Fachbereich 4: Informatik

Mixed Reality Embodiment Platform – Visual coherence for volumetric mixed reality scenes

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Katrin Meng

Mixed Reality Embodiment Platform – Recording and replay of volumetric characters

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Arne Reepen

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Prof. Dr. Holger Regenbrecht
(Department of Information Science, Human Computer Interaction Group)

Koblenz, im September 2016

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

....

(Ort, Datum) (Unterschrift)

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Obwohl Virtual Reality bereits seit Jahrzehnten existiert, hat es in den letzten Jahren erneut an Bedeutung gewonnen. Die Veröffentlichungen der ersten Geräte für den Endverbraucher ermöglichen zum ersten Mal immersive und bezahlbare VR für Privatpersonen. Diese Entwicklungen haben zu einem erneutem Fokus der Forschung auf technische Herausforderungen und psychologische Effekte geführt. Die Konzepte von Presence, dem Gefühl an einem virtuellen Ort zu sein, body ownership und ihr Einfluss sind seit langem zentrale Themen der Forschung und dennoch noch immer nicht vollständig verstanden.
Um weitere Forschung im Bereich von Mixed Reality zu ermöglichen, wollen wir ein Framework vorstellen, das sowohl den Körper als auch die Umgebung eines Nutzers in eine kohärent dargestellte, virtuelle Umgebung integriert. Als einen der Hauptaspekte wollen wir reale und virtuelle Objekte so in dieser gemeinsamen Umgebung zusammenbringen, dass sie visuell nicht mehr unterscheidbar sind. Um dies zu erreichen, soll der Fokus nicht auf einer hohen grafischen Qualität liegen, sondern vielmehr auf einer vereinfachten Darstellung der Realität. Die grundlegende Frage ist, welcher Grad von visuellem Realismus nötig ist, um eine glaubhafte Mixed Reality Umgebung und das Gefühl von Presence für den Nutzer zu erzielen? Der zweite Aspekt betrachtet die Integration von virtuellen Personen. Können reale Personen auf eine Art und Weise aufgenommen und wiederabgespielt werden, sodass sie als glaubhafte Bestandteile der Welt, und damit der Umgebung des Nutzers, wahrgenommen werden?
Das Ziel dieser Arbeit war die Entwicklung eines neuen Frameworks, der sogenannten Mixed Reality Embodiment Platform. Dieses initiale System bietet grundlegende Funktionalitäten, die als Basis für zukünftige Erweiterungen genutzt werden können. Wir stellen außerdem eine erste Anwendung vor, die Studien zur Evaluierung des Systems ermöglicht und zur Beantwortung der zuvorgenannten Forschungsfragen beitragen soll.

Abstract

While Virtual Reality has been around for decades it gained new life in recent years. The release of the first consumer hardware devices allows fully immersive and affordable VR for the user at home. This availability lead to a new focus of research on technical problems as well as psychological effects. The concepts of presence, describing the feeling of being in the virtual place, body ownership and their impact are central topics in research for a long time and still not fully understood.

To enable further research in the area of Mixed Reality, we want to introduce a framework that integrates the users body and surroundings inside a visual coherent virtual environment. As one of two main aspects we want to merge real and virtual objects to a shared environment in a way such that they are no longer visually distinguishable. To achieve this the main focus is not supposed to be on a high graphical fidelity but on a simplified representation of reality. The essential question is, what level of visual realism is necessary to create a believable mixed reality environment that induces a sense of presence in the user? The second aspect considers the integration of virtual persons. Can characters be recorded and replayed in a way such that they are perceived as believable entities of the world and therefore act as a part of the users environment?

The purpose of this thesis was the development of a framework called Mixed Reality Embodiment Platform. This inital system implements fundamental functionalities to be used as a basis for future extensions to the framework. We also provide a first application that enables user studies to evaluate the framework and contribute to aforementioned research questions.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Goals 3

2 Related Work **5**
2.1 Presence and Ownership in VR 5
2.2 Applications 5
2.3 Visual coherence in MR 5
2.4 Embodiment in a VE 6

3 System **9**
3.1 Concept 9
3.2 Requirements and Constraints 10
3.3 Head Mounted Displays 11
3.4 Depth Sensors 12
3.5 Hardware 14
3.6 Setup 15
3.7 Software Infrastructure 16

4 Mixed Reality Embodiment Platform **17**
4.1 Capturing 19
4.2 Rendering 19
4.3 Networking 21
4.4 Voxelspace 24
4.5 Coordinate systems 25
4.5.1 Kinect v2 25
4.5.2 Oculus Rift 28

5 Scene Realism **30**
5.1 Scene Representation 31
5.1.1 Voxel representation 31
5.1.2 Lighting 33
5.1.3 Shadows 34
5.2 Virtual Objects 35
5.2.1 3D-Modelling 37
5.2.2 Voxelization 38
5.2.3 Culling 38
5.2.4 Ray casting 40
5.2.5 Noise 41
5.2.6 Summary 43

6 Recording and Replay **46**
6.1 Recording . . . 46
6.2 Replay . . . 48
6.3 Bone Structure . . . 48
6.4 Bounding Volumes . . . 50
6.5 Separate Body Parts . . . 51
6.6 Summary . . . 52

7 First Application **54**
7.1 Design . . . 54
7.2 Photo Booth Application . . . 55
7.3 Observations . . . 59
7.3.1 Platform Experience . . . 59
7.3.2 Virtual Objects . . . 60
7.3.3 Recorded Characters . . . 60
7.4 Technical Results . . . 61
7.4.1 GPU Utilization . . . 61
7.4.2 CPU and Memory Utilization . . . 61
7.4.3 Network . . . 62
7.4.4 Scene Details . . . 62

8 Conclusion and Future Work **64**
8.1 Future Work . . . 64
8.1.1 Platform Setup . . . 64
8.1.2 Virtual Objects . . . 64
8.1.3 Recorded Characters . . . 65
8.1.4 Study . . . 65
8.2 Conclusion . . . 65

Appendices **67**

A MREP Instructions **67**

B PhotoBooth Instructions **73**

C Pictures **78**

1 Introduction

1.1 Motivation

We live in a world that is more and more influenced and defined by the use of digital content. A lot of Human Computer Interfaces are integrated in our everyday life and play an substantial role in all different parts of the modern world. For example communication and collaboration, entertainment as well as medical applications and production environments. In general the way we access and experience all kind of information can be improved by the use of new devices and interfaces. One approach is to blend digital information into our normal environments to achieve a natural and easy access and interaction. Today Augmented Reality becomes a widely known term describing interfaces that enhance an image of the real world with virtual objects and information. A part of Augmented Reality is to display the virtual contents as if they are a part of the real environment. Another approach is to fully surround the user by computer generated environments. This is achieved by the use of technologies like Immersive Projection Technology (IPT) or Head Mounted Displays (HMD). The goal is to suppress the perception of the real environment and replace it by the virtual one. Although Virtual Reality exists for a long time, in recent years it experienced a big push. The release of the Oculus Rift Development Kit 1 made affordable and public available Virtual Reality a possibility and sparked new interest for consumers and developers alike. This year the consumer versions of the Oculus Rift and HTC Vive have been released. In combination with more powerful hardware these devices allow for fully immersive Virtual Reality systems in home environments but are also capable to be used in professional applications. But this also creates new challenges as there are still many effects and problems that are not fully understood yet. There is still the need for research regarding the technical as well as the psychological aspect. Similar to Augmented Reality where virtual objects are merged into a real environment, it is possible to integrate real objects in virtual environments. To characterize these systems Milgram et al. defined a "virtuality continuum"[MK94]. In Figure 1 the different systems along the continuum are shown specified by the level of used virtual and real components. On the one extreme there is a real environment without virtual components as can viewed directly or captured by a camera and shown on a display. On the other hand there is a complete virtual environment without any real components. In between are any possible variation of combinations. In Augmented Reality systems the primary world is real enhanced with virtual objects. If the primary world is virtual and some objects are real the system is characterized as Augmented Virtuality. Milgram already stated that at some point one can not distinguish whether the experienced environment is mainly virtual or real. For this cases the overall

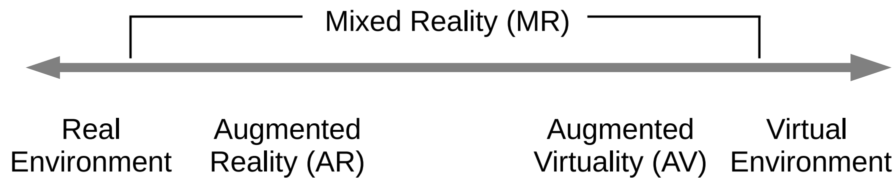


Figure 1: Virtuality continuum [MK94]

term Mixed Reality (MR) is used. Keeping that in mind Mixed Reality is a particular subset of Virtual Reality, where real and virtual objects are presented together in a single display.
One of the problems in MR is to achieve a visual coherence of the real and virtual components, so that it is not longer possible to distinguish them. Most AR systems want to achieve this and there are two different approaches to this problem. First you can try to render the virtual objects as photorealistic as possible and aim the same graphical quality as of the image of the real world. In contrast to that there is the possibility to decrease the graphical fidelity of the scene and step back from the visual realism. Therefor both graphical qualities are changed and rendered in a coherent but unrealistic way. To achieve this non-photorealistic rendering techniques (NPR) are used. It is shown that using this techniques real and virtual objects are significantly more difficult to distinguish.
In addition to these technical problems there are important psychological effects of MR and VR to keep in mind, that have great influence on the user and his experience. In a fully virtual environment we can observe people behaving as they would physically be in this environment. They will duck if a virtual object is thrown at them or they refuse to step over a virtual cliff. This sense of being there is called presence. Presence is a central concept in the research and development of VR systems. It is important to know what characteristics of an environment and the way of representing objects and the user himself are influencing these effects and how. Understanding this may allow the increased and successful use of virtual and mixed environments for medical and therapeutic applications and help to get insights of the human perception and internal models of his environment and himself.
When using VEs for psychotherapeutic purposes the sense of presence is crucial for the possible success. There are Virtual Reality systems that are already used by therapists in virtual reality exposure therapy (VRET) to simulate and replace in vivo exposure. In contrary to the normal in vivo exposure, VR exposure can be offered anywhere in safe and controlled en-

vironments. They can be more affordable for example when used to treat the fear of flying and studies have shown an equivalent success [RHS+00]. Additionally there are therapeutic and medical application relying heavily on a embodiment of the user. This embodiment is needed to enable a manipulation of the body image of the user for example in the treatment of eating disorders [Riv11] or applications for motor rehabilitation [RHM+12].
Being able to present the user an appropriate embodiment is a common problem in Virtual Reality. The virtual body needs to move synchronously to the body of the user to induce a sense of ownership over the virtual body. Hence one need an appropriate method to map the motion, if the user is represented by an 3D model. This would require a method of motion capturing. One problem with virtual models of persons is the uncanny valley. This effect describes the reception of human models. These can become unpleasant or even creepy with a certain level of realism. Both motion capturing and the uncanny valley effect can be avoided by capturing and displaying the real user as 3D data. The user can be captured with a RGB-D sensor and be displayed as a point cloud per frame rather than a 3D Model with transformations. The question is what level of realism and graphic fidelity is needed to achieve the feeling of presence and ownership over this displayed body.

1.2 Goals

In this thesis we want to introduce a new hardware and software framework, the Mixed Reality Embodiment Platform, to merge virtual and real objects to one indistinguishable environment. Therefore we need a coherent visualization of the objects from both worlds. We want to achieve this by reducing the visual realism rather than aim for a high graphical fidelity. The other main goal is to introduce recordings of human beings as a believable part of the users environment. The embodiment of the user himself is a crucial part of the platform as the user should experience the environment in an egocentric and immersive way. He should be able to look down seeing a representation of his own body moving corresponding to his real body.
The framework should work as an enabler for following user studies regarding the perceiving of environments as well as presence in a spatial and social way.
In the long term we want to provide a system that can be used to answer following research questions:

- Which level of visual realism is needed for
  - the user to accept all objects, real and virtual, as a part of his environment?
  - the interaction with real and virtual objects?
  - the user to achieve a sense of presence (spatial and social) in this environment?
- Is it possible to add virtual/captured persons in this environment?
  - Can they be perceived as entities that inhabit the virtual environment?
  - Can they even be perceived as potential interaction partners?

To achieve this there are some necessary requirements for the framework. First of all we need to meet real time characteristics with no recognizable latency. This is an important precondition for body ownership and presence as well as a high frame rate. The representation of the users body, in a way that resembles the body features of the user has to be considered for this. Will the user recognize his own body features in the virtual representation like height, shape, skin color, hair and clothing. The face of a human being is a very import feature, but difficult to implement in a system that uses a HMD. So the main parts we want to implement are the representation of the virtual body moving corresponding to your real body (to achieve agency and ownership), representation of the real environment, possibility to add virtual objects in the environment, representing the virtual objects in a visual coherent way so they become indistinguishable from the real world objects, the possibility to add virtual persons (play recordings), scanning of a person to get a model or a clip and use skeleton tracking for the differentiation of body parts
We want to implement this functionalities in a framework and design a first application to enable research in the aforementioned topics.

2 Related Work

2.1 Presence and Ownership in VR

In immersive VR the term immersion is a technical term defined by objective and measurable factors. For example latency, the presented FOV and the frame rate of a system. Immersion can then lead to the psychological effect of presence which is the central concept of experiencing virtual environments. The sense of presence enables the use of VR in different applications for training, the treatment of phobias or pain distraction [SVS05]. There exist different definitions for presence and influencing factors are extracted to provide a systematic way to measure the presence in virtual environments [WS98] [SFR01]. Another effect in VR is the ownership over a virtual body. The original rubber hand illusion experiment was executed in a real world environment by Botvinick. They induced the sense of ownership over a rubber hand by synchronous visuotactile stimulation [BC98]. This rubber hand experiment was first recreated do work with synchronous movement [KE14] and later brought to VR with the same result of ownership. Kokkinara et al stated that synchronous visuomotor and visuotactile stimuli both work to induce ownership [KS14].

2.2 Applications

Mixed reality systems are used in a wide range of therapeutic and medical applications. They are used as treatment programs in psychotherapy. For this purpose it is essential to induce a sense of presence for the user as described before. If the user is not present he is not affected by the experience in the virtual environment and therefore no effects can be achieved. Studies showed successful treatment of phobias like acrophobia, the fear of heights equally to common in vivo exposure therapy [EKH+02]. VR and MR applications replace and complement mirror therapy systems, as the show the same effects with more flexibility. Regenbrecht et al introduced Augmented Reflection [RFM+11][RHM+12] for motor-rehabilitation for example to treat stroke patients. But mirror boxes are restricted in spatial dimensions and view directions so the implementation of MR systems using HMD can provide more flexibility. This could be used for another mirror based approach for the treatment of phantom limb pain [RRR96].

2.3 Visual coherence in MR

There are two approaches to accomplish visual coherence of virtual objects and real objects in AR systems. The first and commonly used is to improve the realism of the virtual objects. But to get convincing lighting for the virtual objects is still a big issue. Even with very advanced rendering techniques it is hard to adapt in real time to the light conditions of

Figure 2: Results of the stylized augmented reality approach. Conventional rendering, "cartoon-like" style, "sketch-like" style [FBS05]

the real world. Hence the virtual objects look wrong and artificial compared to real objects. So in contrast to the majority of AR systems which try to achieve a photorealistic rendering of virtual objects in the captured real environment some tried to go another way. They used non-photorealistic rendering (NPR) techniques to connect both worlds visually. This way it is possible to adjust both graphical qualities to a common level. Fischer et al [FBS05] used a "cartoon-like" style and a "sketch-like" style for rendering. A filter is applied to the camera image of the real world and the virtual objects are rendered with a NPR technique to match the filter output. After that they are combined and the resulting augmented reality image appears much more coherent than generated with a conventional AR approach. See Figure 2. In a psychophysical study Fischer et al [FCB+06] showed later that it is significantly more difficult for users to distinguish virtual and real objects in a stylized augmented reality system. All of that work used video-see-through AR. Another approach for combining the two components in one AR system is introduced by Klein et al [KM10]. They concentrate on the coherent quality and texture of the rendering of the virtual contents and the image of the real world. Therefore they consider all characteristics of the capturing sensor and try to simulate them in their rendering.

2.4 Embodiment in a VE

There are different ways to present the user a fully immersive virtual environment resulting in different handling of embodiment. In Immersive Projection Technology (IPT) setups, like 3D workbench [KBF+95] and CAVE systems [CNSD93], the user is naturally able to see his own body as his view is not blocked by a device. When the virtual environment is displayed via a non-see-through head mounted display (HMD) the user is blocked from the real environment including his own body. As discussed in the previous sections presence in VEs relies on the possibilities to interact and navigate in this environment. This demands a representation of the users body.

Following different approaches to achieve body representations in a vir-

Figure 3: from left to right: a person in the real world environment of the user, the segmented person rendered in the virtual environment, use of different rendering techniques to achieve a hologram effect [SKB11]

tual environment using non-see-through HMDs are introduced. First as known from virtual environments in computer games a generic virtual avatar based on a 3D model can be used. Generally this will not match the visual properties of the actual user. Skin color, gender or the specific clothing is not part of the 3D model unless it get modified individually. As this model needs to align with the users body and represent it in a spatial sense some tracking method is needed. 2. Using a 3D avatar that is created by previous scanning of the users. Several implementation exist to scan a users and construct a 3D model. Either with a single sensor [LVG+13] or using multiple sensors as in [TZL+12]. Most of these approached do not automatically create a full rigged character that can be directly used as a transformable avatar. Instead they create high quality, but static 3D avatars. Automatic rigging is done in [FSR+13]. 3. Generating the avatar from the live users body. Can be done in different ways with multiple cameras in a silhouette based approach [LKA+04]. Instead of creating a full 3d rig, different approaches exist to capture the users body and displaying it in the virtual environments. Some are based on 2D overlay of a video stream and using background subtraction to only display the user for example. [BSRH09]. More advanced techniques employ depth sensors and create a 3D representation of the users hands. The fully immersive Mixed Reality system introduced by Tecchia et al [TAB+14] captures and generates a texturized mesh of the users hands and body. They mounted the sensor at the HMD and focus on the users hands and body. This makes an external view impossible as it would be needed for collaborative systems where the bodies of the different users should be fully captured and displayed. The capturing and integration of real persons in virtual environments is described by Suma et al. [SKB11]. They also mounted a depth sensor to the front of a head mounted display to allow the capturing of persons in the real environment of the user. The person is then segmented from the background and rendered as a textured 3D point cloud. The process is seen in Figure 3. Although the person could be presented in the virtual environment it does not match in the visual appearance and can easily perceived as not belong-

ing to this environment. By using a different rendering technique they tried to achieve a "hologram "effect to make the existence of other person more believable. But they were not able to show the body of the user himself due to the range of the used depth sensor. Depth sensor mounted externally to capture the users body completely, but from a fixed position. To allow a reconstruction that offers information from all side multiple sensors are needed.

3 System

In this chapter we will give you on overview of the intended system as well as the hardware and software components we used for the MREP project. We will name a few options and state the reasoning for the decisions we made.
First we will give an overview of the system concept in 3.1. In 3.2 some requirements and constraints will be named, which result from the Goals in 1.2. We continue with a discussion of head mounted displays in 3.3 and depth sensors in 3.4. The computer hardware we used for the system is stated in 3.5. An introduction to the physical setup we chose to combine all the components is described in 3.6. The section will finish with a short discussion of the used software infrastructure in 3.7.

3.1 Concept

First we want to summarize what we want to implement. The MREP is supposed to be a basic framework for Mixed Reality embodiment studies. A user is immersed in an mixed environment in which he can freely look and move around. His real world environment is captured and reintroduced into his virtual environment. One important aspect is, that he shall be able to see his own body and might get a sense of ownership over this virtual representation.
Virtual Objects, that are not actually in the real environment, shall be visible to the user in the virtual environment. Similar to this, virtual characters, that have been recorded previously are also part of the virtual environment. The important aspect is that the real and virtual objects shall be indistinguishable from each other.
To achieve this we want to capture the real world and the user with a RGBD sensor to get spatial information about the real world. This should be integrated as 3 dimensional data into the virtual environment. Because of this 3D data a lot of problems found in typical Mixed Reality applications, which relay on video see through, do not apply for our intended system. If virtual objects are supposed to be integrated into a real environment, they have to show the same behavior as real objects. These are effects like the occlusion between object or light sources causing shadows. Due to the integration of all scene elements, real or virtual, into one common rendering system, effects like occlusion, lighting or shadows can be naturally handled by the rendering system.
The graphical representation of the system is not focused on photorealistic rendering as former work has shown that visual realism might be unimportant at some point to induce a sense of presence. We want to find the minimal necessary level of realism that allows the feelings of presence and embodiment while obtaining a coherent visualization. To achieve this we

want to build the platform around the concept of simplification of the real world's 3 dimensional data in analogy to reducing pixel resolution in 2D images. While in 2D the representation unit is a pixel the matching concept in 3D is a voxel. So to reduce the data we define a discrete space for our real world the voxel space. The real world then represented as voxel data and can be captured and shown in different resolutions.

