FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Streaming und Exploration von sich Dynamisch Ändernden Oberflächenrekonstruktionen in Immersive Virtual Reality

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Manuel Kröter
Matrikelnummer 0820478

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Mitwirkung: Dipl.-Ing. Dr. Annette Mossel

Wien, 7. April 2016

Manuel Kröter	Hannes Kaufmann

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Streaming and Exploration of Dynamically Changing Dense Surface Reconstructions in Immersive Virtual Reality

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Manuel Kröter
Registration Number 0820478

to the Faculty of Informatics

at the Vienna University of Technology

Advisor: Priv.-Doz. Mag. Dr. Hannes Kaufmann
Assistance: Dipl.-Ing. Dr. Annette Mossel

Vienna, 7th April, 2016

Manuel Kröter	Hannes Kaufmann

Erklärung zur Verfassung der Arbeit

Manuel Kröter
Wasserburgergasse 5/14, 1090 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, 7. April 2016

Manuel Kröter

Acknowledgements

I am thankful to my advisors Priv.-Doz. Hannes Kaufmann and Dr. Annette Mossel for giving me the opportunity to write this diploma thesis at the Interactive Media Systems group. Besides guiding me through the thesis and providing me with valuable feedback, I want to thank Annette for helping me in conducting the user study. I also want to thank all the volunteers of this study for their time and participation.

My special thanks go to my parents Irene and Werner for always supporting me and making my studies possible. Furthermore, I want to thank my brother David for his help in all situations and especially the time in Vienna. I want to thank Stefan for the great moments in Copenhagen and Vienna and also for providing his laptop for the time of my thesis. Lastly, I am extremely thankful to Stephi for all her support and the last six wonderful years.

Kurzfassung

Günstige Tiefenbildkameras wie die Microsoft Kinect ermöglichen es die Struktur der Umwelt zu erfassen. Mit Hilfe von 3D Rekonstruktionsverfahren kann ein detailliertes 3D-Modell in Echtzeit aus den Kameradaten berechnet werden. Autonome Roboter können diese Techniken anwenden, um eine Karte der Szene zu erstellen, während sie diese erforschen. Dies ermöglicht es dem Roboter, sich selbst in unbekannten Umgebungen zu orten und zu navigieren. Das rekonstruierte Modell kann zudem auch für andere Parteien interessant sein, um sich ein Bild dieser Umgebung zu machen. Besonders weit entfernte oder gefährliche Bereiche können durch Roboter gescannt werden, während entfernte Beobachter in der Lage sind, sicher einen Überblick über die Szene zu bekommen. Um diese Fernerkundung zu erlauben, während die Szene noch gescannt wird, müssen die rekonstruierten Daten inkrementell über ein drahtloses Netzwerk gesendet werden. Da derzeit keine Lösungen mit inkrementeller Netzwerkübertragung vorhanden sind, wird das existierende Rekonstruktionsframework InfiniTAM erweitert, um genau dies für große, dynamische Modelle zu unterstützen. Die Visualisierung und Exploration des Modells wird mit Hilfe der Unreal Engine 4 durchgeführt, einer State-of-the-Art 3D-Engine. Zu diesem Zweck wird eine Darstellnung des Modells als Dreiecksnetz bevorzugt, während dichte Rekonstruktionsverfahren meist mit einer volumetrischen Darstellung arbeiten. In aktuellen Ansätzen wird das Dreiecksnetz in einem Nachbearbeitungsschritt extrahiert. Dies ist jedoch nicht anwendbar, wenn die Szene noch während dem Scannen betrachtet und erforscht werden soll. Das verwendete Rekonstruktionframework ist daher so angepasst, dass ein aktuelles Dreiecksnetz in Echtzeit erhalten wird. Das rekonstruierte Dreiecksnetz wird schließlich in einem Virtual Reality Setup mittels Head-Mounted Display und einem omnidirektionalen Laufband erforscht. Der Einsatz von Virtual Reality Hardware ermöglicht es, auf natürliche Art und Weise zu navigieren. Das entwickelte System ist in Bezug auf die Speicheranforderungen und Datenübertragunsraten evaluert. Zudem ist der Erwerb des Raumverständnisses im Rahmen einer Nutzerstudie analysiert.

Abstract

Low cost commodity depth cameras like the Microsoft Kinect allow to sense the structure of the environment. With the aid of dense surface reconstruction methods, a detailed 3D model can be computed in real-time from the acquired camera data. Autonomous robots can apply this techniques in order to build a map of the scene while they are exploring it. This allows the robot to locate itself and to navigate in unknown environments. Besides that, the reconstructed model can be interesting for different parties, who want to explore these environments as well. Especially, distant or dangerous areas can be scanned by robots while remote observers are able to safely get an overview of the scene. In order to support remote exploration while the scene is still scanned, the reconstructed information has to be streamed incrementally over wireless network. Since currently no solution exists with this feature, the existing reconstruction framework InfiniTAM is extended to support the transmission of a large scale, dynamically changing model. The visualization and exploration of the model is performed with the aid of Unreal Engine 4, a state-of-the-art 3d engine. For this purpose, a triangular mesh representation is favored, while dense reconstruction methods mostly operate on a volumetric representation. In current approaches, the mesh is extracted in a post-processing step, which is not applicable when the scene should be explored while being scanned and updated. The used reconstruction framework is therefore adapted to maintain an up-to-date mesh in real-time. The reconstructed mesh finally is explored in a virtual reality setup using a head-mounted display and an omnidirectional treadmill. The usage of virtual reality hardware enhances the ease of use and makes it possible to navigate in a natural way. The developed system is evaluated in terms of memory requirements and data rates as well as within a user study, that analyzes the effect of the incremental streaming and the virtual reality exploration on spatial knowledge acquisition.

Contents

CHAPTER 1

Introduction

The reconstruction of environmental geometry is broadly researched in computer vision and computer graphics. The goal of this process is to capture the 3D structure of the scene using information from an RGB or depth camera and generate a computer model from it [NIH+11]. Especially in the area of robotics, these methods are widely applied, when the robot has to navigate autonomously in unknown environments. In order to avoid obstacles and to explore new areas, the robot simultaneously needs to build a model of the environment and estimate its position within the generated model. This principle is called Simultaneous Localization and Mapping (SLAM) [TL08]. Another application area of this method is architecture, where a 3D model of a building can be generated by walking through it with a handhold scanning device [FCSS09]. Other persons can then experience this building without actually being there.

The capturing of an environment and its exploration is often performed by two remotely connected parties, such as in a typical setting in robotics: a human observer virtually explores the scene while an autonomous robot is actually scanning it. Using a robot might be cheaper, safer or more convenient than going everywhere in person. For example, a quadcopter can scan large areas without effort [WMG+12] or firefighters can send a robot into dangerous buildings to safely get an overview. In all these cases, data has to be exchanged between the two parties in some way. The aim of this thesis is to create a 3D reconstruction pipeline, which integrates such data transmission over wireless network and allows for live exploration by a remote observer, as illustrated in Figure 1.1.

1.1 Incremental Network Transmission

A trivial solution is to transmit the final model after the reconstruction process has finished. As a consequence, one has to wait a long time before the environment can be analyzed, especially for large scenes. If the data is streamed incrementally while scanning, the scene can be explored from the start and findings are obtained sooner. This can be

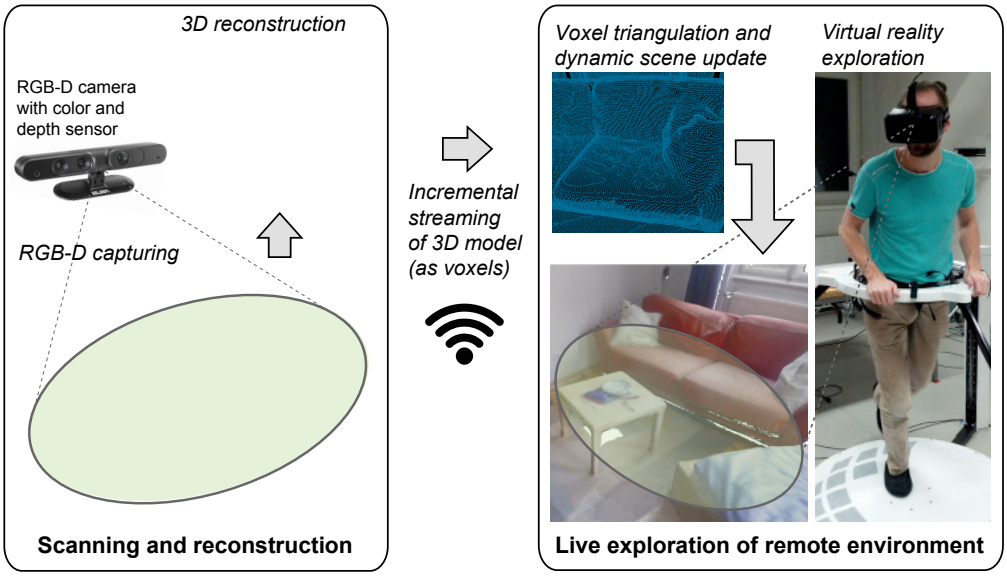


Figure 1.1: A room is scanned using a RGB-D camera and 3D model is reconstructed from the camera data. The model of the room is transmitted incrementally over network to a remote client. On the client side, the volumetric model is triangulated and can be virtually explored while it is gradually expanding.

vital in scenarios like the firefighter example. Moreover, it is possible to influence the way the scene is scanned. For instance, one can give directions to the scanning robot if something interesting is discovered during exploration.

Current reconstruction systems do not incrementally stream the reconstructed model but only the sensor data. The 3D reconstruction process is completely [NSS14], or at least partially [WMG+12], performed on a remote server. This is advantageous since generally, the server is more powerful than the mobile scanning computer (e.g. the robot). Moreover, the complete 3D model of the reconstruction is available on the server and thus, the scene can be remotely explored, independently of the current position of the scanning camera. The major drawback is, that this principle is prone to network failures or bottlenecks. Once the network connection is lost, the reconstruction cannot continue and might fail due to camera tracking errors. Moreover, the robot does not get instructions from the server anymore, e.g. for navigation purposes. Autonomous decisions cannot be made because the robot does not store the 3D model of the environment itself.

In the reconstruction pipeline developed within this thesis, not the camera data, but the model itself is streamed. This way, the reconstruction is both available to the scanning computer and the remote server, and one is less dependent on a stable network connection. If the network connection is lost, the reconstruction still continues on the client side and once the connection is established again, the model on the server is updated. The latency of the model update is not critical compared to the case when streaming camera

images. The major challenge with the incremental transmission of the model is to keep the needed bandwidth as low as possible. Furthermore, the reconstruction process needs to be very efficient in order to minimize the required computing power on the mobile computer.

1.2 Live Exploration

As already mentioned, a reason for the need of incremental model transmission is the desire to explore the scene while still scanning. The observer should be able to examine the reconstruction as quickly as possible and to get to know the spatial layout of the environment.

1.2.1 Exploration in Virtual Reality

Exploring the scene by navigating through the model using standard input and output devices such as a keyboard and a standard computer monitor does not facilitate a natural user experience due to the limitations of both input and output device. Studies have shown, that the spatial knowledge acquisition can be enhanced by exploring a virtual scene as natural as possible [RL09]. Learning the spatial arrangement of a scene requires mental resources and the fewer one has to concentrate on the human-computer interface, the more one can concentrate on the environment [BKJP04]. In addition, non-visual sensory information such as vestibular cues or muscle usage support the building of a mental map, as one can easier keep track of the orientation and position within the scene [RB04].

In order to improve the exploration in terms of ease-of-use and spatial knowledge acquisition, one can use immersive Virtual Reality (VR) technologies. As part of this thesis, the implemented reconstruction pipeline integrates a head-mounted display (HMD) and an omnidirectional treadmill (ODT) for scene exploration. An HMD allows to rotate and look around in the virtual scene like in reality and an ODT enables to walk infinitely in the virtual world while actually staying at the same place in the real world.

While it was previously shown that natural walking significantly contributes to spatial learning [RB04], there are no studies yet that analyze the effect of a low friction ODT. In this thesis, it is evaluated in an experimental user study if walking with an ODT is to be favored over pure virtual walking with keyboard input when considering spatial knowledge acquisition. Apart from that, the study also examines if the fact, that the model is not available in its entirety from the beginning but incrementally expands during exploration, increases or decreases the spatial knowledge.

1.2.2 Visualization

To be able to explore the reconstructed scene, the model needs to be visualized first. Generally, there are two options: Raycasting or standard forward rendering [FVDF$^+$94]. The process of raycasting is an image-based approach. It generates an image by shooting

a ray into the scene for every pixel and gathering information at the intersection with the model. The computational effort therefore depends on the resolution of the final image. Forward rendering is an object-based approach and constitutes the standard approach used by most existing 3D engines. The image is computed by projecting the model on the image plane. The complexity of the model determines the runtime. While raycasting is fine for live feedback during scanning, forward rendering is favored when exploration and interaction with the model is desired. In the latter case, high image resolutions and high framerates are necessary, especially in a VR setup [Ocu16b].

Dynamic Mesh Update In order to apply forward rendering, a triangular mesh needs to be extracted from the representation used for reconstruction. Most of the currently available reconstruction solutions perform this mesh computation in a post-processing step. In this thesis, a real-time extraction is implemented since it is necessary to support live exploration. Note, that the underlying representation does not only grow during reconstruction but can also change in already scanned areas. Thus, the mesh does not only have to be expanded but updated in existing areas, which comprises an additional challenge. Lastly, the final rendering step is adjusted to support this dynamically changing geometry. In standard rendering approaches, the geometry to be rendered is loaded before visualization and is then not changed anymore [FVDF+94].

1.3 Contribution

To sum it up, the aim of the thesis is to create a 3D reconstruction pipeline, which allows to generate a 3D computer model from a real-world environment and to stream that model incrementally over wireless network to another PC. The model can then be remotely explored in a VR setup, which supports the spatial knowledge acquisition. All steps happen in real-time while the environment is still scanned and the reconstructed model grows. For this purpose, the existing state-of-the-art reconstruction framework InfiniTAM, developed by Kähler et al. [KPR+15], represents the base implementation. InfiniTAM is extended with following parts:

- Incremental transmission of the reconstructed model over network
- On-the-fly mesh generation (and dynamic update) from the underlying model
- Communication with UE4 to pass the extracted meshes

In order to visualize and explore the gradually expanding reconstruction in an immersive VR setup, the game engine Unreal Engine 4 (UE4) is applied [Epi16]. In addition to the standard UE4 capabilities, it contains following features:

- Communication with InfiniTAM
- Dynamic update of the scene representation

- Integration of the VR hardware (Oculus Rift HMD and the Virtualizer, an ODT from Cyberith [CH14])

The implemented system is evaluated in terms of frame rate, required memory and network bandwidth. Besides that, a user study is performed, which examines the effectiveness of the VR exploration regarding spatial knowledge acquisition in unknown environments.

A paper summarizing the work of this thesis is also submitted to the International Symposium on Mixed and Augmented Reality (ISMAR) 2016. At the time of writing, the submission is currently under reviewing.

The rest of the thesis is structured in the following way: Chapter 2 introduces the main concepts of 3D reconstruction methods. The state-of-the-art in real-time dense 3D reconstruction, network transmission, mesh computation and visualization and exploration is reviewed in Chapter 3. Afterwards, Chapter 4 covers the methodology, where the concepts of the developed reconstruction pipeline are explained along with the used methods. Implementation details and instructions, how to use the system, are presented in Chapter 5. Chapter 6 describes both, the system performance, as well as an experimental user study, targeting spatial knowledge acquisition during live exploration. Finally, the thesis ends with a conclusion in Chapter 7.

CHAPTER 2

Fundamentals of Real-Time Surface Reconstruction

Before reviewing related work and introducing the methodology, the basic concept of real-time surface reconstruction is explained. Reconstruction methods work either in real-time or offline. While offline approaches compute the model from a captured camera stream in a post-processing step, real-time approaches can reconstruct the scene while it is scanned [NIH+11]. This enables live feedback and allows to see which regions of the scene still need to be scanned and if the reconstruction process is working without failure. Furthermore, these methods can be used in robotics as mentioned in Section 1. This thesis focuses on real-time techniques.

2.1 Depth Sensors

To be able to build a 3D model of the environment, one needs to sense not only the color but also the depth of the scene, i.e. the distance to the nearest object. With that information, it is possible to reconstruct a 3D point in space. Standard RGB cameras only provide color images but not depth information. There exist approaches to estimate this information from the available color data, which can however lead to wrong values and require additional computational resources.

RGB-D cameras spare this step since they are able to measure the depth directly. An example of such an RGB-D camera is the Xtion Pro Live from Asus [Asu16], which works the same way as the first version of the Microsoft Kinect. It projects a known infrared dot pattern into the scene, which is then captured with a built-in infrared sensor. The depth of the scene is estimated by analyzing the distortion of this pattern. This principle is called structured light. Besides the scene depth, the camera measures the color with a separate RGB sensor. Figure 2.1 shows an image of the Asus Xtion. Another type of

RGB-D sensors are Time-of-Flight (TOF) cameras, as for instance the second version of the Microsoft Kinect. They emit infrared light pulses and measure the time until the light is reflected back. The larger the distance to an object, the longer the measured time is. The drawback of using both types of RGB-D cameras is, that they do not work in direct sunlight since the infrared part of the sunlight interferes with the infrared dot pattern or light pulses. As a consequence, the cameras can only be used indoors in a reliable way. A more robust and precise depth estimation can be performed with laser scanners, however, they are also more costly. Within the thesis, the Xtion Pro Live camera is used since only indoor scenes are scanned.

Figure 2.1: RGB-D camera Xtion Pro Live from Asus [Asu16].

2.2 Scene Representation

Independently of the real-time property and the camera type, a further characteristic of a reconstruction system is the used scene representation. It defines how the reconstructed environment is stored in memory and strongly affects the reconstruction process. Generally, the model can be represented either in a sparse or dense way [NIH+11]. In contrast to sparse models, a dense representation allows to maintain fine details of the geometry and is thus well suited for exploration and interaction with the captured scene.

A Signed Distance Function (SDF) [CL96] is a common type of such a dense representation and is also applied in InfiniTAM, the reconstruction framework used in this thesis [KPR+15]. A SDF stores the distance to the closest surface for each point in space, where all values on one side of the surface are negative and all values on the other side of the surface are positive. The scene is therefore represented implicitly as the zero crossing of this function. A drawback of dense representations is the high amount of required memory and, especially for large scenes, an efficient data structure is necessary. An overview of dense data structures suited for large scales can be found in Chapter 3.

2.3 Reconstruction

The actual reconstruction process of real-time methods mainly consists of two steps: Camera pose estimation (or camera tracking) and integration of the current camera

information into the model [NIH+11] (mapping).

Pose Estimation The goal of camera pose estimation is to track the position and orientation of the scanning camera by using its images. This process is also referred to as visual odometry [SSC11]. By comparing the information of two camera frames, it is possible to estimate how the camera moved and rotated in between. As a consequence, the complete camera trajectory can be reconstructed. Like the scene representation, the method for camera pose estimation can be classified as either dense or sparse. Sparse camera tracking approaches use only parts of the available image information. They try to compute and find corresponding points in both images. Then a transformation is computed which best maps the points from frame one onto the points of frame two. This transformation constitutes the camera transformation between the two frames. Corresponding points can be found using RGB image features like SIFT or SURF[PPS13] and the transformation can be estimated using a procedure like RANSAC [FB81]. RANSAC tries to find the best transformation by eliminating wrong point correspondences. Instead of only using a set of points, dense methods estimate the transformation by using the information of every pixel. In order to still achieve real-time performance dense methods often utilize the GPU. The computations for each pixel can performed very well in parallel. A widely used method within dense camera tracking is Iterative Closest Points (ICP) [BM92]. It tries to find the camera transformation matrix by iteratively aligning two 3D point clouds. The point clouds can be generated from the depth images and the internal camera parameters. ICP is also used in this thesis and is therefore explained in more detail in Section 4.

Camera Drift In order to achieve well aligned reconstructions, it is fundamental to keep the error in the camera pose estimation step as small as possible. Since only the relative camera transformations between two frames are computed, the error sums up over time. This problem is known as camera drift [SSC11]. Especially for large scenes, it leads to unpleasant results: walls or floors are not straight anymore and reconstructed objects can overlap. Figure 2.2 shows a reconstructed scene with notable camera drift, where it can be seen that corresponding parts are strongly misaligned. To mitigate this problem, instead of a frame-to-frame tracking, a frame-to-model camera tracking can be applied [NIH+11]. This means, that the camera transformation is not estimated directly between two successive camera frames but between the current frame and a frame which is synthesized from the current model. As a consequence, the resulting model is much more aligned to the model, but the general problem of camera drift still remains. The impact of drift can be perceived especially when scanning regions, that have already been scanned and reconstructed before. By detecting this loops, i.e. regions which have already been scanned, the accuracy of the reconstruction can be improved. Once a loop is detected, the model has to be transformed in such a way to align the corresponding parts. While there exist solutions that perform the so called loop closure [AFDM08] for point clouds, at the time of writing this thesis, there does not exist any publicly available

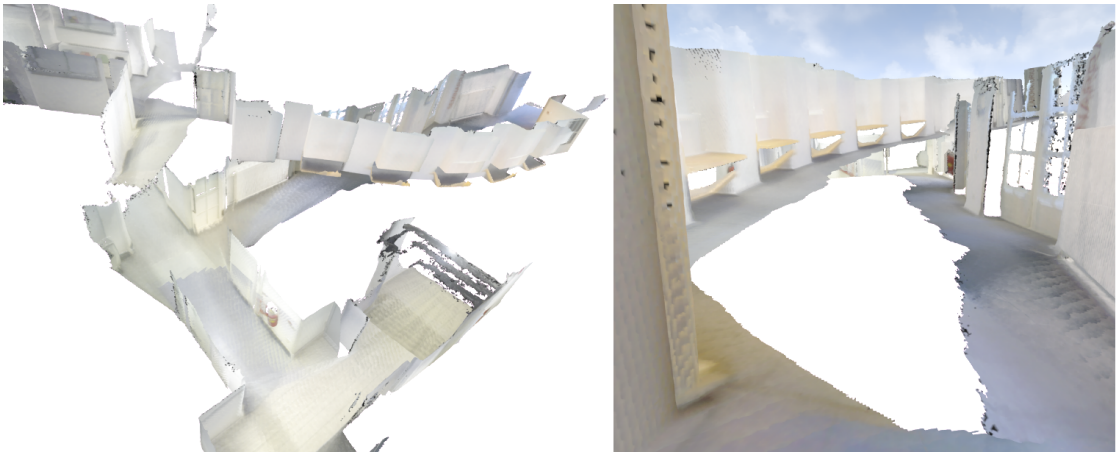


Figure 2.2: Distorted reconstruction due to camera drift.

method to compute this in real-time for large-scale scenes stored as a dense SDF. Thus, the implementation in this thesis does not integrate loop closure.

Data Integration Besides camera pose estimation, the second step of a real-time surface reconstruction method is the integration of the current camera information into the 3D model. By using the inverse of the estimated camera transformation matrices, all images of the camera stream can be mapped to a common world coordinate system. The 3D world position of the individual pixels can then be inferred with the depth information of the individual images, resulting in a point cloud. Finally the individual 3D points of this cloud need to be integrated and combined with the existing model. Since this step depends entirely on the used scene representation, it is not described in more detail here.

CHAPTER 3

State-of-the-Art

At the time of writing, there is no prior art that combine 3D reconstruction, network streaming and a mesh extraction component into a single application. However, all these areas are well studied itself and the most relevant approaches are presented in the next four subsections.

3.1 Dense Real-Time Surface Reconstruction

3D surface reconstruction methods, which incrementally fuse camera data into a single representation of the scene, got very popular in recent years, especially since the release of the Microsoft Kinect RGB-D camera [Cor16a].

3.1.1 KinectFusion

One of the first algorithms, which is able to reconstruct scenes in a dense way in real-time is called KinectFusion[NIH+11]. It estimates the camera pose with ICP where it uses a synthetic depth map from the current model to limit the camera drift. As a model representation, it uses a dense volumetric SDF. KinectFusion achieves real-time rates because both tracking and integration is performed on the GPU. The major limitation of KinectFusion is, that it can only scan desktop-sized scenes and fails to reconstruct larger environments. This is due to the fact, that the SDF is stored in a regular 3D voxel grid on the GPU. Therefore, both space which is occupied by objects and empty space, is stored the same way, which leads to a large memory footprint. By decreasing the size of the individual voxels in order to store finer geometric details or by increasing the total scene size, the amount of the required memory increases cubically. On a GPU with two GB memory, a volume with 512^3 can be stored. This corresponds to a cube with around 2.5m side length when using a voxel resolution of 0.5 cm. There are now several methods, which extend the KinectFusion algorithm to overcome the space limitation.

3.1.2 Extending the Scale

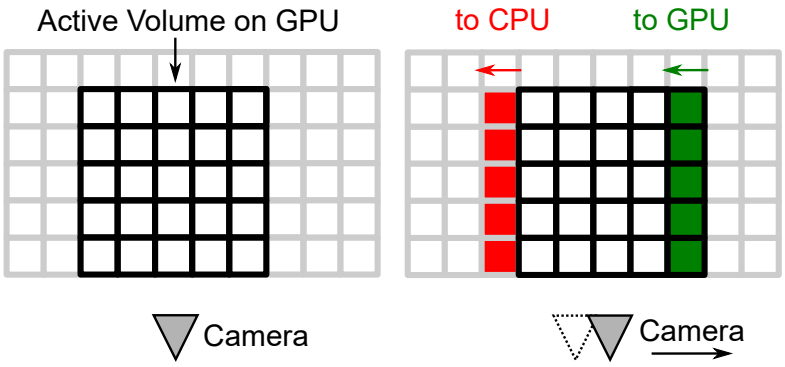


Figure 3.1: Principle of a Moving Volume approach in 2D. Only a small parts of the scene is stored on GPU (black), the rest is maintained on CPU (gray). When the camera moves to the right, the red blocks are streamed from GPU to CPU to make space for the new area (green).

Moving Volume Approaches Moving Volume approaches reconstruct the scene by dividing it into an active and an inactive volume, where only the active volume is stored inside the GPU. This volume is translated according to the estimated camera motion. It always holds those parts of the scene which are in front of the camera. The scene is still represented as a regular grid, but the parts of the scene that fall out of the current active volume are streamed to the CPU to make space for new data on the GPU. The principle is illustrated in Figure 3.1. That way, the supported scene size is solely limited by the available CPU memory. However, the size of the active volume is still restricted. This has the effect that either fine details cannot be reconstructed or that only close-by structures can be scanned (even if the sensor supports a larger depth range). Point Cloud Library's LargeKinfu [HF16], Moving Volume KinectFusion [RM12] or the most recent Kintinuous project by Whelan et al. [WKJ+14] implement this moving volume idea. In Kintinuous, the parts which are streamed out of the active volume to the CPU are converted to point clouds first. This further reduces the required memory on the CPU, but existing point information on CPU is not considered when revisiting an already scanned area. If the information is contained as SDF, it can be uploaded to the GPU again and fused with new data. Implementations of LargeKinfu and Moving Volume KinectFusion are available for public use and at the time of writing this thesis, the authors of Kintinuous have also released a public version of their code.

Hierarchical Approaches Besides the moving volume approaches, there are several methods which try to increase the supported scale by ignoring empty space. This can be achieved by using octrees or similar hierarchical data structures. In a SDF scene representation, all data that is further away from the closest surface than a certain threshold can be considered as empty space. Zeng et al. use a ten level octree on the

GPU and are able to reconstruct a space about ten times larger (8 m x 8 m x 2 m) than the original KinectFusion algorithm at the same resolution [ZZZL13]. Steinbrücker et al. further extend the scale and can scan a corridor consisting of nine rooms (45 m x 12 m x 3.4 m) while still fitting in their 2.5 GB GPU memory [SKCS13]. In contrast to Zeng et al. even the color information of the scene is integrated. They apply a multi-resolution strategy, where not only leaves but also octree branches can hold scene data. Measurements with high distance to the sensor (including more noise) are stored with a lower resolution at higher levels of the octree and close surfaces are represented at lower levels with finer a resolution. A voxel size of up to 0.5 cm is used. A CPU version of this approach is available for public use, however this does not contain a camera tracking component. Chen et al. avoid octrees including their pointer overhead and propose a regular spatial subdivision with are more shallow hierarchy [CBI13]. This results in a better performance and enables efficient GPU-CPU streaming of sub-parts of the scene to support scales only limited by the CPU memory.

