés, E. Montijano, and S. Llorente. Human-Computer Interaction Based on Hand Gestures Using RGB-D Sensors. *Sensors*, 13(9):11842–11860, 2013.

[62] PCL. Euclidean Cluster Extraction. http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php. Online; Accessed: 2016-01-15.

[63] PCL. Plane model segmentation. http://pointclouds.org/documentation/tutorials/planar_segmentation.php. Online; Accessed: 2016-01-15.

[64] PCL. Region growing segmentation. http://pointclouds.org/documentation/tutorials/region_growing_segmentation.php. Online; Accessed: 2016-01-15.

[65] G. Peyre and L. Cohen. Surface segmentation using geodesic centroidal tesselation. In *Proceedings of the 2nd International Symposium on 3D Data Processing, Visualization and Transmission*, pages 995–1002, September 2004.

[66] PROFACTOR. ReconstructMe. `http://reconstructme.net/`. Online; Accessed: 2016-01-15.

[67] PROFACTOR. ReconstructMe for Developers. `http://reconstructme.net/reconstructme-sdk/`. Online; Accessed: 2016-01-15.

[68] T. Rabbani, F. van den Heuvel, and G. Vosselman. Segmentation of point clouds using smoothness constraint. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 36(5):248–253, 2006.

[69] R. B. Rusu. *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*. PhD thesis, Computer Science Department, Technische Universität München, Germany, October 2009.

[70] R. B. Rusu, N. Blodow, Z. C. Marton, and M. Beetz. Close-range scene segmentation and reconstruction of 3D point cloud maps for mobile manipulation in domestic environments. In *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–6. IEEE, 2009.

[71] R. B. Rusu and S. Cousins. 3D is here: Point Cloud Library (PCL). In *IEEE International Conference on Robotics and Automation (ICRA)*, May 9-13 2011.

[72] R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for point-cloud shape detection. *Computer Graphics Forum*, 26(2):214–226, June 2007.

[73] A. Shamir. A survey on mesh segmentation techniques. *Computer Graphics Forum*, 27(6):1539–1556, 2008.

[74] A. L. Simeone, E. Velloso, J. Alexander, and H. Gellersen. Feet movement in desktop 3D interaction. In *2014 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 71–74, March 2014.

[75] Splashtop. Splashtop - Remote Desktop and Remote Support. `http://www.splashtop.com/`. Online; Accessed: 2016-01-15.

[76] A. Subramanian. Integration of Natural User Interface in a Real-World Environment. In *2015 IEEE International Conference on Computational Intelligence Communication Technology (CICT)*, pages 714–718, February 2015.

[77] N. Sugiura and T. Komuro. Dynamic 3D interaction using an Optical See-through HMD. In *Virtual Reality (VR), 2015 IEEE*, pages 359–360, March 2015.

[78] Z. Toony, D. Laurendeau, P. Giguere, and C. Gagne. 3D-NCuts: Adapting normalized cuts to 3D triangulated surface segmentation. In *2014 International Conference on Computer Graphics Theory and Applications (GRAPP)*, pages 1–9, January 2014.

[79] A. J. Trevor, S. Gedikli, R. B. Rusu, and H. I. Christensen. Efficient organized point cloud segmentation with connected components. *Semantic Perception Mapping and Exploration (SPME)*, 2013.

[80] Unity Technologies. Unity Game Engine. `https://unity3d.com/`. Online; Accessed: 2016-01-15.

[81] Visual Paradigm. Visual Paradigm - Software Design Tools. `http://www.visual-paradigm.com/`. Online; Accessed: 2016-01-15.

[82] R. Y. Wang and J. Popović. Real-time hand-tracking with a color glove. *ACM Transactions on Graphics (TOG)*, 28(3):63, 2009.

[83] E. W. Weisstein. Point-Plane Distance. `http://mathworld.wolfram.com/Point-PlaneDistance.html`, 2002. Online; Accessed: 2016-01-15.

[84] M. Y. Yang and W. Förstner. Plane Detection in Point Cloud Data. In *Proceedings of the 2nd International Conference on Machine Control Guidance, Bonn*, volume 1, pages 95–104, 2010.

[85] Z. Zhang. Microsoft Kinect Sensor and Its Effect. *MultiMedia, IEEE*, 19(2):4–10, February 2012.

[86] Z. Zhang, M. Zhang, Y. Chang, E.-S. Aziz, S. K. Esche, and C. Chassapis. Real-Time 3D Model Reconstruction and Interaction Using Kinect for a Game-Based Virtual Laboratory. In *ASME 2013 International Mechanical Engineering Congress and Exposition*, pages V005T05A053–V005T05A053. American Society of Mechanical Engineers, 2013.

[87] L. Zhaoyang and A. Davison. KinfuSeg: a Dynamic SLAM Approach Based on Kinect Fusion. Master's thesis, Imperial College London, September 2013.

[88] Y. Zheng, C.-L. Tai, and O. K.-C. Au. Dot Scissor: A Single-Click Interface for Mesh Segmentation. *Visualization and Computer Graphics, IEEE Transactions on*, 18(8):1304–1312, 2012.