3.2 Requirements and Constraints

The concept described in 3.1 results in some requirements and constraints, which will be discussed in the following.
Like most virtual reality systems, real time is a very important constraint. The users head movement has to be resembled with very low latency, to avoid simulator sickness and increase immersion. For a typical VE, the current HMDs and the specific software handle this very well. As long as a constant frame rate of 90 frames per second is achieved, the VR experience is not straining for most people. Smaller dips in frame rate or tracking imprecisions can be compensated by techniques like Asynchronous Timewarp (ATW), if a frame is missed, meaning it is not rendered fast enough to be displayed. ATW will warp the last rendered image to the new head position based on the tracking data. This gives the impression of smooth head movement even if the scene is not updated fast enough. But it is important to know, that ATW is only capable of eliminating smaller drops in frame rate [VRa].
Relative to complete virtual environments complexity is added in our system by reintroducing components of the real environment into the virtual environment. Especially the representation of the users body has to move synchronously with the real body, to allow the user a sense of body ownership over the virtual body[KE14].
Requirements also exist for the size of the interaction space. It is restricted by the area covered by the depth and tracking camera, but also must allow for a decent range of movement for the user. We want him to be able to take a few steps in each direction. The interaction space must also be big enough to accommodate multiple virtual persons and objects. The virtual characters shall be perceived as plausible parts of the users environment. A virtual component placed outside the interaction space, can not naturally be part of the users real environment. As we want to provoke interactions between the live users and virtual characters and objects, the user should be able to reach the virtual persons and objects.

	Oculus Rift DK2	Oculus Rift CV1	HTC Vive
Resolution	1920 x 1080	2160 x 1200	2160 x 1200
Refreshrate	75 Hz	90 Hz	90 Hz
Tracking Range	0.4m - 2.5m	0.4 - 2.5m	5m x 5m
Tracking FoV	54°x 74°	100°x 70°	120°
Tracked Controllers	No	Yes	Yes
Release	July 2014	March 2016	April 2016

Table 1: Comparison of different HMD specifications

3.3 Head Mounted Displays

There a two major ways to immerse a users into a virtual world. These are namely Head Mounted Displays (HMD) and Immersive Projection Technology (IPT). These can be responsive workbenches [KBF+95] or a Cave Automatic Virtual Environment(CAVE) [CNSD93]. A HMD is a device that enables the user to view computer generated images by placing displays directly in front of the users eyes. The head movement is tracked to show a version of the virtual world from his point of view.
Another way to present a VR experience to a user would be a CAVE setup. In a CAVE the user is located inside a number of screens on which the virtual environment is projected. Depending on the specific setup the user is completely surrounded by projections. Again the users head movement is tracked and the projections are rendered accordingly. An advantage is, that the user can naturally see his own body. While a CAVE is a good system to immerse a user in a virtual environment it has several major drawbacks for our intended system. We want to alter the appearance of the users body, but in a CAVE it is hard to remove components of the real world from the users view. This also applies to objects present in the real environment. A CAVE also requires more specialized equipment, is more expensive and harder to setup.
Overall a HMD is more affordable and allows a more flexible system. Consequently it is the better choice for our intended system.
The MREP system is intended as a desktop based and stationary system. While mobile focused VR headsets like the Gear VR offer untangled VR experiences on mobile devices, they lack the positional tracking capabilities. In addition the computational performance of current smartphones is not comparable with a desktop system. In 2016 several new desktop based HMDs have been released and offer high quality VR hardware at an affordable price. These are for the first time available to a consumer audience.
A comparison of the three most common VR HMDs is shown in 1. The Oculus CV1 and HTC Vive are clearly superior to the DK2 in every aspect. Especially the resolution and refresh rate offer a better VR experience. The larger area in which the HTC Vive can track the HMD, would allow for a

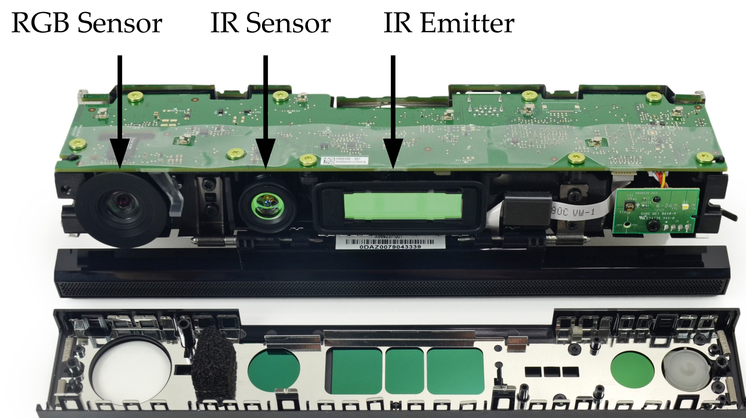


Figure 4: Inside of the Kinect v2 Sensor (adapted from [IFI])

bigger interaction space. The support of tracked controllers could be a useful addition for an interactive system. Unfortunately both were not available at the start of development. Thus we started with the Oculus Rift DK2, as we already had the hardware at hand and experience using it. It offers decent hardware, good availability and wide spread support in different existing frameworks. We used it for the major part of the project, with the Consumer Version just being released as we conclude our work. The results of the transition to the CV1 are mentioned in 7.3.

3.4 Depth Sensors

As stated in 3.2, we want to capture the user and his environment. The captured data shall be used to reconstruct and reintroduce persons and objects into the virtual environment as 3D data and not just 2D image. To capture the geometry of a real world scene depth sensors are used. This spatial data can be combined with the color information of a RGB sensor to get a colored 3D point cloud of the real world.
Although there are several depth sensing cameras available, the Microsoft Kinect camera features the most advanced full body skeleton tracking. Skeleton information will be crucial for determining body parts which is needed for manipulating and changing the users appearance in the system. Moreover the Kinect is a RGB-D sensor so that no additional color camera is required. The sensor is affordable and widely available. There are two versions available so we will briefly discuss these two options. The original Microsoft Kinect and its SDK have been released in June 2011. It is used in a wide range of applications. In July 2014 the Kinect v2 was released and replaced the original sensor. A comparison of the first Microsoft Kinect[Netb]

	Kinect	Kinect v2
Color Resolution	640 x 480 @ 30fps	1920 x 1080 @ 30 fps
Depth Resolution	320 x 240	512 x 424
Depth Sensing Tech.	structured light	time-of-flight
Min. Depth Distance	0.4m	0.5m
Max. Depth Distance	4.5m	4.5m
Horizontal FoV	57	70
Vertical FoV	43	60
Skeleton Joints	20	25
Multiple sensor support	Yes	No

Table 2: Comparison of Kinect and Kinect v2

and the Kinect v2[Netc] can be found in Table 2.

Especially the improved resolution and precision speaks in the favor of the v2 sensor. Obviously the higher resolution results in a higher complexity to process the data. But given the significant quality differences, we consider the higher cost as acceptable. The ideal system should provide a 360 degree scan of the user and his environment, therefore the use of multiple sensors would be required. Unfortunately the official SDK for the v2 does not support the use of multiple sensors on one computer. For each sensor you would need an dedicated computer to access the sensor. For the v1 there are multiple sensors supported by the SDK so you can get the data of multiple connected sensors from one computer.
The depth sensors do not just differ in resolution but also in the accuracy of the values due to different methods of depth sensing. The first version uses a structured light technique. This works by emitting a known infrared light pattern onto the scene and capturing the result. By the distortion in the resulting pattern the depth values for each pixel are calculated. Even for the relatively low resolution image, not every pixel is measured actively. Instead the values for each pixel are interpolated from fewer samples. The Kinect v2 sensor on the other hand uses a true time-of-flight sensor. This allows for higher precision as every pixel a value is measured without interpolation. This results not only in a higher resolution depth image but the individual measurements are more precise [GJRGMS+15]. Another drawback of the structured light method is the inference caused by multiple sensors in the same environment. The different patterns cannot be distinguished and therefore not assigned to the right sensor. This can be solved by adding motion to each sensor. The system from Beck et al [BKKF13] [BF15] uses multiple Kinect sensors for 3D live avatars in a group-to-group telepresence system. They also had this working with v2 sensors but they are using libfreenect, a free library for the Kinect that does currently not support skeleton tracking. As the v2 does use TOF for depth sensing, there

is no interference between multiple sensors.
For the first implementation of the MREP system we decided to use a single Kinect v2 sensor with the official SDK. It offers the best quality depth sensing and skeleton tracking. The ability to capture true 360°of the interaction space has a high priority for future improvements. To overcome the limitation of only one sensor per computer, we could use a network connection between the multiple capturing computers and send the voxel data to a central rendering computer. This requires the separation of the system in a capturing and a rendering part to be extensible in the future.

3.5 Hardware

The system is run on two identical desktop computers. They feature an Intel i7-6700 Quad-Core Processor running at up to 4.0 GHz and 16GB of DDR3 Memory. The graphics card is a NVIDIA GTX 970 with 4GB of VRAM. As the operating system Windows 10 64Bit is used. The network connection is a 1 Gigabit/s Ethernet connection over a local switch.
As mentioned in 1.2 we want to capture a user and his surroundings with a depth sensor. Specifically the goals to create recordings of a user and to use skeleton tracking to identify and segment a user, require an external mounting position for the sensor. The user shall also be able to see his whole body, for example in a virtual mirror. To just capture the users body and display it to his own perspective it could be sufficient to use a head mounted depth sensor. This has the benefit of always facing in the view direction, but also requires a lightweight sensor with very short minimal usable distance. As long as only one sensor is used, the external mounting position is required. As a mounting solution we chose a 2.3m height and 1.1m wide metal frame on wheels as shown in 12a. This provides us with a solid mounting solution for the Kinect v2 depth camera as well as the tracking camera of the Oculus Rift HMD. Additionally the frame is mobile so it is possible to move the system while ensuring unchanged positioning of the sensors.
First we need to decide on the positioning of all components. The depth camera defines the space in which we can capture the user and the environment. One of the defining features is the ability to present the user with a representation of his own body. It is important to keep in mind, that we try to provide him with a mixed reality experience, from his own first person view, although other view points are possible. Thus we have to evaluate possible camera position from the view of a user. The users hands and arms are very important to give a believable body experience. With a camera mounted at shoulder height or lower it would not be possible to capture the users arm if extended in front of the body. The camera has to be able to see the upside of the arms. Thus we chose a mounting position on top of the frame, this puts the Kinect v2 sensor on a height of 233cm. It is

tilted forward in an angle of 30°. From this position the camera can capture the arms, if extended straight. To improve this, multiple cameras would be needed. Either mounted around the user or directly on the HMD.
The tracking camera for the HMD defines the space in which the user can move, while still being tracked. The cameras field of view as well as minimal and maximal reliable range are the defining parameters. For the Oculus Rift DK2 the field of view is 74° horizontal and 54° vertical. With a minimum distance of 40cm and a maximum distance of 250cm. We mounted the Oculus tracking camera at a height of 160cm. The reasoning and figures explaining the positioning of the sensors can be found in 4.4.

3.6 Setup

As mentioned in 1.2 we want to capture a user and his surroundings with a depth sensor. Specifically the goals to create recordings of a user and to use skeleton tracking to identify and segment a user, require an external mounting position for the sensor. The user shall also be able to see his whole body, for example in a virtual mirror. To just capture the users body and display it to his own perspective it could be sufficient to use a head mounted depth sensor. This has the benefit of always facing in the view direction, but also requires a lightweight sensor with very short minimal usable distance. As long as only one sensor is used, the external mounting position is required. As a mounting solution we chose a 2.3m height and 1.1m wide metal frame on wheels as shown in 12a. This provides us with a solid mounting solution for the Kinect v2 depth camera as well as the tracking camera of the Oculus Rift HMD. Additionally the frame is mobile so it is possible to move the system while ensuring unchanged positioning of the sensors.
First we need to decide on the positioning of said components. The depth camera defines the space in which we can capture the user and the environment. One of the defining features is the ability to present the user with a representation of his own body. It is import to keep in mind, that we try to provide him with a mixed reality experience, form his own first person view, although other view points are possible. Thus we have to evaluate possible camera position from the view of a user. The users hands and arms are very important to give a believable body experience. With a camera mounted at shoulder height or lower it would not be possible to capture the users arm if extended in front of the body. The camera has to be able to see the upside of the arms. Thus we chose a mounting position on top of the frame, this puts the Kinect v2 sensor on a height of 233cm. It is tilted forward in an angle of 30°. From this position the camera can capture the arms, if extended straight. To improve this, multiple cameras would be needed. Either mounted behind the user or on directly on the HMD.
The tracking camera for the HMD defines the spaces in which the user can

move, while still being tracked. The cameras Field of view as well as minimal and maximal reliable range are the defining parameters. For the Oculus Rift DK2 the field of view is 74° horizontal and 54° vertical. With a minimum distance of 40cm and a maximum distance of 250cm. We mounted the Oculus tracking camera at a height of 160cm. The reasoning and figures explaining the positioning of the sensors can be found in 4.4.

3.7 Software Infrastructure

As already discussed in 3.4, we will use the official Kinect SDK 2.0 to access the data of the capturing sensor. This also determines the operating system for us, as it is only available for the Windows platform. Additionally it requires the use of Windows 8 or higher. The Oculus Rift is supposedly best supported on Windows 10 systems as it offers a better implementation for Asynchronous Timewarp. Based on this we chose Windows 10 64Bit as our operating system for the computers.
The basis for our rendering system is the Unity Game Engine. We chose this as it offers a good set of features and is still easy to learn for further maintenance of the system. The feature set includes all major features one would expect from a game engine, but also integrated support for virtual reality headsets. This makes it easy to prototype a VR application without recreating all basic functionalities on the way. In addition the big and active community offers great resources on how to use the engine and tips on common problems.
The following is a very brief introduction to Unity, to explain some of the terms that will be used in this work.
Unity application are structured in Scenes. These are used as containers for independent parts of the application. For example the main menu could be a Scene with each level of a game being a Scene again. The basic structure for a Unity object is the GameObject which is used for a wide variety of different functions. It defines for example the Transformation which handles all the basic functionalities like position, rotation and scale of an object. The graphical representation of an object is stored in a Mesh, which contains its vertices, index list, normals and material. Functionality is added by C# scripts.

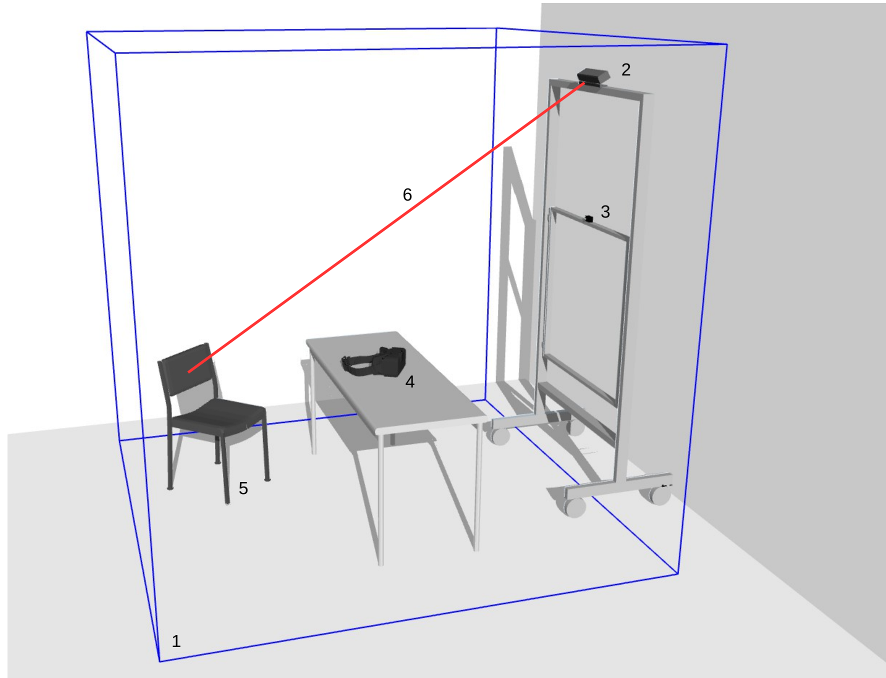


Figure 5: An overview of the MREP system. (1)Interaction space, (2) Kinect v3 Depth Camera, (3) Oculus Rift Tracking Camera, (4) Oculus Rift HMD, (5)Objekt in the Scene, (6) View ray from the Kinect

4 Mixed Reality Embodiment Platform

In this section the specific implementation of the MREP project is described. The decisions for each system component have been discussed in section 3 and a setup for the hardware is defined. To give an impression of the structure a high-level overview of the system is shown in Figure 6. On this level the system is structured in three major parts.
First the capturing components on the left, as described in 4.1. This part consists of the first desktop computer and the Kinect v2 RGB-D camera. It handles the capturing of the interaction space in the real environment, in which the user is located. The second major part is the rendering computer, a more in depth description can be found in 4.2. It runs the Unity application, which renders the voxelspace as an output to the Oculus Rift HMD. The last part is the networking between capturing and rendering computer as described in 4.3.

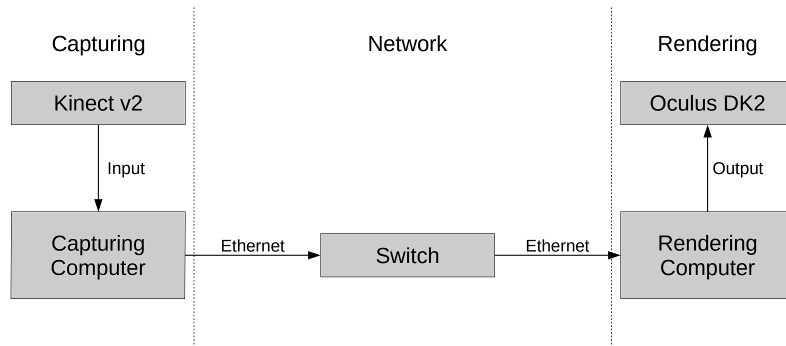


Figure 6: Overview of hardware components

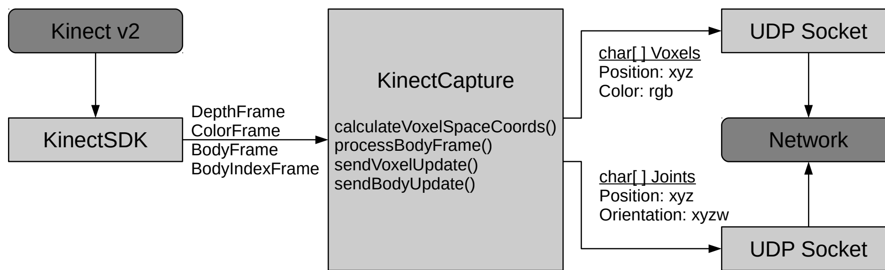


Figure 7: The Kinect capturing application

4.1 Capturing

As already discussed the capturing part is separated from the rendering part to be exchangeable in future versions of the system. This allows the easy modification using multiple or different sensors for capturing as long as it will sent the appropriate data expected by the rendering machine. For the first version of the system the capturing subsystem uses a Kinect v2 to capture the interaction space with the user and his surroundings. A diagram of the KinectCapture application is shown in Figure 7. We use a plain C++ application with the official Kinect for Windows SDK 2.0 to access the data of the Kinect sensor. The specific buffers containing depth data, color data and the skeleton tracking data are processed and send via the network interface described in 4.3.
The input consists of the raw depth and color data. These are mapped and transformed as described in 4.5 to get 3D positions and their corresponding colors. The 3D positions are transformed in a way that they are represented in the coordinate system of the voxelspace defined in 4.4. At this point the data is essentially a colored point cloud representing the real data within the voxelspace.
To give additional meaning to each voxel, we use the BodyFrame and BodyIndexFrame classes. The BodyIndexFrame is based on the depth image of the sensor. It contains 512 x 424 pixels with information whether the given pixel is part of a tracked person or belongs to the background. This allows us to identify voxel that belong to the users and others that make up the real environment. This information is added to each voxel additional to the position and color data.
The data of the skeleton tracking is stored in the BodyFrame. The Kinect sensor can track six persons with up to 25 joints each. It provides positions and orientations for these joints. Similar to the position data, the tracking data is transformed and mapped to voxelspace coordinates. This tracking data is send over a second port to the rendering machine and is used to determine the body parts of the user. This process is explained in 6.3.

4.2 Rendering

The rendering system is build using the Unity Game Engine. As a game engine it offers a wide range of features to get started quickly, without the need to implement basic functionality again. Some of the major components of the MREP system will be described in the following.
As the MREP system is an immersive virtual reality application, the rendering output has to be displayed on an HMD. Although Unity offers build-in VR support for different HMDs, we opted to use the Oculus Utilities for Unity 5 package. This provides a more flexible VR implementation and offers several helper functions.