Voxel Block Hashing Even though Chen et al. improve the speed, the drawback of hierarchical data structures is still, that the required traversal of the hierarchy generally introduces additional computational overhead, which leads to a reduced performance. Especially for mobile reconstruction systems as targeted in this thesis, the performance is critical. To avoid these performance issues, Nießner et al. introduce a technique called Voxel Block Hashing, which is both fast and space efficient [NZIS13]. The main idea is, that they store data only near actual surfaces but without resorting to a scene hierarchy. This is achieved by introducing a spatial hash function which maps a 3D voxel position from world space to a hash table entry. The hash table is maintained on the GPU and only data (SDF and color) for currently visible voxels is stored on the GPU memory. All other voxels are streamed out to the CPU similar to the Moving Volume techniques. Voxel Block Hashing requires only about 10% of the memory in comparison to a regular 3D grid and is superior to other techniques regarding the frame rate. It is integrated in the InfiniTAM framework [KPR+15] and is explained in more detail in Section 4.1. Figure 3.2 shows a comparison of storing the scene in a regular grid (as in standard KinectFusion or Moving Volume approaches), a hierarchical grid or using Voxel Block Hashing.

3.1.3 Improving the Camera Pose Estimation

Most of the mentioned surface reconstruction approaches integrate depth-based ICP for camera pose estimation. For environments with a lot geometric features it works well, however for some scenes, such as hallways, it likely fails. The low geometric information alone does not allow to compute a unique pose in these cases. In order to get a good pose estimate, not only the depth but also the color information of the RGB-D sensor can be used. A common approach is, to estimate sparse visual features and find matches between image pairs. For example, Henry et al. [HKH+12] use FAST features in combination with ICP. In order to better exploit all the available color information, Steinbrücker et

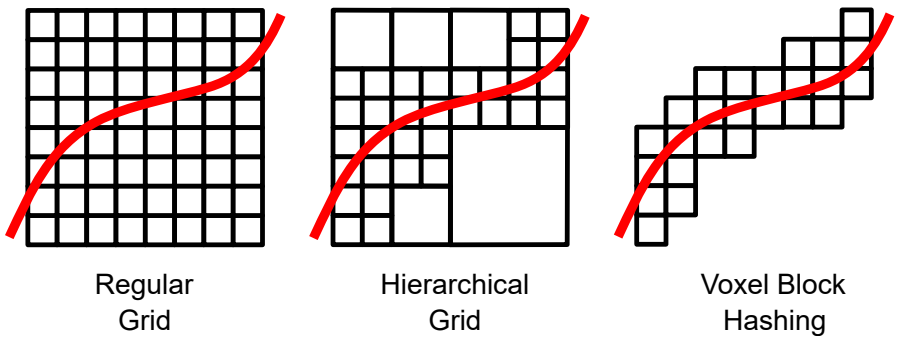


Figure 3.2: Comparison of storing the scene in a regular grid (left), a hierarchical grid (middle) or using Voxel Block Hashing (right).

al. propose a dense estimation where they minimize both the photometric error and geometric error between two consecutive RGB-D image pairs [SSC11]. In the Kintinuous project, Whelan et al. integrate Steinbrücker's RGB-D tracking and combine it with frame-to-frame ICP [WJK+13].

Loop Closure As mentioned in Section 2, another way to improve camera pose estimation is to integrate a loop closure procedure. Current state-of-the-art approaches mostly apply appearance-based methods using a pose graph to detect loops [WKJ+14, LM13]. In a pose graph, each node corresponds to a keyframe and stores a camera image with the corresponding pose estimate. A new node is inserted, whenever the camera moved more than a defined threshold or when some time had passed. The main idea is then to compare the current camera image with the previous keyframes using sparse visual features based on the color image. Whenever a loop is found, a new edge is added to the pose graph and the graph with its poses can be corrected by running a pose graph optimization algorithm. After adjusting the existing pose estimations, the current 3D model is deformed. Usually, a point-based representation is used in these approaches and in this case, the deformation can be done efficiently. A drawback of such appearance-based methods is, that they do not work well in low-light environments or when the camera is moved too fast due to motion blur. Apart from that, to be able to detect loops, previous regions have to be seen from a similar viewpoint, so that corresponding features can be found. Recently, Fioraio et al. introduced a method which does not rely on the color information and which works for large scale volumetric models. They are [FTF+15] able to perform online global optimization without explicitly performing loop closure. The scene is divided into a number of subvolumes, where each subvolume is assigned a pose, and these poses are optimized by performing geometric registration of the volumes. The subvolumes are fused together in order to get a global map. The authors state that their approach is able to produce results similar to state-of-the-art offline in only a fraction of time. An implementation of this approach is not available to the public at the time of writing.

3.2 Network Streaming

The streaming of a 3D reconstruction depends very much on the representation of the data to be sent. In order to achieve a low bandwidth requirement, a proper compression method is necessary. This method can be either lossless, i.e. all the original information is contained, or lossy. In that case, information is sacrificed to achieve a higher compression rate. Since RGB-D sensors already introduce some noise, a loss of information due to compression can be acceptable if it is below the noise level.

3.2.1 Transmission of Images

Most of the related approaches send the current camera images or images of the current reconstruction as seen from the camera's viewpoint over network. Sending raw images is not feasible, as a Kinect camera produces depth and RGB images with about 45 MB per second. In order to keep the data rate low, the images can be compressed using standard image compression methods or algorithms specifically designed for depth data. Coatsworth et al. suggest to compress RGB images using the lossy JPEG compression and depth images using the lossless DEFLATE algorithm [CTF14]. They evaluated in an experiment that this combination offers a good compromise between compression ratio and computing complexity. In their approach, each image is encoded separately. The redundancy usually observed in a camera stream is not exploited. Nenci et al. also consider this temporal relation between the depth images using the H.264 video codec to achieve a higher compression rate [NSS14]. Since video codecs like H.264 usually only support 8 bit per pixel, they are not directly applicable for depth data with 16 bit per pixel. Moreover, they result in blurred edges which results in strong artifacts in the surface reconstruction. Nenci et al. try to solve this by dividing the depth image into multiple channels, where each channel contains a certain (normalized) depth interval. The channels are then compressed individually. Generally, this scheme is targeted at a lossy compression mode. When choosing lossless compression, it even results in a higher bandwidth (0.31 bytes per pixel) than the approach by Coatsworth et al. (0.27 byte per pixel). A further method for depth maps by Mehrotra et al. exploits the fact that the Kinect depth map accuracy decreases with distance. As a consequence, distant values can be compressed higher without introducing additional error. The authors claim, that their near-lossless method achieves a compression rate between 7 and 16.

3.2.2 Transmission of the 3D Model

Besides compressing and streaming images, the second option is to transmit the reconstructed model, i.e. the point cloud, the mesh or the SDF, itself. In that case, an important requirement besides a low computational complexity and a low data rate is the support for incrementally acquired models. There are a number of methods which make use of octrees to compress the space.

Point Clouds Kammerl et al. developed a real-time approach for point clouds streams [KBR+12]. They encode the structural differences between the octrees of two consecutive point clouds and achieve therefore both spatial and temporal compression. The compression is lossy and depends on the desired voxel resolution of the scene. With a resolution of 5 mm, a compression ratio of 1:27 is achieved (0.45 bytes per point) when compared to the uncompressed point cloud with 12 bytes per point. Kammerl's method is available as part of the Point Cloud Library. Recently, Golla and Klein introduced another point cloud progression algorithm for real-time streaming which is based on local 2D parameterizations [GK15]. They divide the scene into a regular 3D voxel grid. The point cloud within each voxel is compressed separately by parameterizing them as planar patches. The actual point positions are encoded with height maps and holes are represented by occupancy maps. For these two maps, standard image compression techniques are used and the planar patches are compressed using the Lempel-Ziv-Markov chain algorithm. Similar to that, Morell-Gimenez et al. achieve a spatial compression by detecting planes [MOCGR14] in the point data and representing those with a more compact form. However, it is not specifically targeted at the incremental transmission of reconstructed model.

Meshes Apart from streaming point clouds, another option is to compress and transmit the reconstruction data as meshes [MLDH15]. However, meshes are usually not the representation used during scene reconstruction. They have to be extracted first which introduces additional computational cost at the mobile scanning computer. Moreover, two different representations of the same scene would have to be stored on the mobile platform which increases the memory footprint. As a consequence, mesh transmission is not considered as an option in this thesis.

SDF Streaming of volumetric reconstruction data stored as a SDF is not done in related work but one can apply compression methods targeted at arbitrary data. Note, that the scene representation chosen for the 3D reconstruction not only determines the supported scene size but also strongly affects the network transmission. When the model is stored very memory efficient, the possible amount of data to be transmitted is already limited. In case a volumetric SDF model is used, it contains many zeroes or similar numbers. This data can be compressed effectively by using a lossless general purpose algorithm like run-length encoding or more advanced methods such as DEFLATE or bzip2. The latter achieve a higher compression but are also more computational expensive [Sal04].

3.2.3 Network Protocols

Independently of the chosen compression scheme, the data needs to be sent over network with an appropriate network protocol. Depending on the data to be streamed, the network protocol may need to satisfy different requirements. If for example images are transmitted and a real-time rate is required, the latency of the transmission is critical, whereas some packet losses might be acceptable. If the reconstructed model is streamed,

the most important factor is the throughput. The Transmission Control Protocol (TCP) is connection-oriented, ensures ordered data delivery and is reliable, i.e. no data is lost [For02]. The User Datagram Protocol (UDP) on the other hand is connectionless, unreliable and does not guarantee ordered delivery, but comes with less overhead than TCP. The advantage of these two main protocols is, that they are known and widely supported. Besides those standard options, there exists the Stream Control Transmission Protocol (SCTP) [SM01], which is reliable, connection-oriented and allows both ordered and unordered data delivery. In contrast to TCP, it preserves message boundaries like UDP. Moreover, it supports multi-streaming, i.e. multiple streams between two endpoints. When comparing SCTP with UDP and TCP regarding their performance, SCTP achieves the highest throughput but with the highest delay [NM12]. Another state-of-the art protocol is the UDP-based Data Transfer Protocol (UDT) [GG07], which is intended for transferring very large datasets over high-speed wide area networks. In contrast to standard UDP, it is reliable and connection oriented.

3.3 Surface Extraction

To supply live feedback during scanning, most reconstruction frameworks apply raycasting to visualize the reconstructed surfaces. However, as already mentioned, a mesh representation is desired for exploration purposes and has to be extracted from the underlying model. As the network streaming, the mesh extraction should work for dynamic data. Thus, it needs to be able to update in real-time upon model changes.

3.3.1 Mesh Extraction from Point Clouds

To compute a mesh from a dense point cloud, a popular choice is Poisson Surface Reconstruction [KBH06], which first computes a mathematical model before triangulating it. This method however does not perform well for reconstruction data since it is developed for continuous surfaces without holes.

In order to extract a mesh from point clouds, which are measured by a camera, the Greedy Projection Triangulation (GPT) can be used [MRB09]. It directly connects the points instead of computing a mathematical model first. The triangulation happens locally, where only the neighborhood of a point is taken into account. The method works in real-time for incremental noisy data sets. The major issue of this approach is, that it keeps all the input points. As a result, a mesh with a huge amount of triangles is generated. Besides requiring a lot of memory, it hinders interactive exploration of the mesh when reconstructing large spaces .

To tackle this issue, Ma et al. introduce an algorithm to detect planar regions and represent those with fewer points, i.e. larger triangles [MWB+13]. Planes are detected by using a region growing segmentation based on the point curvature. The segmented points of a plane are then converted to a triangular mesh using their new QuadTree-Based (QTB) algorithm. Within this method, a quadtree is applied for point decimation. All

boundary points of a planer region are contained and all interior points are replaced by a set of new ones. The number of new vertices should be as small as possible without causing skinny triangles, i.e. triangles with very small angles. All points, not belonging to any planar regions, are triangulated using GPT. While this approach reduces over 80% of planar points, it is not designed for incrementally acquired point clouds. Whelan et al. improve the robustness and speed of the algorithm and most importantly extend it to allow incremental mesh growing [WMB+15]. Apart from that, they added the capability to maintain the color information of the reduced planar segments by automatically generating textures. This incremental meshing is integrated in the Kintinuous project, however the publicly available version does not include it but uses basic GPT instead.

3.3.2 Mesh Extraction from SDF

To create a triangular mesh from a volumetric representation such as a SDF, the standard choice is to apply the Marching Cubes algorithm [LC87]. This algorithm steps through the volume and takes eight SDF values at a time, which would lie on the corners of an imaginary cube. Using these eight values, it is possible to find the triangles, which represent the surface passing through that cube. The final mesh is then just the set of triangles of all cubes. This method is applied in the thesis and is explained in more detail in 4.7. A drawback of this method is, that it produces a very high number of triangles if one wants to generate a mesh with a high level of detail. The tessellation factor, i.e. the size of the triangles, is determined by the resolution of the volumetric SDF and it is the same all over the mesh, also for planar regions. Figure 3.3 shows the dense triangulation of a reconstructed couch.

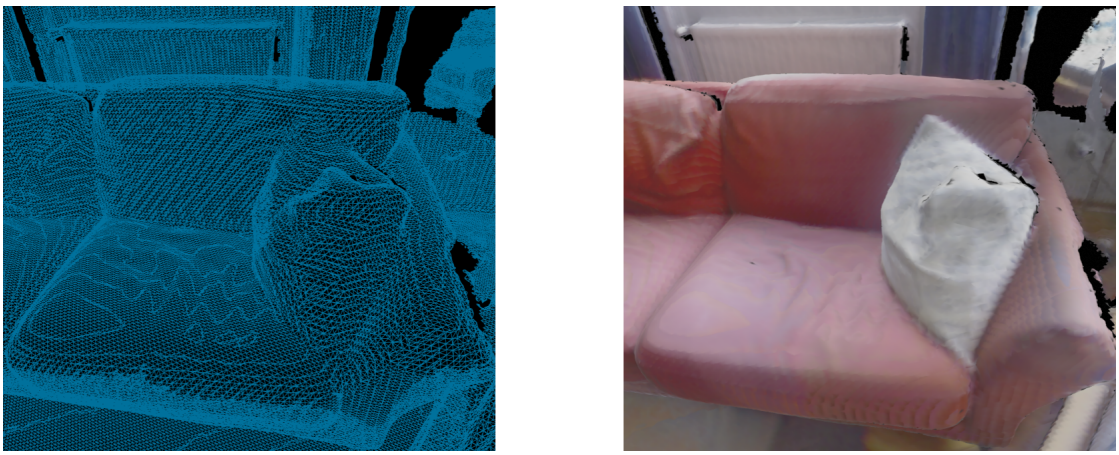


Figure 3.3: Marching Cubes Triangulation from a volumetric 3D reconstruction with a voxel resolution of 1 cm. The left image shows the triangulation and the right image shows the corresponding colored mesh.

There exist adapted algorithms based on the original Marching Cubes algorithm like Marching Tetrahedra or Adaptive Marching Cubes, which try to solve this issue [NY06].

Steinbrücker et al. integrate Marching Cubes in their reconstruction pipeline to dynamically extract a mesh from the SDF model [SKCS13]. Since they store the model at different levels in an octree with different voxel resolutions, also the corresponding mesh is computed with different levels of detail. Areas that are scanned closely, produce smaller triangles than far away regions. Further improvements, such as point decimation in planar regions, are not integrated. The dynamic update of the mesh is achieved by recomputing the mesh of the corresponding parts of the octree.

3.4 Visualization and Exploration

3.4.1 Visualization using 3D Engines

When the triangular mesh representation is available, the standard rendering pipeline using rasterization can be applied. For that purpose, there exist a number of game engines which ease the development process. These engines already incorporate advanced techniques to produce realistic visual and physical effects, such as collision detection, while still providing a high performance. Among the current state-of-the art engines are Unreal Engine 4 (UE4) [Epi16] and Unity 5 [Uni16]. Both have multi-platform support and are free for personal use. UE4 is based on C++ and has Blueprints, a visual scripting system, which allows fast prototyping without any coding. Unity is based on C# and its main advantage is the large asset store, a library which offers a lot of additional functionality. Since in this theses the goal is to explore a reconstruction while still scanning, the main requirement is the ability to render a large dynamically changing mesh. Both in UE4 and Unreal this can be achieved using procedural meshes. UE4 has more advanced lighting capabilities, however the lighting is less important since the reconstruction already contains the measured color. Besides that, another requirement is the support for VR devices, such as an HMD or an ODT, in order to allow the desired immersive exploration. As both have support for an HMD, but not for any ODT, they can be used equally well. The treadmill has to be integrated in both scenarios. In this thesis, Unreal Engine 4 is used, mainly because it builds on C++, which makes the integration with the rest of the reconstruction pipeline easier.

3.4.2 Exploration with VR Hardware

Head-Mounted Displays HMDs (for VR) are displays, worn on the head like glasses, and provide a much higher field of view compared to a normal desktop screen. They enable a stereoscopic view by showing separate images for both eyes. Moreover, these devices track the position and orientation of the user's head which allows to adjust the visualization according to the current head movement. This way, the user is able to explore the virtual surroundings in a natural way just by turning the head. State-of-the art devices include the Oculus Rift [Ocu16a], HTC Vive [Hig16] or PlayStation VR [Son16]. Besides a high display resolution and field of view, a high refresh rate and low latency is critical to fully immerse the user in the virtual world and to avoid symptoms like headache or nausea [MS92]. The so called cybersickness can also be introduced by

the VR content when different sensory cues are in conflict, e.g. when experiencing a virtual rollercoaster ride, while actually sitting at a desk. To enhance the VR experience and to limit motion sickness, the user can move in the virtual environment by actually walking the the real world.

Omnidirectional Treadmills While HMDs allow to look around freely, walking more than a few steps is usually not possible because of the limited available space in the real world. An ODT can be used to tackle this problem. Such a device enables a person to walk in virtual space while actually staying at the same position. In contrast to a typical treadmill, the person can move in any direction. Generally an ODT can be either active or passive. Active systems track the user's movement and try to maintain a still position by moving the user in the opposing direction. With such an approach, the user performs a natural walking movement like in the real world. In passive systems, the users stands on a stationary platform and is fixed to the ODT. The user moves by sliding the feet over a low-friction surface. These systems are lighter and more affordable than active system, but enable a less natural experience. The Infinadeck [Inf16] is an example of an active system. The Cyberith Virtualizer [CH14] and the Virtuix Omni [Vir16] are two VR treadmills, which are both passive systems. In this thesis, the Oculus Rift as well as the Cyberith Virtualizer are integrated in the immersive exploration setup.

CHAPTER 4

Methodology

In this chapter, the concepts of the implemented 3D reconstruction system are introduced and the applied methods are explained. Implementation details and instructions, how to use the system, can be found in the following Chapter 5. The system is based on the InfiniTAM framework [KPR+15] from Oxford University which uses a volumetric SDF as a scene representation. InfiniTAM integrates Voxel Block Hashing [NZIS13] to store the model efficiently in terms of both memory usage and computational performance. UE4 [Epi16] is used for visualization and exploration purposes.

The thesis implementation allows to scan and reconstruct the geometry of large environments using an off the shelf camera like the Microsoft Kinect, which delivers both color and depth information. The acquired reconstruction data is incrementally streamed over network, and the gradually expanding reconstruction can be explored in an immersive VR setup. Since network transmission is involved, the system is divided into a server part and a client part as illustrated in Figure 4.1. Colored parts indicate novel features. Yellow modules modify or extend existing functionality, whereas green modules are newly added. The server performs the actual scene reconstruction, which consists of the camera pose estimation, the integration of the camera images into the model, the swapping of data between the GPU and the CPU memory and the raycasting procedure. This raycasting step creates a synthetic depth map from the volumetric model, used both for camera tracking and live feedback. The network transmission of the volumetric model is performed in a separate procedure after the GPU - CPU swapping stage. The client part of the framework allows to explore the reconstruction. It listens for incoming voxel data and assembles it again, so that the client has an exact copy of the server-side model. Since a mesh representation is used for exploration, it is extracted from the volumetric representation and dynamically updated after integrating new voxel data. The computed mesh is passed to the visualization module without interrupting it. As already mentioned, this visualization and exploration module builds upon UE4 and runs in a separate process.

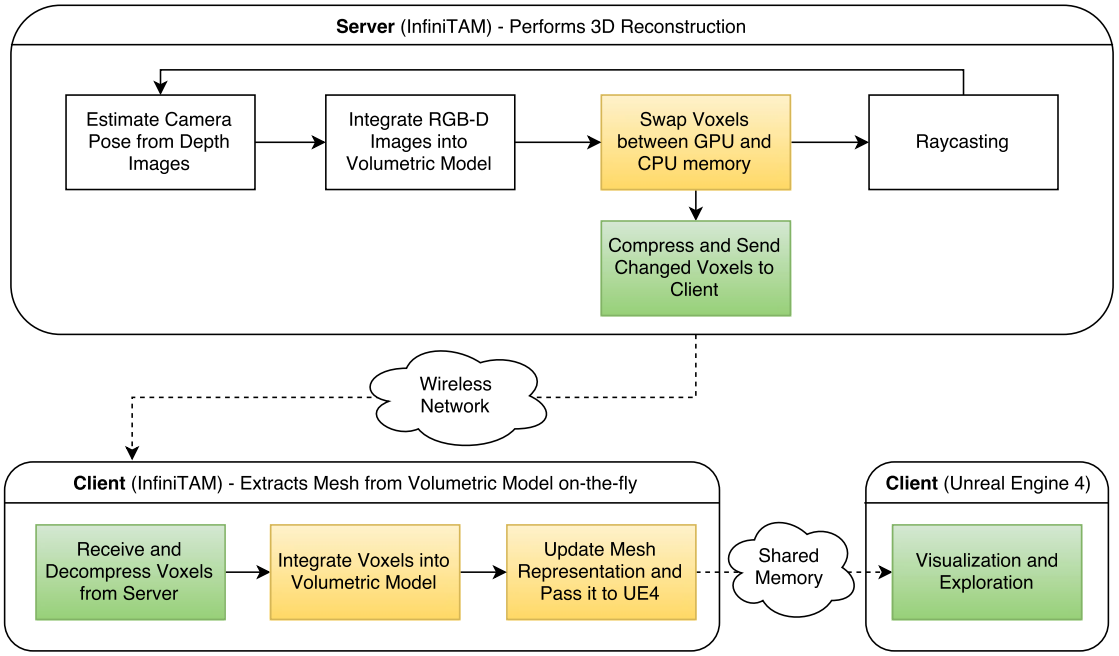


Figure 4.1: Overview of the reconstruction pipeline. The server performs the scene reconstruction and sends the model incrementally to the client. The client maintains an up-to-date mesh representation and enables live exploration of the reconstructed scene. Yellow color indicates, that corresponding parts are modified functionality

Since main parts of the reconstruction system rely on properties of the Voxel Block Hashing data structure, it is introduced first. Afterwards, the individual parts of the implemented pipeline are explained.

4.1 Model Representation

4.1.1 Truncated Signed Distance Function

During the reconstruction, the model is represented as a three-dimensional truncated SDF (TSDF) [CL96]. A SDF stores for the distance each point in space to the closest object surface, where the two sides of a surface are distinguished by the sign of the distance. The zero crossing of the SDF then determines the surface location. In contrast to a standard SDF, a TSDF only considers values within a certain range from the surface. All values further away are set to the maximum distance. The considered range of distances is called truncation band. Figure 4.2 shows the TSDF for a simple rectangular object in 2D, where the width of the truncation band is set to three. While in the implementation, the Euclidean distance measure is used, the Manhattan distance is applied in this example in order to show nicer numbers.

3	3	3	3	3	3	3	3	3	3
3	3	2	2	2	2	2	3	3	3
3	2	1	1	1	1	1	2	3	3
2	1	-1	-1	-1	-1	-1	1	2	3
2	1	-1	-2	-2	-2	-1	1	2	3
2	1	-1	-1	-1	-1	-1	1	2	3
3	2	1	1	1	1	1	2	3	3
3	3	2	2	2	2	2	3	3	3
3	3	3	3	3	3	3	3	3	3

Figure 4.2: Example of a TSDF (with Manhattan Distance) for a simple object The surface is defined as the zero crossing and is illustrated in red.

4.1.2 Voxel Block Hashing

The basic data structure to store such a function is a regular voxel grid, where each voxel stores a distance value. The position of the voxel is encoded implicitly by its index within the grid. Since all values outside the truncation band are the same, they do not contain any valuable information and thus this representation is not space efficient.

Voxel Block Hashing [NZIS13] overcomes this limitation and is applied in the developed system. In this approach, the world is divided into so-called voxel blocks, where each of these blocks holds n^3 voxels. The position of the voxels within the block is determined by its index. The main difference to the basic representation is, that information is not stored densely for every point in space, but only for regions which are close to a surface. In order to store a voxel block, at least one of its voxels has to be inside the truncation band of the TSDF. Note, that in this case, there is no implicit connection between the voxel block position and its index, because the data is stored in an unstructured way. Instead, the blocks are addressed using a spatial hash function, which maps points in space to a hash table. Given a voxel block world position, one can find the corresponding hash table entry. But given the index of an entry, the world position cannot be retrieved. As a consequence, besides the information, if the corresponding block is actually allocated, each hash table entry stores the block position. The hash function to compute an index h from a 3D voxel block position with integer world coordinates x, y and z can be seen in Equation 4.1,

$$h = ((x \times p_1) \oplus (y \times p_2) \oplus (z \times p_3)) \mod n \tag{4.1}$$

where n is the number of hash table entries and p_1, p_2 and p_3 are large prime numbers. In InfiniTAM, these numbers are set to 73856093, 19349669 and 83492791. The $\oplus$ operator defines a logical XOR and $\mod$ stands for the modulo operator.

The hash table usually contains a lot of empty entries to minimize hash collisions. In order to avoid allocating a lot of empty memory, the actual voxel data of the block, i.e. TSDF values and colors, is not saved in the hash table directly, but in an additional voxel block array. Each hash table entry maintains a reference to this array. Figure 4.3 shows the principle of the data structure in two dimensions. For illustration purposes, one voxel block only holds 4^2 voxels. In the implementation, each block stores 8^3 voxels.

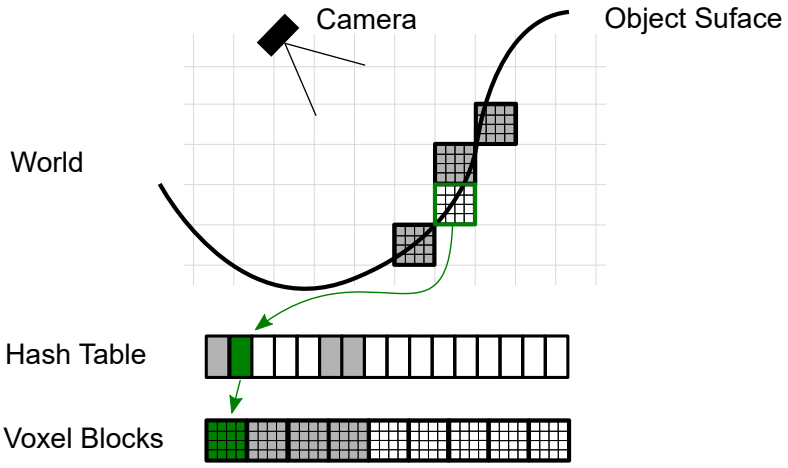


Figure 4.3: The principle of the Voxel Block Hashing data structure. A hash function maps world positions to a hash table. Each hash table entry maintains a block world position and a reference to the voxel block array. The voxel block array stores the actual TSDF and color data.

Handling Hash Collisions Note, that the hash function is not injective, which means several world positions can be mapped to the same index. Such hash collisions are handled by extending the hash table with an additional array, which holds all collided entries. This array is called the unordered part and the standard hash table region is called the ordered part. Figure 4.4 illustrates this concept. Each hash table entry (either in the ordered or unordered part) stores a possible reference to a further entry in the unordered part. Therefore, all voxel blocks with the same hash index form a linked list. Voxel blocks, stored in the ordered part, can be retrieved in constant time, whereas a

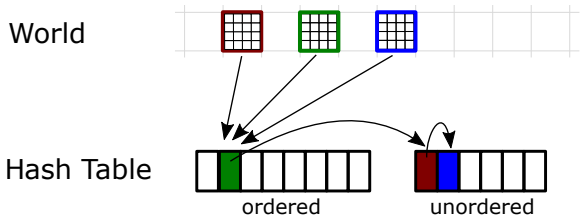


Figure 4.4: Division of hash table in an ordered and an unordered part to resolve hash collisions. Three voxel blocks map to the same hash index in the ordered part. The green block, which is allocated at first, is stored at that index. References are maintained to entries in the unordered part, where the other two blocks are stored.

lookup in the unordered part involves a little overhead, since the linked list has to be traversed. However, by creating a large hash table, which ensures a low load factor, the number of collisions can be reduced and the performance remains high.

Generally, Voxel Block Hashing allows for a very efficient implementation, even faster than the original KinectFusion. According to Kähler et al., InfiniTAM also runs on a Tablet with more than 30 fps.

4.2 Camera Pose Estimation

The first step in the reconstruction pipeline is the camera pose estimation. InfiniTAM integrates the ICP approach used in the orignial KinectFusion implementation [NIH+11], which is solely based on the depth image. Apart from that, InfiniTAM also integrates an option to compute the pose using the color image in a dense frame-to-frame manner but it cannot be combined with the ICP tracker.

4.2.1 Iterative Closest Point

The idea of ICP is to find the transformation which best aligns two point clouds [BM92]. Basic ICP selects a number of points from the first cloud and finds, for each of these points, the closest point in the other cloud. As a result, a set of point pairs is generated where pairs with a too big distance are rejected. This so called data association step is computationally expensive. In order to find the nearest neighbor faster, a kd-tree data structure can be used. After the data association, the translation and rotation, which maps the first cloud onto the other one, is computed. This is achieved by constructing and minimizing an error function such as the sum of squared distances between the associated points (s_i and p_i):

$$Error = \sum \left(Rs_i + t - p_i\right)^2 \tag{4.2}$$

R represents the rotation matrix and t the translation vector to be found. The function has a closed form solution. After aligning the point clouds with the found transformation, the whole process with association and alignment is started again and repeated iteratively until the error converges. Since ICP is a gradient descent method, it relies on a good seed point and can get stuck in a local minimum.

Generally, basic ICP can be applied to depth maps by converting them to 3D point clouds first. However, since basic ICP is time-consuming, a more efficient variant is used in the field of real-time 3D reconstruction. It can be computed in parallel on the GPU and allows to use all the input data instead of just a selection of points.

Projective Data Association One important feature is the usage of projective data association, which makes it possible to find corresponding points very fast without the need for a special data structure. The idea of this data association is, to project the points from one depth image into the other one and associate those points that fall on the same pixel. A point-pair is rejected if the depth difference of the pixels is too big or if the surface normals are not similar enough. Since in live camera tracking, the pose of the current frame is unknown, this projection cannot be performed directly. However, when

scanning the scene at 30 frames per second, the camera motion can be assumed to be very small between consecutive frames. For this reason, the current camera pose estimate is initialized with the pose estimate of the previous frame in the first ICP iteration.

Point-to-Plane Error Metric Another difference to basic ICP is the use of a point-to-plane error metric. Instead of considering point-to-point distances, the goal is to minimize the distances between the points of frame number one and the tangent planes at the corresponding points of frame number two. According to Rusinkiewicz, this is slower but usually results in a better convergence rate [RL01]. Equation 4.3 shows the error function for this case.

$$Error = \sum \left((Rs_i + t - p_i) \cdot n \right)^2 \tag{4.3}$$

The difference to Equation 4.2 is the added scalar product with the normal vector n_i (at point p_i). This error function can be minimized using a general non-linear least squares algorithm like Levenberg-Marquardt or Gauss-Newton [Bjö96]. The latter is applied within InfiniTAM. Figure 4.5 illustrates the difference between the point-to-point and the point-to-plane distance.

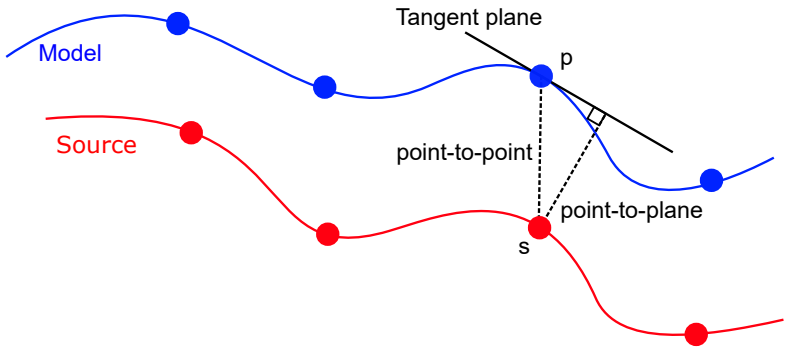


Figure 4.5: Difference between point-to-point and point-to-plane distance. The blue curve represents the reconstructed model and the red curve represents the current camera information, which needs to be aligned to the model. The model point p and the source point s are associated with each other. The point-to-point distance is given as the direct distance between the two points. The point-to-plane distance is the distance between s and its orthogonal projection on the tangent plane of p.

The ICP procedure can be performed multiple times on different resolutions of the input images to further improve the convergence behavior. At first, the pose is computed on a coarse resolution and then, this pose deals as initial estimate for the finer levels, where it is iteratively refined. In the InfiniTAM framework, a five scale image pyramid is used, where in the first three coarse levels, only the three Degree Of Freedom (DOF) camera rotation is estimated. In the last two finer levels, the full six-DOF pose is computed.

4.2.2 Limiting the Drift

Frame-to-Model Tracking In order to limit the drift of the pose estimates, a frame-to-model approach is applied instead of frame-to-frame tracking [NIH+11]. For that purpose, a depth image is synthesized from the current volumetric model using raycasting. To compute the relative camera translation between the current and the previous frame, the synthetic depth image is used in combination with the current camera depth image, instead of two consecutive camera images. Procedures to detect and close loops are not integrated in the reconstruction pipeline. As a result, even the frame-to-model tracking leads to notable drift in larger scenes.

Integration of Inertial Measurement Unit Kähler et al. propose to integrate an Inertial Measurement Unit (IMU), which senses the orientation with a gyroscope, to improve the pose estimates [KPR+15]. They do not apply any form of sensor fusion, but just replace the ICP rotation estimate with the relative IMU rotation. Since only the 3-DOF camera translation has to be estimated with ICP, Kähler et al. use a reduced image pyramid with two levels instead of five. Besides limiting the drift, the IMU integration reduces the required computational which makes it possible to achieve real-time frame rates on Tablets. The publicly available InfiniTAM version only allows to load prerecorded IMU data. The thesis implementation enables to use also live data of an Android device by streaming its orientation via UDP to InfiniTAM.

4.3 Data Integration

After the pose of the camera is computed for the current frame, the depth and color information can be integrated into the volumetric model. Each voxel of the model stores a running average of the color and the TSDF information, where the information of the last n measurements is considered. InfiniTAM uses a value of 100 for n per default.

Find Visible Voxel Blocks The first step of the data integration is to determine which voxel blocks are actually visible, and thus need to be updated. Since the traversal of all voxel blocks is too expensive, a list of visible voxel blocks is maintained in an incremental way. In each frame, it is examined, if the blocks visible in the previous frame are still visible. This is achieved by projecting all eight corners of a block to the current image plane (using the estimated camera pose and the intrinsic camera parameters). If any corner falls inside the image boundary, the block is visible. Blocks, which are not visible anymore, are removed from the list. Previously invisible blocks are found the inverse way. The pixels of the current depth image are projected into the world space. The corresponding blocks are found by evaluating the spatial hash function and performing a lookup in the hash table. If the voxel block is not stored yet for a given position, a new one is allocated. Finally, the block is added to the list of visible blocks.

Update TSDF and Color The actual integration is performed in a similar way than in the original KinectFusion system. All the visible blocks are transformed to the current camera coordinate system using the inverse of the estimated camera pose. The individual voxels of those blocks are then further projected into the current depth image using the known intrinsic depth sensor parameters. The depth of the projected voxels and the values of the depth image at the corresponding positions are compared. Whenever the difference (which represents an SDF value) is within the truncation band or whenever the stored voxel is in front of the measured surface, the voxel has to be updated. The depth difference is truncated at the maximum TSDF value and integrated into the running average. To update the color information, the voxels are projected into the color image to retrieve the corresponding RGB data. Note, that the used depth difference is not a true TSDF value, but a projected one since the distance is only measured along the view direction. There could be a closer distance in the other directions. While for surface extraction (using raycasting) this does not affect the quality, it can lead to problems at 3D gradient computations as performed for normal estimation [KPR+15].

4.4 GPU - CPU Swapping

Generally, the camera pose estimation and data integration is performed on the GPU to ensure real-time performance. The direct use of data stored in the main memory from the GPU is not possible. In order to access it, it has to be copied to GPU memory. Since this is too expensive to perform for every data access, the model is stored directly on the GPU. The major limitation in this case is the low available amount of GPU memory, compared to the main memory. Although Voxel Block Hashing ensures a low memory footprint compared to other scene representations, truly large scenes, such as entire building complexes, are still not possible. This is especially true when requiring a fine voxel resolution.

4.4.1 Existing InfiniTAM Functionality

For this reason, the idea of Moving Volume techniques is integrated in InfiniTAM: Only those parts of the model which are currently needed for processing, are stored inside the GPU memory, all other parts are moved to the main memory. When revisiting previously scanned areas, the corresponding voxel blocks are uploaded to the GPU again, so that this existing information can be combined with new data. The GPU - CPU swapping can be achieved very efficiently, when an unstructured scene representation, such as Voxel Block Hashing, is used. Each voxel block can be accessed and processed individually, without the need of maintaining any hierarchy information. Voxel blocks required for processing, are exactly those, which are currently within the camera's view frustum. Voxel blocks, that turned from visible to invisible between the previous and the current frame, are transferred to the CPU memory. Previously hidden blocks, which are now inside the view frustum are copied to the GPU memory.

As explained in Section 4.1, the actual voxel data is stored in the voxel block array. This array lies inside the GPU memory and when using GPU - CPU swapping, it only holds the visible part of the model. The CPU memory, which stores the complete model, is called the global memory.

The GPU - CPU swapping takes place after data integration and thus, the required voxel block visibility information is already available. However because of this order, an enlarged camera view frustum has to be used for the computation of the visibility information. This way it is ensured, that for the processing of the next frame, all the required voxel blocks are inside the GPU memory. Figure 4.6 shows an example, where one can see which blocks are swapped, when the camera rotates left. The dotted line represents the enlarged frustum. Green voxel blocks (currently on CPU) are swapped in to GPU, whereas red blocks (currently on GPU) are swapped out to CPU. The white (on CPU) and gray (on GPU) blocks remain unchanged.

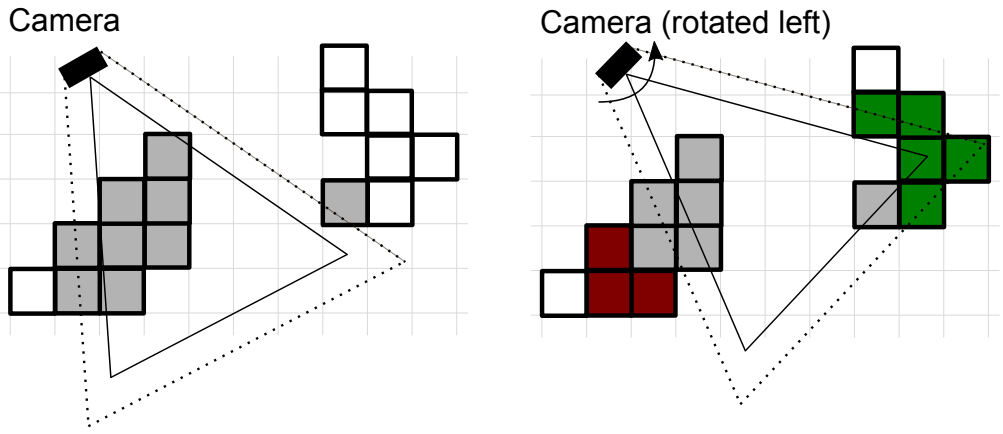


Figure 4.6: Voxel block visibility check for GPU - CPU swapping. All blocks within the enlarged view frustum (dotted line) have to be inside the GPU memory. The red blocks are swapped out to CPU and the green blocks are swapped in to GPU.

4.4.2 Modifications to InfiniTAM

Decreasing Global Memory Footprint In the original InfiniTAM framework, the global memory contains exactly the same amount of elements as the hash table. That means, that there is an implicit one to one mapping from a hash table entry on GPU to a voxel block entry in the global memory, which is advantageous for the swapping process. The drawback is, that a lot of CPU memory is required for the global memory since the hash table is usually large and contains many empty entries to reduce collisions. As a consequence, the potential advantage of GPU - CPU swapping is mitigated and large scenes still do not even fit into the CPU memory. In this thesis, the global memory is adapted to tackle this issue. An additional auxiliary array of indices is stored in the CPU memory with the same size as the hash table. One element of this array then forwards to

an entry in the global memory. This strategy slightly reduces the performance because of the additionally required lookups in the auxiliary array. However, the size of the global memory is greatly reduced since it does not store any empty voxel blocks. Note, that this strategy is similar to the mapping between the hash table and the voxel block array.

Swapping Out Voxel Block Positions In order to allow the transmission of voxel blocks over network to a remote party, they need to be copied to CPU first. Sending them directly from GPU is not possible. InfiniTAM's swapping procedure is therefore also beneficial for the network transmission feature since it performs the mandatory voxel block copying already. To be able to integrate the voxel blocks on the remote side again, not only the voxel data itself, but also the voxel block world position is required. In the original framework however, only the voxel data is swapped out to CPU. As a consequence, the swapping is extended in this thesis to include the voxel block world positions as well. The positions are only copied one-way to the CPU, since the other way is not necessary. More details on the network transmission module can be found in Section 4.6.

Figure 4.7 illustrates both swapping in (right image) and swapping out (left image) of voxel blocks. The modifcations are highlighted yellow. When swapping out a block, also the hash table index is transferred to CPU. Given this index and the auxiliary array, the block within the global memory can be found and updated. After swapping the data to global memory, the data in the GPU voxel block array is deleted. The swapping in from CPU to GPU happens in two stages. First, the hash table indices of the required voxel blocks are transferred to CPU. Given these indices, the CPU is then able to locate the corresponding blocks in the global memory and copy the data to the GPU. Finally, the GPU stores these blocks in newly allocated entries in the voxel block array.

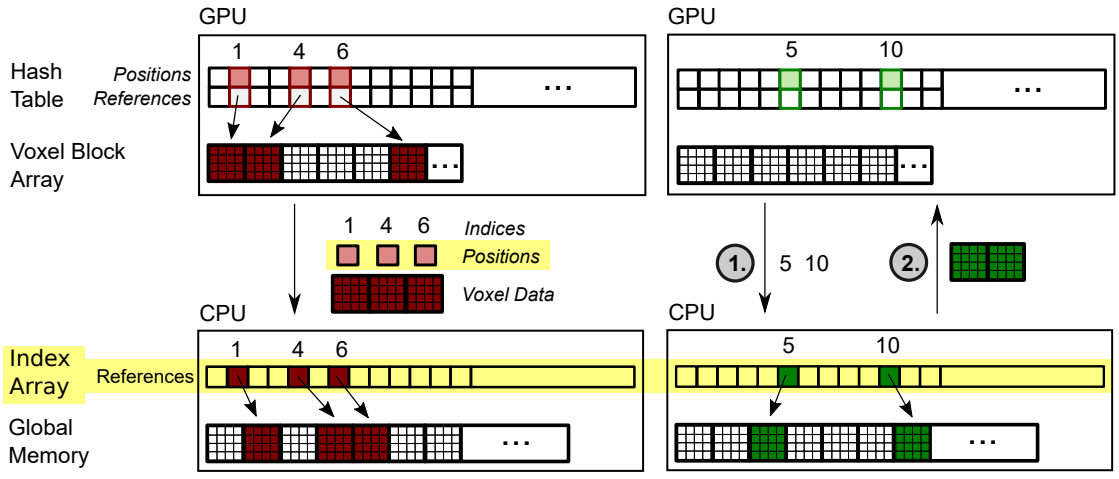


Figure 4.7: GPU - CPU swapping procedure. The left figure illustrates the swapping out to CPU and the right figure shows the swapping in to GPU. Extensions to InfiniTAM are highlighted in yellow.

4.5 Raycasting

Raycasting is the basic method to create an image directly from a volumetric TSDF model [FVDF+94]. It creates the image by shooting rays into the scene, which start at the virtual camera position and run through the pixels of the image plane. For each pixel, a single ray is casted and the first intersection point of the ray with the model is computed. The color of the intersection point determines the color of pixel. If a depth image is desired, the pixel value is set to the distance between the camera and the intersection point.

The live image extraction from the current state of the reconstruction is crucial during scanning in order to known if the camera pose estimation is working correctly and to see which regions still need to be scanned. Besides the live user feedback, raycasting has another major use case. As stated in Section 4.2 a frame-to-model tracking approach is applied in order to minimize the camera drift. This requires the synthetic creation of a depth and normal map from the current model, which can be achieved with raycasting.

In order to find the first object intersection, i.e. the zero crossing of the TSDF, one has to step along the ray and sample the model. The trivial solution is to check all voxels along the ray, which is however too expensive for real-time approaches. Since the model is represented as a TSDF, information about the distance to the surface is available and therefore, larger steps are possible. However because the SDF is truncated, the surface location cannot be inferred directly and the maximum step length is limited by the size of the truncation band, so that the surface is not missed. This is true for the original KinectFusion approach, where the TSDF is stored in a regular voxel grid. The use of Voxel Block Hashing enables to skip large empty portions of the scene since only data near actual surfaces is stored. The possible area of the ray intersection can be limited by computing the minimum and maximum depth of the scene along the ray direction. This greatly improves the overall performance of the reconstruction system in comparison to the standard KinectFusion.

To find the bounds for each ray, all visible voxel blocks are projected into the image space of the virtual camera. Then for each block, the image space bounding box of the eight projected corners is computed along with the minimum and maximum depth of this projected block. Finally, all bounding boxes are rendered to a modified z-buffer. This buffer holds the minimum and maximum observed depth values of all visible voxel blocks.

The minimum depth value now determines the start point of the ray. At first, an allocated voxel block has to be found. If no block is allocated at the current ray position, a step is made with the size of a voxel block. If an allocated block is found, but the TSDF value of the current voxel is outside the truncation band, a step is made with the size of the truncation band. When the sampled voxel is inside the truncation band, the step length is given by its TSDF value. The surface is found when the first TSDF value is negative. Note, that usually the sample points along the ray do not exactly correspond to voxel positions and therefore the TSDF values have to be interpolated. For performance reasons, a trilinear interpolation is only performed when being close

to the surface, otherwise a nearest neighbor lookup is performed. The corresponding color can be interpolated like the TSDF. A final depth map is generated by rendering the depths of the intersection points.

For some applications, like camera pose estimation, additionally the normal vector is required. This vector can be computed either in image space from the computed depth map or in object space from the TSDF. Usually, the normal direction for a given point in a SDF corresponds to the direction with the strongest increase of the values, the gradient. However, as explained in section 4.3, the model does not contain true TSDF values since distances are only measured along the viewing direction of the scanning camera. Therefore, distances are not always maximal in orthogonal direction which can lead to wrong normals. Normal vectors in image space can be evaluated by computing cross products of vectors between neighboring points. This is faster than doing a gradient computation in object space, but also produces artifacts at depth discontinuities. For tracking and live visualization, the latter approach is used. For mesh extraction, a 3D gradient computation is performed since no depth map is available in this case.

4.6 Network Transmission

The network transmission feature allows to send the reconstruction data incrementally over a wireless network to a remote client. The client receives this data and integrates it into its scene representation again, so that both server and client store exactly the same model.

4.6.1 Server Side Compression and Transmission

The network transmission module runs in parallel in order to avoid blocking the reconstruction process. The networking feature is not part of the original InfiniTAM framework and is added as part of this thesis. The major requirement of the transmission is to keep the required bandwidth as low as possible, whereas the latency is secondary. It is not that important if the model update on the client side is delayed by couple of frames, as long as the transmission keeps up with server side model acquisition over time. Note, that this is a clear advantage over streaming live images, which require an interactive frame rate with low latency.

Since Voxel Block Hashing is used, the space is already compressed efficiently. Moreover, individual voxel blocks can be addressed and processed independently of the rest of the scene, which is a big advantage for this streaming feature. In order to further save bandwidth, redundancy is removed by streaming only new or changed data. Generally, voxel blocks outside the view frustum cannot change and can be ignored. Only blocks inside the frustum integrate new information, but due to errors in camera pose estimation or sensor noise, they are likely to change every frame and would have to be retransmitted many times. However, voxel blocks falling out of the current view frustum store new

data and remain unchanged in the future if the scanning camera is not revisiting this area. As a consequence, exactly those blocks are transmitted to the client every frame.

The GPU - CPU swapping module already performs the identification of voxel blocks which fall out of the view frustum. Moreover, they are moved from GPU to CPU. The swapping module marks those blocks, so that the network streaming module is informed. In InfiniTAM's original swapping module, only the TSDF and color data of the voxels is exchanged between GPU and CPU memory. The world position of a corresponding voxel block is not included. Since this position is mandatory for the client in order to integrate the voxel block in its scene representation correctly, the swapping procedure is adapted, so that it also copies the block position one way from GPU to CPU.

The network transmission module waits for voxel blocks to be marked by the GPU - CPU swapping module. In order to achieve a higher compression, it gathers multiple voxel blocks in larger chunks instead of compressing each block alone. Moreover, the TSDF, color and position of all blocks are grouped into three arrays, so that common data types are stored together. Finally, the arrays are compressed individually and transmitted to the client. The network transmission is designed in such a way, that it detects when the connection to the client is lost. Since the networking runs in parallel to the main processing, the scene reconstruction continues. The network module waits until the connection is reestablished and once the client is reachable again, all the data reconstructed in the meantime is transmitted.

Voxel Block Compression For one voxel block with 8^3 voxels, 512 TSDF values, 512 color values and one position is transmitted. Generally, the voxels also contain weights for the integration of new color and TSDF values, however they can be omitted at transmission since they are not required at the client side. In an uncompressed format, the amount of data to be streamed for a single block would be equal to 1024 bytes for the TSDF, 1536 bytes for the color and 6 bytes for the position. Because the color is used only for visual appeal and makes up for most of the data, it may be deactivated to further decrease the bandwidth. For compression, the DEFLATE algorithm is used [Deu96]. This is a lossless general purpose method, which is also used for the PNG image format or the ZIP archive file format. DEFLATE uses concepts of both Huffmann coding and LZ77. It achieves the compression in two stages: At first, the data stream is analyzed to find duplicate strings, which are then replaced by pointers. Afterwards, individual symbols are exchanged with new ones according to their number of occurrence. Common symbols are replaced by shorter representations, whereas rarely used symbols are replaced by longer representations.

Network Protocol Regarding the selection of a network protocol, the main goals are that no data is lost and that the throughput is maximized. The guaranteed order of the data, i.e. data is received in the same order in which it is sent, is not important as long as it is taken care that old data cannot overwrite newer data during the client side integration. Besides that, the latency is secondary as already mentioned above. In this

thesis, the network protocol is not of primary concern and thus, only the two standard protocols TCP and UDP are considered because they are widely supported. UDP is not an option in its standard form since it is unreliable, i.e. data packets can be lost during transmission. As a consequence, TCP is selected, even though it contains unrequired features such as ordered transmission, which results in a higher bandwidth.

4.6.2 Client Side integration

The client stores the reconstructed scene as a lossless copy in the same format as the server. A hash table maintains voxel block positions and references to a voxel block array, which stores the actual voxel data. In contrast to the server, the representation is stored completely on the CPU memory and not (partly) on the GPU. The power of the GPU is not required at the client since it does not perform expensive camera tracking or raycasting. As a consequence, the voxel block array (now on the CPU) is used directly to store the whole scene instead of the additional global memory. The size of the array should be similar to the size of the server side global memory. GPU - CPU swapping is not required. Note, that even though the same representation is used on both server and client, the data might be stored in a different way in the hash table. The order in which voxel blocks are encountered is usually not the same on client and server. It can happen, that on the server side, there already exists a voxel block with the same hash value and thus, the new block has to be stored in the unordered part of the hash table. On the client side however, there might be free space in the ordered part where the block must be stored. For this reason, the client cannot just inherit the server side hash table index of the voxel block but has to perform its own voxel block allocation.

The client listens for incoming voxel data and whenever new data arrives, the individual arrays (TSDF, colors and positions) are decompressed and assembled to voxel blocks again. Given the position of a voxel block, the client then computes the corresponding hash value and checks if there already exists a block for that position. If so, the TSDF and color values are overwritten with the new values. Otherwise, a new voxel block entry is allocated in the hash table and filled with the new data. Furthermore, the integration procedure informs the meshing procedure that a new voxel block arrived at the corresponding part of the scene so that the mesh of this region is updated.

4.7 Dynamic Scene Update

In order to visualize and subsequently explore the scene at the client side, a triangular mesh is computed from the underlying volumetric representation using Marching Cubes [LC87]. The mesh is updated on-the-fly upon model changes so that it can be explored while still scanning the scene. The dynamic update is achieved by partitioning the scene into several subvolumes, each holding an individual mesh. The meshes are recomputed whenever underlying voxel blocks changed.

Every subvolume has a unique identifier ID which can be computed from the world

position of any underlying voxel block. The client's Network Transmission module performs this computation for every newly arrived voxel block to identify the mesh block to be updated. The computed ID is checked for existence within a map of mesh block identifiers MID, which holds the identifiers of all mesh blocks to be updated. If the index does not exist, it is added to MID. The Scene Update module can also access MID. It regularly reads and removes an ID from the map. Given the ID, the world position of the mesh block is reconstructed and the corresponding volume is triangulated.

The following subsections explain the scene partitioning, the mapping between the scene parts and the corresponding voxel blocks and the actual update of individual meshes in more detail.

4.7.1 Scene Partitioning

Inspired by Steinbrücker et al. [SKCS13], the mesh representation of the scene is composed of a number of smaller individual meshes, where each mesh covers a certain region of the volumetric scene. In contrast to Steinbrücker's approach, the volumetric scene is only stored at one resolution, which eases the mesh computation. The scene is divided into a regular 3D grid of mesh blocks, where each mesh block holds its own mesh and covers a region of n^3 voxel blocks. An example is shown in Figure 4.8. Real-time updates are

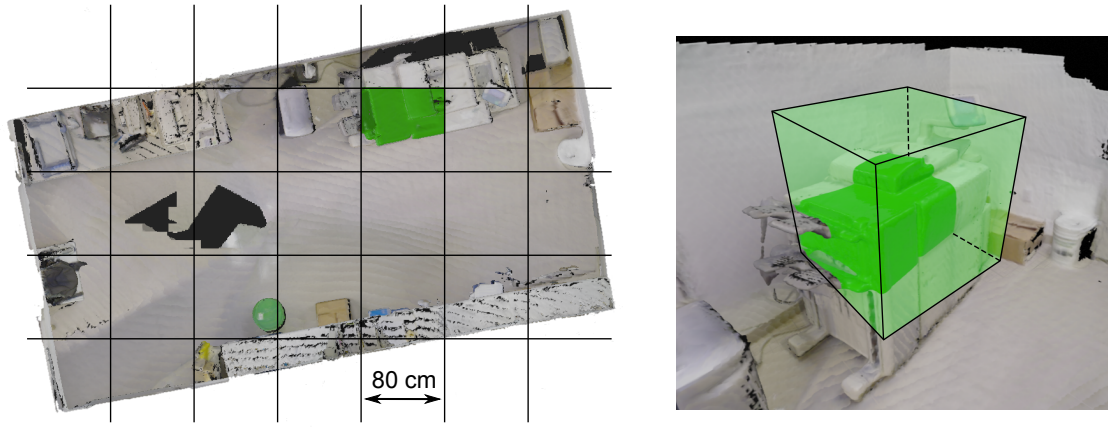


Figure 4.8: Partitioning of a test scene into individual mesh blocks, where each block contains its own mesh. The mesh of a single mesh block is highlighted in green.

achieved by recomputing an individual mesh, whenever new data for any of its underlying voxel blocks arrived. The triangulation of the voxel blocks is performed using an extended InfiniTAM Marching Cubes implementation, as described in Subsection 4.7.3.

The size of the mesh blocks is defined by the number of voxel blocks they should cover. Increasing the mesh block size leads to fewer but larger meshes that need to be updated even when only small parts contain new information. Decreasing the mesh block size optimizes updating of a each single mesh but increases the overhead for mesh maintenance

and rendering since the total number of meshes becomes very large. Moreover, the number of vertices increases since a vertex on the border of a mesh block is contained in each adjacent mesh block, too. For the thesis setup, sufficient update rates were found with ten voxel blocks for each dimension per mesh. Note, that the metric size of a mesh block also depends on the chosen voxel resolution and number of voxels per voxel block. If one uses voxel blocks with 8^3 voxels at a resolution of 1cm, one mesh block covers a volume of 80x80x80cm.

The scene is divided into a regular 3D grid of mesh blocks, where one grid element contains multiple voxel blocks. In the thesis implementation, such a mesh block covers 10^3 voxel blocks, which leads to an area of $80x80x80$ cm (at 1 cm voxel resolution). A maximum number of 1000 mesh blocks per dimension is used. As a consequence, the maximum size of the mesh representation is 800 m^3 at a 1 cm resolution. Figure 4.8 shows a the partitioning of a reconstructed test scene into 80 cm^3 mesh blocks, where the mesh of a single mesh block is highlighted in green. In the right image, one can also see the corresponding mesh block. In total, the scene consists of 130 individual meshes.

4.7.2 Mapping between Voxel Blocks and Mesh Blocks

To establish a correct mapping between the voxel blocks vB and their mesh block mB, each mesh block is identified by a non-negative and unique integer ID. This ID can be computed – as described by Algorithm 4.1 – from a given 3D voxel block position $vbPos \in \mathbb{Z}^3$. First, $vbPos$ is shifted to the positive octant of the coordinate system. Afterwards, the shifted position is divided by the number of voxel blocks per mesh block (integer division). This results in a three dimensional mesh block $ID_3 \in \mathbb{Z}^3$. Finally, ID_3 is flattened to a one dimensional integer ID. Note, that due to the division by the number of voxel blocks per mesh block, all voxel blocks within a mesh block produce the same mesh block ID.