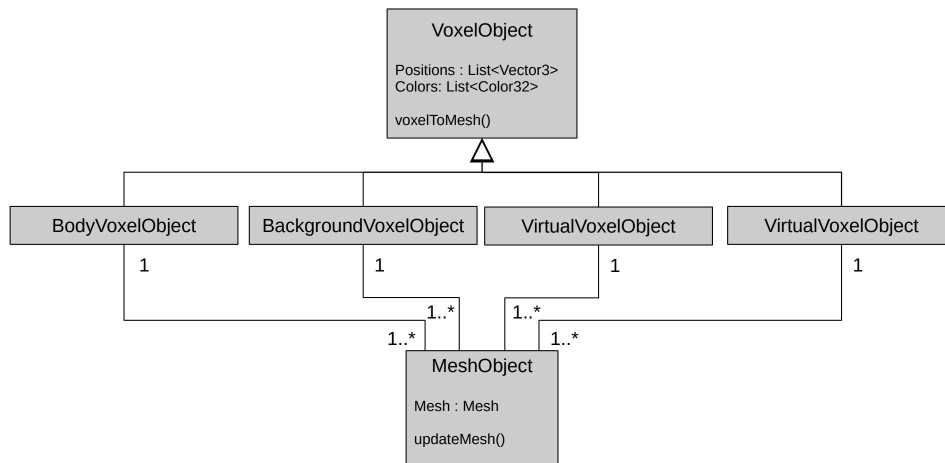


Figure 8: relation between VoxelObjects (containing voxel data) and MeshObjects (containing the Mesh needed for rendering by Unity)

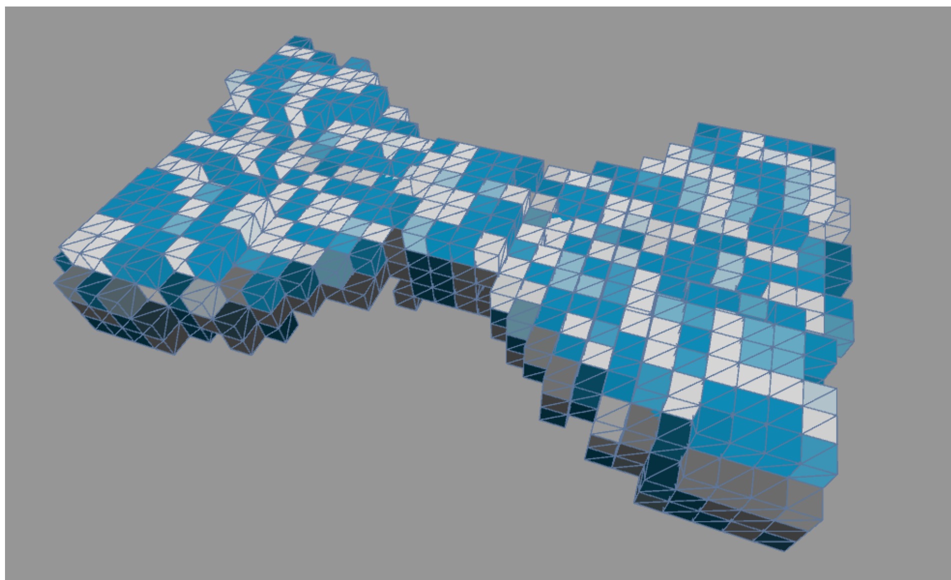


Figure 9: A closeup of voxels with highlighted edges

Every Scene in Unity needs a main camera that captures and renders the virtual world. The OVRCameraRig provided by the Oculus Utilities 5 package has this camera component and handles the rendering according to the position of the head determined by the tracking.
As a central GameObject to hold configuration details, regarding the overall setup, we used the MREPManager. It is used to set the dimensions of the voxelspace and also the real world position and orientation of the Kinect v2 and Oculus Rift tracking camera. The same settings are configured in the capturing application, as it does the processing to convert the sensor data to our voxelspace.
One of the central components are VoxelObjects, as shown in Figure 8. VoxelObjects are used for everything that is supposed to be rendered as voxels. The system defines different kinds of VoxelObjects to handle the different classes of objects present in the scene. First is the BodyVoxelObject which contains all voxels which have been identified as part of the user. It further differentiates these voxels into different body parts, based on the skeleton tracking data. This process in explained in detail in 6.3. Second the BackgroundVoxelObject. It holds all voxels that do not belong to a user, but are present in the real world and captured by the sensor. Last the VirtualVoxelObject. Different from the other two types, this does not represent real world objects. It is used to handle virtual objects which are added to the scene, without a physical version being present. The process to create these objects is explained in 5.2.
Each VoxelObject has one or more MeshObject defined as children. Unity is not a voxel based game engine, instead it works with vertices which form triangles to make up a Mesh. Thus Unity expects Mesh objects as the input for their rendering implementation. To achieve the voxel optic, we discussed earlier, we need to translate the simple position and color data to a Unity Mesh. Each voxel is rendered as a cube with uniform color, thus there is no need to work with all 4 vertices and 36 triangle indices. Instead we keep the simple position and color structure until all modification to these are done. At the end of each frame the position and color lists are converted to a Mesh. If necessary multiple MeshObjects are used, because a Unity Mesh uses a 16-bit unsigned integer to address vertices in a Mesh and thus is limited to 65,536 vertices per Mesh. Ultimately to create the representation of a voxel, a geometry shader is used to create the necessary vertices that make up a cube. This cube is uniform colored with the one color we mapped to his position in the capturing application.

4.3 Networking

The network component allows to extend the system in the future with additional capturing machines or to separate the capturing and the rendering locally. To connect the two subsystems for capturing and rendering we

time	packetNum	voxelCount	x_1	y_1	z_1	r_1	g_1	b_1	i_1	...

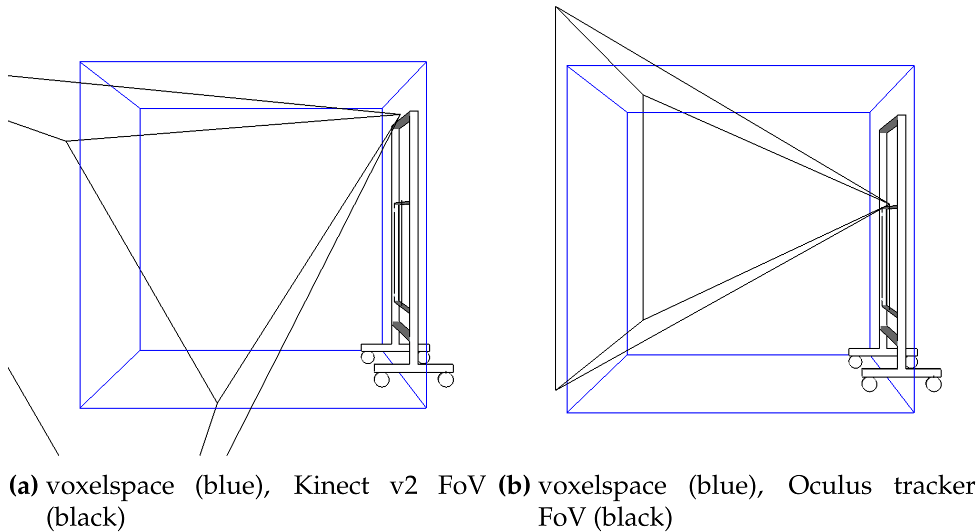
Table 3: structure of a voxel packet

time	jointCount	jointType	x_1	y_1	z_1	rx_1	ry_1	rz_1	rw_1	...

Table 4: structure of a joint packet

need a network component to transfer the real world data to the virtual environment. After processing the raw sensor data, the KinectCapture application will send the data to the render computer.
This is achieved via an 1 Gigabit/s Ethernet connection between the capturing computer and the rendering computer. We are using a local connection over a separate switch to limit external factors and ensure a stable connection. The network implementation is based on a Winsock UDP socket connection. The User Datagram Protocol (UDP) is a connectionless minimal protocol that gives no guarantee of delivery or even the order of the arriving data. It is often used in real-time systems where it is more important to reduce latency at the cost of potential packet loss.
As we have two independent data sets, the general voxel data and the tracking data, two different sockets are used. The first one is used for the voxel data. Each voxel is defined by a 3D position in the voxelspace and its corresponding RGB-color. We chose a 16-bit unsigned integer to address each dimension of the voxelspace and 8 bit per color channel. Additionally we encode the BodyIndex in an 8-bit integer. This adds up to a total size 10 byte per voxel. The structure of a packet is shown in Figure 3. Each captured frame has to be split into multiple packets. To identify potential data loss and identify packages of one frame on the rendering side, each packet stores additional data. This is in particular the time stamp from the Kinect sensor, an ongoing number for each packet of a frame and the total number of voxel of the corresponding frame. These are additional 16 byte per packet. Because the maximum datagram size of an UDP packet is 65,507 we set a limit of 6,000 voxels per packet, which results in a overall datagram size of 60,016 bytes.

The second type of packet sent over the other port handles the data related to skeleton tracking. It bundles the position and orientation data for all tracked joints of a user into one packet. Each joint takes up, 1 byte to identify its JointType, 2 byte per dimension for its position, and a total of 16 byte for a rotation quaternion. This results in a total size of 23 bytes per tracked joint. The structure of a JointPacket is shown in Figure 4. It carries additional information in form of an 8 byte time stamp and 4 byte corresponding to the total number of tracked joints. Even for a theoretical number of 6 users with 25 tracked joints each, the total datagram size stays well below the aforementioned limit of an UDP packet, thus no splitting is



(a) voxelspace (blue), Kinect v2 FoV (black)

(b) voxelspace (blue), Oculus tracker FoV (black)

Figure 10: A comparison between the Kinect v2 FoV (left) and the Oculus Tacking FoV (right). The voxelspace is shown in blue.

needed.
In the current state the network bandwidth is not a limiting factor. Although a potential system, using multiple Kinect sensors could increase the network load significantly. In this case some optimization or compression techniques could be applied to reduce the required network bandwidth.
The Kinect sensor only provides data at a frame rate of 30 frames per second. However the rendering to the HMD has to be done at a rate of at east 90 frames per second to ensure a smooth and stable experience for the user. To unlock the Unity main thread from the network traffic, we outsourced the network handling to a seperate thread. This allows the rendering to happen at a high frame rate with potential slow downs due to network handling. At the start of each new frame the rendering thread will poll the network thread for new data, but will immediately continue if no new and complete frame has been received since the last poll.
The described network implementation has been tested and works as expected. Due to the fast local connection, the network communication between two distinct computers does not add noticeable delay to the users experience. For ease of use we opted to use a local host loop back between the capturing and rendering application to execute both on the same computer.

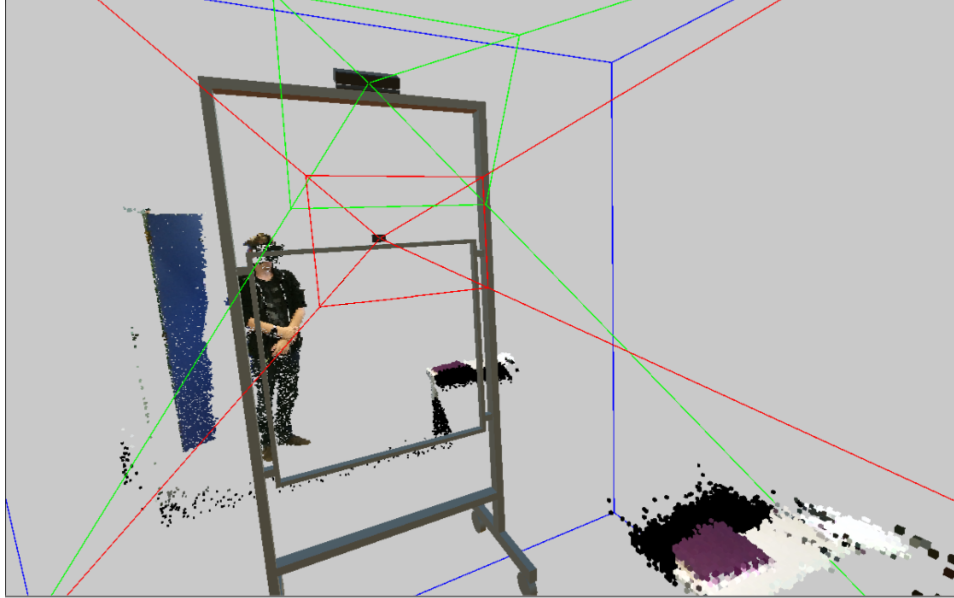


Figure 11: Voxelspace boundaries(blue), Kinect v2 FoV (green), Oculus DK2 camera FoV (red)

	Kinect v2 depth	Kinect v2 color	Oculus DK2
Range	0.5 - 4.5m	0.5 - 4.5m	0.4 - 2.5m
FoV	70°x 60 °	84.1°x 53.8°	74°x 54°

Table 5: Comparison of the field of view for the used sensors

4.4 Voxelspace

After choosing the hardware and installing it, we have to define the environment we want to use. We need to consider the space captured by the Kinect sensor and the range of the Oculus tracking. Another determining factor is the height of the user and how much interaction space we want to have. We want to give the user the option to move around in an area, instead of being fixed in place.
To determine the area we can cover, with the chosen hardware, we have to take a look at their field of view. Table 5 shows the specifications for the hardware we use. Notice how the horizontal field of view of the color camera is much wider than the one of the depth sensor. But the vertical one is slightly smaller. We have to respect the smaller one in each dimension to correlate all depth value with color values. The a visualization field of view of the Kinect can be found in Figure 10. Another visualization is shown in Figure 11. This captured inside the MREP system and can be toggled on for the user, but is mainly used for debugging purposes.

The amount of space we need is mainly determined by the user. An

average male of 1.81m male has an arm span 1,80m and measures a height of 2.30m with extended arms. To allow more natural movement the user should not be limited to a fixed position but have some extra space to move around. Now we have to define our voxel space. Initially we tested a resolution of 256 x 256 x 256 with a size of 1cm per voxel. This area of 2.56m x 2.56m meet the aforementioned requirements. This results in a total count of 16,777,216 voxels. With each dimension measuring 256 positions, it can be addressed in 8 Bit. Later we could improve the system performance to allow for greater voxel resolutions. The effect of this can be seen in 18

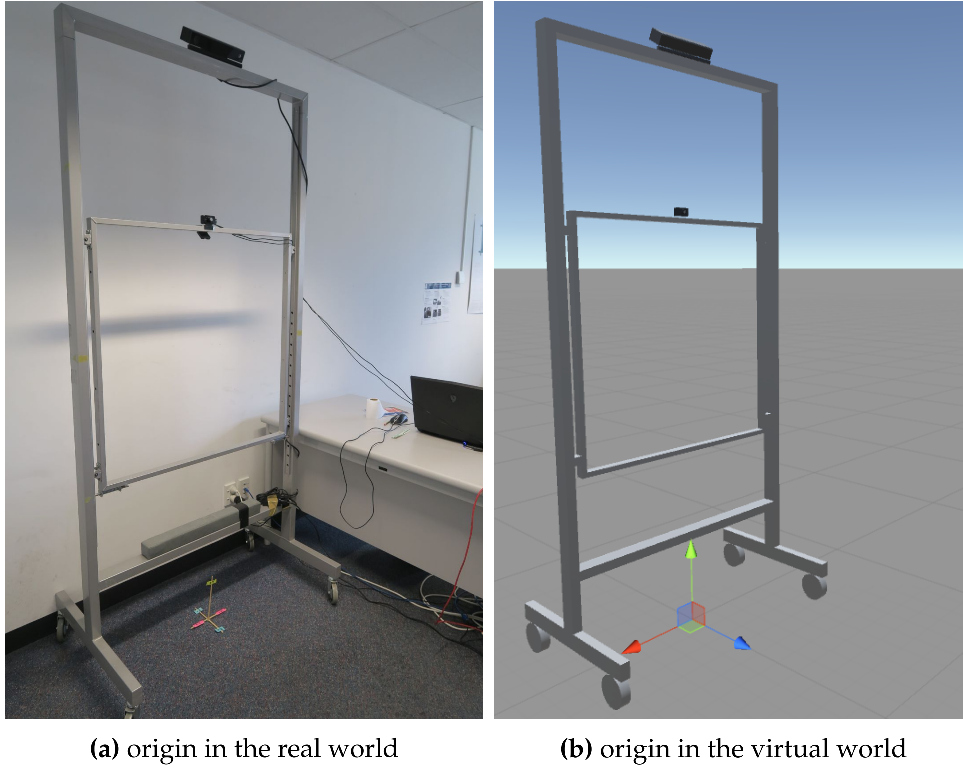
4.5 Coordinate systems

As we want to merge virtual and real objects to one shared environment we need to define a common reference coordinate system. Without this it is not possible to align both worlds. This alignment is key as we want to achieve synchronous visual and haptic feedback for real objects. When the user sees the voxelized real table, his voxelized hands have to touch the voxelized real table at the same time as his real hands touch the real table. To achieve this the origin of the shared coordinate system needs to be represented in the real world and the virtual world. Unity has a defined origin so a corresponding position in the real world needs to be set. Recommendable is an accessible and unique point in the world from where it is easy to measure. In this case the origin was placed on the floor in the middle of the frame and was marked as shown in Figure 12a. The axes are oriented in a way that positive y-axis is up. When the user stands in front of the frame he is looking along the negative z-axis and positive x-axis is to the left. With this as an orientation the exact position of each object can be measured in the real environment. Because one unit in Unity corresponds to one meter in the real world, measurements can be easily transferred. We placed a model of the real world frame and the sensors in the virtual environment to have a visible reference to the real world. The virtual frame and the origin in the virtual environment is shown in 12b.

The next step is to get the captured data of the real world into this virtual environment. After that the virtual camera has be to placed in the right position to match the users eyes. To align all data in the defined coordinate system we have to know what kind of data we get from the Kinect sensor and the Oculus tracking system.

4.5.1 Kinect v2

As described in section 4.1 there are different frames containing different information provided by the Kinect SDK to work with. One is the ColorFrame, containing the data of the color sensor in a 2D color image. This is



(a) origin in the real world

(b) origin in the virtual world

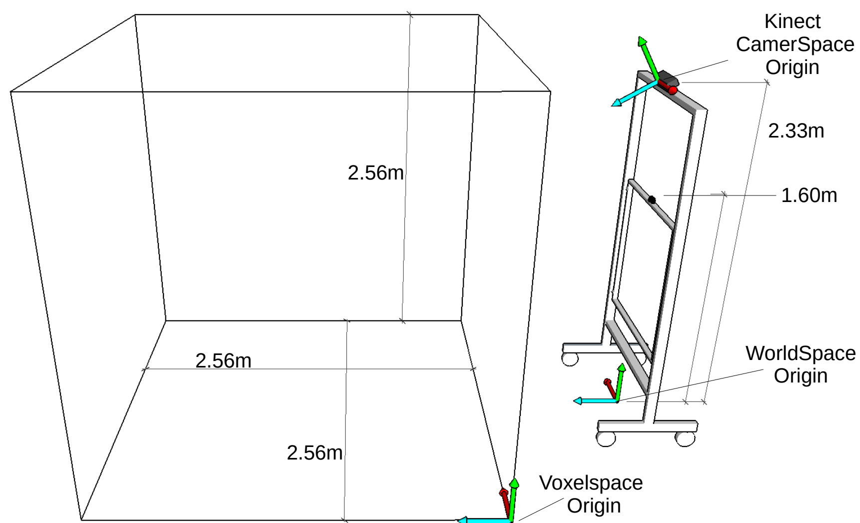


Figure 13: Overview of different coordinate systems

provided in an array with 1920 x 1080 RGB-values. Color space describes a 2D location on the color image. The second is the DepthFrame, also providing a 2D image with 512 x 424 values. Each value contains the distance in millimeters of that point to the sensor. Accordingly depth space describes a 2D point on the depth image. The third frame is the BodyFrame containing the classification of user and background data. As it is based on the data of the depth sensor it has the same resolution and can be assigned directly. All remaining skeleton and tracking information, for example a joint position, are 3D points in camera space. Camera space describes a 3D location using the coordinate system of the Kinect. The origin of the coordinate system is located at the center of the depth or IR sensor as shown in Figure 14. The positive x-axis goes to the left side of the sensor, the positive y-axis goes up and is based on the sensor's tilt, the z-axis grows out in the direction the sensor is facing. One unit equals one meter.
First of all some calibration is needed to align the color image with the depth information. As the color and the depth sensor have different resolutions, FOVs and are at slightly different positions, the images need to get mapped to obtain the matching data for one specific point. The depth points need to be transformed into color space to be able to look up the right color information. This look up is a common operation for a lot of Kinect applications and is provided in the SDK by the coordinate mapper.
What we have now are 512 x 424 2D points in depth space with their corresponding color information. The next step is to transform the 2D points to 3D positions in camera space. This is achieved by un-projecting from depth space using again the coordinate mapper. This refers to intrinsic calibration and is based on the camera parameters of the Kinect sensor. As this is provided by the SDK we do not need to do a manual calibration to calculate these parameters and transformation.
The result of these 3D points displayed in Unity is shown in Figure 15a. The positions need to be transformed to the defined shared world coordinate system. To do so the position and rotation of the Kinect have to be measured in world coordinates. This can be done manually or using the approximation of the floorplane from the Kinect SDK. With these information a transformation matrix can be calculated to map camera to world space. After this transformation the voxel in the world coordinate system are rendered as in Figure 15b.
The final transformation is the mapping to voxelspace coordinates. The voxelspace is defined as described in 4.4. As our voxelspace has a lower resolution as the Kinect depth sensor especially close to the sensor several positions are mapped to one voxel. By now we have a data structure representing the voxelspace where is stored whether a voxel is already set. Just the first color of a point falling into that voxel is stored, the others are ignored. One could think about more sophisticated methods of handling this as interpolating between all points. When all points are mapped the central

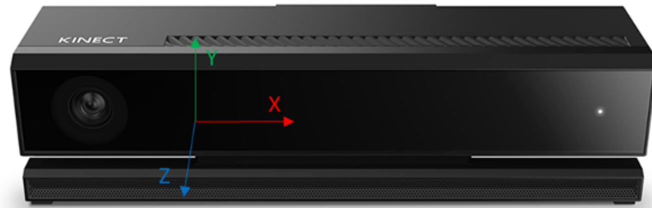
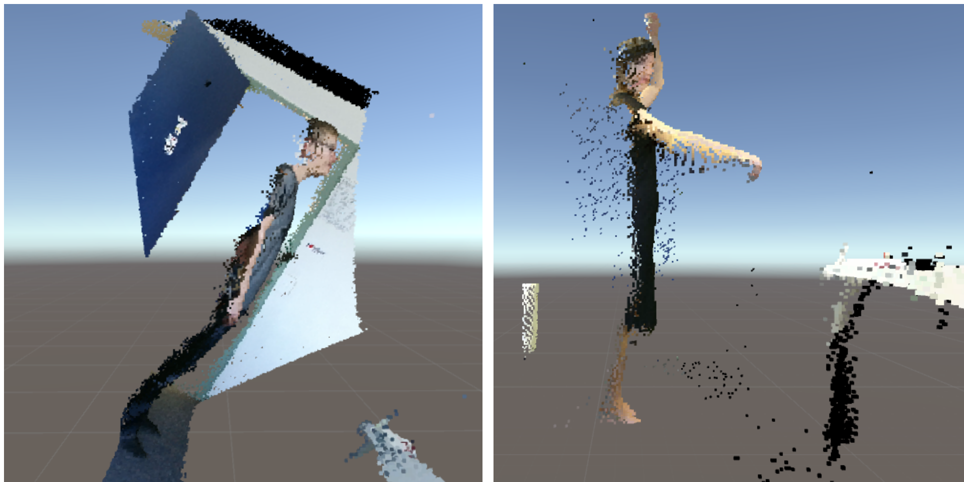


Figure 14: Kinect v2 sensor with coordinate system of the camera space [Neta]



(a) no calibration of the sensors tilt and height, positions in camera space but rendered as world space coordinates

(b) after the transformation, tilt and height are compensated the positions are stored in world coordinates

Figure 15: Kinect data in MREP

positions and colors of the set voxel are sent over the network as described in Section 4.3.