Algorithm 4.1: Mesh Block ID from Voxel Block Position

Input: Voxel Block Position $vbPos \in \mathbb{Z}^3$, Number of Mesh Blocks per Dimension mB, Number of Voxel Blocks per Mesh Block (per Dimension) vB
Output: Mesh Block $ID \in \mathbb{Z}$

$ID \leftarrow -1$;
$vbPos_{shifted} \leftarrow vbPos + \frac{mB}{2} \cdot vB$;
$ID_3 \leftarrow vbPos_{shifted}$ / vB; //3D Mesh Block ID
if $ID_3(x) \in [0, mB)$ **and** $ID_3(y) \in [0, mB)$ **and** $ID_3(z) \in [0, mB)$ **then**
  $ID \leftarrow ID_3(x) \cdot mB^2 + ID_3(y) \cdot mB + ID_3(z)$; //1D Mesh Block ID
end
return *ID;*

In order to allow this ID computation, the maximum number of mesh blocks per dimension mB has to be predefined. Limiting it to 1000 keeps the generated index below one billion

(four bytes), but still provides a sufficient large address space. Furthermore, in this case, the flattened integer ID is constructed in simple way: the first three digits represent $ID_3(x)$, followed by three digits for $ID_3(y)$, and three for $ID_3(z)$.

Given a mesh block ID, one can reconstruct the world position of that mesh block $mbPos \in \mathbb{Z}^3$ according to Algorithm 4.2. First, the three dimensional ID_3 can be extracted from ID (lines 3 - 5). Next, the shifted mesh block position is computed (line 6) and finally the shifting is inversed (line 7). The resulting mesh block world position $mbPos$ corresponds to the voxel block with the smallest position in the mesh block, i.e. each coordinate is smaller than or equal to the corresponding coordinate of all other voxel blocks.

Algorithm 4.2: Mesh Block Position from Mesh Block ID

Input: Mesh Block $ID \in \mathbb{Z}$, Number of Mesh Blocks per Dimension mB, Number of Voxel Blocks per Mesh Block (per Dimension) vB
Output: Mesh Block Position $mbPos \in \mathbb{Z}^3$

1 $mbPos = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$;
2 **if** $ID >= 0$ **then**
3 $\quad ID_3(x) \leftarrow ID \ / \ (mB^2)$;
4 $\quad ID_3(y) \leftarrow (ID - mB^2 \cdot ID_3(x)) \ / \ mB$;
5 $\quad ID_3(z) \leftarrow ID - mB \cdot (ID_3(y) + mB \cdot ID_3(x))$;
6 $\quad mbPos_{shifted} \leftarrow ID_3 \cdot vB - \frac{mB}{2} \cdot vB$;
7 $\quad mbPos \leftarrow mbPos_{shifted} - \frac{mB}{2} \cdot vB$;
8 **end**
9 **return** *mbPos;*

4.7.3 Triangulation of a Mesh Block

The triangulation of individual scene parts is performed with Marching Cubes. The Marching Cubes algorithm can be performed efficiently in parallel on the GPU for large meshes. However in this implementation, the mesh generation is performed by the CPU only because the client side representation lies on the CPU. This avoids the need for the expensive data transfer between the CPU and the GPU memory. Moreover, since only small regions of the mesh have to be updated at a given time, the power of the CPU is enough to ensure real-time performance. The GPU of the client is utilized for rendering only.

The main idea of Marching Cubes is to walk through the volume and compute the triangles representing the isosurface at each step individually. Eight voxels representing the corners of an imaginary cube are processed at a time. By comparing the SDF values of these voxels, the isosurface passing through the cube can be inferred. If the SDF of one voxel is positive and the SDF of another voxel is negative, the surface has to pass anywhere in between. The information, if the eight SDF values are in front of or behind

the surface, is enough to determine the general triangle configuration, i.e. the number of required triangles and their approximate position and orientation. The information, on which side of the surface the voxel are, is encoded by the signs of the SDF and can be stored as a single byte, where one bit is used for each voxel. This byte then serves as an index to perform a lookup in a table, which stores the precomputed triangle configurations for all 256 bit combinations. These 256 combinations can be reduced to 15 families, where combinations within one family represent triangle configurations which are symmetrical or rotations of each other. Figure 4.9 illustrates the principle in two dimensions (Marching Squares).

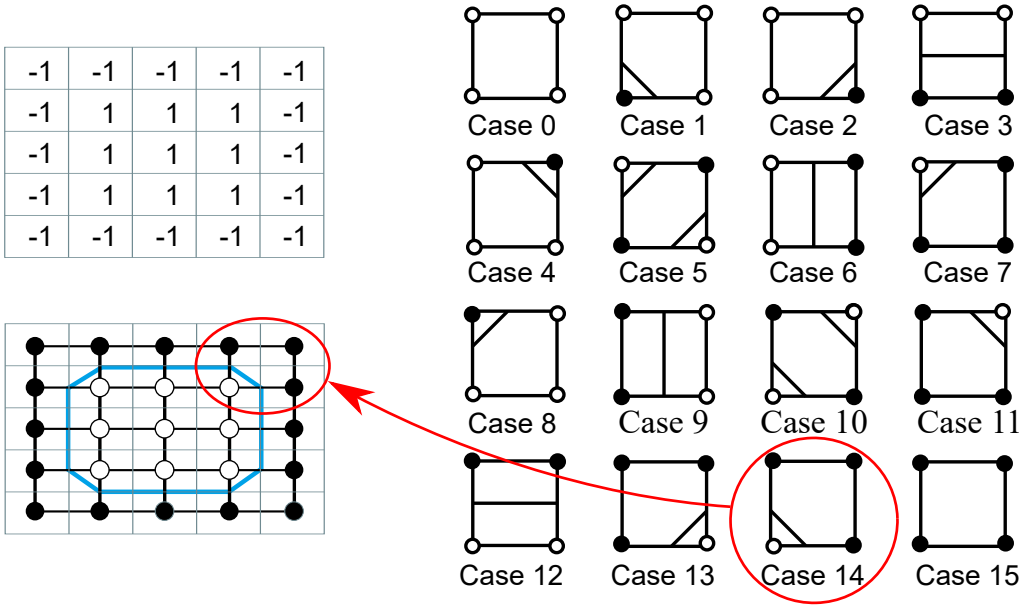


Figure 4.9: Principle of Marching Cubes in two dimensions (Marching Squares). The right image shows the lookup table storing the possible line configurations. The top left image represents the SDF data and the bottom left image shows the computed surface. The dot color indicates if the point is inside (white) or outside (black) the surface. Marked in red is one square, where the inside-outside combination corresponds to configuration 14. Figure adapted from Wikipedia[1].

Given the correct triangle configuration, the exact vertex positions of the triangles have to be determined. Note, that the vertices always lie on the edges of the imaginary cube. The final position P of a vertex is computed by taking the corresponding two SDF values (SDF_1, SDF_2) and positions (p_1, p_2) of the edge and performing linear interpolation as shown in Equation 4.4.

[1] https://commons.wikimedia.org/wiki/File:Marching_squares_algorithm.svg

$$P = p_1 + \frac{-SDF_1}{SDF_2 - SDF_1} * (p_2 - p_1) \tag{4.4}$$

The vertex colors are interpolated the same way as the positions. The vertex normals can be extracted by performing a 3D gradient computation at the computed vertex positions. Once all triangles have been computed for the current imaginary cube, the processing continues with next cube. Note, that this cube contains four of the same voxels as the previous cube, if the previous cube is adjacent. Previously computed information could be reused in these cases but is currently not done in the thesis implementation for simplicity reasons. Once the whole volume is processed, the final mesh is given as the set of all computed triangles.

Extension to InfiniTAM's Marching Cubes InfiniTAM includes a basic Marching Cubes approach to compute an uncolored mesh in a post-processing step. Besides the ability to update the mesh in real-time, the implementation is extended to include the vertex normals and vertex colors. Moreover, duplicate vertices within one mesh block are avoided by utilizing a hash map to check if a given vertex already exists. This check requires additional resources, but it also allows to skip normal and color computation for existing vertices. Apart from that, it reduces the required amount of memory. Instead of delivering an array of triangle structures, the meshing procedure now generates separate position, color, normal and vertex index arrays. The latter contains indices referencing vertices in the other arrays, where three consecutive indices define one triangle. This is the standard representation for rendering and is also required by UE4, which is used for visualization.

4.7.4 Shared Memory Connection and Dynamic Mesh Update in UE4

As already stated, the mesh of the whole scene is divided into a regular grid of smaller meshes. All these small meshes are stored and maintained by the rendering and exploration procedure, which runs as a separate process and uses UE4. Within UE4, the scene is represented as a set of procedural meshes, which can be added and modified at runtime. The UE4 process communicates with the client's meshing procedure using shared memory. With shared memory, different processes on the same computer can directly access the same memory region and communicate with each other efficiently. Once the meshing module generates a new mesh of a mesh block, it writes the resulting arrays (positions, colors, ...) along with a unique mesh block index to the shared memory. The UE4 process regularly checks if new data is available and if yes, it reads the data and cleans the shared memory region to make space for a new mesh. Given the mesh block index, UE4 knows which procedural mesh requires an update. If the index is unseen, a new procedural mesh is added at runtime.

4.8 Visualization and Exploration

The reconstructed scene can be explored right from the beginning of the scanning process, so the scene can change and grow while exploring it.

4.8.1 Rendering with Unreal Engine 4

As previously mentioned, the existing game engine UE4 is integrated, since it already incorporates advanced rendering and physical effects, which are both vital for exploration.

When providing the color information of the scene, the exploration is enhanced because the depth perception is increased and one can recognize known objects easier. Furthermore, the reconstructed scene looks more like reality, which can lead to a higher degree of immersion during exploration in a VR setup. At the scanning process, the RGB-D camera already captures the color of the scene which contains the full lighting information including shadows. As a consequence, the camera RGB information can be used directly for rendering without any further shading calculations. Note, that the specular lighting component depends on the current view point of the person looking at the scene. This property is very notable on glossy surfaces like metal where one can perceive specular highlights. The highlight should change when moving the head, which is not true for the captured model. As a consequence, when scanning the scene, the measured color is only correct for the corresponding camera position. Since the objects are usually scanned from various different locations, the color information of multiple view points is averaged and the specular lighting problem is less notable. For rough surfaces, which only have a small specular lighting component, the measured color represents the true lighting information much better. In order to include dynamic specular highlights and to enhance the 3D perception during exploration, three artificial points lights are added to the reconstructed scene.

Besides basic lighting calculation for those lights, no advanced rendering effects are applied in order to enhance the rendering frame rate. Keep in mind, that even without specular surfaces, the color quality of the reconstructed model is still degraded compared to a synthetic 3D model due to slightly wrong camera pose estimation, motion blur, auto exposure or errors in the depth and color image alignment. Whelan et al. propose to improve the color information by ignoring measurements near depth discontinuities [WJK+13] at color integration. Moreover, they apply a weighted color update, where colors measured at a low angle (between surface normal and camera view direction) are included more strongly. This improvement is currently not included in the InfiniTAM framework and is therefore neither contained in the thesis implementation.

4.8.2 Exploration with VR Hardware

The thesis implementation supports the exploration with various input and output devices. Regarding the output device, one can use a standard desktop screen or a HMD. For input, one can use a keyboard/mouse setup, a gamepad or an ODT. In this thesis the Developer

Kit 2 (DK2) of the Oculus Rift HMD is used, which is illustrated in Figure 4.10 [Ocu16a]. The DK2 has a single full HD display which provides a field of view of 100 degrees. It is shared by both eyes, so for each eye there are 960 x 1080 pixels available. The rendering can be updated with maximum rate of 75 Hz. Besides tracking the orientation at 1000 Hz using a built in inertial measurement unit (IMU), the DK2 supports optical positional tracking with an update rate of 60 Hz. However, the positional tracking only works when the Rift is facing the camera and therefore, it does not work for all orientations. Since in this thesis, the Rift is used in combination with an ODT, the positional tracking is turned off.

Figure 4.10: Oculus Rift HMD (Developer Kit 2)[Ocu16a].

The Cyberith Virtualizer ODT [CH14] can be used instead of the gamepad/keyboard input when using a HMD for display. The Virtualizer allows to move in the virtual world while actually staying at the same position in the real world. It also enables to decouple the walking and viewing direction, i.e. the user can walk in one direction while looking in another direction. This is usually not possible with standard input devices. A mechanical construction keeps the user at the same place while he walks on a low friction surface. Walking is performed by leaning the body forward so that one foot slides backwards while the other one performs a step. Figure 4.11 illustrates the construction. The treadmill's surface has built-in sensors and detect the speed and direction of motion. Depending on how strong the user leans into the fixed construction, the walking speed changes since the feet slide backwards faster or slower. The user wears textile overshoes in order to ensure a level of friction which allows effortless walking while not being too slippery. The user can rotate freely 360 degrees in the Virtualizer so that any walking direction is possible. Moreover, the device allows to crouch, jump or sit which distinguishes it from similar treadmills like the Virtuix Omni.

While the Oculus Rift is natively supported by UE4, a plugin is developed in this thesis in order to integrate the Virtualizer. The UE4 plugin allows to retrieve the current body rotation, the walking direction and speed and the current height of the user's hip. The returned information is then used to update the position and orientation of the virtual character.

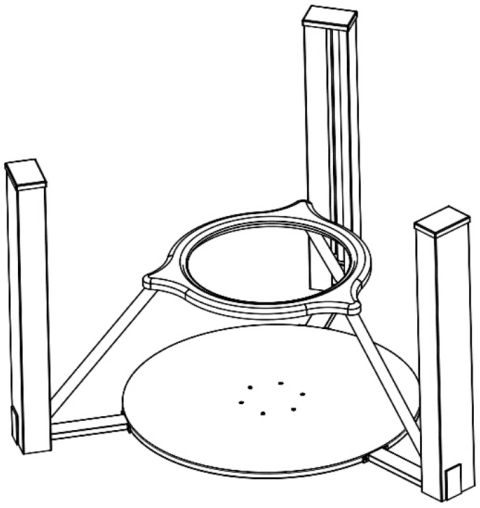


Figure 4.11: The principle of the Virtualizer. The user is placed in the middle of the device and is fixed to the ring with a belt system. Walking is performed by leaning into the belt system so that the feet slide backwards. Optical sensors, integrated into the low friction surface, measure the motion. [CH14]

4.9 Current Limitations

In order to be used in general real world applications, some parts of the implemented system have to be improved in future work.

Camera Pose Estimation The used InfiniTAM framework enables computationally efficient 3D reconstruction and allows to large scale scenes with a low memory footprint. Capturing a large environment successfully is however challenging, due to the applied camera pose estimation. The reconstruction process cannot continue once a camera pose could not be estimated because the pose of the previous frame is always required for the current pose estimation. Such tracking failures can occur when the scanning camera is moved too fast or when the scanned environment does not contain enough features, as in hallway scenes. An option to find the camera pose again is to match the current depth image to the model, where only a certain region around last known camera position has to be considered. The developed reconstruction pipeline however does not detect tracking failures, nor does it integrate any form of pose recovery. Even when a large-scale environment is reconstructed successfully, the resulting model is likely to be misaligned due to the accumulated camera drift. Loop closure for large-scale dense SDF representations is still an open problem and is not integrated in the framework.

Mesh Extraction A further issue with the developed system is the huge amount of triangles produced by the Marching Cubes triangulation. Currently, the extracted

mesh is only used for rendering without advanced effects. For this purpose, the high triangle count is still manageable with modern hardware. However, the mesh needs to be simplified when interaction with the model is desired. Furthermore, the mesh representation is currently stored completely inside the client's GPU memory and large meshes could exceed the available space.

Exploration Collision detection between the virtual character controlled by the user and the reconstructed model is currently not performed. Wrong depth measurements can lead to artifacts, which block the way and permit the user to pass. Because of the missing collision detection, the user does not follow the terrain of the scene but walks on a fixed invisible plane and may also pass through walls. Moreover, the position and orientation of the reconstructed model is determined by the initial pose of the scanning camera. In order to ensure, that the invisible plane fits to the floor of the reconstructed scene, the depth camera has to be horizontally aligned at a height of 150 cm above the floor at the beginning of the scan. In future work, the model has to be cleaned from blocking artifacts in order to perform collision against the extracted mesh. This would also allow to reconstruct and explore multi-storey buildings, which is currently impossible.

CHAPTER 5

Implementation

In this chapter, the implementation of the 3D reconstruction pipeline is highlighted. The focus lies on those parts which are adapted or newly added to the reconstruction pipeline as part of this thesis. For all existing and unchanged parts, it is referred to the original publications of InfiniTAM[PKC+14][KPR+15].

Used Frameworks and Libraries Both server and client applications are implemented in C++ on Windows 7 and use following libraries.

Server:

- InfiniTAM 2 [KPR+15]
- OpenGL/GLUT (freeglut 3.0) [Fre16]
- OpenNI 2.2 [Occ16]
- CUDA 7.0 [Cor16b]
- Windows Sockets 2 (Winsock2)
- zLib 1.2.8 [DG96]

Client:

- InfiniTAM 2
- Unreal Engine 4.9 [Epi16]
- Ultimate Shared Memory (usm) [Cho16]

- Windows Sockets 2 (Winsock2)
- zLib 1.2.8
- Optional: OpenGL/GLUT (freeglut 3.0)
- Optional: CUDA 7.0

While the original InfiniTAM framework, as well as UE4, supports a wide range of platforms including Windows, Linux or Android, the developed reconstruction pipeline currently only runs on Windows due to the usage of Winsock and usm.

The rest of this chapter is structured in the following way: first, an overview of the implementation is given in Section 5.1. Afterwards, it is explained in Section 5.2, how to run the 3D reconstruction system. The remaining Sections 5.3 to 5.6 describe the structure and usage of the changed and newly added features more detailed.

5.1 Implementation Overview

This section explains the structure of the most important classes and shows how the individual modules and their threads communicate with each other.

5.1.1 Class Structure

As it can be seen in Figure 5.1, the reconstruction pipeline consists of three different processes: The InfiniTAM Server, the InfiniTAM Client and the UE4 Visualization. The yellow color indicates, that corresponding classes are changed, compared to the original InfiniTAM implementation. Note, that this just reflects important changes. There can also be minor changes in unmarked classes. Green classes are newly added to the framework.

InfiniTAM Server The UIEngine is the entry point to InfiniTAM and controls the application. It processes user input and displays an image of the reconstruction's current state. The UIEngine maintains three other engines: The ImageSourceEngine, the IMUSourceEngine and the ITMMainEngine. The ImageSourceEngine handles the connection to the camera and provides images for the reconstruction. By extending this class, one can add support for any image source. In this thesis, the OpenNIEngine is primarily used and allows to process live data from the Asus Xtion Pro Live camera as well as already captured OpenNI sequences. The IMUSourceEngine makes it possible to integrate IMU orientation data to improve the camera tracking. Since the default implementation allows to load only saved data from file, it is extended in this thesis with the IMUNetworkEngine to use live data. The UIEngine gathers new input data from these two source engines and forwards it to the ITMMainEngine, which performs the actual reconstruction process. The ITMMainEngine is divided into three main parts: The ITMVisualizationEngine, the

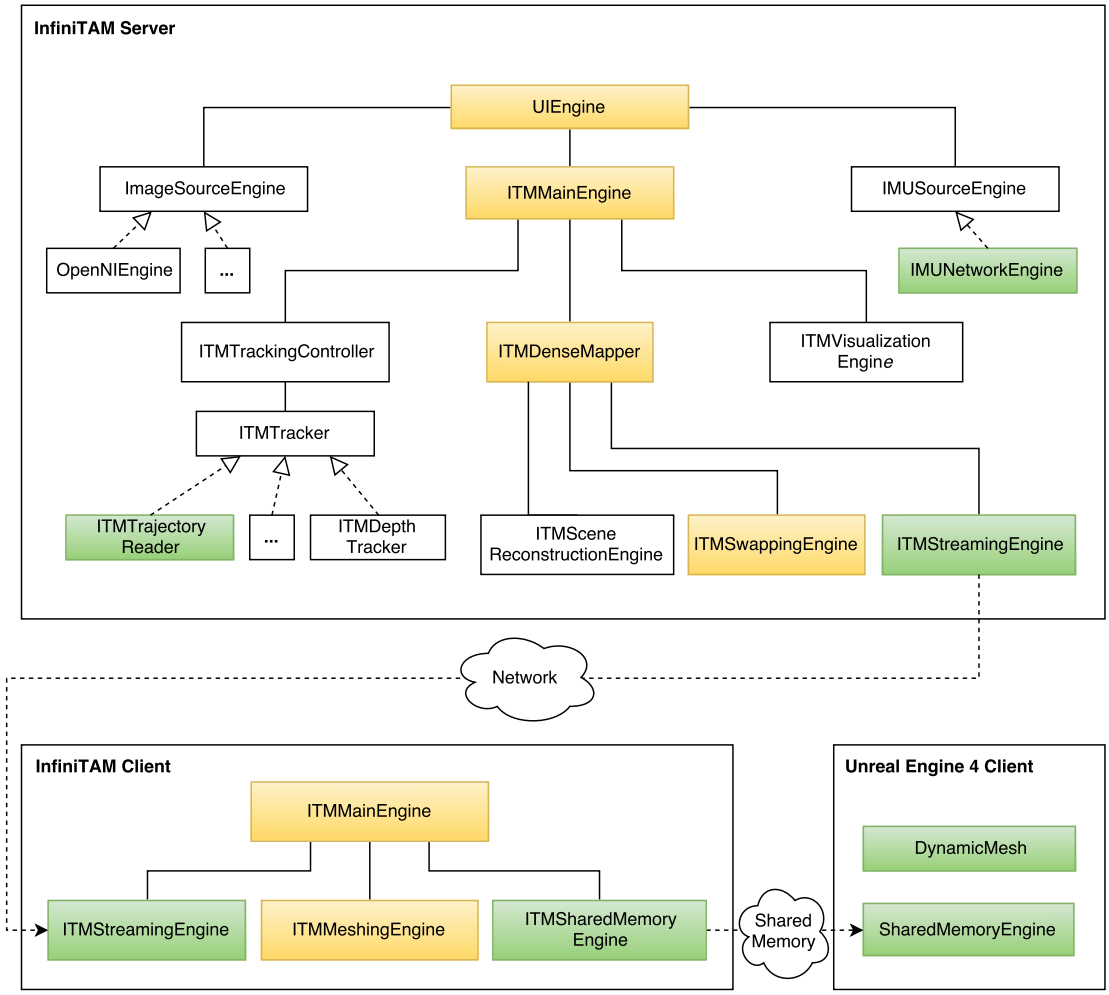


Figure 5.1: Overview of the most important classes and their relations. The yellow color indicates changed parts compared to the original InfiniTAM implementation. The green color indicates newly added parts.

ITMTrackingController and the ITMDenseMapper. The main task of the ITMVisualizationEngine is to compute an image of the reconstructed model using raycasting, which is used both by the UIEngine as well as the ITMTrackingController. The ITMTrackingController maintains a specific camera tracker, which handles the camera pose estimation. By extending the ITMTracker class, one can add new tracking methods. As mentioned in the previous chapter, InfiniTAM uses a depth-based ICP tracker per default. Within the thesis, the ITMTrajectoryReader is added, which allows to read and use precalculated camera poses instead of performing live tracking. The ITMDenseMapper handles both the integration of camera data into the model as well as the GPU-CPU swapping. The integration is performed by the ITMSceneReconstructionEngine and the swapping is performed by the ITMSwappingEngine. Besides those two engines, the ITMDenseMapper

contains the newly added ITMStreamingEngine, which transmits the reconstruction data to the client.

InfiniTAM Client and UE4 Visualization The client application is divided into an InfiniTAM process and an UE4 process. In contrast to the server, large portions of the original InfiniTAM framework are not required, since the client does not need to perform pose estimation, integration of camera images into the model or raycasting. What is left, is the general voxel block data structure as well as a trimmed ITMMainEngine. The ITMMainEngine contains the ITMStreamingEngine directly, which receives the data from the server side ITMStreamingEngine. Furthermore, the ITMMainEngine maintains an ITMMeshingEngine and an ITMSharedMemoryEngine. The ITMMeshingEngine generates the mesh representation and the ITMSharedMemoryEngine passes the computed meshes to the UE4 process via shared memory. The UE4 process also contains a SharedMemoryEngine, the counterpart of the ITMSharedMemoryEngine. Apart from that, it contains a DynamicMesh object, which allows to update the UE4 scene representation on-the-fly. Lastly, a UE4 plugin is implemented to integrate the Cyberith Virtualizer.

5.1.2 Thread Communication

The developed reconstruction pipeline uses several threads to keep the performance high and to avoid unnecessary blocking. Figure 5.2 shows all threads and how they communicate with each other. The threads are shown in gray. They communicate via commonly accessible data structures shown in white.

The InfiniTAM server is composed of two threads. The main thread performs the actual scene reconstruction and the additional network transmission thread sends the reconstruction data via TCP to the client. The interface between these two threads is the StreamingSet. This set contains references of all those blocks which have to be transmitted to the client. The InfiniTAM client is composed of at least two threads. The network transmission thread receives data from the server and integrates it into the model. One or more meshing threads take care of the mesh generation. They communicate with the network transmission thread via the MeshIDMap. This map contains information which mesh blocks require an update. By using more than one meshing thread, multiple meshes can be recomputed at the same time. Apart from the InfiniTAM threads, there are additional threads within UE4. The shared memory thread receives the mesh data from InfiniTAM utilizing a shared memory region, where the meshing threads store their results. The shared memory thread converts and stores the mesh data in temporary arrays. The UE4 game thread finally reads these arrays and handles the dynamic scene update.

5.2 Usage Instructions

As already mentioned, the reconstruction pipeline consists of three different processes: The InfiniTAM Server, the InfiniTAM Client and the UE4 Visualization. Each process has

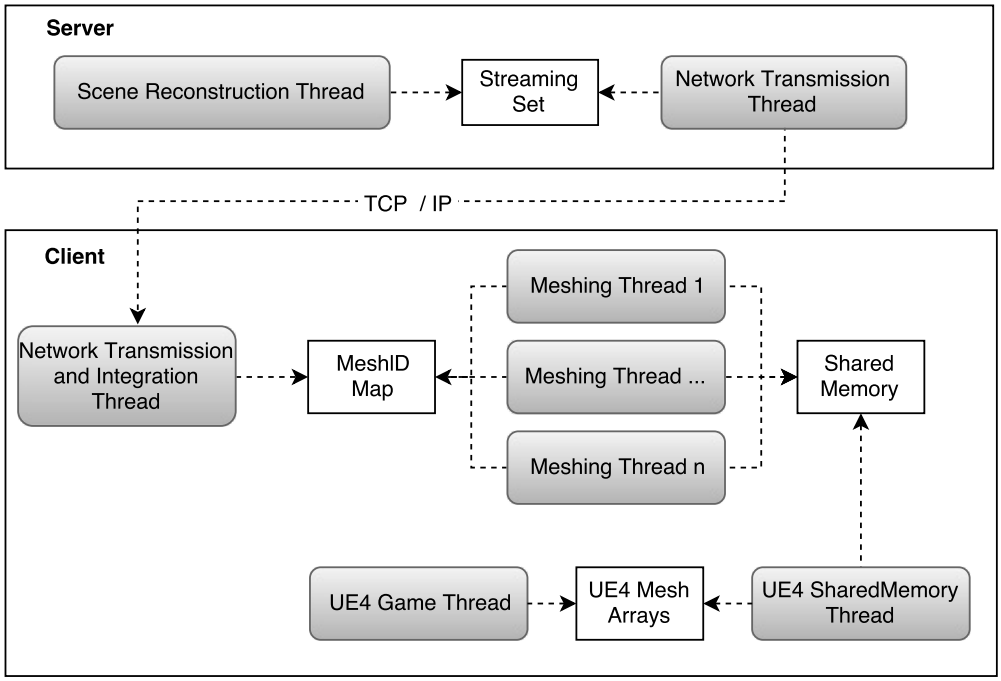


Figure 5.2: Overview of the individual threads, shown in gray. The threads communicate with each other via commonly accessible data structures shown in white.

to be started separately, so on the client side, one has to run two processes. After starting one side (client or server), the system waits for the other side to be connected before the reconstruction process can be started. The system does not contain a configuration file, which can be loaded at runtime. All the settings are hardcoded and have to be adjusted at compile time. The InfiniTAM settings of both server and client can be found in two different files: *ITMLibSettings.cpp* and *ITMLibDefines.h*.