4.5.2 Oculus Rift

The system is supposed to be egocentric. Hence the main camera in the Unity scene has to be at the right position to give the user the feeling of being in his own body. Therefore we need to know the position of the HMD in our main coordinate system. The position is not fixed but has to be updated as the user will move during the application. The tracking of the HMD and the updating of the point of view as well as the stereoscopic rendering is handled automatically. The reference frame for the tracking of the Oculus is set at the start of an application. The origin of this tracking space is set to the current position of the users head as seen in Figure 16. Seen from the user, the positive y-axis is up, the positiv x-axis goes right and he is looking along the negative z-axis. The x-z plane is always aligned to the ground.

This origin is represented in Unity in the TrackingSpace GameObject. All other components like eye anchors are its children and transformed relative to this origin. When an Unity application is started everything is set accordingly to this conventions. But there is no real world reference. With every start the reference frame is placed in the world coordinate system by a guess from the underlying Software. For example the height is set to the parameter of the character height you can chose in the Oculus setup. But this is obviously not the right world coordinate for every user. If we assume the HMD is in the real world at a height of 1.7m at the start of the application it will appear in the virtual environment at a height of 1.9m. Let tracking camera be at a height of 1.6m in the real world. As it is transformed relative to the tracking space (which is at the start position of the HMD) it will appear in Unity at a height of 1.8m. As this would not match with the view of the user we have to transform the TrackingSpace in Unity in a way it matches our chosen world coordinate system. Unfortunately the origin of the tracking space is hard to measure and changes with every start of the application depending on the position of the HMD. What is measurable and stays in a fixed position is the tracking camera. The position from our tracking camera is set to a height of 1,60m above the world origin with no rotation. The GameObject representing the tracking camera is the TrackerAnchor. GetPose will provide the position of the tracker in Unity world coordinates. To get the tracker in the right position the current transformation needs to be inverted and translated to the right world position after that. It does not help to transform only the position of the tracking camera, but the whole tracking space has to be adjusted. As this is the parent object everything else including the TrackerAnchor will transform accordingly. This calibration needs to be done at every start. We get a pretty good result by that and interacting with the real world, like touching a table feels believable.

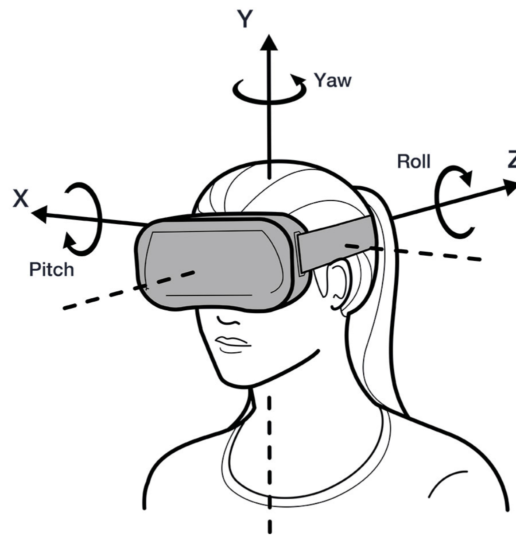


Figure 16: Oculus Coordinate System [VRb]

5 Scene Realism

Most known MR applications are AR systems where the real world appearance is the dominating factor and for visual coherence the virtual components are rendered as photorealistic as possible. It is the principal goal to make virtual objects look like reality. This has a high computational cost and requires a lot of hardware resources. But as discussed in Section 2.1 the visual realism of a scene is surprisingly not defined as an important factor for presence. Because of this and the question how much we can simplify the visual realism of a scene without losing the sense of presence, we chose a voxelized representation of the virtual environment. This corresponds to

Figure 17: first voxel rendering (1cm voxel)

Figure 18: result of different voxel sizes from left 0.5cm, 0.8cm, 1cm, the smaller voxel sizes preserve more details, especially noticeable at the hands and the face, but introducing missing voxel due to oversampling the Kinect data

reducing the pixel resolution of an 2D Image, but here we are reducing 3D data. It is also shown that non-photorealistic rendering can make a mixed environment significantly more believable as visual coherence between virtual and real objects is more important than high fidelity (see Section 2.3). As we get a point cloud of the real environment including the user we have to think of an appropriate reduction of this data. In section 4.4 we already discussed the dimensions of the voxelspace. Now we have to consider the resolution of our data. Figure 17 shows a first voxel rendering of the Kinect point cloud, where each voxel is 1cm in each dimension. But is this a good size?

5.1 Scene Representation

One main difference to most AR systems, is the fact that we do have 3D informations of the real world and not just a 2D image. This enables the automatically handling of the most problems AR systems do have to accomplish a coherent scene. We first discuss our general scene representation, including the chosen graphical representation and factors influencing the hole scene including real and virtual objects. These are specifically lighting and shadows. The problem of occlusion between virtual and real objects of the users perspective is one of the automatically right factors due to the known geometry. As every spatial information is combined in the Unity scene, Unity's rendering pipeline generates the correct images.

5.1.1 Voxel representation

The spatial dimensions of the voxel space were already discussed in Section 4.4. Now we have to determine a appropriate resolution. The size of each voxel and therefore the resolution of our voxelspace is limited by the

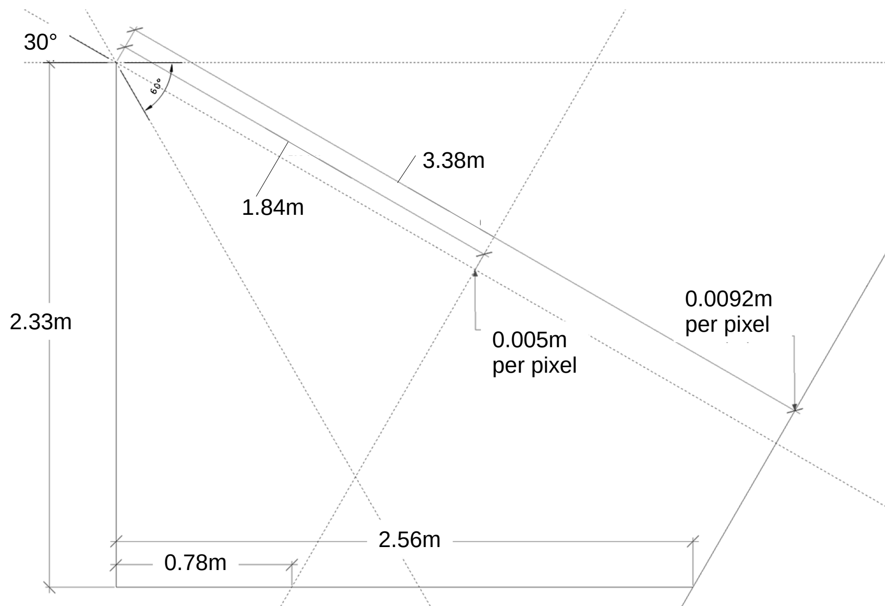


Figure 19: precision of the Kinect data at different distances

(a) voxel size = 0.5cm

(b) voxel size = 0.8cm

Figure 20: Effect of different voxel sizes at further distances

resolution of the Kinect data. Figure 19 visualize the changing area each pixel covers at different distances to the sensor. As we need a depth and a color value for every voxel the limiting resolution is the lower one of the depth sensor. The resolution of the depth sensor is 512 x 424 pixels with a FOV of 70 x 60°. Using this values and trigonometry the relation of pixel and the covered area can be determined. Following this one can calculate that one degree in the FoV is covered by 7x7 pixel. As the area covered by one degree gets bigger with increasing distance, increasingly more space of the real world needs to be mapped on these 7x7 pixel. Now you can calculate the space that is mapped to one pixel at a certain distance. So for a distance greater than 1.84m from the Kinect sensor, each pixel of the depth image cover more than 0.005m. This distance is measured along the view direction of the sensor and equates to a distance of 0.78m from the frame. The consequences are that details smaller than that cannot be captured by the sensor. In contrast the space covered by one voxel is consistent for every distance.
If you choose a voxel size bigger than the precision we loose details provided by the sensor. On the other hand by choosing a smaller size there will be gaps between the voxel as there are not enough values provided at further distances. You can see this effect in Figure 18. The image on the left shows a voxel size of 0.5cm. This results in a very sparse voxel grid, because the sensor can not provide values for each voxel at this distance. But it preserves a more detailed especially noticeable at the hand and face of the user. Considering the given interaction space the user would usally be at this position or even farther away. Consequently the effect and the holes get worse with increasing distance as can be seen in Figure **??**. We experienced the missing voxel as more distracting than the more details are an advantage. Increasing the voxel size to 0.8cm gives a much better representation at this distance.
When increasing the voxel resolution too far, only a very narrow interaction space is usable without big gaps. Thus we settled on a voxel size of 0.8cm, as this offers a good trade off between a detailed image and usable interaction space.

5.1.2 Lighting

A common problem for MR applications is to achieve a consistent lighting for virtual and real components. There are different algorithms to simulate the lighting of the real environment in the virtual scene to illuminate the virtual objects in a consistent way. The most common methods are summarized in the work from Jacobs and Loscos [JL06]. As we have a relative course structure little changes of lightings are less noticeable than in a photorealistic approach. We manually try to match the lights in the real environment by using the light sources Unity offers. We use the ambient light

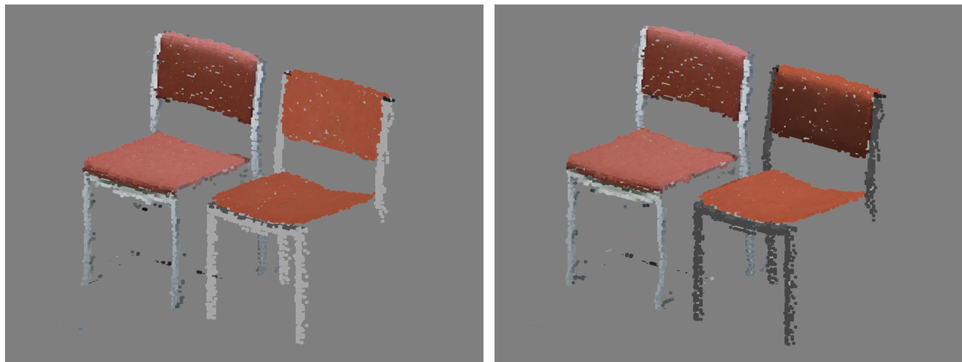


Figure 21: difference made by directional lighting, left off right on, the left chair is the real chair illuminated by the real ligth situation

from Unity and one directional light source to achieve a lighting situation similar to our real world. As the main light in the real environment is at the ceiling we use one directional light facing down to match that. The position of directional lights in Unity is not relevant. The light sources have an intensity property that needs to be adjusted for the right brightness. Due to other light sources in the real environment like daylight we manipulate the intensity of the ambient light in Unity as well to match the situation as good as possible. As we do not have any technique to measure the real lighting situation this adjustments need to be made every time the real situation has noticeably changed. Although it is just a rough approximation of the real world situation it makes a big difference. The result of illuminated virtual object and a rendering without illumination is shown in Figure 21 each time compared to the real chair.

5.1.3 Shadows

Additional to the right lighting the right shadows should be casted similar to the real world. The advantage of our system is that we do not need to generate shadows as close to the real world as long as both real and virtual objects cast the same shadows in the MREP environment. Unity provides shadow generation for directional lights. By calculating shadows for every voxel in our scene we achieve the same shadows for real and virtual objects. Because we have the information of the geometry from both worlds it is given that they can influence each other as seen in Figure 22. The left images shows the scene without virtual shadows. In the upper left picture a soft shadow from the real mask is seen on the real table caused by the light in the real environment. The images on the right show the scene with virtual shadows. One can tell that these differ from the shadow in the left image. This is because the shadows are casted by each individual voxel of

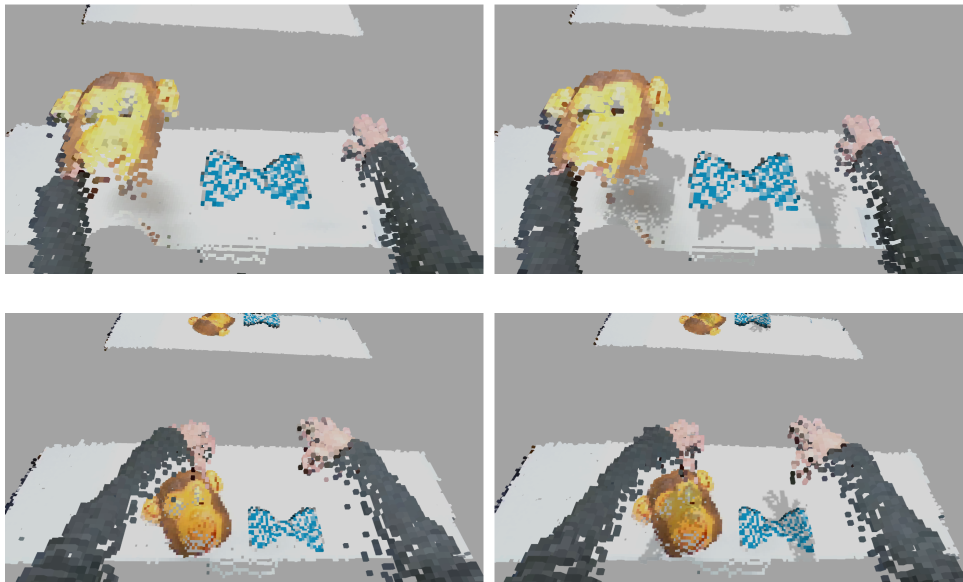
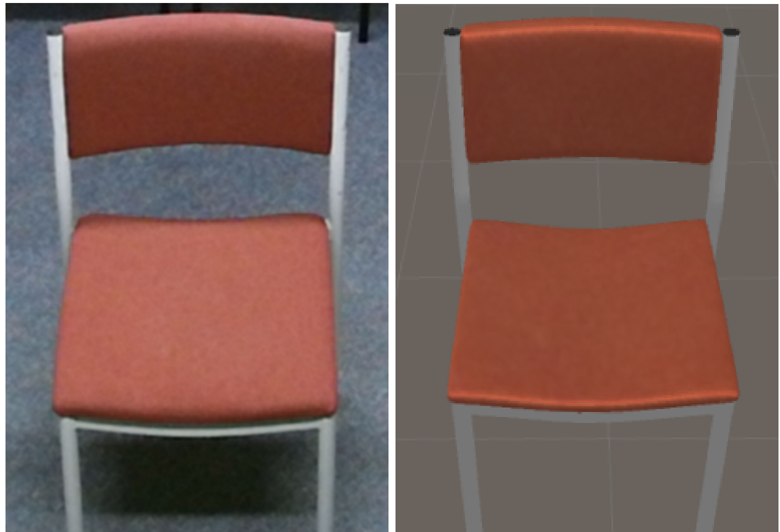


Figure 22: shadows off vs shadows on between real and virtual objects

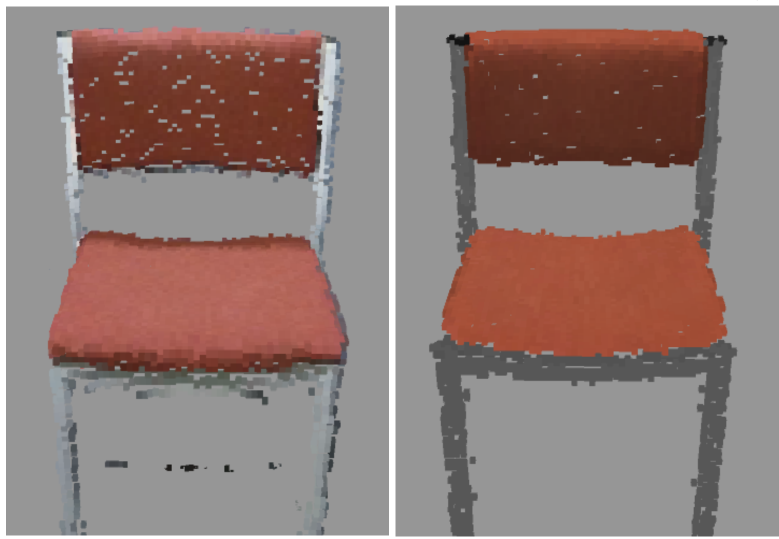
the voxelized objects showing holes and noise. However, it is more important that the objects do influence each other in any way than that the behavior is physically correct. Sugano et al. showed that even if the shadow of virtual objects are behaving contradictory to the real shadows of the scene, the virtual objects felt more present than without any shadow [SKT03]. The images show that the shadows help to connect real and virtual objects. Additional there are spatial clues provided by the shadows in the scene. In our framework these spatial information are useful, but not as crucial as in the 2D images as we have already some spatial information due to the stereoscopic rendering. Altogether the most important point in our system is the coherent behavior of all objects. As seen in the upper right image of Figure 22 the real object (mask), the virtual object (bowtie) and the user all cast similar shadows.

5.2 Virtual Objects

Now that we have defined an environment and a representation for our real data, the next step is to add virtual objects. We start with a "perfect" 3D Model corresponding to the real object in the real world. Now we have to figure out how the data of this real object is captured, processed and displayed to transfer this steps to the data of the virtual object. There are different factors influencing the appearance of the representation of the real objects:



(a) the real chair Kinect **(b)** model of the chair in Unity



(c) the representation of the real chair **(d)** the representation of the virtual chair

Figure 23: chair comparison

1. characteristics of the sensor; resolution, transformation and FOV
2. the voxel representation in a given resolution and size
3. noise and holes depending on the view of sensor and user
4. colors are affected by the illumination in the real world

To get a matching representation all of these factors have to be reproduced for the virtual objects. We use the following steps to achieve this. The steps are described in more detail in the following sections. First of all we need a 3D Model to start with. This model needs to be converted from a triangulated mesh to a voxel representation. Then the view restrictions of the sensor are applied by culling. If we want to consider the resolution of the sensor and be closer to the way of capturing of the sensor we have to use a raycasting technique. With this we can do voxelization and culling in one step. In either way we need to apply noise to the static voxel model.

5.2.1 3D-Modelling

First of all for displaying any virtual object a 3D model is needed. There are different ways of achieving such a model. One way is to scan real objects and digitalize them. Kinect Fusion is a algorithm to scan and store a 3D Model from captured sensor data. This did not work very well as there are many inaccuracies and artifacts in the resulting model. We want a model without artifacts from any sensors to start with. Our models were made with the 3D Creation Software Blender. We needed one virtual model with a real counterpart to compare our representations with the sensors output. Our first model was a chair in our office. This real chair was the ground truth for our method. It is essential for the modeling to have the right sizes and proportions of the real chair to transfer it to the model. The most difficult part was to texturize the model in the right way.
Colors are influenced by a lot of factors:

- the material of the object
- the lighting situation in the surrounding environment
- the capturing sensor
- the way of storing the color values/the color space
- the displaying medium
- the eye of the observer

To get the same color impression with the virtual model and the real model is nearly impossible. As we want to keep it simple we texturized the model with a texture captured by the Kinect color sensor in the usual lighting situation. For our system it is not needed to achieve the same color impression. Normally the question will not be whether two objects look the same but whether they are in the same environment.

5.2.2 Voxelization

To get a voxel representation of a mesh there are different techniques. We want a surface voxelization of the object. This is achieved by an intersection test between a voxel grid and the triangles of the mesh. This has to be done for every voxel of the grid with every triangle of the mesh. One of the intersection test that can be used for testing triangles against boxes in 3D is based on the Separating Axis Theorem (SAT) and described by Akenine-Möller in [AM01]. SAT tests two objects for each potential separating axis using projection. The algorithm will return as soon as one separating axis is found. When importing the virtual objects after modeling, Unity creates a GameObject with a corresponding triangulated Mesh. The triangles of the Mesh can be accessed via script. The grid used for the voxelization is defined by our voxelspace with the parameters height, width, depth and origin in world coordinates as described in Section 4.4. Each unit of this grid is a cube with the size given by the value voxelsize. For every voxelization of a virtual object at the start of the application a new GameObject is created. To calculate which voxels of the voxelspace are occupied by the object, intersections are checked between the triangles of the mesh and the cubes of the voxelspace. In order to avoid unnecessary intersection tests the axis aligned bounding box (AABB) of the triangle is determined. By mapping them to voxelspace coordinates, only the voxel inside the AABB are used for the intersection test. This is illustrated in images 1 and 2 of Figure 24. Each voxel inside the triangles AABB, colored in gray, is now tested with the algorithm of Akenine-Möller. The resulting voxels are marked in blue color in the last image. As a result the central positions of these voxel and corresponding lists of colors and normals are stored. The color and normal data are extracted from the underlying Mesh. In our approach we store the color and normal of the first vertex of the triangle as we want to keep it simple and assume small triangles. The voxelization is recalculated, when the GameObject was transformed. The correct rendering of the voxel is managed in the geometry shader as described in 4.2.

5.2.3 Culling

As said before the virtual objects should match the data we get from the sensor, which means that they need to behave like the captured data with

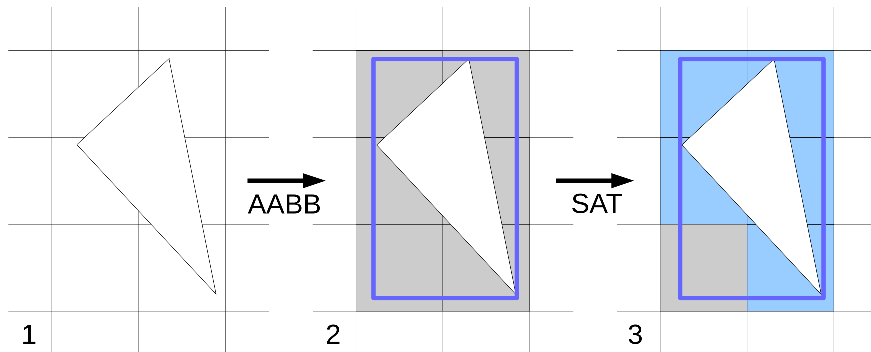


Figure 24: from left to right: triangle in voxel grid, AABB of the triangle and the voxel used for intersection test, stored voxel as the result of the intersection test

all its restrictions. Due to the fact that there is only one sensor in the real world everything is captured from this point of view. Consequently all parts of the real environment and of the objects in it that can not be seen by the sensor is not part of our captured data and the resulting voxel environment. What is visible or not is defined by sensor itself. Obviously it depends on the FOV and the position of the sensor. Everything outside of the view frustum cannot be captured. Additional the backsides of all objects and occluded parts cannot be seen. As in our virtual world the virtual camera for rendering differs from the camera generating the real world data the user is able to move independent from the capturing. So he is able to see holes and missing parts he would not see if the cameras would be the same. The same static capturing needs to be simulated for the virtual objects. In computer graphics culling algorithms are used to avoid the rendering of objects or object parts that are not seen by the camera. These algorithms can be used to simulate a virtual camera identical to the Kinect sensor in the real world. The virtual objects needs to be changed according to this camera to match the real world objects. There are different types of Culling handling different causes.

Frustum Culling Frustum culling discards the parts of a scene not seen by the camera caused by its view frustum. These is defined by the FOV and the positioning of the camera. Everything outside of this frustum cannot be seen.

To implement frustum culling we first calculate the planes of the Kinect frustum. They are defined by a point on the plane and a normal, facing inwards. In a first simple and not optimized approach we test every voxel against every plane and check with the dot product, whether the voxel is inside or outside the frustum. To do the vector from the point of the plane

to the position of the voxel is calculated. A negative result of the dot product with the normal of the plane indicates a voxel outside the frustum.
Backface Culling Backface culling discards the parts of a scene that are not seen by the camera because of their orientation. To implement backface culling we need the normal for each voxel. Because of this in the voxelization step additional to the position the normal from the underlying mesh is stored for each voxel. Both parameter are stored in world coordinates. In the culling step each voxel is tested whether its facing the Kinect or not. To do so the view vector from the position of the Kinect to the position of the voxel is generated. By calculating the dot product between this vector and the normal the visibility can be checked. If the result is positive the voxel is not visible. At the moment we "delete" the voxel by setting the position to a value which is automatically culled by Unity afterwards, to avoid changing the size of the underlying Mesh.
Occlusion In our framework we have two different types of occlusion. In other MR systems the occlusion Occlusion is the reason for a lot of holes in the real objects of the scene. These parts are not seen by the camera because they are occluded by other objects in front of them or themselves. In this case the rendering camera is not the same as the camera we need the depth test for. This could be achieved by making a simple depth test.

5.2.4 Ray casting

Alternatively to the methods described so far ray casting can be used for the simulation of the capturing by a virtual camera. This method is the closest to way the Kinect sensor works and the real data is generated. For each pixel of the sensor a ray is casted into the scene and the position and color of the first hit mesh is stored. This takes all of the camera characteristics in account. The transformation of the sensor and the FOV determine the rays that are casted. In that way no geometry outside of the view frustum is checked. Additionally as only the first position and color seen by the sensor is stored we do not have to handle occlusion separately. In Figure 25 the result of the ray casting is compared to the result of the voxelization step described in 5.2.2. In particular the stripes of missing voxel in the ray casting result are interesting. Comparing the results of the voxelization and culling steps with the real objects in the scene there are some artifacts besides the noise that are not reproduced yet. These are coupled to the resolution of the sensor. It is the same effect as described in 5.1.1 but horizontally instead of vertically. As this is taken into account in the ray casting technique by casting as many rays as pixel in the depth image, these artifacts can be reproduced by this, resulting in the stripes. Figure 26 shows what causes these stripes. With a smaller angle of the viewing ray to the surface of a captured object the distance between the rays increases. Figure 26 shows a sample for 3 viewing rays. So we are able to do the voxeliza-

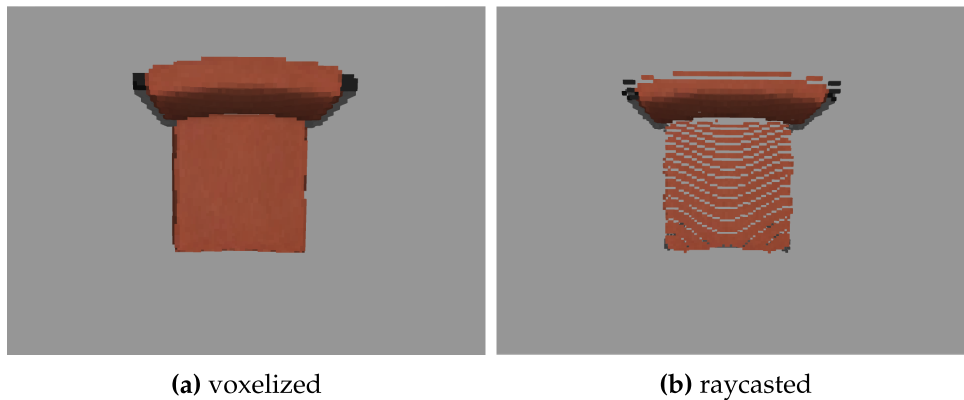


Figure 25: difference between SAT-voxelized chair and the raycasted result

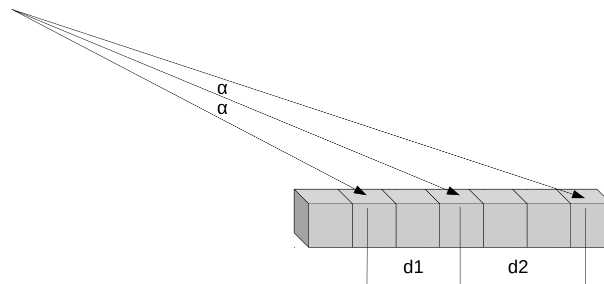


Figure 26: effect of the sensor modeled by ray casting

tion,culling and occlusion in one step. The finished results of the virtual objects voxelized in the two different ways is shown in Figure 29.

5.2.5 Noise

By now the rendering of the virtual objects take into account the voxel representation in the given resolution, the FOV, resolution and position of the Kinect sensor and the lighting of the real scene. But at this point the rendering is static. This is a very noticeable difference to the objects of the real world as they are changing over time due to the noise of the sensor. The noise in the data is caused by the depth sensor. The raw data is used without any processing. Most applications using depth data use smoothing filters for example an edge preserving bilateral filter. As we do not want to improve the data we get to achieve a higher fidelity, we do not use any filter to keep the processing of the real data as simple as possible. Consequently we have to figure out the characteristics of TOF sensors in generally and the Kinect v2 specifically. What factors are influencing the noise and how can this be modeled and transfered to the virtual objects?

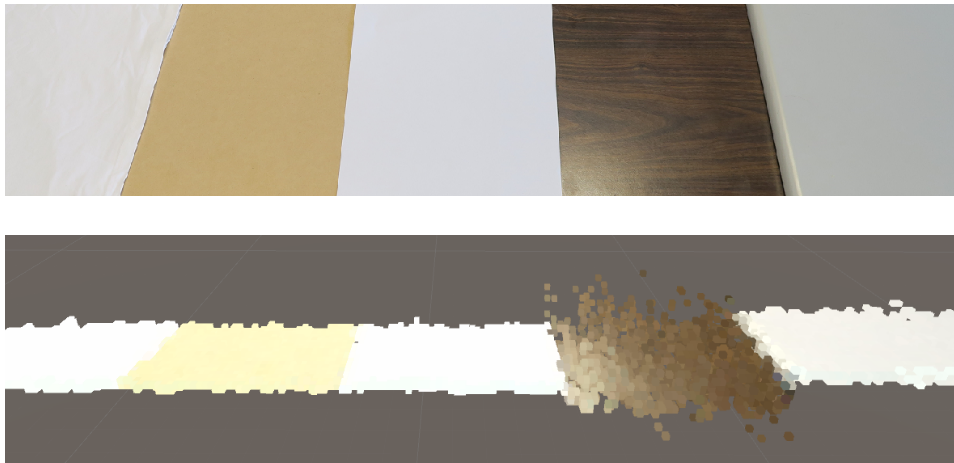


Figure 27: Set of different materials, a photo (top) and in MREP (bottom). From left to right: white cloth, MDF wood, white sheet of paper, wooden table, gray table.

To achieve this we have to know how the data is captured, processed and what errors are produced. As introduced in 3.4 Depth Sensors the Kinect v2 depth sensor works with TOF. A metrological characterization for the TOF sensor of the Kinect v2 is provided by Corti et al [CGMS16] and Gonzales-Jorge et al [GJRGMS$^+$15] who compared the v1 and the v2 sensors. They analyzed the random and systematic components of uncertainty in the measurements of the depth sensor. They identified the following parameters that influence the noise present in the depth image of the Kinect v2 depth sensor.

1. The distance of the captured object to the sensor
2. The angle between the camera and the captured surface
3. The distance of a pixel in the depth image to the central pixel
4. The reflectivity of the captured material
5. The additive noise present in all TOF sensors but not specified yet
6. The wiggling error, also an specific TOF characteristic
7. The mixing pixels effect shown between objects

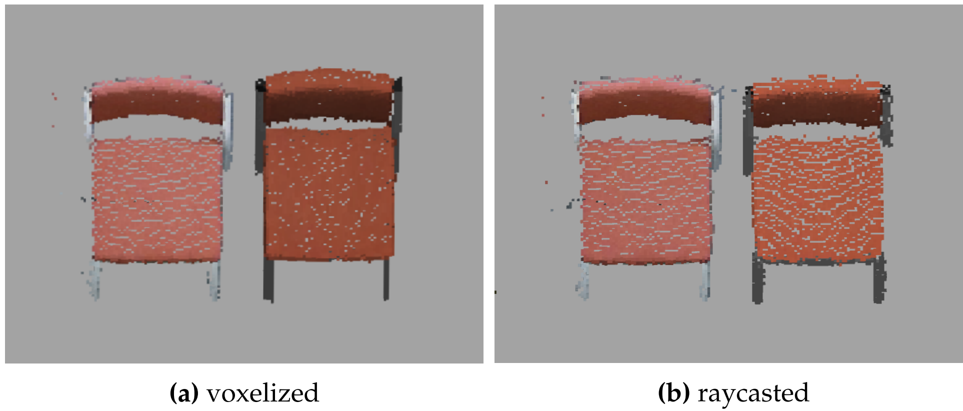
These factors were found by Corti et al [CGMS16] due testing the behavior of the Kinect depth sensor in different situations. The resulting depth values follow a gaussian distribution with different standard deviations, depending on the mentioned parameters. To introduce these errors to the

Figure 28: Mixing pixel effect

virtual objects a model would be used. Belhedi et al [BBB+12] introduced a model for the noise of TOF sensors. They used a 3D thin-plate-spline function to get the right standard deviation for each distance and pixel position. They also state that the noise for each pixel follows a gaussian distribution. The ideal approach would be similar to the one discussed by Belhedi (2012), but using the standard deviation values for the Kinect sensor determined by Corti et al. Due to insufficient time and to keep it simple in the first approach. We used a much simpler function to generate noise. As most of the effects just causes errors of a few millimeter this would not make any difference for our system, because in the voxelization step they are mapped to the same value nonetheless. More noticeable is the mixing pixel effect, which is introduced by interpolating between depth values in a preprocessing step included in the Kinect sensor. This effect can be seen in Figure 28 but is not implemented in our system. Another factor which has a big influence is the reflectivity of different materials. We have tested this in Figure 27. This has consequences in the scene as the environment seems a lot more disturbing when using a reflective table for example than a less reflective. As we have no material properties for our virtual objects this is not something we have implemented.

5.2.6 Summary

Due to the fact, that the capturing and the rendering camera for the real object differ there is the need for rendering the virtual objects in a way that adapts the static real world camera. The best way to implement the effects of the sensor on the geometry is a ray casting technique. Figure 29 shows the result of a straight forward surface voxelization combined with culling



(a) voxelized **(b)** raycasted

Figure 29: comparison of real chair (left) and virtual chair (right)

methods and the result of the ray casting each next to the real chair. The virtual objects in our system are mostly static as there is no possibility for interaction right now. Consequently the costs of raycasting is no problem at the moment as it can be done once at the start of the application. This technique is closest to the function of the Kinect sensor and reproduces the artifacts caused by the resolution and the view of the camera.

(a) external view

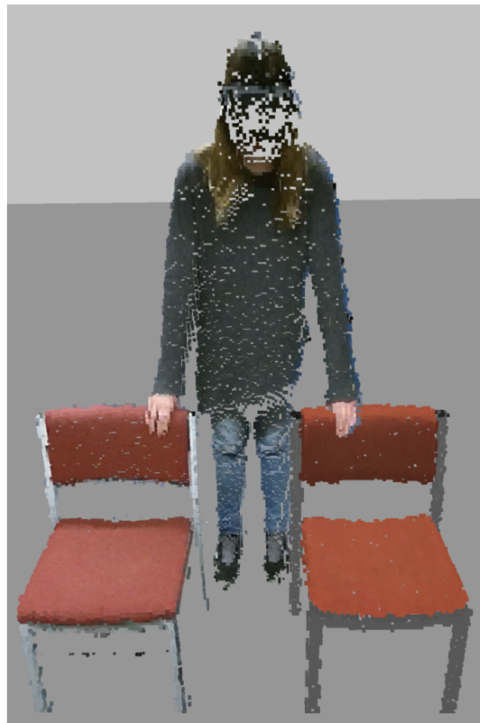

(b) internal view

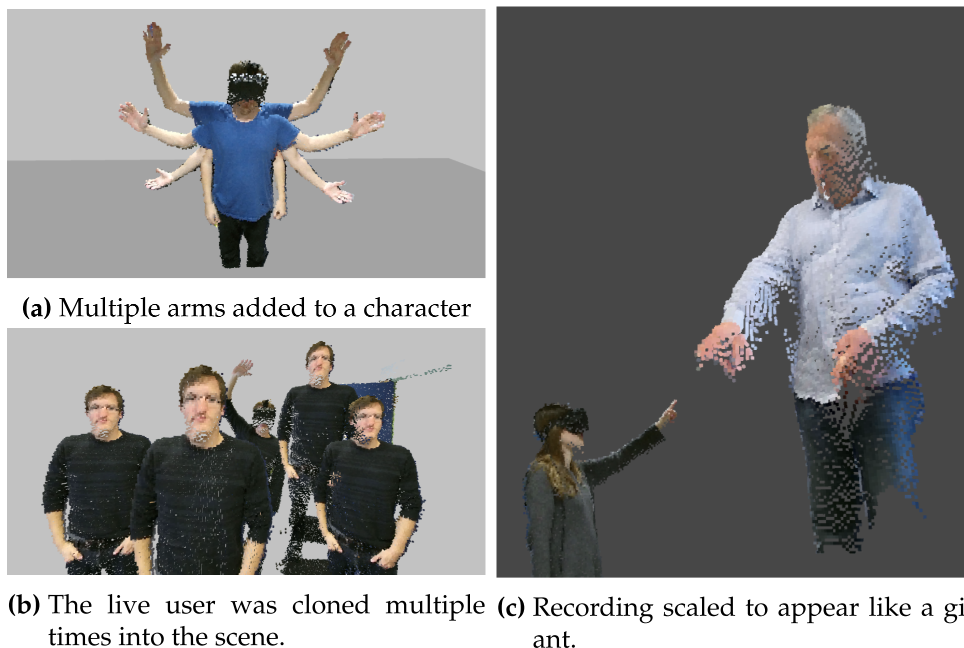
Figure 30: real chair(left) and virtual chair (right) with a user.

6 Recording and Replay

The second major component for the MREP system is the recording and replay of volumetric characters. We want to be able capture a user inside the interaction space, save the voxels that represent his body as clip and later replay it inside the voxelspace. This already give us the option to present a user a recording of himself or a different person inside of a coherent virtual environment. He can freely move around and watch independently from the original recording.
Further we want to structure the basic point cloud data, to include more than just position and color information. The information each voxel carries shall be enriched with data retrieved from the skeleton tracking. To achieve this we can use Joint positions provided by the Kinect skeleton tracking. Based on these and the Bone Structure described in 6.3 we can come up with a process, to collect the voxels a specific body part is represented by. This information can be used in a variety of ways, some of which have been already implemented in MREP, while others can be focus of future work. For example the separation in different body parts can be used to individually address and modify these. This includes individual modifications to position, size, color or even hiding a body part completely. This means we can identify and hide a specific limb from the users view and make it appear as if his limb is no longer there. The gathered body information is not limited to the live user. The separation into different body parts can also be stored when recording a character.
Once we have the body part information for live users and recordings alike, we can come up with ideas to manipulate the users body. Some of which are shown in 31. For example we could exchange his body parts with parts from a recording. Another example could be to scale a users body in different ways, this could give the user the impression of being very tall or much wider than he really is.
First in 6.1 we describe how to create a basic recording and in 6.2 how to replay it. To improve on this basic functionality we describe the process to assign voxels to body parts. In 6.3 the basic structure of the Kinect skeleton tracking data is described. Followed 6.4, in which we present different options to create bounding volumes around the body parts. And in 6.5 we describe some possibilities and drawback when handling separate body parts instead of the whole users point cloud as one.

6.1 Recording

To record a character, we have to capture and store the live stream from the depth sensor. We already have the BodyVoxelObject, which contains the live representation of the user. To record a characters movement we utilize the same structures. Body identification and background subtraction is



(a) Multiple arms added to a character

(b) The live user was cloned multiple times into the scene.

(c) Recording scaled to appear like a giant.

Figure 31: Different options to manipulate a recorded character

already done at this point, which allows us to only record the person independently from the surroundings.
A recording is stored as a series of frames that contain the voxel data of the captured person. Each frame is the direct exact copy of the current position on color data of the BodyVoxelObject. While we render the scene at 90 frames per second, the Kinect sensor only updates its buffers at a rate of 30 frames per second. It would not have any benefits to record the users at the higher frame rate. This would unnecessarily increase the file size. Thus we only add a new frame to the recording, when new data is received from the capturing application.
The recording size depends on the current voxel resolution, as higher resolution increase the overall voxel count of the scene. At a voxel size of 0.8cm this result in about 200 kB to 250 kB per frame. A second takes up about 7 MB and a 30 second recording adds up to about 200 MB. When the recording is finished the full set of frames is serialized and stored as a .binary file. We have not invested in compression technique for this yet. As the recorded clips are less than 30 seconds long we can store and load them without a significant delay.
This data is stored relative to the voxelspace origin and hence can be replayed at its original location, if no modifications are done.
When the voxel assignment process is active, each body part is also stored relative to its bone origin. For future version of the system this can be used

to allow more control over the recorded characters. For example to alter their appearance or exchange them with recorded data of a different person.

6.2 Replay

After recording a clip, as described in 6.1, we want to load and replay it. To load a clip it is attached to a VirtualBodyVoxelObject. The contained data is deserialized and converted into the usual voxel data structure of position and color information. At the moment the complete series of recorded frames is loaded and attached to the GameObject. While this requires considerable amounts of memory, similar to the size of the stored binary file, we opted not to implement a streamed loading process. As this might introduce inconsistencies in the loading process and is not necessary as long as free memory is available.
It is important to keep in mind, that the recorded data is only provided at 30 frames per second by the sensor. Thus we have to account for this in replay phase. While the rendering is fixed to 90 frames per second, it is not advice able to couple the playback to the actual render frame rate. Potential variations of the render rate could negatively impact the playback quality. Instead the playback is triggered on a separate timer, which adds the next frame of body data every 33.33 ms.
The GameObject with the attached recording can be manipulated at will. This means the clip can be replayed at an arbitrary position and orientation. But to keep a believable virtual world, only configuration possible with the current setup should be used. If the scene is only captured with one sensor, a recorded character is only then plausible if his voxels appear to be captured form this viewpoint.
The system also allows for basic playback controls during run time or through scripting. This includes pausing and resuming a playback as well as a looping after a complete series of frames is played back. To create a seamless loop, the recording has to be done accordingly. The first and last frame have to match each other as precise as possible. The playback can also be configured to start and stop at specific times. This allows us to create different kinds of scripted scenarios in which virtual characters enter and leave the interaction space or loop for example in an idle animation. These functionalities can be also be used in a supervised scenario where an operator controls the recordings to create interactions between user ans virtual characters.