5.2.1 Server

Server Settings The size of the scene memory does not grow dynamically during reconstruction, but has to be defined beforehand in the ITMLibDefines header. Here, one can set the maximum number of hash table entries, where *SDF_BUCKET_NUM* corresponds to the ordered part and *SDF_EXCESS_LIST_SIZE* to the unordered part. The *SDF_GLOBAL_BLOCK_NUM* definition represents the maximum number of voxel blocks to be stored in the global CPU memory. One block requires 4 KB if color is used and 2 KB if no color is used. The number of voxel blocks to be stored in GPU memory is defined by *SDF_LOCAL_BLOCK_NUM*. It needs to be large enough to store all voxel blocks, which are currently in the field of view of the scanning camera. This number depends on the chosen voxel resolution and the size of the view frustum. In most cases, the default value does not have to be changed. Beside the memory sizes, a further setting in ITMLibDefines.h is the voxel type to be used. It defines if the reconstruction should

contain color or not. A C++ typedef controls the type of *ITMVoxel*:

```
typedef ITMVoxel_s_rgb ITMVoxel; // With Color
typedef ITMVoxel_s ITMVoxel;     // Without Color
```

The voxel type on the server and the client need to be set to the same type, otherwise the client's network module expects too much or too little data from the server. All other server settings are defined in the constructor of the ITMLibSettings class. Here, one can choose the size of the SDF truncation band, the voxel resolution or the minimum and maximum considered depth range for data integration. For instance, the line

```
sceneParams(0.02f, 100, 0.01f, 0.2f, 3.0f, false)
```

selects a 2 cm truncation band with a voxel size of 1 cm. Only depth measurements with a value between 0.2 m and 3 m are integrated into the model. The value 100 defines, that the color and the TSDF value of a voxel is computed using a running average of 100 measurements. If the value is reached, older measurement are removed to integrate new ones. The last boolean value can be used to avoid replacing existing measurements. If the flag is set to true, new measurements are not integrated anymore. Besides these scene parameters, the ITMLibSettings class controls the camera tracking method and the network streaming. By setting the *trackerType* variable to *TRACKER_ICP*, a standard depth-based ICP tracker is applied. More details on the available trackers can be found in Section 5.3. The variable *useNetworkStreaming* enables network streaming and the network port to be used is defined by *serverPort*. One can also disable GPU - CPU swapping with *useSwapping*, however, it is mandatory for network streaming.

Server Execution When executing the compiled server program with no command line parameters, the system tries to use live image data from a connected RGB-D camera (such as the Asus Xtion Pro Live) with a default camera calibration. To improve tracking and integration, one can provide a path to an existing calibration file as the first command line argument. The content of the default calibration file can be seen below:

```
640 480
525.0 525.0
319.5 239.5

640 480
525.0 525.0
319.5 239.5

1 0 0 0
0 1 0 0
0 0 1 0

0 0
```

The first three lines correspond to the RGB sensor and the following three lines correspond to the depth sensor. For each sensor, the size of the image (first line), the focal length (second line) and the principal point (third line) are given. All values are expected to be pixels. Besides that, the file contains a 4x4 matrix, which maps points from the RGB image to the depth image. In the default calibration, the identity matrix is used. The values in the last line are required for the conversion of Kinect disparity values to depth values. For OpenNI devices, as used within the thesis, these values are ignored. When applying standard ICP tracking, only the depth sensor information is required for camera tracking. The RGB sensor information is only used to integrate the color into the model correctly. The *OpenNICalibTool*[1] performs a calibration for OpenNI supported cameras like the Asus Xtion and generates a file in the InfiniTAM format. Apart from the camera calibration file, a path to an existing OpenNI file can be provided as the second command line argument to use a prerecorded scene instead of live camera data.

After running the program, the user has to start live reconstruction by pressing the *b* key. Since the first camera pose determines the orientation and position of the whole reconstructed model, the scanning camera has to be aligned with the real floor plane before starting the process. Moreover, the camera should be positioned at a height of 1.5 meters above the floor in order to fit the size of the virtual character used for exploration at the client side. After the process has started, one can switch between different live visualization modes or pause the integration of new data. The corresponding keys are displayed at the bottom of the screen. The program is closed by pressing the Escape key. Figure 5.3 shows a screenshot of the server's user interface.

Figure 5.3: Screenshot of the InfiniTAM server application.

[1] https://github.com/carlren/OpenNICalibTool

5.2.2 Client

Client Settings Equally to the server, the ITMLibDefines header defines, if color should be used or not. Both server and client need to use the same voxel type. The hash table size is chosen using *SDF_BUCKET_NUM* and *SDF_EXCESS_LIST_SIZE*, which should be set as large as at the server side. The maximum number of supported voxel blocks, i.e. the scene size, is controlled by *SDF_LOCAL_BLOCK_NUM*. Note, that this is in contrast to the server, where this definition controls the number of blocks on the GPU. This is due to the fact, that the client does not use the GPU but stores everything on the CPU memory. Therefore, the *SDF_LOCAL_BLOCK_NUM* corresponds to the server's *SDF_GLOBAL_BLOCK_NUM* and should be set equally large. Moreover, the ITMLibDefines header controls meshing characteristics such as the size of one mesh block (in voxels blocks) and the number of supported mesh blocks per dimension (*MESH_CHUNK_SIZE* and *MESH_CHUNKS_PER_DIM*). The ITMLibSettings class contains network transmission parameters such as the name/IP address of the server (*serverName*) and the corresponding network port (*serverPort*).

Client Execution In order to run the client, the UE4 process should be executed at first to avoid any problems. Afterwards, the InfiniTAM process has to be started. Both processes do not require any command line arguments. At startup of the UE4 process, it is checked which input and output devices are connected. If a HMD is detected, it is used automatically instead of the standard display. The same is true for the Virtualizer. If a Virtualizer device is found, navigation is only possible with the treadmill. Otherwise, the user can navigate with either the gamepad or the keyboard. When using a gamepad, one moves with the joystick or with the arrow buttons. When using a keyboard, movement is performed with WASD or the arrow keys. In this non-Virtualizer setup, the walking direction is defined by the viewing direction and the viewing direction is determined directly by the HMD or using the mouse (when using a standard display). Figure 5.4 shows a screenshot of the client's user interface.

The following sections describe the structure and usage of the changed and newly added features in more detail. At first, the camera tracking module is examined, followed by the network transmission. Afterwards, the dynamic scene update including the mesh computation, the shared memory connection between InfiniTAM and UE4, and the actual mesh update within UE4 is explained. Finally, features of the visualization and exploration are highlighted.

5.3 Camera Pose Estimation

The camera tracking behaviour is defined by the used camera tracker object. During initialization of the ITMMainEngine, the camera tracker is created by the ITMTrackerFactory and then maintained by the ITMTrackingController. The selection of specific camera tracking method happens in the ITMLibSettings class by adjusting the *trackerType* variable. By default, depth-based ICP tracking is used. InfiniTAM supports standard

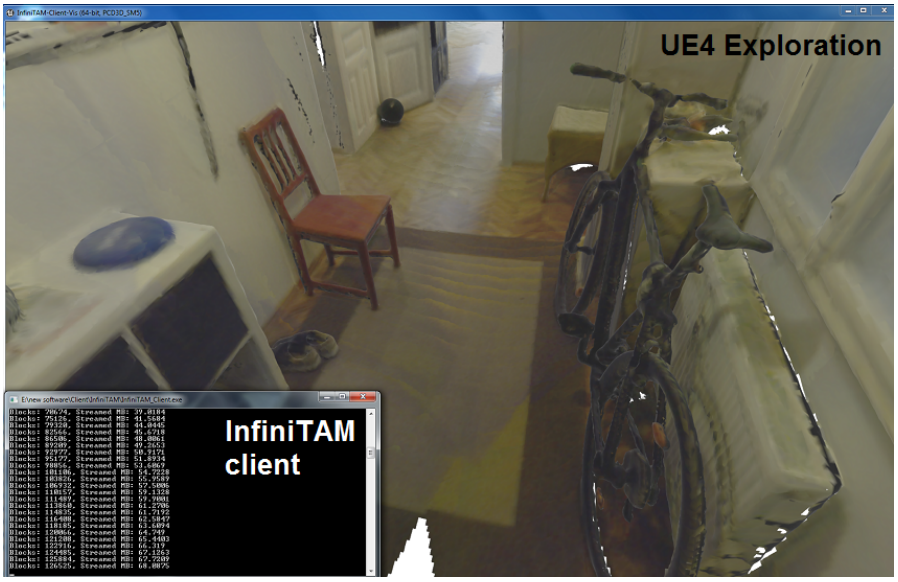


Figure 5.4: Screenshot of the InfiniTAM and UE4 client application.

ICP tracking (TRACKER_ICP) as well as weighted ICP tracking (TRACKER_WICP). The latter uses an ICP weighting scheme which integrates a model of the sensor noise [NIL12]. When using one of the two ICP trackers, the tracking behavior can be further influenced with the ITMLibSettings variables *depthTrackerICPThreshold* and *depthTrackerTerminationThreshold*. The ICP threshold defines the maximum depth difference for the ICP correspondence. Two points are only associated as pair, if the distance between them is below the threshold. The termination threshold represents the error value when to stop the ICP iterations. Once the error is reached, the ICP process stops early before the maximum amount of iterations. Besides depth-based ICP tracking, one can also choose color-based tracking (TRACKER_COLOR), however, it resulted in stronger drift and more frequent tracking failures during testing.

IMU Integration In order to get an improved tracking performance, one can add IMU rotation estimates to ICP tracking by selecting the IMU tracker (TRACKER_IMU). In this case, the camera rotation is initialized in each frame with the IMU rotation followed by a limited ICP pose estimation. In contrast to standard ICP, a smaller image pyramid with less levels is used where the focus lies on the estimation of the camera translation. The rotation is only estimated on the finest resolution. Since the default implementation does not support the integration of live IMU data, it is added in this thesis. The IMUNetworkEngine is a subclass of IMUSourceEngine and listens for IMU data coming via a UDP network connection. After creating the IMUNetworkEngine, it waits for new IMU orientation data at port 5555. The expected format of an orientation are four numbers (separated by a comma), which correspond to a timestamp and a 3D

Android Rotation Vector [2]. The most recently received orientation is converted to a rotation matrix and is used as the rotation estimate as described above. For this thesis, the integrated IMU of a Nexus 4 smartphone is used. The smartphone is attached to the RGB-D camera and its orientation is streamed to the scanning computer with the existing Android application HyperIMU [Cam16]. This application is able to read the Android Rotation Vector and send it via TCP or UDP. In order to minimize the latency, a tethered connection is used between the smartphone and the scanning computer. Due to the reduced ICP computations, the IMU Tracker makes it possible to run InfiniTAM directly on tablets with a Tegra K1 chip, such as the Google Nexus 9 or the NVIDIA Shield. Note, that the thesis implementation however only runs on Windows.

Loading a precalculated trajectory Besides the support for live IMU data, the tracking module of InfiniTAM is extended to enable the usage of a precalculated camera trajectory instead of performing live tracking. Two different trajectory readers, which support different data formats, are implemented. The TRACKER_READER allows to load camera poses of the scenes from Choi et al. [CZK15] and Zhou et al. [ZK13]. For each frame, the corresponding camera pose is given as a 4x4 matrix [3]. Following line shows an example pose for a single frame:

```
0       0       1
1.0000000000    0.0000000000    0.0000000000    2.0000000000
0.0000000000    1.0000000000    0.0000000000    2.0000000000
0.0000000000    0.0000000000    1.0000000000    -0.3000000000
0.0000000000    0.0000000000    0.0000000000    1.0000000000
```

One pose is encoded by five lines, where the first line holds the frame number (third number) and the other four lines store the 4x4 matrix. Parts of the code to load the trajectory are taken from Choi et al. [CZK15].

The second trajectory reader TRACKER_READER_RTAB makes it possible to read a trajectory as generated by RTAB-Map [LM13]. RTAB-Map stands for Real-Time Appearance-Based Mapping and is graph-based SLAM application integrating loop closure. The loop detection is based on sparse features of the RGB image. The trajectory format is similar to the TUM RGB-D SLAM format [4]. One pose is given as eight numbers, which correspond to a timestamp, a 3D translation vector (x, y, z) and a rotation quaternion (qw, qx, qy, qz):

```
timestamp x y z qw qx qy qz
```

Note, that RTAB-MAP enforces real-time processing. If the processing of a single frame takes longer than the update rate of the camera, frames are dropped. As a result, the generated trajectory does not store a pose for every input frame. One option is to

[2] http://developer.android.com/guide/topics/sensors/sensors_motion.html#sensors-motion-rotate
[3] http://redwood-data.org/indoor/fileformat.html
[4] http://vision.in.tum.de/data/datasets/rgbd-dataset/file_formats

ignore frames with missing poses, i.e. the data of those frames is not integrated into the model. The second option is to use the trajectory reader, similar to the IMU tracker, in combination with the ICP tracker. The precalculated RTAB-Map pose, if available, is only used as a seed point for a following ICP procedure. This second option is set as the default behaviour for TRACKER_READER_RTAB.

5.4 Network Transmission

The incremental transmission of reconstruction data over network is enabled by the adapted ITMSwappingEngine at the server and the newly added streaming engines at both server and client. The network transmission can be turned on via the *useNetworkStreaming* variable in the ITMLibSettings class. The streaming engines of server and client communicate via TCP, where the default port is 5445. Note, that both the client and the server need to use the same scene parameters. The voxel resolution is transmitted to the client directly after a new connection is established. However, since the information if voxels contain color is hardcoded, it still has to be ensured at compile time, that both server and client use the same voxel type.

5.4.1 Server Side Compression and Transmission

As mentioned in the methodology chapter, all voxel blocks, that fall out of the view frustum (and are swapped to CPU), are transmitted to the client, along with their world positions. For this purpose, the server's ITMSwappingEngine is adapted, that it informs the ITMStreamingEngine about newly swapped out blocks. This is achieved by adding the corresponding blocks to the commonly accessible StreamingSet. In order to save memory, each element of this set does not actually store the voxel data but only a reference to the global CPU memory, where the swapped out data is stored anyway. Moreover, each set entry contains the block world position, which is required for the client side integration. For this purpose, the swapping out function of the ITMSwappingEngine is changed, so that also the block positions are copied to CPU. Unlike the voxel data, the positions are however not stored permanently in the global memory. The ITMStreamingEngine runs in parallel to the reconstruction process in a separate thread and regularly checks, if the StreamingSet contains new entries. All blocks referred to in the queue, are gathered and removed from the queue. They are compressed and transmitted to the client via TCP. When the queue is empty, the ITMStreamingEngine checks again after the next frame. If the connection is lost, the current blocks to be transmitted are added to the StreamingSet again and the ITMStreamingEngine waits for the same client to reconnect. During this time, the StreamingSet keeps growing and remembers which voxel blocks need to be transferred upon reconnection. The transmission is based on the Winsock2 library, which is contained in the Windows operating system. The main steps of the server side transmission process are illustrated in Figure 5.5.

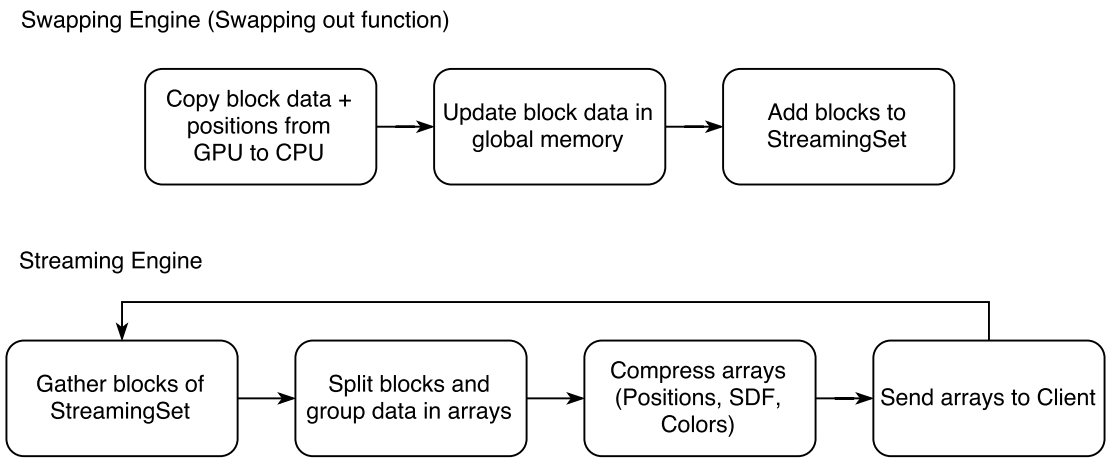


Figure 5.5: Main steps of the ITMSwappingEngine and ITMStreamingEngine for network transmission at the server side.

Keeping the data rate low By using a set data structure for the StreamingSet, each element is unique. If a block is swapped out to the CPU, which is already in the set (i.e. it was swapped out before but it was not transmitted since then), it is not added again. Even though, the StreamingSet is not updated in this case, always the most recently swapped out voxel data is to be transmitted since references to the global memory are stored instead of a data copy. This feature decreases the data rate because multiple retransmissions are reduced. In order to further minimize the required bandwidth, the voxel blocks are compressed in a lossless way using the zLib library (version 1.2.8) before transmission. All gathered voxel blocks and their positions are splitted up into three arrays, where the first array contains the 3D world positions of all blocks, the second array contains all TSDF values and the last array stores the color values. The grouping of the same data types is performed with the intention to achieve a higher compression. Besides that, any padded bytes (due to memory alignment) are avoided, which further decreases the memory footprint. The SDF and color arrays are then compressed individually. The positions are not compressed because the amount of data compared to the SDF and color data is negligible. One uncompressed block requires 6 bytes for the position, 1024 bytes for the SDF data and 1536 bytes for the color. By limiting the speed of the streaming engine thread, the data rate can be reduced even more at the cost of latency for two reasons. If data is for example only transmitted once a second instead of every frame, more blocks have to be streamed and compressed at a time and a possibly higher compression can be achieved. Besides that, if more blocks accumulate in the StreamingSet, the chance is higher that a block is already contained. This limits repeated transmission of the same blocks. Currently, blocks are transmitted to the server, even if they did not change since the last time they fell out of the view frustum. By detecting and sending only changed blocks, the data rate could be further improved.

5.4.2 Client Side Decompression and Integration

Since the client does not have a data integration module, the ITMStreamingEngine takes care of the integration process itself. After receiving the voxel resolution of the server, it initializes the hash table and the voxel block array. Keep in mind, that on the client side, this array stores the whole model on CPU. After the initialization, it starts listening for incoming data in a separate thread. Whenever new data arrays arrive, they are decompressed using zLib and composed to voxel blocks again. The missing voxel weights are filled with default values, since they are not required at the client. These fields could be removed from the voxel type at the client side in order to save memory. Afterwards, the arrived blocks are allocated into the hash table, if necessary, and the voxel block array is updated. Figure 5.6 shows the main steps of the client's streaming process. The ITMStreamingEngine informs the ITMMeshingEngine about new data via

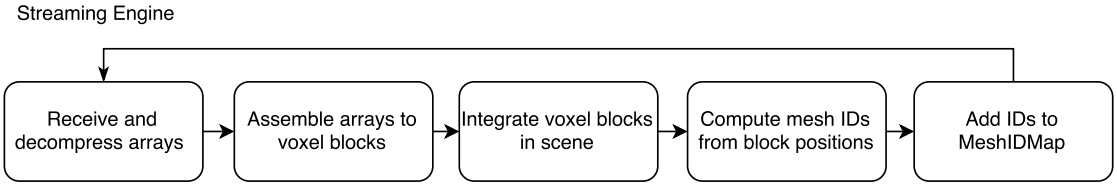


Figure 5.6: Main steps of the ITMStreamingEngine at the client side.

a commonly accessible data structure, similar to the server side ITMStreamingEngine and ITMSwappingEngine. The ITMMeshingEngine maintains the HashIDMap, a hash map containing key value pairs of integer IDs (key) and integer timestamps (value). The ID identifies a mesh block and the timestamp refers to the time when the ID was added to the map. The ITMStreamingEngine computes for each arrived voxel block, the corresponding mesh block ID from the 3D voxel block position and adds it to the HashIDMap with the current timestamp. This timestamp is increased every second. Note, that all voxel blocks within a mesh block result in the same mesh ID. When the map already contains a computed mesh ID, only the existing timestamp is updated with the newer one.

5.5 Dynamic Scene Update

The client is able to dynamically recompute the meshes of regions, where voxels changed, using the ITMMeshingEngine. The ITMSharedMemoryEngine then passes the computed meshes to the UE4 process.

5.5.1 Mesh Extraction

The ITMMeshingEngine runs in parallel and can use more than one thread for an improved performance. The number of desired threads is specified when creating the ITMMeshingEngine, where four threads are used per default. Each meshing thread

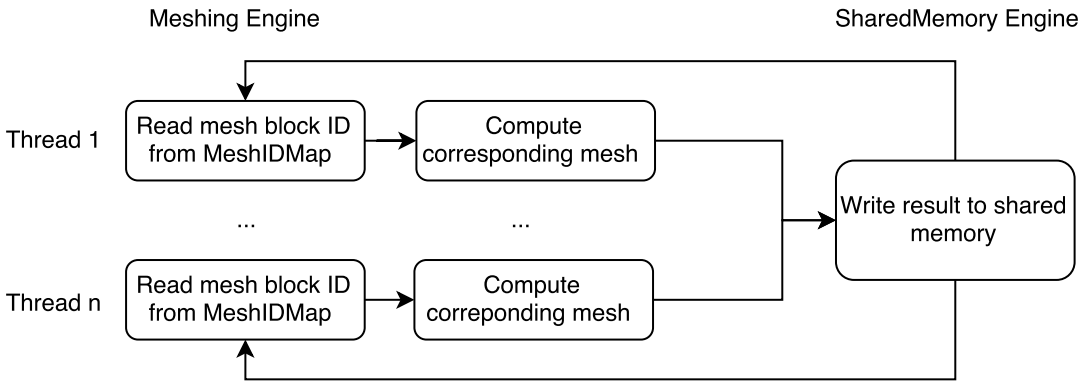


Figure 5.7: Main steps of the scene update in InfiniTAM.

maintains its own mesh memory but all access the same MeshIDMap. This map tells the threads, which mesh blocks require a mesh update. Every thread regularly checks the MeshIDMap for new entries using a thread-safe function. With the aid of the stored timestamps, always the oldest entry is returned. This entry corresponds to the mesh block, which did not change for the longest time and thus, has the highest probability to remain this way. Using this strategy, the available computing power is utilized in an efficient manner since the required number of mesh updates is minimized. Note, that if there is enough computing power available, mesh blocks are still updated even if they are changing from frame to frame. The ITMMeshingEngine also ensures, that two threads cannot process the same mesh block at a time. This achieved by maintaining a separate set of currently processed mesh block IDs. In the case, that the oldest ID is currently processed by another thread, the thread updates the next oldest. The returned mesh block ID is converted to a world position and the mesh for the corresponding cubic mesh block area is generated.

5.5.2 Shared Memory Connection

The computed meshes then have to be transferred to the UE4 process. The connection between the InfiniTAM client and UE4 is implemented with shared memory. For that purpose, the library Ultimate Shared Memory [Cho16] is used, which provides a straightforward interface for shared memory access on a Windows machine. An ITMSharedMemoryEngine is added to the InfiniTAM client, which performs the writing of mesh data to a shared memory region. This region is defined to have space for two meshes. It holds two vertex arrays, two color arrays, two normals arrays and two index arrays. As a consequence, one mesh can be written by InfiniTAM while the other mesh is read at the same time by the UE4 process. However, all meshing threads use the same ITMSharedMemoryEngine. Only one of the meshing threads can pass its computed mesh to UE4 using the ITMSharedMemoryEngine at a time. If multiple meshing threads finished processing at the same time, all but one have to wait. Besides the data arrays, the

shared memory also holds an info field for each mesh. This info field tells the UE4 process, how many vertices and triangles are contained in the arrays, so that it knows how much data is needs to read. Moreover, the info field stores the mesh ID. Apart from identifying meshes, the ID field is also used to indicate, that UE4 is finished with reading the data from shared memory by setting the ID to -1. This way, the ITMSharedMemoryEngine of InfiniTAM is informed, that the corresponding mesh data can be overwritten with new data. Figure 5.7 illustrates the main steps of the scene update in InfiniTAM.

5.5.3 Scene Update in UE4

In order to update the scene in UE4, the experimental procedural meshing feature is used. This feature allows to create and change meshes at runtime. The procedural mesh class allows to store multiple mesh sections, which can be updated individually. This would fit the mesh block characteristic very well. However, it turned out, that updating the mesh sections of a single procedural mesh is slower than having multiple procedural meshes with only one section. As a result, the DynamicMesh class is created, which stores several procedural mesh objects with only a single section. At initialization of the DynamicMesh, 650 of such (empty) procedural meshes are created to avoid any performance issues at runtime. However, if more meshes are required, they are dynamically added in groups of ten meshes. Besides the DynamicMesh class, the UE4 process has a SharedMemoryEngine in order to access and read the mesh data. It runs in a separate thread in parallel to the UE4 game thread and regularly checks if new data is available in the shared memory region. Whenever the ID field is bigger or equal to 0, data is available and the UE4 scene has to be updated. Since the scene update cannot be performed from a parallel thread, the UE4 SharedMemoryEngine converts and saves the data into temporary UE4 arrays which are also accessible by the game thread. The game thread can then update the DynamicMesh in the next frame.

5.6 Visualization and Exploration

The rendering and exploration of the scene is performed with UE4 per default. In order to explore the reconstruction in a VR setup, the hardware needs to be integrated into the pipeline. In contrast to the Oculus Rift, the Virtualizer is not supported natively by UE4. As a consequence, a plugin is developed. Besides the default UE4 exploration, the reconstruction pipeline can be also used with InfiniTAM only. The Virtualizer integration in the standard UE4 mode is highlighted first, followed by an explanation of the standalone rendering.

5.6.1 Virtualizer Integration in Unreal Engine 4

Within the thesis, a UE4 plugin is developed to integrate the Cyberith Virtualizer. It acts as a wrapper to the existing DLL from Cyberith and exposes the functionality to UE4. The DLL is able to connect to the Virtualizer hardware and retrieve information without any other drivers to be installed.

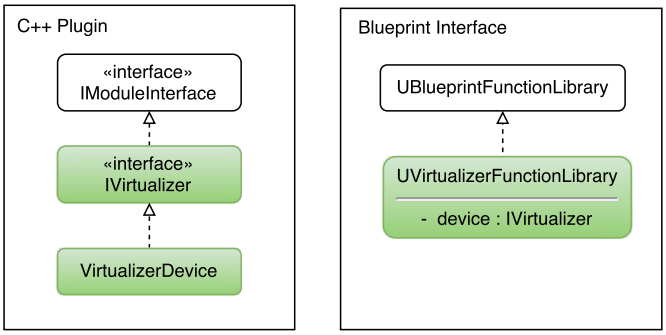


Figure 5.8: Class structure of the UE4 Virtualizer plugin.

Figure 5.8 shows the class structure of the plugin. It is divided into two parts: The actual C++ plugin and a second interface which allows to call the C++ plugin functions via the UE4 blueprint system. The C++ plugin is added by extending UE4's module interface with the IVirtualizer interface. The actual implementation of the IVirtualizer interface is contained in the VirtualizerDevice class. This class handles the connection to the underlying Cyberith DLL and forwards the function calls. Most importantly, the C++ plugin allows to retrieve following information from the Virtualizer:

- Orientation of the user's hip (GetPlayerOrientation)
- Walking direction relative to current body rotation (GetMovementDirection_Local)
- Walking direction (GetMovementDirection_Global)
- Walking speed (GetMovementSpeed)
- Height of the user's hip (GetPlayerHeight)

Besides passing the function calls to the DLL, the plugin handles any required type conversions since the Cyberith DLL returns scalar values for all functions. These values are converted to more meaningful types, such as 3D direction vectors for the player orientation or the movement direction. The retrieval of the global movement direction is also not offered by the DLL. It is added, in order to avoid the repeated manual computation from the player orientation and local movement direction. Note, that the Virtualizer plugin is not implemented as a standard input controller like a keyboard or a gamepad. As a consequence, one has to call the plugin functions manually every frame and adjust the players orientation and position accordingly. The additionally implemented Blueprint Function Library can be used for that purpose, which contains only static functions. Internally, it maintains a VirtualizerDevice object, which handles the function calls. Figure 5.9 shows the integration of the Virtualizer with a UE4 character using it's blueprint. The characters position is updated at every frame by retrieving the Virtualizer information and passing it to the UE4 function AddMovementInput.