6.3 Bone Structure

The body part distinction is based on Kinect v2 skeleton tracking. We use the tracked joints to create a bone structure that resembles the whole body

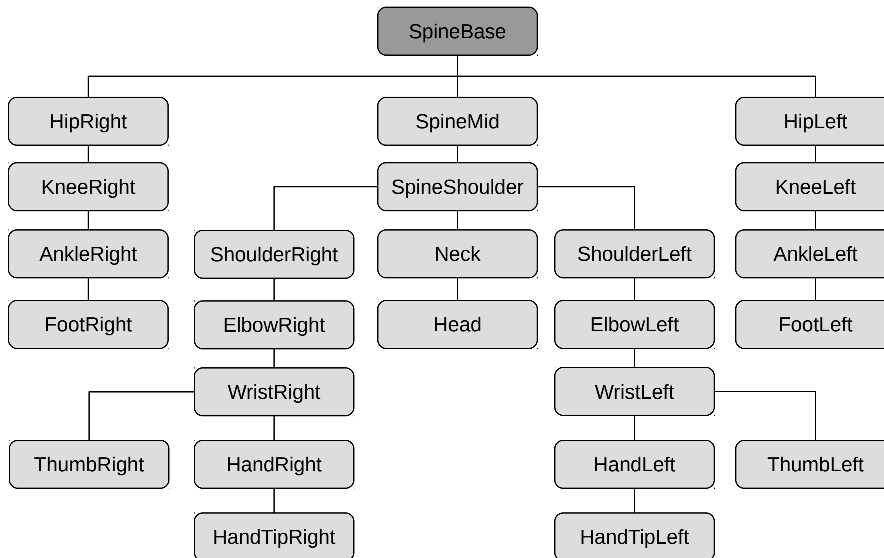


Figure 32: Hierarchy of Joints used by the Kinect SDK

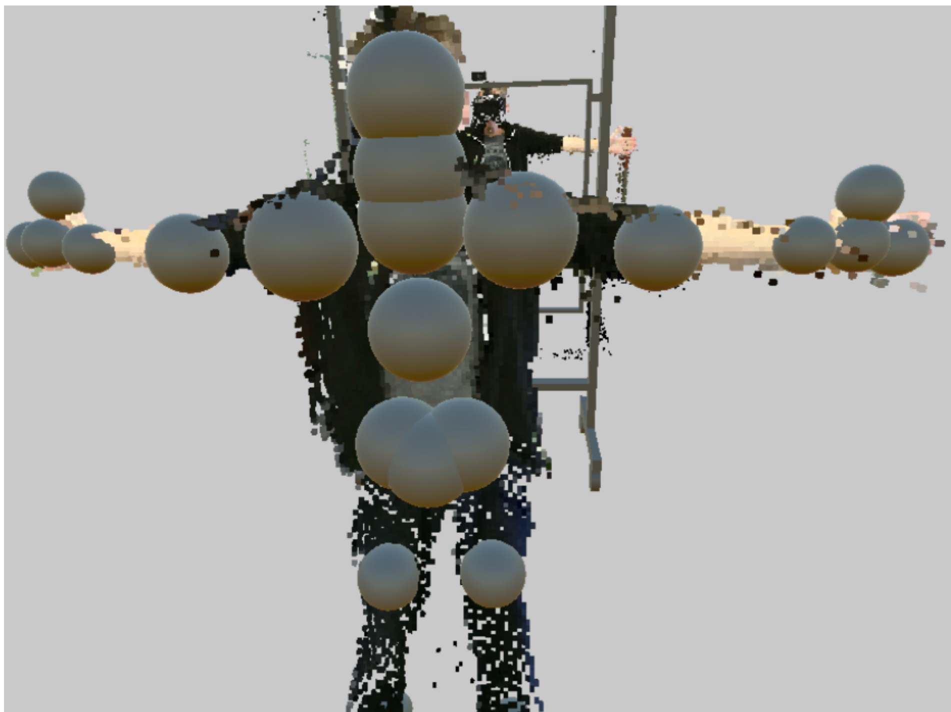


Figure 33: Skeleton tracking. Each tracked Joint is represented by a sphere

(a) Rotation of fore arm can be detected.

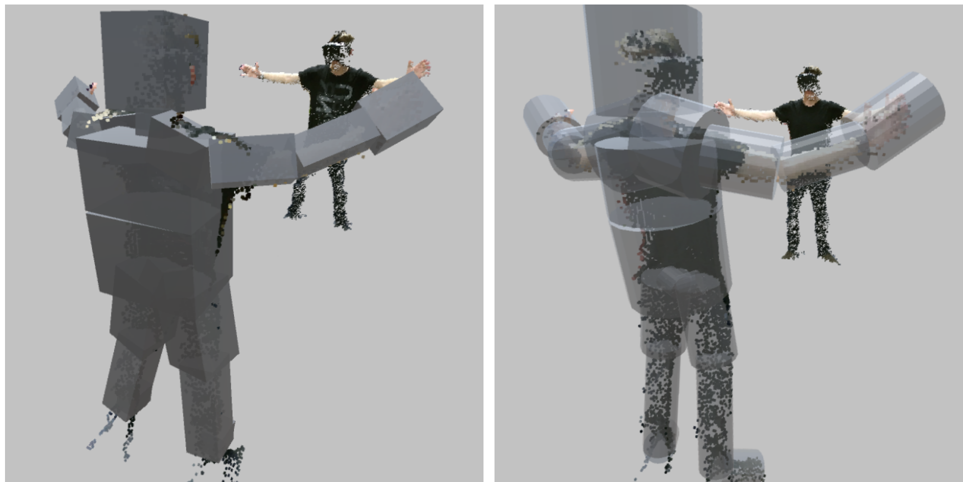
(b) No distinction between front and back side of the user is possible.

Figure 34: Bone orientation data visualized with colored cuoids.

of the user. These bones get individual bounding volumes to collect all voxels that are part of a specific body part. All voxels that are located inside a bones bounding volume are associated with this bone and structured accordingly inside the BodyVoxelObject for further processing.
In Figure 32 the hierarchy of the tracked joints can be seen. The SpineBase is the root of this hierarchy with other joints as child nodes. Based on this we define a set of bones or bodyparts.
The tracking data provides a lot of detail in some parts. For example the hands are tracked at multiple points, being: Wrist, Hand, HandTip and Thumb. We decided to bundle all these joint into a simpler Hand bone. The overall voxel resolution the system uses at the moment does not provide fine detail in this areas anyways. Separating the thumb from the hand would not add much detail at this point.
In additon to the position data, the sensor also calculates orientation data per joint. This can be used to visualize the rotation of different body parts, see Figure 34. While the system can detect the orientation of the users forearm (see Figure 34a) it can not determine whether the user is facing the camera or not (see Figure 34a). Due to these limitations and also relatively poor precision and stability of the provided tracking data, we opted to not use it for know.

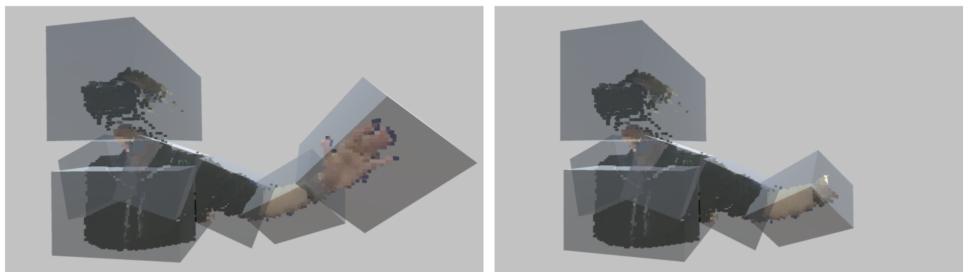
6.4 Bounding Volumes

As a first shape for the bounding volumes we tried cuboids, shown in Figure 35a. Their length is determined by the distance between their start and end joint. This is calculated for each new frame of tracking data. The width and height are fixed values per body part and have been manually set. A trade off between over and under fittings has to be made. Ideally this would happen dynamically as well, with the users height and width at certain points as parameters.



(a) cuboid bounds **(b)** cylindrical bounds

Figure 35: different shapes as bounding volumes per body part



(a) all parts **(b)** hand voxel turned off

Figure 36: body parts and corresponding geometry

To determine whether a voxel is located within a bounding volume we test its location against the normals of each cuboid face. If the position is in the rear hemisphere of each face, it lays inside the bounding volume. As it shows, a cuboid is not an ideal shape to approximate human body parts. A cylinder resembles basically all limbs and the torso more precise, as shown in Figure 35b. This results in less overlap, if body parts are close together. It also allows for a very fast test to determine the voxel as an inlayer or the bounding volume. The voxel positions distance to the cylinder axis is used in this case.

6.5 Separate Body Parts

Now we have a way to separate the voxels that make up the users representation into individual body parts. The before mentioned BodyVoxelObject

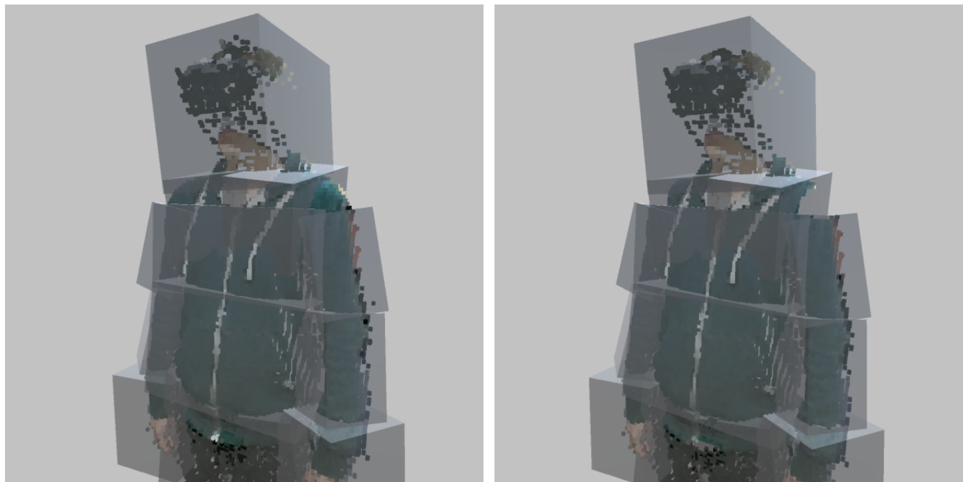


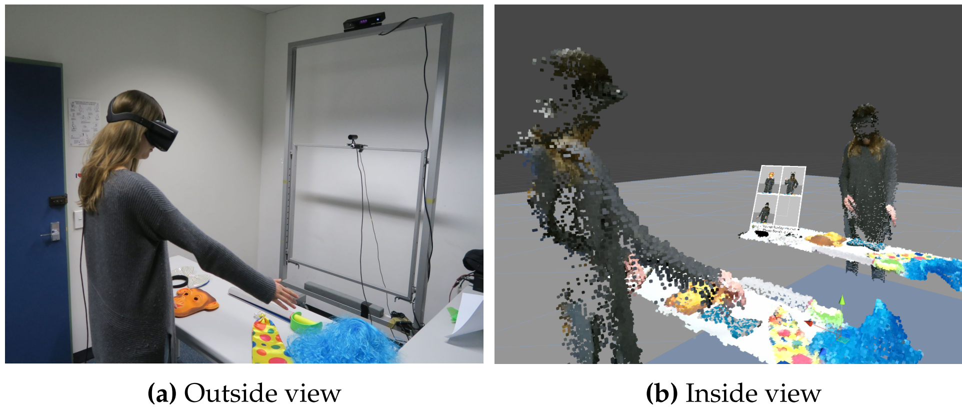
Figure 37: Joints need special treatment

features a child for each one of the body parts. If the separation process is enabled, the voxels will be assigned to the corresponding GameObject. This allows us to manipulate them individually, as they get their own transformation parameters. An example of this is shown in Figure 36. While in Figure 36a all parts are visible. In Figure 36b, the users hand has been disabled. This could be used to disable the users head and replace it with a version captured before the session, remove the HMD from his face. But obsiously the recorded head model would no longer correspond to the users actual facial expressions.
This process introduces some artifacts that have to be considered as well. First the dimensions of the bounding volumes have to be chosen appropriately. Otherwise parts can be missed when assigning voxels to body parts. Especially the junction of two adjacent body parts at a joint is a problematic area. As shown in Figure 37, the shoulder joint is not covered by any of the two adjacent bone volumes. This results in missing voxels and not properly connected transition between shoulder and upper arm. This could be solved by overlapping both adjacent bounding volumes and add the contained voxels to both body parts. This could fill up the gap and make both parts appear complete if watched on their own.

6.6 Summary

We have implemented basic recording and replay capabilities for character inside the voxel space. The functionality can be used to create different scenarios for a user to experience either himself or the recordings of others.
The process to determine relations between voxels and body parts has been

implemented using cylindrical shaped bounding volumes, as these most closely resemble human limbs. They are placed according to the skeleton tracking data to enclose the limb of a user and collect all the voxels that belong to this body part. While this works and can be used to individually address the body parts. Different limitations have yet to be addressed. These are mainly the size of each bounding volume, which has to be set manually at the moment. The other limitation is the adjacent area between bounding volumes. Overlapping volumes could be used to close these holes in the voxel representation.



(a) Outside view **(b)** Inside view

Figure 38: User in the photo booth application

7 First Application

After the basic implementation of MREP is done, we can create a first application. In the first place the application is meant to demonstrate the platform to an audience. But if it proves itself use able and stable, it can potentially be used for a first user study.
We defined a new set of requirements for this application, to make it use able for demos and studies. First it should demonstrate the basic mixed reality experience to a user, especially to users to may have never used any VR devices. It should show the main features, being the coherent rendering of virtual and real objects and the recording and replay of characters. The demo setup should work mostly self contained and without extensive supervision. This limits some of the functionalities as they require manual input from an operator, but these functionalities are not crucial to demonstrate the platform. The individual time each user spends with the system should be limited, to reach a wider audience. We also want to reuse the afore mentioned setup, as the frame on wheels has shown its benefits during development. For evaluation purposes we will record the first person view of the user, as it is presented to him in the HMD. We also added a camera that records an external view of the interaction space. This clearly shows the differences between real and virtual workd. The recorded material can be analyzed to collect noteworthy situation during a user session. A screenshot of a session recording is shown in figure 40
This first application will be described in the following.

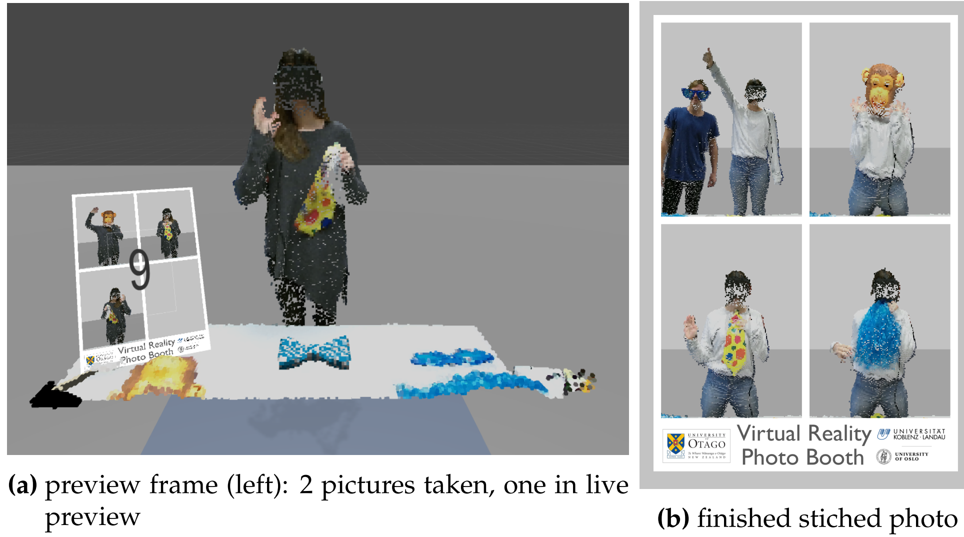
7.1 Design

Based on the above mentioned requirements, we came up with a concept. The Mixed Reality Photo Booth. This allows the user to step into a virtual photo booth, and take pictures of himself with a set of different props. We

chose a photo booth because it naturally encourages the user to interact with the presented props. Also it does not feel like a forced demo situation, but can be experienced without much introduction to the specific scenery. Figure 38 shows the photo booth application being used. An outside view of the user on the left and an inside view rendered with MREP on the right. The pictures are taken from an external camera, the user himself sees his first person perspective. He can see his own body, look and walk around.
On the table in front of him are several real props positioned, this can be seen in Figure 42a. In addition to the real props, two spots on the table are taken up by virtual props, see Figure 42b. As one can see the mustache to the far left and the bow tie in middle are only virtual and not present in the photograph of the table. Before each session the user is encouraged to use the props in his pictures. The idea is to observe the users interaction with the props. If the virtual props are accepted and recognized as objects that could be present in the real world, they will be approached as such. During playback of the session records we can see whether the user tries to interact with the virtual objects.
On a timer four photos are taken, from a separate camera position. These get stitched together and printed as a handout. In front of the user is a virtual mirror plane which exactly recreates the actual interaction spaces. This is used to give the user a view of himself and his appearance in the voxelized rendering. The only difference between the normal and mirrored voxelspace is the preview frame on the table. Figure 39a shows it in use, after two pictures have been taken. The preview frame gives the user the ability to position himself in the photo and review the already taken photos.It starts blank, but displays a live preview from the photo booth camera view. Which is a separate camera that takes the photos of the user. In Figure 39b a finished stitched photo is shown.

7.2 Photo Booth Application

The implementation of the photo booth application required some additional functionalities to be added to the basic framework. As we want to minimize the interaction from an operator we tried to automate the process as far as possible. This resulted in a system that automatically loops through different scenes, shown in 43. The Setup scene is used to do the inertially start of the system. The way Unity handles Scene transitions requires some Assets to be loaded beforehand to ensure short loading times. The IntroScene presents a Welcome message to the user and asks for his consent to be recorded for later evaluation.
As the user has no physical input devices like mouse or keyboard we opted for a gaze based navigation interface. A view ray is casted from the users hand position and can trigger buttons presented on the interface. The user has to look on a button for a set amount of time to activate it. This approves



(a) preview frame (left): 2 pictures taken, one in live preview

(b) finished stiched photo

Figure 39: different stages during the photo booth

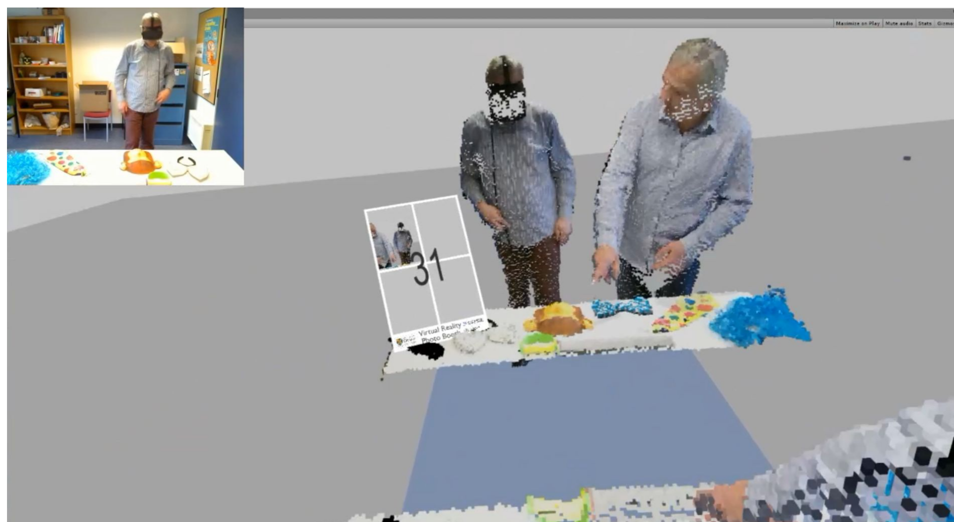


Figure 40: Screenshot of a session recording. The first person view of the user (big image). Overlay of the camera that records the interaction space(top left)

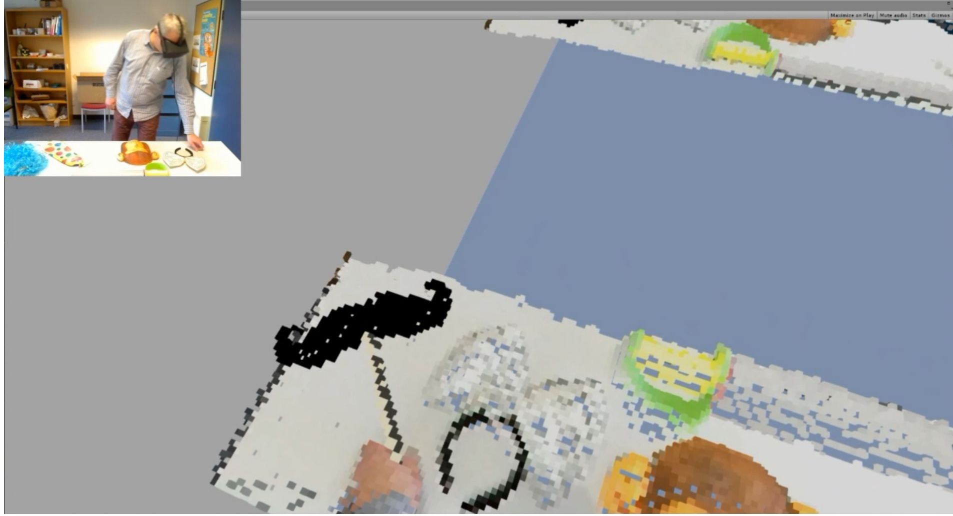
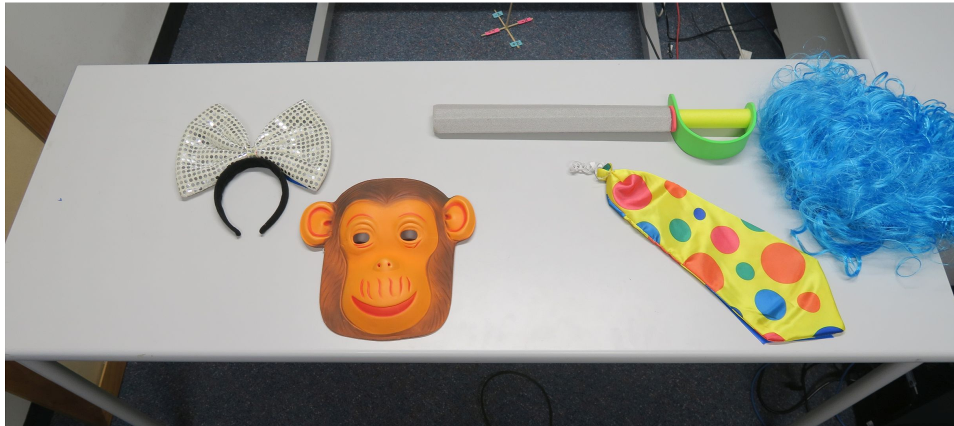
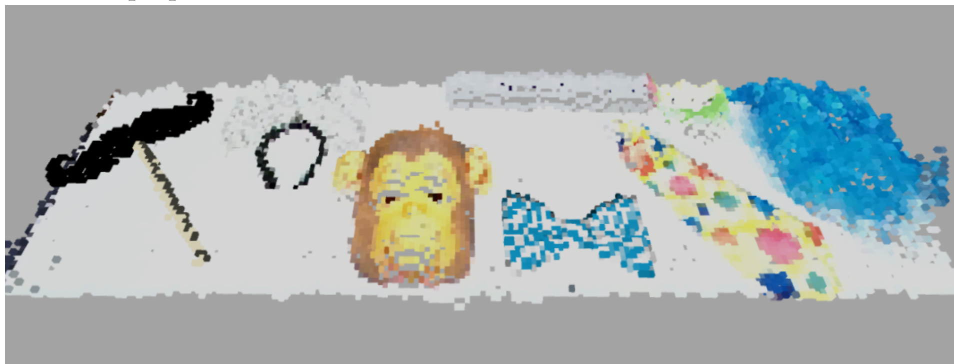


Figure 41: Screenshot of a session recording. The user tries to grab the virtual mustache prop(big image). The camera overlay shows that the prop is not actually on the table (top left)

or denies the recording and proceeds through the Scene loop process. Next one of multiple MainScenes is automatically selected. We constructed 10 different Scenes for the first exhibitions. These differ in the replayed recordings although different sets of virtual object would be also possible. We recorded more than 30 different virtual characters that are used in these Scenes. They feature character that walk through the interaction space, approach the user or point on the props. As an extra we recorded a character wearing a gorilla costume, that walks or jumps into the interaction space. We hope to provoke different reactions with these recordings and learn which could work best for a user study. Different Scenes could allow us in the future to present varying scenarios to a user group. With some of the Scenes used as a control group.
After all pictures are taken the user is transfered to the OutoScene. This thanks the user for his participation. After a delay, in which a new user can enter the booth, the loop will restart with the IntroScene.
To later analyze the user sessions, we record the first person view, that is displayed to the HMD user. In addition an external view, that shows the real environment. It helps to identify the attempted interactions with virtual props or virtual characters. We also record an audio stream from the camera to capture potential reactions of the user. We hope to use this collected data directly or indirectly for the following user studies.



(a) Photo of the real table. Only the real props are visible with two spots free for virtual props.



(b) Inside MREP. Real and virtual props are displayed on the table with not obvious differences.

Figure 42: View on the prop table for the photo booth application.

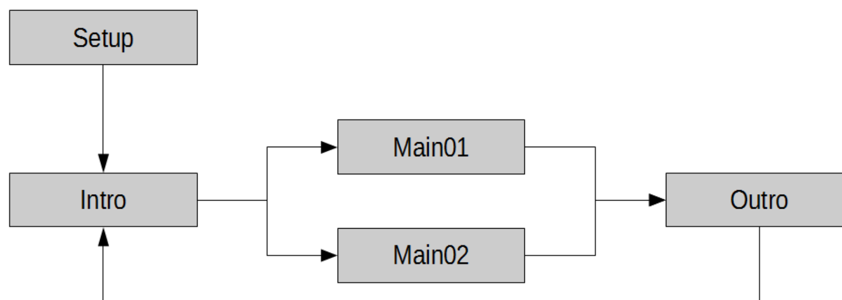


Figure 43: Scene structure for photo booth application. Each user proceed through Into, one of multiple MainScenes and the Outro. The loop will automatically restart for a new user.

7.3 Observations

The application, as described in 7.1, has been demonstrated on two exhibitions and several smaller demo sessions. Overall more than 150 sessions with unique users have been conducted and recorded. Cause by the very uncontrolled environment during most of these session, the recorded data is of varying quality. Thus we omit the formal analysis of these recordings. Instead we will present the observations we made during the presentations and collected from the recordings. Due to the informal nature of these results and to further investigate the effects of the presented system, a formal study will be conducted in the near future.

7.3.1 Platform Experience

First of all we can say that the users very much enjoyed the experience and most of them seemed engaged by the photo taking process. For many of them this was the first use of any kind of virtual reality application. Many of them did not react to the ability to see their own body at all. This was unexpected for someone used to virtual reality, as most application do not feature any kind of representation of the users body. We expected this to be a more prominent advantage of our system, but it seems like users completely new to virtual reality simply expected to see their own body. Users hat have used virtual reality applications before noticed the representation of their own body as a feature. The main criticism was noted about the very noisy and incomplete reconstruction of the users hands. This is mainly caused by the positioning of the depth camera. When the user hold this hand in front of this face to look at them, he effectively blocks the sensor from seeing the side facing the user. This results is a representation that is recognized as their own hands, but mostly due to the accurate movements of both real and virtual hands. A feeling of body ownership is likely achieved. But a way to improve the representation of the users hands would be beneficial. A lightweight head mounted depth sensor could be a good solution as it always faces in the users view direction and is able to see the users hands in front of his face. But has to offer a very small minimal distance at which the sensor functions. Alternatively the use of multiple externally mounted sensors could improve this situation, although there are most likely configuration in which even multiple sensor can not see an area of interest that the user is looking at.
An important detail about the used hardware is the change from the Oculus DK2 to CV1. The higher resolution screen and higher refresh rate make a very noticeable difference when using the system. Even though the graphical fidelity offer low details, the higher resolution makes the experience more comfortable and enjoyable. But the biggest improvement is caused

by the ability to completely turn around and look in the opposite direction of the tracking camera. In contrast to the DK2 headset, the CV1 headset features tracking LEDs at the back of the headset. This allows for a reliable tracking at almost all orientations of the HMD. The tracking camera also offers a wider field of view and greater maximum distance from the sensor. This enables bigger interaction spaces which can be reliably tracked. An even bigger interaction space would be possible by using multiple tracking cameras, which is supposedly supported by the SDK though we did not try this our selfs.

7.3.2 Virtual Objects

The integration of the virtual objects also works as expected. Even tough the user can see the table and the real props before putting on the HMD, observations have shown that nobody identified the virtual objects. Users seem to accept them as part of the scene. Basically all users tried to grab one of the virtual props and reacted noticeably surprised when they realized it is not really there. Interesting is also the fact that even after being aware of the existence of virtual items in the scene, they still can not identify the second virtual prop. Most of them will still try to interact with it like they did with the first. Only after detailed analysis of the scene some users can point out the virtual objects. In most cases this was accounted to differences in the lighting between real and virtual object. The virtual object appeared either to bright or to dark compared to the real ones. As mentioned in 5.1.2 we expected it to be very difficult to simulate an accurate lighting situation with the simple techniques we used. The lighting configuration also needs to be adapted specifically to every new setup and lighting situation. Exhibitions that ranged over multiple ours required changes to the virtual lighting to adapt to the changed real world lighting during the day. Thus a more advanced lighting system that takes the environment lighting into account could improve the coherence between real and virtual object.
Still the virtual objects are clearly perceived as a part of the virtual environment. It feels like a lot of this is caused by the life experience of the users. The objects appear real enough to trigger known behavior instead of skepticism that would the user evaluate his actions more closely. A chair that I can clearly see in front of me, must be meant to sit on. Even if it was not there seconds ago, before I stepped into the virtual environment.

7.3.3 Recorded Characters

The replays of recorded characters are more difficult to analysis. The users reacted very different to them. Some got scared, wile others showed no reaction at all as if they did not notice the character at all. It is hard to say how

this wide range of reactions is caused. The users who seemingly did not notice the character may have mistaken it for a real person. While this would speak in favor of the system, we have no way of proving this after the fact. Due to the uncontrolled setup it would have been possible for someone to just walk into the interaction space and thus appear inside the virtual environment. While this would have been plausible for the recordings of normal humans, the appearance of the gorilla recording should have caused some note able reaction. Unfortunately many users did not complete the full session and thus did not see the human recording and the gorilla. This would have allowed us to combine both reactions an eventually spot a difference and conclude a result. And lastly it is also possible, that the users just were to engaged with the props and photo process, that they did actually not notice to appearance of a recorded character.
We observed several occasions that one can categorize as social presence induced by the virtual characters. We users felt like the recorded characters stepped to close to them. They felt uncomfortable in the very close proximity of the recordings and stepped out of the way of an approaching recorded character.

7.4 Technical Results

From a technical perspective the Photo Booth test application, proved it self as stable. The more than 150 test sessions did not produce crashes or unexpected behavior. Some technical observations regarding different parts of the hardware are described in the following.

7.4.1 GPU Utilization

The systems mentioned in 3.5 are capable of running the application with a constant frame rate of above 90 frames per second. This frame rate is necessary for the Oculus CV1 to produce a good experience with stable tracking and without stutter or noticeable latency for the head movement. The Kinect v2 sensor data can be processed with low enough latency to give a believable and functional representation of the users body. When using the system for extended periods of time, no additional strain is produced. It fells comparable to other VR experiences, using the same hardware.

7.4.2 CPU and Memory Utilization

For exhibition purposes both the capturing and rendering application were executed on only one system. This does in increase the system load compared to the usage of distinct capturing and rendering computers. The

overall CPU load for the Quad-Core CPU stays below 40% with RAM utilization reaching up to about 9 GB of the 16 GB of the system memory. The capturing application only makes up a small portion of this. With about 10% CPU load, including the load produced by the Kinect SDKs KinectService executable. Memory usage is also low, at less than 200MB. Compared to the rendering application, the capturing application does not produce significant load in the current implementation. The CPU and RAM load produced by the capturing application stays consistent for different configurations. Increasing the dimensions or resolution of the voxelspace does not affect its complexity significantly, as the size of the processed raw data is fixed based on the used Kinect sensor.
The rendering application on the other hand is affected by these changes. An increased voxel resolution results in more voxels that need to be processed and rendered. Up to the point when the voxel resolution surpasses the depth sensors resolution. At this point a higher voxel resolution does not produce more actual voxels, as all depth values are already represented by a voxel. Consequently the mentioned frame rate targets can the achieved for all sensible configurations of the system. It is important to keep in mind that optimization has not been a focus for the current implementation. Especially the memory usage can most likely be reduced, if necessary in the future.

7.4.3 Network

The 1 GigaBit/s connection is being saturated by the current system, that only uses one Kinect sensor. There are two major factors that effect the used network bandwidth. First is the resolution of the voxel space. The scene in 18 consists of about 35,000 voxels for the 256^3 resolution and about 90,000 voxels for the 512^3. Second the structure of the environment that is captured. A free standing user has a relatively small footprint on the depth image and thus requires less voxels to be transmitted per frame. If the voxelspace is configured to include the floor or multiple walls the scene can contain 300,000 voxels or more.

7.4.4 Scene Details

During usage with multiple virtual objects and multiple character replays we observed total voxel counts of up 500,000 voxels. It is important to note, that this includes the mirrored scene. Even in these situation we stay within the frame rate limits. At the moment only static virtual object are possible. This is due to a scene ray cast taking to long to achieve the constant high frame rate. But static virtual objects only add their visible voxels to the scene after being voxelized. This means that several virtual object can be

added to the scene simultaneously. This is only limited by the total amount of voxels in the scene.

8 Conclusion and Future Work

8.1 Future Work

The developed framework and demonstrated first application are a first step for further research. This needs to be evaluated to get first usable results for the further progress. Some aspects we had in mind for a full featured system are still to be implemented.

8.1.1 Platform Setup

The first and probably most noticeable improvement for a future system, is to use multiple Kinect v2 sensors. This would enable true 360°capturing and recording of characters. The underlying system was designed with multiple sensors in mind. A system with manually aligned point clouds from multiple sensors should therefore be not to difficult to implement, but give a good impression of such a system. Although an automated calibration system would most likely be necessary to achieve precise alignment of multiple sensors. Even for a single sensor an automated calibration process without the need of manual measurements of the sensors transformation would be beneficial for an easier and faster setup in different locations.
There are several other improvements that could increase the complexity of the capturing application. For example preprocessing of the raw depth or color data, to reduce the visible noise as well or fill in missing voxels by interpolation. At this point it might be beneficial to distribute the load from the rendering computer to the capturing computers.
The next step to a fully immersive system would be the interaction with virtual objects. This would require to collect more information about the real world. Spatial information, is not sufficient to. Another interesting aspect would be to allow manipulation of real world objects. By detection and segmentation it could be possible to manipulate a real object.
The first version did not focus on maximizing the graphical fidelity, instead to find the minimal required level of realism. If this changes in the future there are several points that allow for improvements. These are mainly, a higher voxel resolution, more stable voxels on a static surface and the filling of missing voxel data. In places where multiple depth values are captured for one voxel, all color information used instead of only the first value.

8.1.2 Virtual Objects

The virtual objects are now represented in a way that makes it hard for the user to distinguish them from real objects. But there are not yet all sensor characteristics and environmental effect simulated. The most noticeable effect is the missing occlusion between both worlds from the perspective of

the Kinect sensor. But as stated in 5.2.3, the depth image obtained by raycasting the virtual objects could be compared to the depth image captured by the Kinect sensor and discarding voxel by a simple depth test. But to do so there are some changes needed in the implementation and it has to kept in mind that this might not work for dynamic virtual objects or virtual persons. This would require a new raycast each frame which is far to expensive to be executed in real time with the current implementation.
It could be interesting to create a much more detailed virtual environment that extends beyond the current interaction space. While characters and object are not directly reachable, this could create a very interesting environment to test the impact of several virtual characters and objects.

8.1.3 Recorded Characters

The current character recording system, can be extended to allow more advanced manipulations of the live users representation as well as the replayed recordings. First this would include to implement the same voxel assignment process currently used on the live users, for the recordings. This would result in recorded characters with already seperated body parts. These body parts could than be manipulated or used individually. For example to create different body experiences, by assigning recorded body parts to the live users representation.
If bigger scenes with more than a few simultaneous replays are desired. A compression oder streamed loading technique could result in greatly reduced file sizes and RAM usage.

8.1.4 Study

While we discussed observations and technical aspects of the system. A detailed and controlled user studies has yet to be conducted. The study can hopefully help to understand the psychological effects the use of such a Mixed Reality application has. Such a study is currently in preparation and will be conducted as a cooperation between the Information Science Department from the University of Otago and the Department of Psychology at the University of Oslo. The latter will provide the psychological expertise to correctly analyze the results.

8.2 Conclusion

We have presented a framework called the Mixed Reality Embodiment Platform. It presents a coherent mixed reality environment to the user, by capturing him and his surroundings with a depth camera. The captured scene is reintroduced into the virtual environment and presented alongside fully virtual objects and characters. Looking at 1.2 and the goals we

set for the first version of the platform, we have reached these goals. The intend was to create platform that enables further research in the area of mixed reality applications and can be extended on in the future.
The platform was tested on multiple occasions and with big user groups for general functionality and stability. From our observations we can says, the anticipated effects of presence and embodiment have been achieved. But especially these non technical aspects have to be further investigated by a user study which is planned for the near future. We hope to produce further insight in the world of Mixed Reality and enable functional applications based on this research.

Appendices

A MREP Instructions

Hardware and physical Setup:

Hardware Setup:

- The PC is set up as usual:
  - Oculus with HDMI and USB 3.0
  - Oculus Camera with USB 3.0
  - Kinect USB 3.0 + Power
  - Webcam in USB 2.0
  - Printer USB 2.0 + Power (don't forget to turn it on)
- User is the usual ".\cmrl" login

The physical setup:

- Kinect:
  - Height: preferably higher than 2m
  - Angle: tilt it forward as far as possible
- Oculus tracking camera:
  - Position: In one line under the Kinect
  - Height: 1.4 - 1.6m should work
  - Angle: should point straight forward
- Webcam:
  - Height: same level as the oculus camera
  - Angle: tilted so the whole table with props is visible
- Table:
  - Centered in front of the Kinect
  - we used a distance of about 1.45m to the front of the table

The Project:

- Everything is located at: C:\KatArMaster\Projects\MREP
  - Resources contains most of the pictures etc.
  - contains the actual source code and Unity project

Change the size and resolution of the voxelspace:

- <u>Change the settings on the capturing side</u> (KatArMaster\Projects\MREP\KinectCapture) in KinectCapture.h
  - spaceWidth, spaceHeight and spaceDepth determine the number of voxel in each direction
  - voxelSize is the size of one voxel in meters (changing this requires adjustments in the GeometryShader as well)
- <u>Change the settings on the rendering side</u> (KatArMaster\Projects\MREP\VoxelRender)
  - The same values are found in the MREP Manager Script, they are changeable in the editor, selecting the MREPManager Prefab or GameObject of the Scene
- Make sure the values are the same on both sides

Adjusting the Positioning (Look here if the output looks wrong):

- First check the devices are correctly aligned as mentioned above
- This may need some tinkering as not all these alignments and measurements are 100% accurate

Change Kinect position:

- All parameters are found in KinectCapture.h in the KinectCapture project
- you can change the *height* and the *angleX* of the kinect manually or use the flooplane approximation of the sensor by changing *useFloorPlane* to true

Change Oculus Tracking camera position:

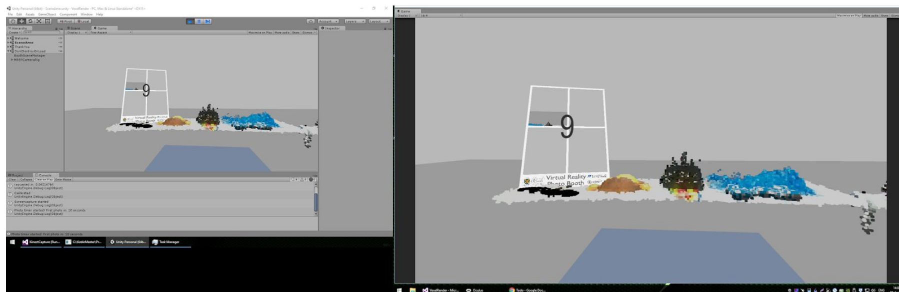
- The MREPManager GameObject has parameter for the Position of the oculus tracking camera
- The camera should always be at 0° tilt and facing the user.

Change the mirror plane:

- Change the parameter Mirror Offset of the MREP Manager Script
- it is available in the editor, selecting the MREPManager GameObject. If the offset is 0 the mirror plane will be at the x/y plane of the world coordinate system.

Using the Unity project:

General:

- On a dual monitor system(highly recommended), configure the Unity Editor like this:
- 

    - This keeps the Game view on the main monitor to be recorded with Shadowplay
    - It also allows to view the Scene View without the rendering pausing
    - Right click the tab handle and click *Add Tab -> Game*, maximise this on the main screen and use the Unity Editor and the secondary screen

- To edit GameObjects present in multiple Scenes, e.g. parameters in the MREP Manager or the position of a virtual Object, use the respective Prefab to change all instances at once

Add a new scene:

- You can use the scene BasicMainScene as a starting point
- Open the scene and make some changes
- Save the scene with a new name
- If you want to use the scene in the photobooth setup don't forget to add the scene to the build manager. Open File -> Build Settings and click "Add open Scenes". After that switch to the Setup scene and add the new scene per drag and drop to the Main Scenes Assets field of the BoothSceneManager

Change the lighting:

- We use tow different light sources: the ambient light of the scene and a directional light from above. You might have to change the intensities to match your real world light situation. The ambient light, found under Window -> Lighting is set for each scene independently. There is a Prefab for the directional light called LightCeiling, change the intensity of the Prefab to get the same light in every scene.

Add a virtual object:

- Create or get a 3D model of the object you want to add to the scene, it needs to provide vertex colors, we are using .fbx models, but any other format that works with unity and stores vertex colors should be fine (tested with .dae)
- Import the model to Unity, in the editor simply drag the file into the model folder
- Add a VirtualVoxelObject to the Scene, you find it in the Prefab folder, drag it on the hierarchy field of the editor (you need one VirtualVoxelObject for each model you want to add)
- Find the model you want to add in the Model folder and drag it on the VirtualVoxelObject to make it a child of it
- Tag the model with VirtualModelObject
- To transform the model, always change the transform of the model itself not the VirtualVoxelObject

Record a Recording:

- A Recording will always record the current BodyVoxelObject in the Scene. It won't record any background or virtual objects.
- You can start the Recording by pressing C.
- You stop the Recording by pressing X.
- Every frame in between will be stored in a .binary file at Assets\Recordings.
- If you want to store only one frame (Snapshot) you can press Z.

Replay a Recording (virtual person):

- Add a VirtualBodyObject to the Scene, you find it in the Prefab folder, drag it on the hierarchy field of the editor (you need one VirtualBodyObject for each model you want to have at the same time, it is possible to change the played recording in a VirtualBodyObject at runtime).

- Find the file with the recording you want to add in the Recordings folder and drag it on the File field of the VirtualBodyObject
- With the Play Recording toggle you can choose whether you want to play the recording or not. At runtime it will pause a running recording.
- With the Start Delay time you can set the time in seconds till the recording will start to play. Note that this time will also delay the start of a paused recording at runtime. If you want the recording to start instantly set the Start Delay time to zero.