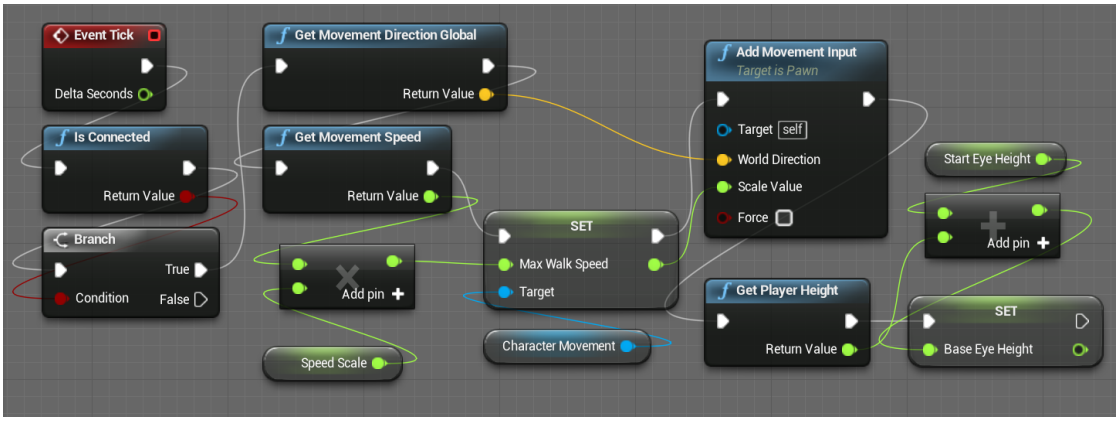


Figure 5.9: Integration of the Virtualizer via UE4's blueprint system.

5.6.2 Client Standalone Visualization

Apart from the standard UE4 rendering and unlike previously explained, the client can be used with InfiniTAM only. In this case, the UI Engine is used the same way as on the server, where standard raycasting of the volumetric representation is performed. As a consequence, the ITMVisualiationEngine is used also on the client side. The ITMMeshingEngine and the ITMSharedMemoryEngine are not required and are disabled. Figure 5.10 shows the modified class structure of the client for this case. The standalone

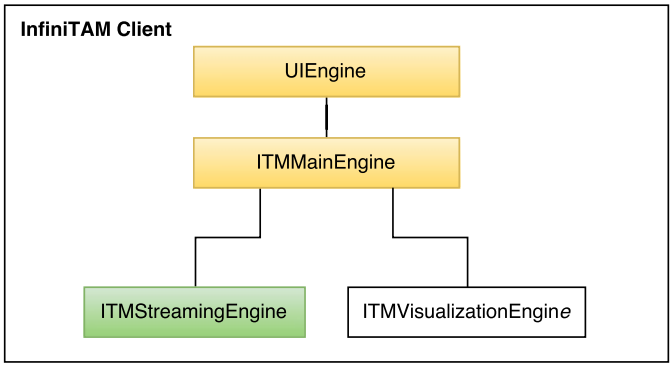


Figure 5.10: Class structure of the client when using the standalone visualization mode instead of UE4.

mode is mainly used for testing and allows to analyze the latency of the network streaming because the individual voxel block updates are immediately visible. When using the UE4 mode, updates are only visible after the corresponding mesh is recomputed. In order to perform raycasting in real-time, the client has to be used with CUDA support, where the model is stored on the GPU. The CUDA support has to be switched on at compile time by disabling the flag #COMPILE_WITHOUT_CUDA. However, since the client does not integrate any form of GPU-CPU swapping, the whole model needs fit

inside the GPU memory and thus, the voxel block array needs be big enough. Note, that only the ITMSharedMemoryEngine is implemented with CUDA support but not the ITMMeshingEngine and the ITMSharedMemory Engine. Thus, if using the UE4 mode, CUDA must be disabled to avoid any errors. The visualization mode can be changed in the client's ITMLibSettings class.

Within the thesis, the UI Engine is also adapted in such a way, that it is possible to render the raycasted image to the Oculus Rift using the Oculus SDK 0.6. Note, that this feature is however disabled by default because high frame rates over 30 fps could not be achieved for a high definition resolution, as required by the Rift, even with CUDA support. The Oculus Rift support can be enabled using the flag #WithRift at compilation. Both at the server and at the client, the Rift support can be enabled. Of course in this case, the standalone mode is mandatory at the client.

CHAPTER 6

Experimental Results

This chapter covers the evaluation of the implemented reconstruction pipeline and consists of two parts. At first, the results of a technical evaluation of the system performance are presented in Section 6.2. Besides that, a user study is performed, which examines the effectiveness of the system regarding the spatial knowledge acquisition in unknown environments. This study is presented in Section 6.3 along with its results.

6.1 Test Data

While the used InfiniTAM framework allows to store large scale environments in memory effectively, the scanning is challenging and for scenes with few geometric details like hallways often not possible. It has to be performed by skilled users since the camera always needs to see enough features to avoid the loss of the tracked camera pose. A tracking failure requires to start the scan process from the beginning since InfiniTAM does not integrate a pose recovery feature. Even if one manages to reconstruct large scenes without tracking failure, the resulting models are not well aligned and overlap because InfinTAM suffers from camera drift and does not integrate any form of drift correction like loop closure. Kähler et al. propose to use the rotation estimate of an IMU to improve the camera tracking, but tests in this thesis only found minor improvements and do not solve the general problem. Since the aim is to explore the reconstructed environments, well aligned models are however crucial. Thus, for evaluation purposes, live ICP camera tracking is suspended and precomputed camera trajectories are used instead. All other steps of the pipeline are still performed in real-time.

In order to compute optimized trajectories for prerecorded camera streams, a state-of-the-art offline reconstruction system by Choi et al. [CZK15] is used. This system considers both color and depth information and tracks the camera using either the RGB-D odometry [SSC11] or the Kintinuous [WJK+13] approach. The key idea is that the recording is divided into several smaller fragments, where in a first step, each fragment produces its

own geometry with its own trajectory. All fragments are then pairwise aligned using geometric registration. A pose graph for the whole recording is generated and globally optimized. The generated trajectory can be loaded with the developed trajectory reader.

During evaluation, three different prerecorded camera streams are used in order to reproduce the same scene multiple times. The data sets are referred to as *Copy Room*, *Lounge* and *Flat*, where the first two sets are taken from Zhou et al. [ZK13] and represent smaller scenes consisting of a single room. These two sets already come with precomputed trajectories. The Copy Room stream consists of 5490 frames (around 2:45 min) and the Lounge recording has 3000 frames (around 1:30 min). The Flat data set is recorded for this thesis and represents a Viennese apartment with 93 m^3 consisting of a hallway, four rooms, a bathroom and a toilet. The recording contains 22175 frames (around 11:10 min) and the corresponding camera trajectory is estimated in a prior step with the mentioned offline approach. The computation is, however, tedious and takes over 20 hours with modern hardware (Core i7-4940MX, Gefore GTX980M, 16GB RAM). Since one needs to find appropriate parameters to yield a well aligned reconstruction, the system has to be executed multiple times. As a consequence, only a single large scale scene is captured and optimized for this thesis evaluation. Figure 6.1 shows a top view on reconstructions of all three test scenes, where the Flat scene is shown at the top, the Lounge scene at the bottom left and the Copy Room at the bottom right. All data sets are OpenNI RGB-D sequences with both a depth and color resolution of 640 x 480 and are captured with an Asus Xtion Pro Live camera at 33 Hz. The depth and color image are already registered using the default internal calibration parameters.

6.2 System Performance

Used Hardware The technical system evaluation is performed on a standard notebook with an Intel Core i7-4940MX processor (3.10 GHz) and a GeForce GTX 980M graphics card. Besides that, the notebook has 16 GB of memory and runs Windows 7 64 bit. Both client and server application are executed on this single notebook, so the network streaming happens only locally, which simulates a perfect network connection.

This section examines the performance of the developed reconstruction pipeline, but note, that the accuracy of the camera pose estimation is not the focus of this thesis. Instead, the ability to store, transmit and explore large environments is at the heart of this technical evaluation. For this reason, the precomputed camera trajectories are also used in this section. Details regarding the camera tracking performance can be found in the original InfiniTAM paper [KPR+15]. The rendering performance of UE4 is also not analyzed in depth. In all test cases, the number of achieved frames per second lies above 200 and a drop of the frame rate during a mesh update is not noticeable.

The following evaluation focuses on three parts: *3D Reconstruction*, *Network Transmission* and *Meshing*. The reconstruction pipeline is executed eight times for each of the three test data sets with different scene parameters. The voxel size is changed between 0.5 cm and 1 cm and the width of the TSDF truncation band is set to either 2 cm or 4

Figure 6.1: Top view on reconstructions of the three used test data sets. The *Flat* data set can be seen at the top, the *Lounge* data set is illustrated at the bottom left and the *Copy Room* data set is shown at the bottom right. All scenes are represented with the same scale to compare the sizes.

cm. Besides that, the scenes are reconstructed with and without color, but the meshing is only evaluated with color. The Copy Room and Flat data sets are processed at the

original camera frame rate of 33 fps. The Lounge recording is processed at only 25 fps since the original speed looks unnaturally fast.

6.2.1 3D Reconstruction

The 3D reconstruction module must be able to process the data at least at the same speed as the camera's frame rate in order to be real-time capable. Since the used Xtion Pro Live camera provides images at 33 Hz, the maximum processing time for a single frame is 30 ms. On the test system, InfiniTAM's reconstruction process takes only 5 ms per frame in average to track the camera and integrate the data into the model. Another 4 ms are required for rendering the live feedback, consisting of the raycasted image and the input RGB-D image pair. Kähler et al. already analyzed InfiniTAM's processing speed in more detail [KPR+15].

The required memory to store a reconstructed scene in the used volumetric data format is now analyzed in more detail. Both server and client need to be able to store this amount of data. Additionally, the client stores the mesh representation but the memory consumption of the mesh is not considered in this section. See Subsection 6.2.3 regarding details about the mesh generation.

As per default, InfiniTAM uses a hash table with a maximum of 1179648 addressable entries, where the ordered part holds 1048576 entries (88.8%) and the unordered part stores 131072 (11.1%). In order to store this hash table and auxiliary data, only about 20 MB are necessary since each entry is represented with 16 bytes. The hash table size should be kept high, because, if either the ordered part or the unordered part is fully occupied, the reconstruction process cannot continue. Keep in mind, that the actual voxel data is not contained in the table directly. The global memory (or the voxel block array on the client side), which holds this data permanently on CPU, is allocated with 327680 entries. This results in a memory usage of around 640 MB for uncolored data (4 × 512 byte / voxel block) and 1280 MB for colored data (8 × 512 byte / voxel block). The global memory can be smaller than the hash table because big portions of the hash table usually remain empty. Note, that the last byte (4th or 8th) of each voxel is actually an empty byte and is only added for memory alignment reasons. By disabling this data padding, the memory footprint can be reduced by 13% (colored) or 25% (uncolored). However, this goes hand in hand with a reduced performance since processors are optimized for aligned data.

Table 6.1 shows the occupancy of the hash table for each test scene after the reconstruction process has finished. The parameter μ stands for the width of the SDF truncation band, i.e. the maximum considered distance from an object surface. One can also see the amount of actually used voxel block memory, which constitutes the minimum size of the global memory. The single room environments require below 70 MB of memory for an uncolored reconstruction at a voxel resolution of 1 cm. When enabling color, a twice as large global memory has to be provided. Furthermore, the required memory is raised by 30% to 40% in average if the truncation band width is increased from 2 cm to 4 cm.

Scene Parameters			Hash Table Occupancy			Memory	
Scene	Voxel Size	μ	Entries	Ordered	Unordered	Colored	Uncolored
CopyR.	1 cm	2 cm	25K	2.24%	1.03%	97 MB	49 MB
		4 cm	30K	2.66%	1.51%	117 MB	58 MB
	0.5 cm	2 cm	139K	12.06%	9.59%	544 MB	272 MB
		4 cm	175K	14.79%	14.81%	682 MB	342 MB
Lounge	1 cm	2 cm	27K	2.49%	0.42%	104 MB	52 MB
		4 cm	33K	3.10%	0.65%	130 MB	65 MB
	0.5 cm	2 cm	152K	13.44%	8.37%	594 MB	298 MB
		4 cm	201K	17.37%	14.33%	786 MB	394 MB
Flat	1 cm	2 cm	207K	18.02%	13.94%	811 MB	406 MB
		4 cm	279K	23.55%	24.75%	1093 MB	547 MB
	0.5 cm	2 cm	-	-	100.00%	-	-
		4 cm	-	-	100.00%	-	-

Table 6.1: Hash table occupancy with default hash table size.

Scene Parameters			Hash Table Occupancy			Memory	
Scene	Voxel Size	μ	Entries	Ordered	Unordered	Colored	Uncolored
Flat	0.5 cm	2 cm	1.28M	26.36%	33.33%	5009 MB	2508 MB
		4 cm	1.85M	35.81%	66.89%	7247 MB	3629 MB

Table 6.2: Hash table occupancy with enlarged hash table. Both ordered and unordered part are increased by a factor of four.

This is due to the fact, that more voxel blocks fall into the truncation band and have to be maintained. An enlarged truncation band can lead to a better camera tracking behavior and avoids artifacts at the mesh extraction process. The biggest impact on the memory however has the voxel size. By dividing the side length in half, one original voxel has to be replaced by eight new ones. But the finer resolution also allows to discard some of the new voxels if they fall out of the truncation band. As a consequence, a finer resolution of 0.5 cm increases the hash table occupancy by a factor of around six. The total occupancy for the single room scenes changes from around 2% up to 16%, while the Flat data set cannot even be processed completely since it leads to a full unordered part of the hash table.

To be able to reconstruct the whole scene, the hash table is enlarged. Both the unordered and the ordered part are increased by a factor of four. While a larger unordered part allows to handle more hash collisions, a bigger ordered part reduces the probability that collisions appear in the first place. Since the memory requirement for the hash table is low, its size should be set too large than too small. Figure 6.2 shows the case with the expanded hash table. Up to 1.85 million blocks are allocated for the Flat scene, which leads to a memory footprint between 5 and 7 GB (colored). The reconstruction of the same scene at a 1 cm resolution requires only 0.8 - 1.1 GB of memory.

Scene Parameters				Colored		Uncolored	
Scene	Voxel Size	μ	Sent Blocks	Sent Data	CR	Sent Data	CR
CopyRoom	1 cm	2 cm	70K	51 MB	3.34	28 MB	2.48
		4 cm	87K	69 MB	3.09	40 MB	2.11
	0.5 cm	2 cm	379K	365 MB	2.54	202 MB	1.84
		4 cm	490K	488 MB	2.46	295 MB	1.63
Lounge	1 cm	2 cm	80K	60 MB	3.24	31 MB	2.52
		4 cm	104K	78 MB	3.24	43 MB	2.35
	0.5 cm	2 cm	440K	428 MB	2.52	225 MB	1.93
		4 cm	608K	552 MB	2.70	312 MB	1.91
Flat	1 cm	2 cm	540K	238 MB	5.54	154 MB	3.44
		4 cm	776K	392 MB	4.85	229 MB	3.33
	0.5 cm	2 cm	3.172M	2010 MB	3.86	1104 MB	2.82
		4 cm	4.913M	2768 MB	4.34	1652 MB	2.92

Table 6.3: Amount of transmitted voxel blocks and (compressed) data in MB along with the achieved compression ratio (CR). The shown data does not contain any TCP/IP overhead.

6.2.2 Network Transmission

As described in Section 4.6, those voxel blocks are streamed to the client, which fall out of the current camera's view frustum. Besides the TSDF and color information, each voxel contains weights, which can be excluded from network transmission to save bandwidth. Each voxel then only uses 2 bytes (uncolored) or 5 bytes (colored) in an uncompressed format, instead of 4 or 8 bytes.

Table 6.3 illustrates the results of the network transmission for all three test scenes. The amount of sent data, shown in the table, considers only the actual payload and does not include any overhead created by the TCP/IP protocol. One can notice, that the number of sent voxel blocks is significantly higher than the number of all allocated blocks in the scene as shown in Tables 6.1 and 6.2. For example, in the CopyRoom scene with a voxel size of 1 cm and a truncation band of 2 cm, 70000 voxel blocks are transmitted to the client but the scene consists of only 25000 blocks. In average over all test scenes, the number of streamed voxel blocks is 2.6 - 3.1 times higher than the number of allocated voxel blocks. This is due to the fact, that the same voxel blocks fall out of the view frustum multiple times, when revisiting areas during the scan process.

Table 6.3 also shows, that the compression ratio (CR) is higher when using color compared to using no color. As a consequence, the cost of using color is mitigated since not 2.5 times more data has to be sent but only 1.5 - 2.0 times more. Another visible trend is the positive effect of a lower voxel resolution on the compression ratio. In all test cases, a better compression is achieved at a voxel size of 1 cm compared to 0.5 cm. The width of the truncation band has no effect on the compression. Besides the parameters, also the scene itself has an influence on the compression as the Flat data set leads to higher ratios

Scene Parameters		Colored		Uncolored	
Voxel Size	μ	Avg MBit/s	Max MBit/s	Avg MBit/s	Max MBit/s
1 cm	2 cm	3.1	16.5	1.7	11.7
	4 cm	4.3	24.2	2.5	14.8
0.5 cm	2 cm	23.0	113.8	12.4	68.8
	4 cm	30.7	175.0	18.0	110.6

Table 6.4: Average and maximum data rates of all three test scenes.

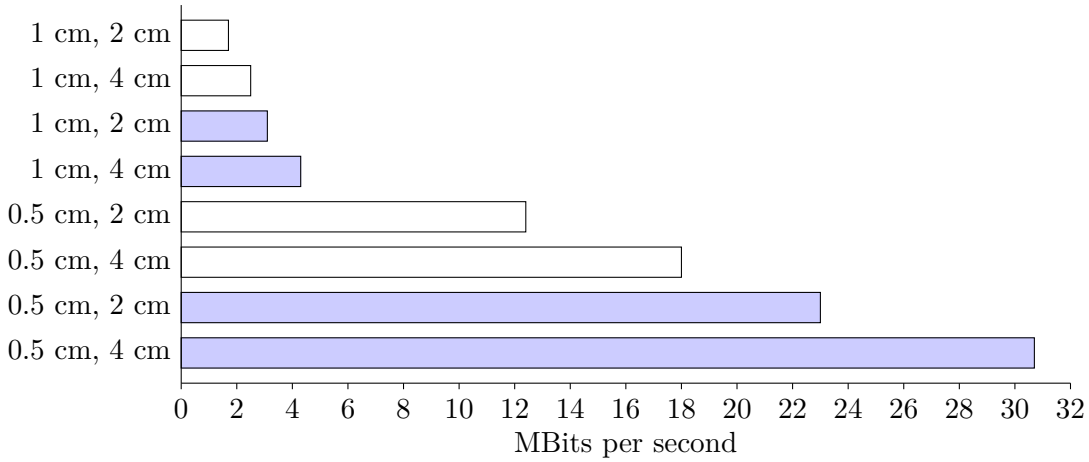


Figure 6.2: Average data rates. The first value of the bar label represents the voxel size and the second value represents the width of the truncation band. Blue bars indicate colored reconstructions.

compared to the other two scenes. A higher compression ratio is generally achieved when the data has many identical values or repeating patterns.

The resulting data rates during the transmission are listed in Table 6.4, where one can see the average and the maximum rates in MBits per second of all test cases. The average data rates are also illustrated in Figure 6.2 to inspect the differences visually. A colored reconstruction with a voxel size of 1 cm and a truncation band of 4 cm led to an average data rate of 4.3 MBit/s, however there occurred peaks up to 24.2 MBit/s. In average, these maximum data rates are 5 - 6 times higher than the normal data rate. As already explained above, the data rate for colored reconstructions is between 1.5 and 2.0 times higher than for uncolored reconstructions. Furthermore, it can be seen, that the finer 0.5 cm resolution results in a seven times higher data rate. Finally, the enlarged truncation band of 4 cm leads to a 40% increase of the bandwidth requirement.

It has to be noted, that the data rate is very dependent on how the scene is scanned. If the robot or the person scanning the scene is moving very fast, the data rate also increases because more blocks fall out of the view frustum every second. Especially fast rotations lead to peak data rates and should be avoided. By decreasing the maximum

considered depth (i.e. moving the frustum's far plane closer), one also minimizes the bandwidth since less blocks can fall out of the smaller view frustum. The full range of the depth camera should not be used anyway since the depth error increases quadratically with increased distance to the sensor. In all tests of this evaluation, the maximum depth is set to three meters.

6.2.3 Meshing

In this section, the processing speed of the meshing engine as well as the extracted meshes are analyzed. Regarding the processing speed, the meshing module requires 65 ms in order to extract the mesh of one mesh block. Note, that this number represents an average value. The mesh extraction can involve peak values ranging from 700 ms to 1 second per mesh. When using only a single thread for the mesh extraction, the resulting mesh can be written to the shared memory with no delay. If multiple threads are used, a little overhead is introduced due to synchronization. For instance, four meshing threads increase the processing time for a single mesh by around 3 ms. The advantage of using more threads is the increased update rate of the mesh representation. When using four meshing threads during the reconstruction of the Flat scene, the whole mesh representation is updated around 40000 times in contrast to only 10000 times, when using a single thread. As a consequence, the mesh representation grows more smoothly and changes in the volumetric model are reflected earlier. With the used hardware, also one thread is enough to achieve a real-time capable update rate. However, the mesh representation expands in bigger blocks.

The remaining part of this subsection presents details about the extracted meshes. In Table 6.5, one can see the number of individual meshes (which corresponds to the number of used mesh blocks) along with the total amount of vertices and triangles. Moreover, the required memory to store the meshes is listed. One vertex can be stored with 28 bytes, where 12 bytes are needed for the position, 12 bytes are used for the normal vector and 4 bytes are required for the color. A triangle defined by three vertex indices requires another 12 byte (4 byte per index). In average, one individual mesh consists of around 7000 vertices and 13000 triangles, however, there is the tendency, that the size of the individual meshes decreases with a bigger truncation band. For instance, the mesh representation for the Flat scene (0.5 cm voxel size) requires 1141 MB at a truncation band of 4 cm but 1245 MB at 2 cm. Note, that this is in contrast to the volumetric representation, which requires more memory at a larger truncation band. The total number of meshes is not affected by the truncation band parameter but by the voxel resolution. The Flat scene at 1 cm voxel size and 4 cm truncation band leads to 741 individual meshes, whereas a voxel size of 0.5 cm already results in 3353 meshes. In average, the number of meshes is four to five times higher when using the smaller voxel size of 0.5 cm. Note, that by using the smaller voxel size, one mesh block is subdivided into eight new ones, but not all of the new ones actually contain meshes.

Scene Parameters			Extracted Mesh Representation			
Scene	Voxel Size	μ	Meshes	Memory	Vertices	Triangles
CopyRoom	1 cm	2 cm	128	47 MB	0.98M	1.81M
		4 cm	129	44 MB	0.92M	1.73M
	0.5 cm	2 cm	582	212 MB	4.42M	8.25M
		4 cm	584	191 MB	3.95M	7.47M
Lounge	1 cm	2 cm	142	45 MB	0.95M	1.71M
		4 cm	152	42 MB	0.88M	1.63M
	0.5 cm	2 cm	578	216 MB	4.52M	8.32M
		4 cm	490	165 MB	3.42M	6.42M
Flat	1 cm	2 cm	724	270 MB	5.71M	10.29M
		4 cm	741	262 MB	5.47M	10.13M
	0.5 cm	2 cm	3335	1245 MB	26.07M	47.97M
		4 cm	3353	1141 MB	23.69M	44.47M

Table 6.5: Number and size of the extracted meshes (for colored reconstructions only).

6.2.4 Streaming Meshes versus Streaming Voxels

When comparing the required memory for the mesh representation (Table 6.1) and the volumetric one (Table 6.5), it is apparent that a mesh is much cheaper. This is also illustrated by Figure 6.3, which shows the required memory to store the Flat scene as mesh (blue) or as TSDF voxel blocks (green) for different scene parameters.

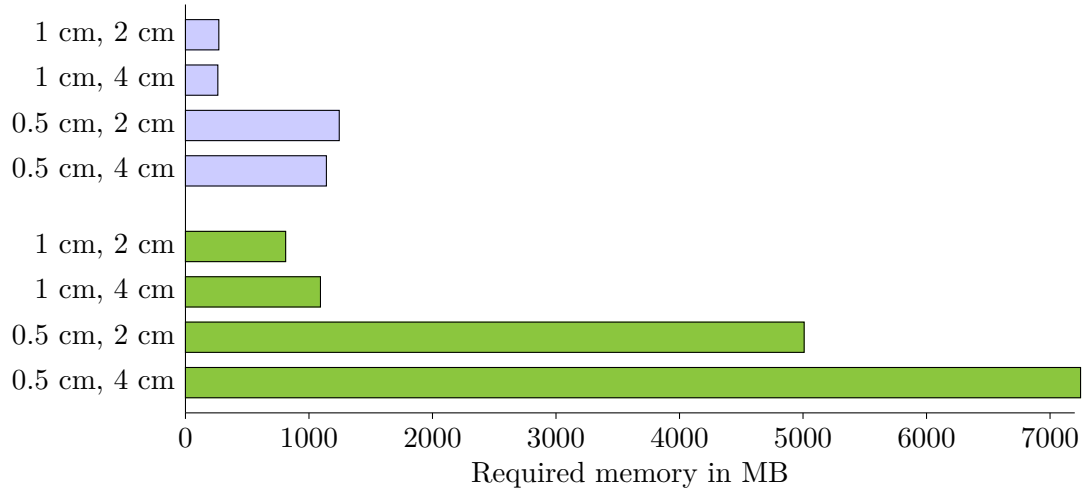


Figure 6.3: Required memory to store the Flat scene either as mesh (blue) or as TSDF voxel blocks (green). The first value of the bar label represents the voxel size and the second value represents the width of the truncation band. Only colored reconstructions are considered.

The network transmission of the extracted meshes instead of voxel blocks would therefore also require significantly less bandwidth, if the streaming happens after the reconstruction has finished. For instance, the Flat data set at 1 cm resolution and 4 cm truncation band requires 1093 MB (colored) of voxel block data to be streamed in contrast to only 262 MB for the mesh representation. Both values represent the data before compression. Challenging in mesh streaming is, however, the incremental update of the scene during scanning. While at voxel block streaming, each voxel block can be updated and transmitted individually, a mesh has to be retransmitted completely whenever any of the underlying voxel blocks have changed. To avoid frequent mesh retransmissions, one could send a mesh only after there are no changes for a certain time. This strategy implies, that the scene does not build up as continuously on the client side compared to voxel block streaming. Another option is to triangulate and update each voxel block individually to avoid retransmission of unchanged mesh parts. In both cases, increased resources are required at the mobile server to perform the mesh update in real-time, in addition to the reconstruction. An advantage of mesh streaming, besides the lower data rate, would be the decrease in the required memory on the client side. Only the mesh representation, but not the volumetric one, needs to be stored in this case.

6.3 User Study

Besides the technical analysis of the system performance, a user study, based on the developed framework, constitutes the second part of the evaluation.

One aim of this thesis is to provide the means to explore and get to know unknown remote locations. The user should get an overview quickly and should be able to navigate in these scenes. It was shown in previous studies, that the spatial knowledge acquisition is enhanced when performing virtual exploration as natural as possible [RL09]. With standard I/O devices, the user sits at a desk and navigates through the scene in a purely virtual way by pressing buttons. As a consequence, the user has to build a mental map of the environment solely from the visual information displayed on a desktop monitor. However, when walking in the real world, one can also resort to non-visual cues in order to track the orientation and position within the environment. Several studies have come to the conclusion, that especially the feedback of real body rotation, coming from the inner ear and the muscles, helps to orientate in the world [RB04]. By integrating a HMD like the Oculus Rift, it is possible to provide this feedback also in the virtual world. The user can control the virtual rotation by rotating the head in the real world. While real body rotation can be integrated very well, natural walking is more complex to achieve. Usually, the space available in the real world is limited and therefore, also the virtual space is constrained, when one wants to map real and virtual walking one to one. An ODT tries to solve this issue by enabling unlimited virtual walking, even though walking in the treadmill is not entirely natural. While real walking helps to better estimate distances, it is unclear if this is true for walking in a passive ODT.

6.3.1 Objective

While related work has studied spatial knowledge acquisition in artificially-generated virtual environments using immersive VR input- and output devices [RL09], no prior art exist, that investigates user perception of streamed 3D reconstructions as well as spatial knowledge acquisition in streamed reconstruction. With this study, the gap should be closed. It should constitute a foundation for understanding users' perception in remotely captured and incrementally reconstructed dense 3D scenes while using means for active navigation in immersive VR. Therefore, the focus lies on investigating the following questions:

- How do input device and scene representation influence spatial knowledge acquisition?
- How do users perceive streamed 3D reconstructions in an immersive virtual environment?

Besides analyzing the general perception and the spatial knowledge acquisition, it is interesting to see, if using the ODT has an effect on cybersickness. Walking virtually, while actually remaining seated in the real world, leads to a sensory conflict and promotes cybersickness. The user receives visual feedback which is in contrast to the information provided by the muscles and the vestibular system. By mapping virtual and real walking one to one, the conflict can be resolved. However, it is unclear, if this is also true for an ODT. In this case, the user still remains at the same place in the real world and receives a different sensory feedback compared to natural walking.