Photo Booth:

- To enable/disable the photos and the printer you have to change the BoothCamera object in the scene or the prefab for all scenes. The object has a Booth Manager component where you can toggle Booth Mode and Print Photos. You can also change the time between the pictures.
- For further information check the Checklist Science Festival document

Keys:

- O: calibrates the positions of the Oculus (should work automatically)
- V: toggles on/off the bounds of the voxelspace
- T: toggles on/off the bounds of the tracking frustum of the Oculus
- K: toggles on/off the bounds of the frustum of the Kinect sensor
- S: takes a screenshot of the Game view and stores it VoxelRender/Screenshots
- F: takes a screenshot from the position of ThesisCamera
- P: pauses the updates of the Kinect data
- B: toggles on/off the boundary shapes of the body parts
- A: assigns voxel to the body parts
- Z: stores a snapshot of the users body
- C: starts the recording of the users body
- X: stops the recording and stores a .binary file with the captured data

Starting the Application:

- Start KinectCapture.exe (always before the rendering)
- Start the Setup scene in the VoxelRender project
- Select the Setup scene for the Booth Expierence
- Press J to activate the first welcome screen or use the HMD to select the Start button.

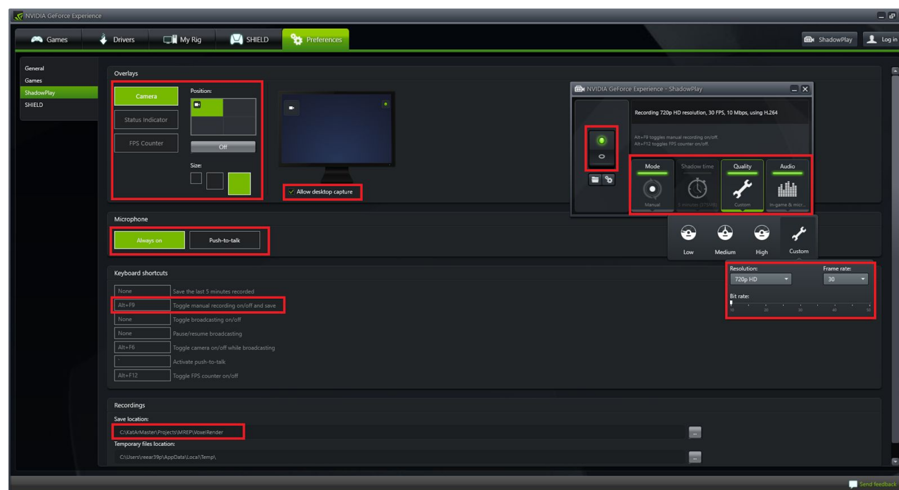
- You can also use the VoxelRendering Scene
    - It contains all the features implemented including the bone boundaries etc.

Instructions Video:
https://drive.google.com/open?id=0B20RQIZ1nTJ1QlIFNlJvRTJfNjA

- Open and run KinectCapture 00:04
- Open and run VoxelRender 01:02
- Check Oculus devices 01:28
- Asset structure 01:47
- GameObjects of the Scene 02:10
- add/change a Recording 03:01
- Manipulate virtual objects 03:25
- Add new virtual objects 03:53
- Change the Lighting 04:53
- Add a second Game View 05:31
- Runtime 05:52
- Create a new Scene 07:09

Screen and external view recording:

- uses Nvidia ShadowPlay, which requires the NVIDIA GeForce Expierence Software to be installed
- a screenshot of the settings:



- Hotkey to start/stop recording must be Alt+F9
- “Allow desktop capture” must be enabled
- “Camera” is the webcam used for external view recording
- To record external sounds “Microphone” must be always on, it uses the default recording devices set in the Windows settings, select the Webcam there
- The default quality settings create big files, the lowest setting for framerate and bitrate are sufficient
- ShadowPlay always captures the main display

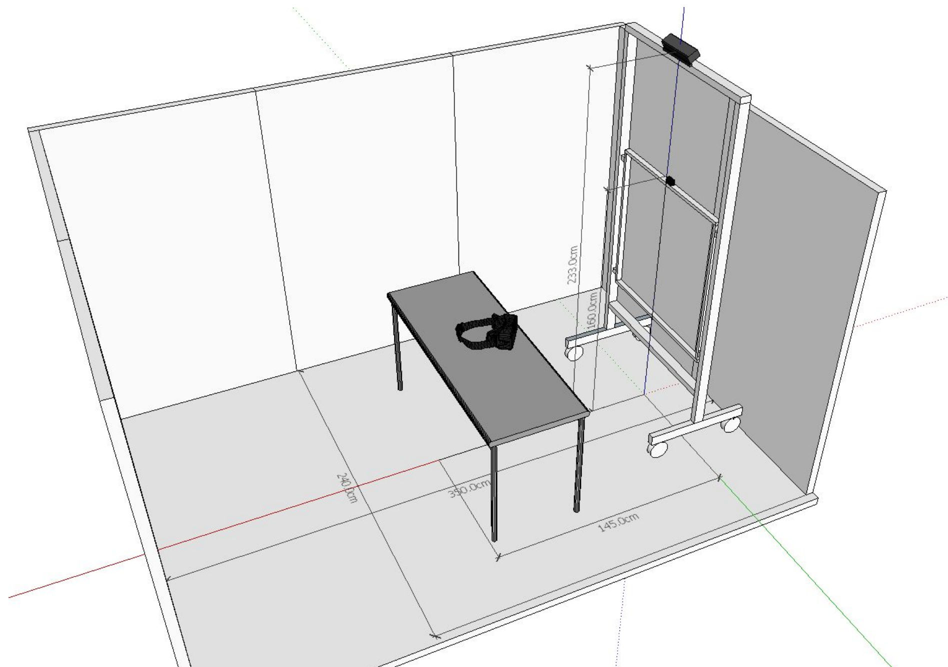
B PhotoBooth Instructions

Hardware Setup:

- ❏ The PC is set up as usual:
  - ❏ Oculus with HDMI and USB 3.0
  - ❏ Oculus Camera with USB 3.0
  - ❏ Kinect USB 3.0 + Power
  - ❏ Webcam in USB 2.0
  - ❏ Printer USB 2.0 + Power (don't forget to turn it on)
- ❏ User is the usual ".\cmrl"

Physical Booth Setup:

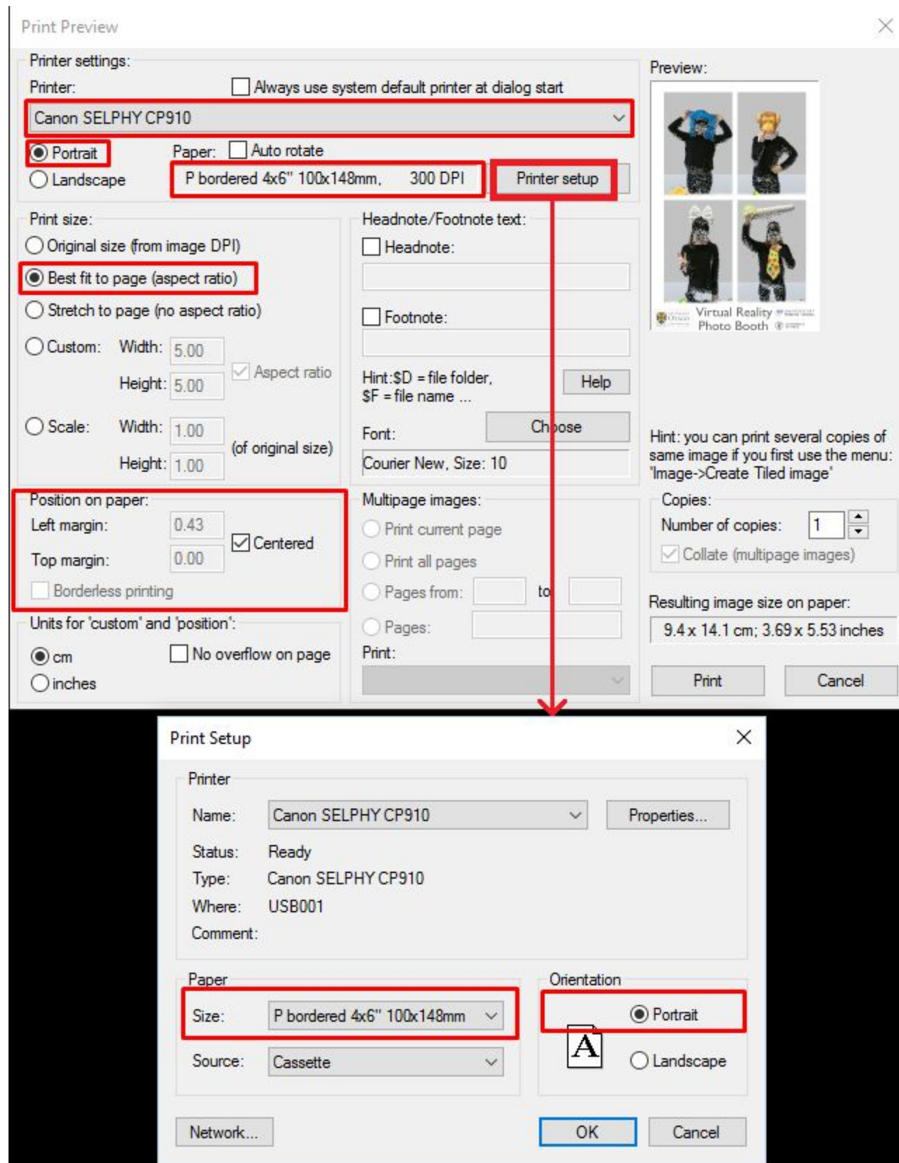
- ❏ The Booth should look somewhat like this:



- ❏ The Booth should be at least 1.5m wide and 3m deep.
- ❏ The front of the table should be at 1.45m from the front of the frame
- ❏ The table should be centered in front of the frame
- ❏ Make sure the Kinect is tilted at 30 degrees (if the voxelspace seems a bit off, try to change the height and angle in the KinectCapture.h (KatArMaster\Projects\MREP\KinectCapture))
- ❏ Align the webcam, so that it captures the whole table including the sword prop
- ❏ The props should be placed on the respective outlines

Printing Settings:

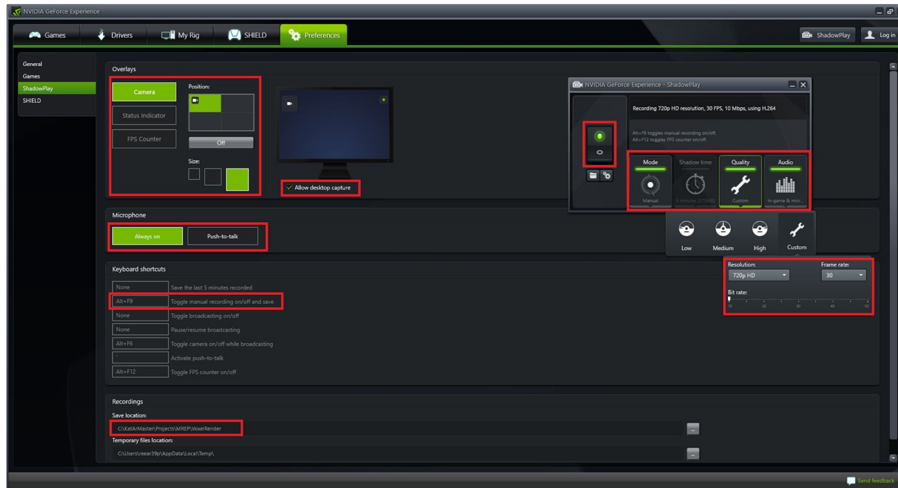
- ❑ Open Irfanview and check the Print Settings, it should look like this:



- If you need to change the settings, turn off the printer and click print to save the new settings. Afterwards remove the pending print job and start the printer again.

Screen recording with Nvidia Shadowplay:

- Setting can be found in the Nvidia Experience and should look like this:



In Unity:

- Open the VoxelRender project in Unity (KatArMaster\Projects\MREP\VoxelRender)
- Select the BoothCamera prefab (in the prefab folder) and make sure that the Print Mode flag is set. Check the times for initial photo delay and photo delay. Initial photo delay should be 60, photo delay 20 (all found in the Booth Manager component).
- Set Unity to “Maximize on Play” and do not open any other windows, because the screen will be recorded
- Make sure the Setup scene is selected

Starting the Application:

- Start KinectCapture.exe
- Start the Setup scene in the VoxelRender project, press J to activate the first welcome screen or use the HMD to select the Start button.

The Printer:

- Don’t forget to turn it on
- Each set of paper lasts for 18 prints
- Each Ink Cartridge lasts for 36 prints

Notes:

- ❏ If something breaks or stutters, restart Unity
- ❏ Keep an eye on the folder for the screen recordings “VoxelRender/Desktop/”, we might fill up our SSD quickly. Maybe transfer some files on a different hard drive?

C Pictures

Figure 44: Exhibition setup, Koblenz, Germany, 08.07.2016,

Figure 45: Setup in Oslo, Norway, 25.08.2016

Figure 46: Exhibition setup, Dunedin, New Zealand, 09.07.2016

Figure 47: Exhibition setup, Dunedin, New Zealand, 09.07.2016

References

[AM01] Tomas Akenine-Möller. Fast 3D Triangle-Box Overlap Testing. 2001.

[BBB+12] Amira Belhedi, Adrien Bartoli, Steve Bourgeois, Kamel Hamrouni, Patrick Sayd, and Vincent Gay-bellile. Noise modelling and uncertainty propagation for tof sensors. pages 476–485, 2012.

[BC98] M Botvinick and J Cohen. Rubber hands 'feel' touch that eyes see. *Nature*, 1998.

[BF15] Stephan Beck and Bernd Froehlich. Volumetric Calibration and Registration of Multiple RGBD-Sensors into a Joint Coordinate System. In *3D User Interfaces (3DUI), 2015 IEEE Symposium on*, pages 89–96, 2015.

[BKKF13] Stephan Beck, André Kunert, Alexander Kulik, and Bernd Froehlich. Immersive group-to-group telepresence. *IEEE transactions on visualization and computer graphics*, 19(4):616–625, 2013.

[BSRH09] Gerd Bruder, Frank Steinicke, Kai Rothaus, and Klaus Hinrichs. Enhancing presence in head-mounted display environments by visual body feedback using head-mounted cameras. *2009 International Conference on CyberWorlds, CW '09*, pages 43–50, 2009.

[CGMS16] Andrea Corti, Silvio Giancola, Giacomo Mainetti, and Remo Sala. A metrological characterization of the Kinect V2 time-of-flight camera. *Robotics and Autonomous Systems*, 75:584–594, jan 2016.

[CNSD93] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality. *Proceedings of the 20th annual conference on Computer graphics and interactive techniques - SIGGRAPH '93*, pages 135–142, 1993.

[EKH+02] P M G Emmelkamp, M Krijn, A M Hulsbosch, S de Vries, M J Schuemie, and C A P G van der Mast. Virtual reality treatment versus exposure in vivo: a comparative evaluation in acrophobia. *Behaviour research and therapy*, 40(5):509–16, may 2002.

[FBS05] J. Fischer, D. Bartz, and W. Strasser. Stylized augmented reality for improved immersion. In *IEEE Proceedings. VR 2005. Virtual Reality, 2005.*, pages 195–325. IEEE, 2005.

[FCB+06] Jan Fischer, Douglas Cunningham, Dirk Bartz, Christian Wallraven, Heinrich Bülthoff, and Wolfgang Straßer. Measuring the Discernability of Virtual Objects in Conventional and Stylized Augmented Reality. *Proceedings of the Eurographics Symposium on Virtual Environments (EGVE 2006)*, pages 53–61, 2006.

[FSR+13] Andrew Feng, Ari Shapiro, Wang Ruizhe, Hao Li, Mark Bolas, Gerard Medioni, and Evan Suma. Rapid Avatar Capture and Simulation Using Commodity Depth Sensors. page 2006, 2013.

[GJRGMS+15] H. Gonzalez-Jorge, P. Rodríguez-Gonzálvez, J. Martínez-Sánchez, D. González-Aguilera, P. Arias, M. Gesto, and L. Díaz-Vilariño. Metrological comparison between Kinect I and Kinect II sensors. *Measurement*, 70:21–26, jun 2015.

[IFI] IFIXIT. Xbox one kinect teardown. `https://www.ifixit.com/Teardown/Xbox+One+Kinect+Teardown/19725`. viewed on: 07.09.2016.

[JL06] Katrien Jacobs and C??line Loscos. Classification of illumination methods for mixed reality. *Computer Graphics Forum*, 25(1):29–51, 2006.

[KBF+95] Wolfgang Krüger, Christian-A. Bohn, Bernd Fröhlich, Heinrich Schüth, Wolfgang Strauss, and Gerold Wesche. The responsive workbench: A virtual work environment. *Computer*, 28(7):42–48, July 1995.

[KE14] Andreas Kalckert and H Henrik Ehrsson. The moving rubber hand illusion revisited: comparing movements and visuotactile stimulation to induce illusory ownership. *Consciousness and cognition*, 26:117–32, may 2014.

[KM10] Georg Klein and David W Murray. Simulating low-cost cameras for augmented reality compositing. *IEEE transactions on visualization and computer graphics*, 16(3):369–80, jan 2010.

[KS14] Elena Kokkinara and Mel Slater. Measuring the effects through time of the influence of visuomotor and visuotactile synchronous stimulation on a virtual body ownership illusion. *Perception*, 43(1):43–58, jan 2014.

[LKA+04] Sang-Yup Lee, Ig-Jae Kim, Sang C Ahn, Heedong Ko, Myo-Taeg Lim, and Hyoung-Gon Kim. Real Time 3D Avatar for Interactive Mixed Reality. i, 2004.

[LVG+13] H Li, E Vouga, A Gudym, L J Luo, J T Barron, and G Gusev. 3D Self-Portraits. *Acm Transactions on Graphics*, 32(6):9, 2013.

[MK94] Paul MILGRAM and Fumio KISHINO. A Taxonomy of Mixed Reality Visual Displays. *IEICE TRANSACTIONS on Information and Systems*, E77-D(12):1321–1329, 1994.

[Neta] Microsoft Developer Network. Coordinate mapping. `https://msdn.microsoft.com/en-us/library/dn785530.aspx`. viewed on: 02.09.2016.

[Netb] Microsoft Developer Network. Kinect hardware. `https://msdn.microsoft.com/en-us/library/jj131033.aspx`. viewed on: 09.09.2016.

[Netc] Microsoft Developer Network. Kinect hardware. `https://developer.microsoft.com/en-us/windows/kinect/hardware`. viewed on: 09.09.2016.

[RFM+11] Holger T. Regenbrecht, Elizabeth a. Franz, Graham McGregor, Brian G. Dixon, and Simon Hoermann. Beyond the Looking Glass: Fooling the Brain with the Augmented Mirror Box. *Presence: Teleoperators and Virtual Environments*, 20(6):559–576, 2011.

[RHM+12] Holger Regenbrecht, Simon Hoermann, Graham McGregor, Brian Dixon, Elizabeth Franz, Claudia Ott, Leigh Hale, Thomas Schubert, and Julia Hoermann. Visual manipulations for motor rehabilitation. *Computers & Graphics*, 36(7):819–834, nov 2012.

[RHS+00] B O Rothbaum, L Hodges, S Smith, J H Lee, and L Price. A controlled study of virtual reality exposure therapy for the fear of flying. *Journal of consulting and clinical psychology*, 68(6):1020–6, dec 2000.

[Riv11] G. Riva. The Key to Unlocking the Virtual Body: Virtual Reality in the Treatment of Obesity and Eating Disorders. *Journal of Diabetes Science and Technology*, 5(2):283–292, mar 2011.

[RRR96] V S Ramachandran and D Rogers-Ramachandran. Synaesthesia in phantom limbs induced with mirrors. *Proceedings. Biological sciences / The Royal Society*, 263(1369):377–86, apr 1996.

[SFR01] Thomas Schubert, Frank Friedmann, and Holger Regenbrecht. The Experience of Presence: Factor Analytic Insights. *Presence: Teleoperators and Virtual Environments*, 10(3):266–281, jun 2001.

[SKB11] Evan A. Suma, David M. Krum, and Mark Bolas. Sharing space in mixed and virtual reality environments using a low-cost depth sensor. *ISVRI 2011 - IEEE International Symposium on Virtual Reality Innovations 2011, Proceedings*, pages 349–350, 2011.

[SKT03] Natsuki Sugano, Hirokazu Kato, and Keihachiro Tachibana. The effects of shadow representation of virtual objects in augmented reality. *Proceedings - 2nd IEEE and ACM International Symposium on Mixed and Augmented Reality, ISMAR 2003*, pages 76–83, 2003.

[SVS05] Maria V. Sanchez-Vives and Mel Slater. From presence to consciousness through virtual reality. *Nature Reviews Neuroscience*, 6(4):332–339, apr 2005.

[TAB+14] Franco Tecchia, Giovanni Avveduto, Raffaello Brondi, Marcello Carrozzino, Massimo Bergamasco, and Leila Alem. I'm in VR! *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, pages 73–76, 2014.

[TZL+12] Jing Tong, Jin Zhou, Ligang Liu, Zhigeng Pan, and Hao Yan. Scanning 3D full human bodies using kinects. *IEEE Transactions on Visualization and Computer Graphics*, 18(4):643–650, 2012.

[VRa] Oculus VR. Asynchronous timewarp on oculus rift. `https://developer3.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/`. viewed on: 07.09.2016.

[VRb] Oculus VR. Documentation: Initialization and sensor enumeration. `https://developer3.oculus.com/documentation/pcsdk/0.5/concepts/dg-sensor`. viewed on: 07.09.2016.

[WS98] Bob G. Witmer and Michael J. Singer. Measuring Presence in Virtual Environments: A Presence Questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, jun 1998.