6.3.2 Means of Evaluating Spatial Knowledge

Types of Spatial Knowledge This section introduces the used methodology to measure the spatial knowledge of the test persons. Before evaluating knowledge of the environment's spatial arrangement, one needs to know, that this knowledge can be classified into three types: *Landmark knowledge*, *route knowledge* and *survey knowledge* [SW75]. Landmark knowledge can be considered as the lowest form of spatial knowledge. A person has this kind of knowledge if he or she is able to recognize already visited places. However, the location of two or more different places in relation to each other is unknown. With landmark knowledge only, the person is not able to find a way through the environment between place A and place B, even if both are known. Route knowledge enables exactly this wayfinding between two known locations, if a path connecting theses places was already traveled before. Note, that only known routes can be taken but no alternative ones because the overall layout of the scene is still unknown. The person only remembers which turns to take but not the exact distance or orientation of the whole path. The last type, the survey knowledge, constitutes the most complete form of spatial knowledge. Having a survey knowledge allows to imagine the environment from a bird's eye perspective in the correct scale. As a consequence, one is able to find routes between

arbitrary places in the environment. Shortcuts can also be inferred, even if they were not traveled before.

Evaluation Methods Depending on the spatial knowledge type to be tested, different evaluation tools can be used. Landmark knowledge is mainly evaluated using landmark recognition tests. Since landmark knowledge represents only a very rough idea of the environment, it is not considered to be evaluated within this thesis. The most common and natural manner to test route knowledge is, to let the user reproduce an already visited path between a starting point and a target point. The user performs this so called *wayfinding task* or route replication task while being in the test environment [BTG97]. In order to measure the degree of spatial knowledge, one can record the number of wrong turns, the amount of required time, the length of the traveled path or the body rotations performed. The more junctions are contained in the path, where the user has to decide which way to follow, the more expressive results can be obtained. The wayfinding tasks can also reveal survey knowledge if the user is able to find previously unknown shortcuts on purpose. A further means to test survey knowledge is *sketch mapping*[Moo76]. In sketch mapping, the test person has to draw a map of the explored environment from a bird's eye perspective. In principle, this task can be considered an effective tool since it requires to reproduce exactly the user's mental representation of the scene. However, the results depend very much on the drawing skills of the user and it is challenging to distinguish map distortions due to bad drawing skills and map distortions due to lack of spatial knowledge [Wal99]. Besides that, no standard means of evaluating the drawn maps exist. In many approaches, the maps are only evaluated qualitatively using independent reviewers. A method derived from sketch mapping, which does not involve any drawing, is called the *map placement task* [Wal99]. Within that task, the user has to arrange magnets on a white board, which represent landmarks of the scene. The goal is to place all landmarks in the correct relative location to each other. The final arrangement can be rotated and uniformly scaled in order to produce a normalized representation. The angles and distances between the magnets can then be used to infer the degree of spatial knowledge. Finally, spatial knowledge can be also tested using *pointing tasks* [Sie81]. As the name suggests, the test person has to point in the direction of an unseen target, where the angle between the pointed direction and the correct direction provides the main error measure. An advanced version of pointing to unseen objects is called *projective convergence* [Sie81]. Within that method, the user has to estimate the direction and distance from three different locations to the same target as illustrated in Figure 6.4. The three distance-angle pairs result in three points, which form a hypothetical triangle shown in red. This triangle is then the basis for four spatial knowledge measures: *Mean angle error*, *Mean Miss Distance*, *Locational Error* and *Consistency*. The *Mean angle error* represents the average of the absolute angular errors, i.e. the difference between the estimated direction and the correct direction. The *Mean Miss Distance*, or locational accuracy, is given as the sum of the distances between each triangle corner and the target point. The *Locational Error* is very similar to the *Mean Miss Distance* and represents an alternative measure. It is defined as the distance between the triangle's centroid and the

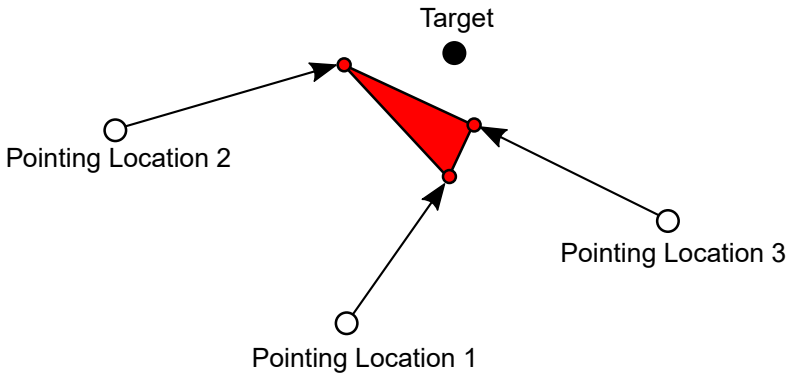


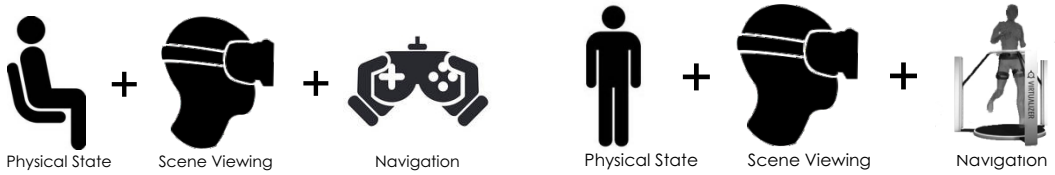
Figure 6.4: Principle of the projective convergence task. The user has to estimate the direction and distance to a target (black dot) from three different locations (white dots). The three estimated positions from a triangle (red), which is the basis of the spatial knowledge measures.

target point. Finally, the *Consistency* is given as the perimeter of the triangle.

6.3.3 Apparatus

Input Device and Scene Representation The experiment is conducted with the developed 3D reconstruction system. The entire scene is presented to the users at once (referred to as *Data 1*) or the scene is incrementally updated in a frame-wise manner to simulate live exploration while the remote capturing is still running (referred to as *Data 2*). As navigational input devices, the two-handed XBox gamepad with joystick functionality, and the Virtualizer ODT, are used. All tests are performed by viewing the scene with an Oculus Rift DK2, and combined either with the gamepad (referred to as *Setup 1*) or with the ODT (referred to as *Setup 2*), as shown in Figure 6.5. When using *Setup 2*, the viewing direction and the walking direction is decoupled, i.e. the participant can walk in one direction while looking in another direction. When using *Setup 1*, the walking direction is determined by the orientation of the HMD. The test participant sits on a swivel chair, which can be rotated 360 degrees, to be able to walk in all possible directions. The swivel chair is chosen according to Nybakke et al.[NRI12], so that vestibular cues of real body rotation are integrated in both setups. This way, only the actual walking in the ODT can be evaluated.

Test Environment All participants explore the same environment since scanning and reconstructing multiple large scale environments is challenging, especially if the scene complexity should be the same. The Flat data set from Section 6.1 is used for this study in combination with the precomputed camera trajectory to provide a well aligned model. The size and layout of the Flat with its three rooms, two bathrooms, kitchen and long nested hallway challenges the user in terms of navigation and orientation, and thus is well suited to examine users' subjective perception and spatial knowledge acquisition.



(a) Setup 1 (b) Setup 2

Figure 6.5: Two different input/output device setups.

Since the data is prerecorded, the reconstructed scene is also known beforehand. As a result, basic collision information is fitted manually to the reconstructed model in order to prohibit the users walking through any objects. Invisible cuboids are used for that purpose. Note, that reconstruction errors do not pose a problem this way. If performing collision detection against all reconstructed triangles instead, reconstruction errors can block the way. Besides avoiding to walk through objects, the invisible collision geometry also ensures, that the user's height above the floor is the same all over the scene. Since the floor can be bent due to camera drift, it is necessary to adjust the user's position accordingly, using invisible floor planes.

System Settings In the Flat data set, the scanning party proceeds very slowly through the environment in order to avoid any tracking failures. In consequence, the scene also expands only slowly. For the user study, the processing is speeded up by a factor of two, so that the users have the full scene available within a reasonable amount of time. After five minutes and 30 seconds, the environment is fully reconstructed. The scene is reconstructed with color information at a voxel resolution of 1 cm^3 and only camera information within a distance of three meters is integrated into the model. The width of the truncation band is defined to be 4 cm. The size of a single mesh block is set to the standard value of 80 cm^3, i.e. one mesh block covers the area of 10^3 voxel blocks.

Used Hardware Both server and client run on the same computer and streaming happens only locally on this single PC in order to avoid any networking problems during the user tests. The user study is performed on a desktop computer with an Intel Core i7-4790K processor (3.10 GHz) and a GeForce GTX 980Ti graphics card. Besides that, the PC has 32 GB of memory and runs Windows 8.1.

6.3.4 Participants

Forty-two (42) participants (12 females, 28.5%) were involved in the experiment, while forty (40) participants (12 females, 30%) successfully finished the experiments. Two male participants aborted their experiments due to cybersickness symptoms, both encountered while using the ODT. Participants' ages ranged between 18 and 57 years (mean $\mu = 31.15$, standard deviation $\sigma = 7.9$ years). 25 participants reported previous experience with HMDs while seven own a HMD.

6.3.5 Study Design

The study procedure for a single user consists of five stages: 1) introduction and pre-questionnaire, 2) training, 3) exploration, 4) evaluation, and 5) a post-questionnaire. At stage 1, users are informed about the study and the procedure. Additionally, they have to fill out a pre-questionnaire. Stages 2) - 4) are performed by each user wearing the immserive VR hardware. At stage 2, users are introduced to the input- and output hardware – either *Setup 1* or *Setup 2* – by explanation and demonstration. Next, users have up to 5:00 minutes to familiarize with the hardware by freely walking in a test environment, which comprises a simple UE4 scene with some artificial virtual objects. As soon as the user feels confident or the maximal training time is reached, the exploration stage (3) starts. Therefore, the 3D reconstruction is either presented in *Data 1* or *Data 2* mode. The users can freely walk up to 6:00 minutes with the instruction to explore the scene, more information are not given. Upon completion of the exploration, users have to perform two tasks in the evaluation phase (4) to assess their spatial knowledge. Therefore, they keep wearing the VR hardware and perform a projective convergence task and a wayfinding task as explained in Section 6.3.2. Projective convergence is done first to avoid further spatial knowledge acquisition by free walking during the wayfinding task. At the end of the procedure, users have to fill out a post-questionnaire (5).

Projective Convergence and Wayfinding Figure 6.6 illustrates the target point as well as the three pointing locations for the projective convergence task. As spatial knowledge measures, *Mean Angle Error*, *Consistency* and *Mean Miss Distance* are applied. In Figure 6.7, one can see an example result for the projective convergence task. During the wayfinding task, the *Total Trajectory Length*, the *Total Amount of Head Rotation* and the *Total Time* is recorded. The starting point and the target location are also shown in Figure 6.6.

Sketch mapping is not integrated in the user study due to the issues related to drawing skills and also due to the lack of a standard quantitative analysis. Map placement is not performed because the user is required to remember various landmarks. If the test person cannot recognize the test landmarks, the user is not able to perform the task, even if he or she has a good understanding of the overall layout of the scene. Note, that this is also true for the used wayfinding and projective convergence task, but in this case, only two landmarks are required. The toilet is selected as the wayfinding target and the washing machine in the bathroom is selected as the pointing target because both are unique and noticeable locations within the flat.

Pre-and Post-Questionnaire The used questionnaires can be found in Appendix A and B. In both pre-and post-questionnaire, it is asked for cybersickness symptoms using the simulator sickness questionnaire (SSQ) [WS98]. The indicator *Simulator Sickness* is computed as the difference between the sums of perceived symptoms before and upon test completion. In the post-questionnaire, further subjective measures are included to collect participants' perceptions of the immersive exploration. The measures are evaluated using

Figure 6.6: Locations used for the evaluation tasks. The filled green circles represent the three pointing locations for the projective convergence task and the green unfilled circle shows the corresponding target point. The red circles illustrate the starting (filled) and target location (unfilled) for the wayfinding task.

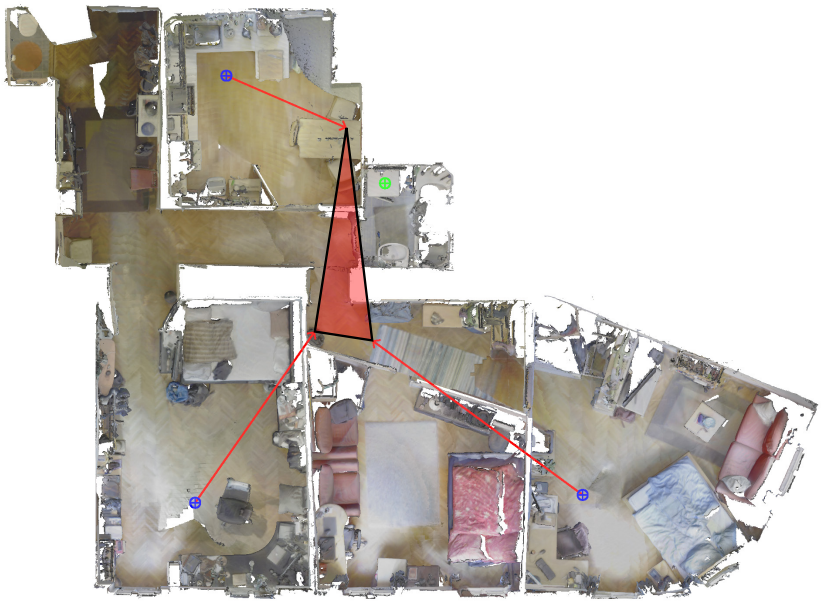


Figure 6.7: Result of a single projective convergence task.

5-point Likert scale ratings, denoting participants' degree of agreement with the following statements:

- *Perceived Satisfaction of Input* measured as participants' evaluations of the statement "Please rate how much you enjoyed using the device for navigation."
- *Perceived Scene Quality* measured as participants' evaluations of the statement "Please rate how much you enjoyed the quality of the virtual environment."
- *Perceived Spatial Understanding* measured as participants' evaluations of the statement "I was able to quickly get an overview of the indoor environment."

For both *Perceived Satisfaction of Input* and *Perceived Scene Quality*, scale levels are: not at all (1), not much (2), moderate (3), much (4), very much (5). For *Perceived Spatial Understanding*, scale levels are: strongly disagree (1), disagree (2), neither agree nor disagree (3), agree (4), and strongly agree (5).

Furthermore, the task load and system usability is measured as follows:

- *Perceived Task Load* is measured with the NASA Task Load Index (TLX) [HS88] and is computed as a mean of questionnaire's results. Scale levels are: very low (1), low (2), average (3), high (4), and very high (5).
- *Perceived System Usability* is measured with the System Usability Scale (SUS) [Bro96] and is computed as a mean of questionnaire's results. Scale levels are: strongly disagree (1), disagree (2), neither agree nor disagree (3), agree (4), and strongly agree (5).

6.3.6 Results

The study is conducted using an independent factorial design with the independent variables *Setup* and *Data*, while the dependent variables are the computed performance measures *Mean Angle Error*, *Consistency*, *Trajectory Length*, *Total Rotation* and *Total Time*, and the subjective measures, as denoted in the previous subsection. The quantitative data is analyzed using two-way independent ANOVA, when suitable pairwise t-tests with Bonferroni adjustment are employed.

For each combination of *Setup* and *Data*, ten randomly assigned participants (three females, 33.3%) performed the test. Upon explanation of the hardware at stage two, users took in average $\mu = 1{:}13$ min ($\sigma = 0{:}37$ min) to familiarize with the setup within the virtual test scene with no significant difference between *Setup 1* and *Setup 2*. For *Perceived Satisfaction of Input*, an average satisfaction of $\mu = 3.95$ ($\sigma = 0.876$) is found, with no significant effect by neither *Setup* ($p = 0.377$) nor *Data* ($p = 0.489$). No participant using *Setup 1* reported difficulties with using the input device, while only five participants using *Setup 2* (25%) were able to constantly navigate without any reported or observed

problems. Three of them were novice users. 75% of the users with *Setup 2* had various levels of difficulties to explore the scene with the Virtualizer. This observation is backed by the users' subjective measure *Perceived Task Load* as highlighted in Figure 6.8.

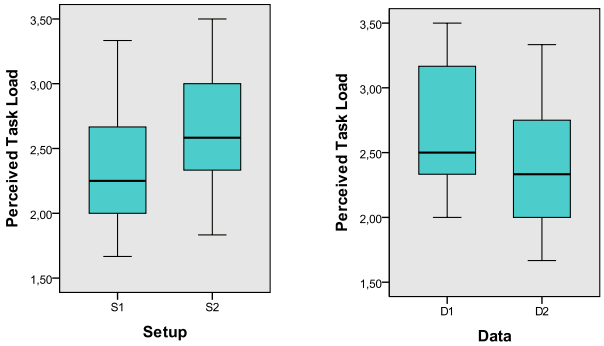


Figure 6.8: Qualitative results of the *Perceived Task Load*.

Here, a significant main effect of *Setup* can be reported with $p < 0.042$, while no significance is found for *Data* ($p = 0.068$), nor for the interaction between *Setup* and *Data* ($p = 0.205$). When analyzing the TLX results in particular, a main effect of *Setup* for the question "How physically demanding was the task?" is found with $p < 0.007$, reflecting the higher perceived physical demand when using *Setup 2*. Furthermore, a main effect of *Data* for "How hard did you work to accomplish your level of performance?" can be reported with $p < 0.03$, reflecting the higher perceived mental load for *Data 2*. Overall, users explored the scene in average for $\mu = 4{:}49$ min ($\sigma = 1{:}22$ min). Users performing the test with *Data 1* explored in average for $\mu = 3{:}59$ min ($\sigma = 1{:}18$ min) while users with *Data 2* required the entire 6:00 minutes due to the time it takes until the entire model is streamed. For *Simulator Sickness*, an increase of symptoms is found with $\mu = 3.60$ ($\sigma = 4.01$), however no significant effect can be reported by neither *Setup* ($p = 0.350$) nor *Data* ($p = 0.277$). For the perceived visual quality of the 3D reconstruction *Perceived Scene Quality*, an average satisfaction of $\mu = 3.78$ ($\sigma = 0.947$) is found. All users were astonished by the size of the flat and constantly commented on details they saw while exploration, such as the coffee machine and pictures. Three users reported to perceive the 3D geometry with low quality due to the HMD resolution, while two users commented on 3D model artifacts and the subjectively perceived low quality of the model's geometry and texture. Four users reported they would feel like snoopers and stated to feel actually being in the flat. Finally, users reported similar *Perceived System Usability* for both *Setup* and *Data* with no significant differences.

Analyzing the quantitative performance data of the spatial knowledge acquisition tasks, users performed similarly for the projective convergence measures *Mean Angle Error*, *Consistency* and *Mean Miss Distance* as well as for the wayfinding measures *Trajectory Length*, *Total Rotation* and *Total Time*. A main effect of *Setup* and *Data* cannot be determined for any of the measures ($p > 0.1$ for all measures). This finding is backed by

the subjective measure *Perceived Spatial Understanding*, where significant results can be reported by neither *Setup* ($p = 0.453$) nor *Data* ($p = 0.707$). Even though there are no significant differences, the best results of the projective convergence task, i.e. the lowest values, are achieved with *Setup 2*. For instance, only one of the top ten *Consistency* values (the 4^{th}) is accomplished with *Setup 1* and only two of the top ten *Mean Angle Error* values are accomplished with *Setup 1* (2^{nd} and 7^{th}). Figure 6.9 shows the results of projective convergence task. It illustrates the tendency, that *Setup 2* leads to lower errors in the spatial knowledge tasks in average, but with a higher variance. In contrast to the *Setup*, the *Data* mode does not affect the results.

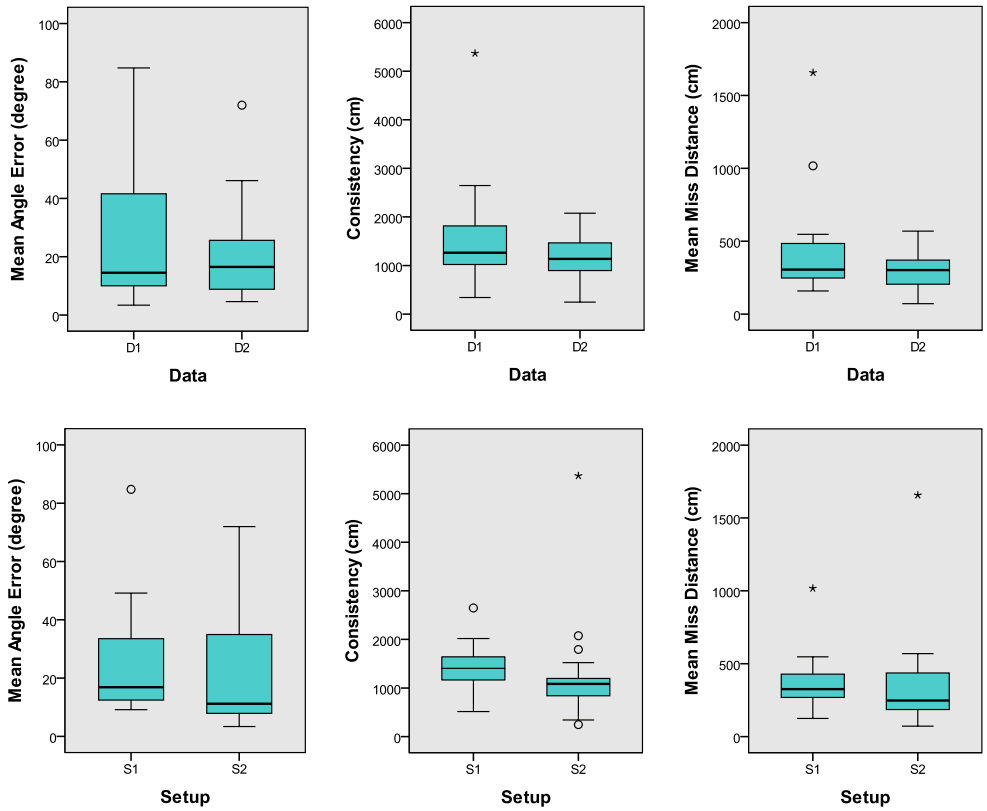


Figure 6.9: Quantitative results of the projective convergence Task.

With *Setup 1*, users tend to navigate quicker and straighter than with *Setup 2*, indicated by *Total Time*, *Total Rotation* and *Trajectory Length*. This is also visible in Figure 6.10, which shows typical trajectories for each setup.

However, *Trajectory Length* varied heavily for *Setup 1* due to false navigation turns that have been observed, also indicated by *Total Rotation*. These findings can be observed in Figure 6.11, which illustrates the quantitative results of the wayfinding task. Similar, to the projective convergence task, the *Data* mode does not change the results.

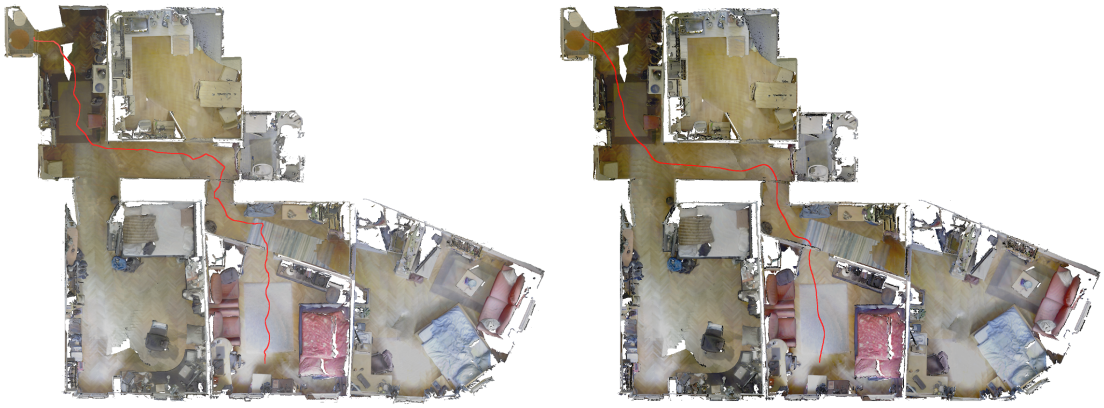


Figure 6.10: Example paths traveled during the wayfinding task. The left path represents a typical trajectory performed with the Virtualizer (*Setup 2*) whereas the right path shows a trajectory performed with the gamepad (*Setup 1*).

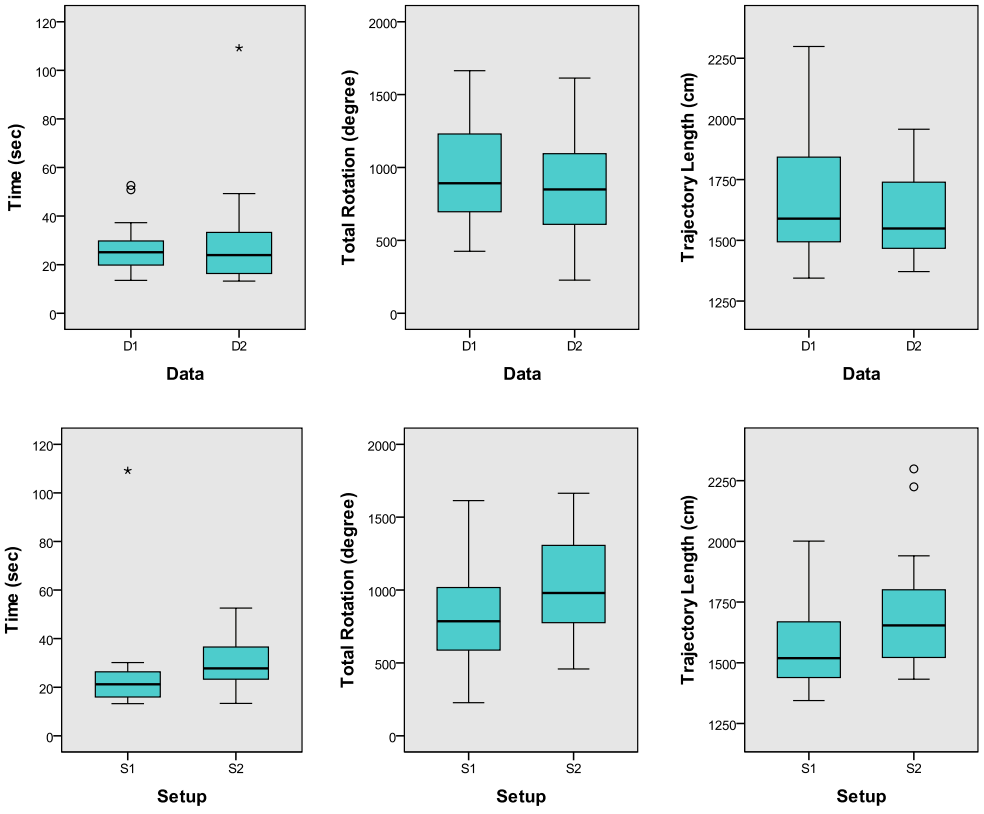


Figure 6.11: Quantitative results of the wayfinding task.

6.3.7 Discussion

The user study is performed to gain insights into general user perception of streamed 3D reconstructions as well as to investigate spatial knowledge acquisition in streamed reconstruction. The quantitative as well as qualitative feedback on the virtual 3D scene reveals high acceptance of the concept by the forty test subjects. Many users immediately suggested real-world use cases such as "I would love to have such an application to be able to have a look into a hotel before I visit it". Users' subjective measures as well as qualitative feedback indicated a high degree of immersion. Comments ranged from "I feel like being in the flat", "I feel like a snooper". Furthermore, test participants wanted to actively get engaged with the virtual 3D reconstruction, indicated by comments such as "I would like to open the entrance door.", "Can I take something out of the shelf?". No participant using *Data 2* reported to feel distracted by the data representation. They rather showed high excitement, indicated by comments such as "This is so Matrix-like". They reported positively on the concept of being decoupled from the view of the remote capturing entity and stated to explore the environment as in reality, by strolling around or investigating one room after the other. Three users asked for a functionality to control the view of the capturing, and four for a faster reconstruction. Given the quantitative performance as well as subjective measures, the evaluation did not indicate a significant influence on spatial knowledge acquisition, neither by the input device nor by the data representation. Regarding the input device, it can be noted that the majority of participants using *Setup 2* had difficulties in navigating with the ODT, as they did not have prior experience with this input device. Thus, they had to concentrate on using the device and could not fully focus on the environment that mitigates the advantages of this direct navigational input device. This is also supported by Waller [Wal99], who found that the proficiency with the navigational interface strongly affects the ability to acquire spatial information. Therefore, the author cannot draw conclusions on the influence of a low-friction ODT with its proprioceptive feedback on spatial knowledge acquisition within a 3D real-world reconstruction. The user study revealed the tendency, that experienced users are able to build up spatial knowledge more efficiently with the Virtualizer, however, further research is required to study this question. Promisingly, at the time of writing this thesis, a Virtualizer software update is released that filters sensor readings and thereby improves ease of walking for novice users. Furthermore, no significant effect of the data representation on spatial knowledge acquisition can be determined which is an interesting finding. It indicates, that spatial knowledge acquisition is not reduced when exploring the scene from the very beginning of the reconstruction process, compared to exploring the scene in its entirety after the reconstruction has finished. This result is in contrast to the finding of Passini [Pas92], who states, that the spatial knowledge is decreased when less decisions are made during exploration, such as when following a guide. With the proposed system, the map is expanded only at one place at a time, which also limits the decision where to go. With a streaming representation, solely the perceived mental load is increased. Both aspects are vital finding to real-world uses cases.

CHAPTER 7

Conclusion

In this thesis, a 3D reconstruction system is developed, which allows to reconstruct an environment and, at the same time, explore the acquired 3D model of the environment in Virtual Reality at a remote location.

Existing systems in the field of 3D reconstruction already make it possible to create such 3D models of real environments by capturing the scene with a RGB-D camera. However, all existing prior art lacks the ability to explore the model while it is still scanned. A live visualization is provided, which shows the current state of the reconstruction. However, this visualization shows the model as seen from the scanning camera's viewpoint. A free exploration, independent from this viewpoint, is only supported after the reconstruction process has finished and a mesh has been extracted in a post-processing step. The scanning of the environment and the exploration of the reconstructed model is usually performed by two different parties. At location A (server), a person or an autonomous robot, equipped with a mobile computer and a RGB-D camera, is reconstructing an unknown environment. At location B (client), another person wants to explore this remote environment in a virtual way. If the transmission of the reconstructed model and the mesh extraction happens only after the scanning is finished, the person at the client side has to wait a long time before any findings can be obtained.

7.1 Contribution

Within this thesis, the existing state-of-the-art 3D reconstruction framework InfiniTAM is adapted and extended in such a way, that it allows to scan large scale environments and explore these environments at the same time at remote location. The support for large scale models is already provided by InfiniTAM's efficient data representation. It stores the model internally as a volumetric signed distance function. In order to save memory, empty space is ignored by applying Voxel Block Hashing. To be able to store a scene with 90 m^2, 800 - 1000 MB are required, when the model is stored with color at a

resolution of 1 cm. Apart from the low memory footprint, InfiniTAM is implemented efficiently regarding the computational performance and runs on a tablet (NVIDIA Shield, Google Nexus 9) at real-time rates (over 30 fps). Note however, that the current thesis implementation only runs on Windows platforms.

The live and remote exploration is enabled by adding incremental transmission of the reconstructed 3D model and real-time mesh extraction. With these two additions, the waiting time is avoided and the reconstructed model can be remotely explored from the beginning of the scan process. Environmental knowledge is gained quicker and decisions can be made earlier. Apart from that, the scanning of the environment could be influenced by the remote party if anything interesting is detected during exploration.

Incremental Model Transmission The novel networking module streams all those voxels, which fall out of the scanning camera's view frustum, to a remotely connected client. The voxels are compressed in a lossless way using DEFLATE and are transmitted with TCP. In combination with the already space-efficient model representation of InfiniTAM, average data rates between three and four MBits/sec are achieved, when reconstructing a scene with color at a voxel resolution of 1 cm. Note, that this value only corresponds to the payload, and does not integrate any networking overhead.

Dynamic Mesh Update At the client side, the model is additionally stored in a mesh representation since a mesh is favored for efficient rendering and interaction. The scene is divided into a regular 3D grid, where each grid element holds its own mesh. Each mesh is recomputed using CPU-based Marching Cubes, whenever any underlying voxel changed. Even though the computation is performed on CPU, the meshing module is able to update the mesh fast enough to keep up with the scanning and reconstruction process. On modern hardware (Core i7-4940MX, Gefore GTX980M, 16GB RAM), the computation time for a single mesh is below 100 ms in average.

Integration of Unreal Engine 4 For visualization and exploration purposes, the extracted meshes are rendered using the existing Unreal Engine 4. A shared memory based communication between the client side InfiniTAM application and the UE4 project is implemented in order to forward the mesh data. Within UE4, the rendering is performed with over 200 fps, even for large scenes consisting of over ten million triangles. The dynamic update of the mesh data during runtime does not lead to notable drops in the frame rate.

Exploration in Virtual Reality The remote environment can be explored in an immersive Virtual Reality Setup, consisting of an HMD and an omnidirectional treadmill. A UE4 plugin is developed in order to integrate the Virtualizer, an omnidirectional treadmill by Cyberith. Using the VR Setup, the scene can be explored in a more natural way. A user study is performed to examine the users' general perception of and spatial understanding within 3D reconstructed environments using the VR devices. The

results indicate a high acceptance of the general concept, combined with a high degree of perceived immersion into the virtual real-world environment. A significant effect of the input device on spatial knowledge acquisition could not be found which might be due to problems users encountered when using the ODT for the first time. The results revealed that spatial understanding is not affected by the way the data is presented to users, which is promising for use cases that require exploration from the very beginning of the reconstruction process. The presented work provides a foundation for enabling immersive exploration of remotely captured and incrementally reconstructed dense 3D scenes and understanding users' perception within this real-world virtual environments.

7.2 Future Work

The implemented system allows to scan large environments and store them efficiently in memory. However, in order to be used by untrained users in arbitrary scenes, the camera pose estimation needs to improved. Due to frequent tracking failures, it is challenging to process longer video sequences, especially in areas with few geometric features. Even if one manages to reconstruct the whole environment, camera drift inevitably leads to misaligned models. The effect of tracking failures can be mitigated by integrating a pose recovery procedure. If a valid camera pose is found again, the scanning process does not have to be started from the beginning, but can continue from the current position. Camera drift can be minimized by detecting already visited places and closing the loop. Challenging in this case, is the fact, that the model of both server and client needs to be updated. One option, which avoids retransmitting all changed parts of the model, is to send only the loop closure instructions and to perform the actual deformation on both server and client. Apart from loop closure, the general tracking accuracy can be improved by fusing various types of sensors. Currently, only the depth images are used for tracking, but one can also use the color information, IMU data or the strength of Wi-Fi or bluetooth signals.

As the technical evaluation revealed, a mesh (or a point cloud) representation is more memory efficient than the TSDF. Streaming the mesh data, instead of the voxels, can significantly reduce the required data rate. For that purpose, the meshing has to be performed on the limited mobile platform and therefore, a computationally more efficient mesh extraction has to be developed. Moreover, one needs to find a strategy, how to cope with the dynamic nature of the volumetric data structure. Currently, changes in the volumetric representation are reflected at the client side right away by recomputing the corresponding meshes. If the mesh extraction happens on the server side, the meshes would have to be retransmitted after every update. This also includes the unchanged parts of the mesh.

Independently of the data type to be sent, the data rate can be improved by choosing a proper network protocol. The current implementation uses the TCP protocol because it is widely supported. However, it contains unrequired features, which increase the data rate. In future work, a more efficient protocol with less overhead can be evaluated and

integrated. Apart from that, the network transmission should be ported from Windows to a mobile operating system, like Android or iOS, by replacing Winsock2 with a multi-platform library. This way, the 3D reconstruction process can be performed conveniently with a handheld tablet.

Regarding the mesh computation, a current drawback is the high number of triangles generated by Marching Cubes. In order to allow real-time interaction for larger scenes, the meshes need to be optimized. One possibility is geometry simplification, such as plane detection. The planes can then be represented with fewer triangles. Challenging is however the fact, that the volumetric data does not only expand, but can also change in existing regions. One could detect meshes, which did not change in the recent past and fuse those to bigger meshes, so that larger optimizations are possible. Whenever underlying voxels changed, the bigger mesh is replaced by the smaller ones again.

Finally, the influence of navigational input on spatial knowledge acquisition requires further investigations, i.e. by examining standard input devices in a non-immersive VR, the updated version of the Virtualizer ODT as well as real walking in immersive VR.

APPENDIX A

User Study Pre-Questionnaire

Participant Number: _____

Age _____ (years / months)

[ ] Male [ ] Female

Do you have previous experience with Virtual Reality? [ ] Yes [ ] No

Do you own a head mounted display (HMD)? [ ] Yes [ ] No

Pre-Exposure Simulator Sickness

Please circle below if any of the symptoms apply to you now. You will be asked to fill this again after the experiment.

1.	General discomfort	None	Slight	Moderate	Severe
2.	Fatigue	None	Slight	Moderate	Severe
3.	Boredom	None	Slight	Moderate	Severe
4.	Drowsiness	None	Slight	Moderate	Severe
5.	Headache	None	Slight	Moderate	Severe
6.	Eyestrain	None	Slight	Moderate	Severe
7.	Difficulty focusing	None	Slight	Moderate	Severe
8.	Salivation increase	None	Slight	Moderate	Severe
9.	Salivation decrease	None	Slight	Moderate	Severe
10.	Sweating	None	Slight	Moderate	Severe
11.	Nausea	None	Slight	Moderate	Severe
12.	Difficulty concentrating	None	Slight	Moderate	Severe

13. Mental depression	No	Yes (	Slight	Moderate	Severe)
14. "Fullness of the head"	No	Yes (	Slight	Moderate	Severe)
15. Blurred vision	No	Yes (	Slight	Moderate	Severe)
16. Dizziness eyes open	No	Yes (	Slight	Moderate	Severe)
17. Dizziness eyes close	No	Yes (	Slight	Moderate	Severe)
18. Vertigo	No	Yes (	Slight	Moderate	Severe)
19. Visual flashbacks	No	Yes (	Slight	Moderate	Severe)
20. Faintness	No	Yes (	Slight	Moderate	Severe)
21. Aware of breathing	No	Yes (	Slight	Moderate	Severe)
22. Stomach awareness	No	Yes (	Slight	Moderate	Severe)
23. Loss of appetite	No	Yes (	Slight	Moderate	Severe)
24. Increased appetite	No	Yes (	Slight	Moderate	Severe)
25. Desire to move bowels	No	Yes (	Slight	Moderate	Severe)
26. Confusion	No	Yes (	Slight	Moderate	Severe)
27. Burping	No	Yes (	Slight	Moderate	Severe)
28. Vomiting	No	Yes (	Slight	Moderate	Severe)
29. Other	No	Yes (	Slight	Moderate	Severe)

APPENDIX B

User Study Post-Questionnaire

Participant Number: _____

Please indicate what kind of navigation input you have been using during the experiement:

[ ] Gamepad [ ] Omnidirectional Treadmill

Please indicate how the virtual environment was presented to you:

[ ] All at once [ ] Incrementally over time

Post-Exposure Simulator Sickness

Please circle below again if any of the symptoms apply to you now.

1. General discomfort	None	Slight	Moderate	Severe
2. Fatigue	None	Slight	Moderate	Severe
3. Boredom	None	Slight	Moderate	Severe
4. Drowsiness	None	Slight	Moderate	Severe
5. Headache	None	Slight	Moderate	Severe
6. Eyestrain	None	Slight	Moderate	Severe
7. Difficulty focusing	None	Slight	Moderate	Severe
8. Salivation increase	None	Slight	Moderate	Severe
9. Salivation decrease	None	Slight	Moderate	Severe
10. Sweating	None	Slight	Moderate	Severe
11. Nausea	None	Slight	Moderate	Severe
12. Difficulty concentrating	None	Slight	Moderate	Severe
13. Mental depression	No	Yes (Slight	Moderate	Severe)

14. "Fullness of the head"	No	Yes (Slight Moderate Severe)
15. Blurred vision	No	Yes (Slight Moderate Severe)
16. Dizziness eyes open	No	Yes (Slight Moderate Severe)
17. Dizziness eyes close	No	Yes (Slight Moderate Severe)
18. Vertigo	No	Yes (Slight Moderate Severe)
19. Visual flashbacks	No	Yes (Slight Moderate Severe)
20. Faintness	No	Yes (Slight Moderate Severe)
21. Aware of breathing	No	Yes (Slight Moderate Severe)
22. Stomach awareness	No	Yes (Slight Moderate Severe)
23. Loss of appetite	No	Yes (Slight Moderate Severe)
24. Increased appetite	No	Yes (Slight Moderate Severe)
25. Desire to move bowels	No	Yes (Slight Moderate Severe)
26. Confusion	No	Yes (Slight Moderate Severe)
27. Burping	No	Yes (Slight Moderate Severe)
28. Vomiting	No	Yes (Slight Moderate Severe)
29. Other	No	Yes (Slight Moderate Severe)

Exploration & Orientation

1. Please specify how much you agree with the following statement: "I was able to EASILY find my way the environment?

Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
1	2	3	4	5

2. Please specify how much you agree with the following statement: "I was able to QUICKLY find my way through the environment?

Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
1	2	3	4	5

3. Please specify how much you agree with the following statement: "I was able to QUICKLY get an overview of the entire indoor environment?

Strongly disagree	Disagree	Neither agree nor disagree	Agree	Strongly agree
1	2	3	4	5

4. Please indicate how well you estimate your spatial understanding of the presented scene.

Very poor	Poor	Acceptable	Good	Very Good
1	2	3	4	5

5. Please rate how much you enjoyed using the device for navigation:

Not at all	Not much	Moderate	Much	Very much
1	2	3	4	5

6. Please rate how much you enjoyed the quality of the virtual environment:

Not at all	Not much	Moderate	Much	Very much
1	2	3	4	5

Task Load (TLX)

Please indicate your perceived amount of the following statements regarding the exploration task:

1. How mentally demanding was the task?

Very low	Low	Average	High	Very High
1	2	3	4	5

2. How physically demanding was the task?

Very low	Low	Average	High	Very High
1	2	3	4	5

3. How hurried or rushed was the pace of the task?

Very low	Low	Average	High	Very High
1	2	3	4	5

4. How successful were you in accomplishing what you were asked to do?

Very Poor	Poor	Acceptable	Good	Very Good
1	2	3	4	5

5. How hard did you work to accomplish your level of performance?

Very low	Low	Average	High	Very High
1	2	3	4	5

6. How insecure, discouraged, irritated, stressed and annoyed were you?

Very low	Low	Average	High	Very High
1	2	3	4	5

System Usability (SUS)

Please specify how much you agree with the following statement:

1. I think that I would like to use this concept frequently

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

2. I found the concept unnecessarily complex

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

3. I thought the concept was easy to use

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

4. I think that I would need the support of a technical person to be able to use this concept

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

5. I found the various functions in this concept were well integrated

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

6. I thought there was too much inconsistency in this concept

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

7. I would imagine that most people would learn to use this concept very quickly

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

8. I found the concept very cumbersome to use

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

9. I felt very confident using the concept

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

10. I needed to learn a lot of things before I could get going with the concept

Strongly disagree	Disagree	Neither agree or disagree	Agree	Strongly agree
1	2	3	4	5

Open Questions

Please describe in a few words your experience with the application.

Please describe in a few words your exploration strategy.

What other uses can you suggest for the concept? (please think of at least 3 uses)

List of Figures

List of Tables

Bibliography

[AFDM08] A. Angeli, D. Filliat, S. Doncieux, and J.-A. Meyer. Fast and Incremental Method for Loop-Closure Detection using Bags of Visual Words. *IEEE Transactions on Robotics*, 24:1027–1037, 2008.

[Asu16] Asus. Xtion Pro Live [Online]. *https://www.asus.com/3D-Sensor/Xtion_PRO_LIVE/*, Last accessed on February 10, 2016.

[Bjö96] A. Björck. *Numerical Methods for Least Squares Problems*. Siam, 1996.

[BKJP04] D. A. Bowman, E. Kruijff, LaViola J., and I. Poupyrev. *3D User Interfaces: Theory and Practice*. Addison-Wesley, 2004.

[BM92] P. J. Besl and N. D. McKay. A Method for Registration of 3-D Shapes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 14:239–256, 1992.

[Bro96] J. Brooke. Sus - a quick and dirty usability scale. *Usability Evaluation in Industry*, 189(194):4–7, 1996.

[BTG97] J. Bliss, P. D. Tidwell, and M. A. Guest. The Effectiveness of Virtual Reality for Administering Spatial Navigation Training to Firefighters. *Presence: Teleoperators and Virtual Environments*, 6:73–86, 1997.

[Cam16] S. Campisi. Hyper IMU (version 1.5) [Software]. *https://play.google.com/store/apps/details?id=com.ianovir.hyper_imu*, Last accessed on February 11, 2016.

[CBI13] J. Chen, D. Bautembach, and S. Izadi. Scalable Real-time Volumetric Surface Reconstruction. *ACM Transactions on Graphics (TOG)*, 32:113:1–113:16, 2013.

[CH14] T. Cakmak and H. Hager. Cyberith Virtualizer: A Locomotion Device for Virtual Reality. In *ACM SIGGRAPH Emerging Technologies*, pages 6:1–6:1, 2014.

[Cho16] M. Chourdakis. Ultimate Shared Memory: A Flexible Class for Interprocess Memory Sharing [Software]. *http://www.codeproject.com/Articles/835818/Ultimate-Shared-Memory-A-flexible-class-for-interp*, Last accessed on February 10, 2016.

[CL96] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. In *ACM Conference on Computer Graphics and Interactive Techniques*, pages 303–312, 1996.

[Cor16a] Microsoft Corporation. Microsoft Kinect [Online]. *https://www.microsoft.com/en-us/kinectforwindows*, Last accessed on February 10, 2016.

[Cor16b] Nvidia Corporation. Cuda (version 7.0) [software]. *https://developer.nvidia.com/cuda-toolkit*, Last accessed on February 10, 2016.

[CTF14] M. Coatsworth, J. Tran, and A. Ferworn. A Hybrid Lossless and Lossy Compression Scheme for Streaming RGB-D Data in Real-Time. In *IEEE International Symposium on Safety, Security, and Rescue Robotics*, pages 1–6, 2014.

[CZK15] S. Choi, Q.-Y. Zhou, and V. Koltun. Robust Reconstruction of Indoor Scenes. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5556–5565, 2015.

[Deu96] P. Deutsch. DEFLATE Compressed Data Format Specification Version 1.3. Technical report, RFC 1951, 1996.

[DG96] P. Deutsch and J.-L. Gailly. Zlib Compressed Data Format Specification Version 3.3. Technical report, RFC 1950, 1996.

[Epi16] Epic Games. Unreal Engine 4.9 [Software]. *https://www.unrealengine.com*, Last accessed on February 10, 2016.

[FB81] 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*, 24:381–395, 1981.

[FCSS09] Y. Furukawa, B. Curless, S. M. Seitz, and R. Szeliski. Reconstructing Building Interiors from Images. In *IEEE International Conference on Computer Vision (ICCV)*, pages 80–87, 2009.

[For02] B. A. Forouzan. *TCP/IP Protocol Suite*. McGraw-Hill, Inc., 2002.

[Fre16] FreeGLUT. Free OpenGL Utility Toolkit (version 3.0) [Software]. *http://freeglut.sourceforge.net/*, Last accessed on February 10, 2016.

[FTF+15] N. Fioraio, J. Taylor, A. Fitzgibbon, L. Di Stefano, and S. Izadi. Large-Scale and Drift-Free Surface Reconstruction Using Online Subvolume Registration. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4475–4483, 2015.

[FVDF+94] J. D. Foley, A. Van Dam, S. K. Feiner, J. F. Hughes, and R. L. Phillips. *Introduction to Computer Graphics*. Addison-Wesley Reading, 1994.

[GG07] Y. Gu and R. L. Grossman. UDT: UDP-Based Data Transfer for High-Speed Wide Area Networks. *Computer Networks*, 51:1777–1799, 2007.

[GK15] T. Golla and R. Klein. Real-time point cloud compression. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 5087–5092, 2015.

[HF16] F. Heredia and R. Favier. Kinfu Large Scale [Online], Last accessed on February 10, 2016.

[Hig16] High Tech Computer Corporation / Valve Corporation. HTC Vive [Online]. *http://www.htcvr.com*, Last accessed on February 10, 2016.

[HKH+12] P. Henry, M. Krainin, E. Herbst, X. Ren, and D. Fox. RGB-D Mapping: Using Kinect-Style Depth Cameras for Dense 3D modeling of Environments. *The International Journal of Robotics Research*, 31:647–663, 2012.

[HS88] S. Hart and L. Staveland. Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research. *Advances Psychology*, 52:139–183, 1988.

[Inf16] Infinadeck. Infinadeck [Online]. *http://infinadeck.com/*, Last accessed on February 10, 2016.

[KBH06] M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson Surface Reconstruction. In *Eurographics Symposium on Geometry Processing*, pages 61–70, 2006.

[KBR+12] J. Kammerl, N. Blodow, R. B. Rusu, S. Gedikli, M. Beetz, and E. Steinbach. Real-Time Compression of Point Cloud Streams. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 778–785, 2012.

[KPR+15] O. Kähler, V. A. Prisacariu, C. Y. Ren, X. Sun, P. Torr, and D. Murray. Very High Frame Rate Volumetric Integration of Depth Images on Mobile Devices. *IEEE Transactions on Visualization and Computer Graphics*, 21:1241–1250, 2015.

[LC87] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *SIGGRAPH Comput. Graph.*, 21:163–169, 1987.

[LM13] M. Labbe and F. Michaud. Appearance-Based Loop Closure Detection for Online Large-Scale and Long-Term Operation. *IEEE Transactions on Robotics*, 29:734–745, 2013.

[MLDH15] A. Maglo, G. Lavoué, F. Dupont, and C. Hudelot. 3d mesh compression: Survey, comparisons, and emerging trends. *ACM Computing Surveys*, 47:44:1–44:41, 2015.

[MOCGR14] V. Morell, S. Orts, M. Cazorla, and J. Garcia-Rodriguez. Geometric 3D Point Cloud Compression. *Pattern Recognition Letters*, 50:55–62, 2014.

[Moo76] G. Moore. *Environmental Knowing*, chapter Theory and Research on the Development of Environmental Knowing, pages 138–164. "Dowden, Hutchinson and Ross", 1976.

[MRB09] Z. C. Marton, R. B. Rusu, and M. Beetz. On Fast Surface Reconstruction Methods for Large and Noisy Datasets. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 3218–3223, 2009.

[MS92] M. E. McCauley and T. J. Sharkey. Cybersickness: Perception of Self-Motion in Virtual Environments. *Presence: Teleoperators & Virtual Environments*, 1:311–318, 1992.

[MWB+13] L. Ma, T. Whelan, E. Bondarev, P. HN de With, and J. McDonald. Planar Simplification and Texturing of Dense Point Cloud Maps. In *IEEE European Conference on Mobile Robots (ECMR)*, pages 164–171, 2013.

[NIH+11] R. A. Newcombe, S. Izadi, O. Hilliges, D. Molyneaux, D. Kim, A. J. Davison, P. Kohli, J. Shotton, S. Hodges, and A. Fitzgibbon. KinectFusion: Real-Time Dense Surface Mapping and Tracking. In *IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 127–136, 2011.

[NIL12] C. V. Nguyen, S. Izadi, and D. Lovell. Modeling Kinect Sensor Noise for Improved 3d Reconstruction and Tracking. In *IEEE International Conference on 3D Imaging, Modeling, Processing, Visualization and Transmission*, pages 524–530, 2012.

[NM12] Nagesha and S. S. Manvi. Performance Analysis of SCTP Compared to TCP and UDP. In *Advances in Computing and Information Technology*. Springer Berlin Heidelberg, 2012.

[NRI12] A. Nybakke, R. Ramakrishnan, and V. Interrante. From Virtual to Actual Mobility: Assessing the Benefits of Active Locomotion through an Immersive Virtual Evironment using a Motorized Wheelchair. In *IEEE Symposium on 3D User Interfaces (3DUI)*, pages 27–30, 2012.

[NSS14] F. Nenci, L. Spinello, and C. Stachniss. Effective Compression of Range Data Streams for Remote Robot Operations using H.264. In *IEEE International Conference on Intelligent Robots and Systems (IROS)*, pages 3794–3799, 2014.

[NY06] T. S. Newman and H. Yi. A Survey of the Marching Cubes Algorithm. *Computers & Graphics*, 30:854–879, 2006.

[NZIS13] M. Nießner, M. Zollhöfer, S. Izadi, and M. Stamminger. Real-time 3D Reconstruction at Scale using Voxel Hashing. *ACM Transactions on Graphics (TOG)*, 32:169:1–169:11, 2013.

[Occ16] PrimeSense / Occipital. OpenNI (version 2.2) [Software]. *http://structure.io/openni*, Last accessed on February 10, 2016.

[Ocu16a] Oculus VR. Oculus Rift [Online]. *https://www.oculus.com/en-us/rift*, Last accessed on February 10, 2016.

[Ocu16b] Oculus VR. Oculus Rift Requirements [Online]. *https://www.oculus.com/en-us/blog/powering-the-rift/*, Last accessed on February 10, 2016.

[Pas92] R. Passini. *Wayfinding in architecture*. Environmental design series. Van Nostrand Reinhold, 1992.

[PKC+14] V. A. Prisacariu, O. Kähler, M.-M. Ren C. Y. Cheng, J. Valentin, P. Torr, I. Reid, and D. Murray. A Framework for the Volumetric Integration of Depth Images. *Computing Research Repository (CoRR)*, abs/1410.0925, 2014.

[PPS13] P. M Panchal, S. R. Panchal, and S. K. Shah. A Comparison of SIFT and SURF. *International Journal of Innovative Research in Computer and Communication Engineering*, 1:323–327, 2013.

[RB04] B. Riecke and H. Bülthoff. Spatial Updating in Real and Virtual Environments: Contribution and Interaction of Visual and Vestibular Cues. In *ACM Symposium on Applied Perception in Graphics and Visualization*, pages 9–17, 2004.

[RL01] S. Rusinkiewicz and M. Levoy. Efficient Variants of the ICP Algorithm. In *IEEE International Conference on 3-D Digital Imaging and Modeling (3DIM)*, pages 145–152, 2001.

[RL09] R. A. Ruddle and S. Lessels. The Benefits of using a Walking Interface to Navigate Virtual Environments. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 16:5:1–5:18, 2009.

[RM12] H. Roth and V. Marsette. Moving Volume KinectFusion. In *Proceedings of the British Machine Vision Conference (BMVC)*, pages 1–11, 2012.

[Sal04] D. Salomon. *Data Compression: The Complete Reference*. Springer Science & Business Media, 2004.

[Sie81] A. W. Siegel. The Externalization of Cognitive Maps by Children and Adults: In Search of Ways to Ask Better Questions. In *Spatial Representation and Behavior Across the Life Span*. Academic Press, 1981.

[SKCS13] F. Steinbrücker, C. Kerl, D. Cremers, and J. Sturm. Large-Scale Multi-Resolution Surface Reconstruction from RGB-D Sequences. In *IEEE International Conference on Computer Vision (ICCV)*, pages 3264–3271, 2013.

[SM01] R. Stewart and C. Metz. SCTP: New Transport Protocol for TCP/IP. *IEEE Internet Computing*, 5:64–69, 2001.

[Son16] Sony. Playstation VR [Online]. *https://www.playstation.com/en-us/explore/playstation-vr/*, Last accessed on February 10, 2016.

[SSC11] F. Steinbrücker, J. Sturm, and D. Cremers. Real-time Visual Odometry from Dense RGB-D Images. In *IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 719–722, 2011.

[SW75] A. W. Siegel and S. H. White. The Development of Spatial Representations of Large-Scale Environments. *Advances in Child Development and Behavior*, 10:9–55, 1975.

[TL08] S. "Thrun and J." Leonard. *"Springer Handbook of Robotics"*, chapter "Simultaneous Localization and Mapping", pages "871–889". "Springer Berlin Heidelberg", "2008".

[Uni16] Unity Technologies. Unity 5 [Software]. *https://unity3d.com*, Last accessed on February 10, 2016.

[Vir16] Virtuix. Virtuix Omni [Online]. *http://www.virtuix.com*, Last accessed on February 10, 2016.

[Wal99] D. A. Waller. *An Assessment of Individual Differences in Spatial Knowledge of Real and Virtual Environments*. PhD thesis, University of Washington, 1999.

[WJK+13] T. Whelan, H. Johannsson, M. Kaess, J. Leonard, and J. McDonald. Robust Real-Time Visual Odometry for Dense RGB-D Mapping. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 5724–5731, 2013.

[WKJ+14] T. Whelan, M. Kaess, H. Johannsson, M. Fallon, J. J. Leonard, and J. McDonald. Real-Time Large-Scale Dense RGB-D SLAM with Volumetric Fusion. *The International Journal of Robotics Research*, 34:598–626, 2014.

[WMB+15] T. Whelan, L. Ma, E. Bondarev, P. de With, and J. McDonald. Incremental and Batch Planar Simplification of Dense Point Cloud Maps. *Robotics and Autonomous Systems*, 69:3–14, 2015.

[WMG+12] A. Wendel, M. Maurer, G. Graber, T. Pock, and H. Bischof. Dense Reconstruction on-the-fly. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1450–1457, 2012.

[WS98] B. Witmer and M. Singer. Measuring Presence in Virtual Environments: A Presence Questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998.

[ZK13] Q.-Y. Zhou and V. Koltun. "Dense Scene Reconstruction with Points of Interest". *ACM Transactions on Graphics (TOG)*, 32:112, 2013.

[ZZZL13] M. Zeng, F. Zhao, J. Zheng, and X. Liu. Octree-Based Fusion for Realtime 3D Reconstruction. *Graphical Models*, 75:126–136, 2013.