Jörg Hermann Müller, BSc

Mobile Telepresence

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisors

Univ.-Prof. Dipl-Ing. Dr.techn. Dieter Schmalstieg
Institute of Computer Graphics and Vision

Dipl. Mediensys. wiss. Dr. techn. Tobias Langlotz
University of Otago, Department of Information Science

Graz, April 2015

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz, am _____ _____

Datum Unterschrift

Abstract

Telepresence systems bridge the big distances that people have to overcome in our globalized world and make us feel present at remote locations. We developed a novel mobile telepresence system based on the WebRTC technology developed by Google. It allows its users to look around freely in the remote environment, without having to direct the remote user. As a symmetric and purely mobile system that works on any recent Android device, it can immediately be used in unprepared environments as a panorama is generated on-the-fly. Compared to other algorithms that create a visual representation of the environment like the simultaneous localization and mapping or depth camera based algorithms, the panorama has the big advantage that it works very well for outdoor scenes where objects are usually far away. Furthermore, we also created a basic user interface that aids in gaze awareness and spatial awareness, and allows users – local or remote – to point and even draw in the scene.

The purpose of this thesis was the development of a telepresence system which is now ready to be evaluated in a user study. Derived from existing research, we expect the system to increase the spatial presence of its users, and to facilitate the feeling of being at another location, which is the main goal of the application. The thesis is concluded with a positive outlook on limitations and how to solve these issues, followed by suggestions for future research.

Acknowledgements

First I want to thank the people at the University of Otago, where I spent the first six months working on my thesis. I was under the great supervision of Tobias Langlotz who is not only a good supervisor but also helped me to come to New Zealand and to settle into life there. Assoc. Prof. Holger Regenbrecht also supervised me and brought in a lot of knowledge and wonderful ideas with his "slow thinking". Jonny, Chris and Elias are the best lab mates anyone can wish for and I also wished that Mohammed would have spent more time with us, you are great! Simon was there for many interesting discussions about all different parts of science and life. Summing up, it was great to meet everyone who ever attended the weekly Wednesday meetings. I thank all of you for motivating me to work hard, your feedback and help and I hope to meet all of you again soon.

At my home university I first of all want to thank Prof. Dieter Schmalstieg my primary supervisor who also supported me to go to New Zealand and helped me with getting scholarships which enabled me to work abroad. Denis Kalkofen helped not only to maintain the contact between New Zealand and Austria but also to finish the thesis after I returned home. A big thank you to all the people I met along my studies, especially Astrid, Thomas, Georg, Chri, Hannes and Phil with whom I had the best teams, the greatest fun and the best comrades through hard times such as the operating systems exercises. Another big thank you to Bernhard Geiger, who with his unsurpassed kindness and helpfulness is one of the best lecturers I know. Without all of you I would have never gotten this far. Thank you all for joining me on this journey.

My utmost gratitude goes to my family who always supported me in all possible ways. I thank my mother, my grandparents, my brother Thomas, Wolfgang and also my cutest of all cats, Grisu and Diego. I missed you a lot when I was abroad. I am grateful for all my friends, most notably Flo, Simon, Christoph and Michael who also supported me from afar when I got homesick, even when it was caused by them in the first place. To the new friends I found in New Zealand, Alex, Brian, Patrick and all the great people I got to know in the Lovelock House: you gave me the chance to relax after hard work in the lab.

Last but not least, I want to dedicate this thesis to Kimmy for her love, support, kindness, power and motivation. You are amazing. Thank you for being there like when I needed you over Christmas and let there be many more adventures for us to come.

Contents

1 Introduction

We live in a globalized world and the main goal of telecommunication is to bring us closer. Telecommunication technology is used to bridge the big distances. Telepresence is the term used to achieve the feeling of being present at another place. This thesis focuses on mobile telepresence. How can presence be achieved with a mobile device like a smartphone that many people own nowadays.

1.1 Telepresence

Before we describe in more detail what telepresence is, we want to explain why it is necessary. Companies are not only operating in their local community, but they are spread over countries and continents in a global market. Families and friendships become less localized as well, as people move for jobs, love and other reasons. Yet, even people that were never located closely to each other build communities over the internet. Therefore, it is important to advance and build technology to support us and bridge the distances to feel closer.

The internet is a global and digital network which connects almost any point on earth and is even reaching out into space. Additionally, the internet has another amazing property, which is being lightning fast, allowing for real-time communication and other real-time applications. In computer-mediated realities we also define the term interactivity, which means that we cannot feel a significant delay between our actions and the system's response. E-mail, though usually fast, is not considered an interactive communication medium, while telephony is. In telephony it is critical that the delay is low, as waiting for five seconds to hear a response would make the conversation daunting. As the internet allows real-time communication, analog telephony is more and more replaced by "Voice over Internet Protocol" (VoIP) telephony.

Most people nowadays know Skype[1] as a program that enables us to call other people anywhere on the globe and not only talk to them, but also see them with the help of webcams. It is a successor of the telephone as a real-time long distance communication medium. Next to Microsoft's Skype, there are other well known programs such as Apple's FaceTime[2] and Google's Hangouts[3]. They all share the ability to transfer audio and video over the internet between two or more users. Such technology is called either video calling software, or in the case of more than two people in one conversation, it is called video conferencing. Figure 1.1 shows such a video calling application in use.

[1] http://www.skype.com/
[2] https://www.apple.com/ios/facetime/
[3] http://www.google.com/hangouts/

Figure 1.1: In a typical desktop computer video call the cameras are facing the two participants. In the lower right edge the picture of the own camera is shown, while most of the screen is used to display the remote image.

But why stop after adding a simple video stream to the call? Is it possible to bridge long distances better and bring people closer together than plain audio and video signals can? Telepresence systems try to improve on that by going one step further in increasing the presence of the users. Presence is the feeling of "being there" and can be further divided into spatial and social presence. Spatial presence (Schubert, Friedmann, and Regenbrecht, 2001) is the feeling of being at a location. In the real world this is so obvious that probably no one would have ever thought of this, but we might feel totally detached from a virtual world, instead of feeling like we are there. Spatial presence is important so that we feel like we are actually there and are immersed in this virtual world. Social presence in contrast relates more to the communication with another (human) being and the "sense of being with another" (Biocca, Harms, and Burgoon, 2003). Both aspects are important in telepresence and it is thus the technology that enables this feeling of presence over a longer distance, as the Greek word "téle" meaning "far" describes.

Telephony was the first step towards telepresence. Adding video adds a first feeling of spatial presence to it as we can now see through a window into the remote environment. Modern telepresence systems try to further increase presence. One example are systems trying to increase eye contact like in Gemmell et al. (2000). Eye contact is a very important aspect in social communication. The problem is, that in simple video calls, people look at the screen. However, to look into the other person's eyes it would be necessary to look at the camera, just as we do when we take photos. It doesn't end with eye contact. A more extreme example is blue-c (Gross et al., 2003), where the user is standing in a cube with three sides being projection panels shown in figure 1.2. Two projectors for each of these three panels create a stereoscopic image, one projector for each eye of the user. At the same time the user is captured and 3D reconstructed. The 3D reconstruction allows remote users to see the local user in the virtual environment,

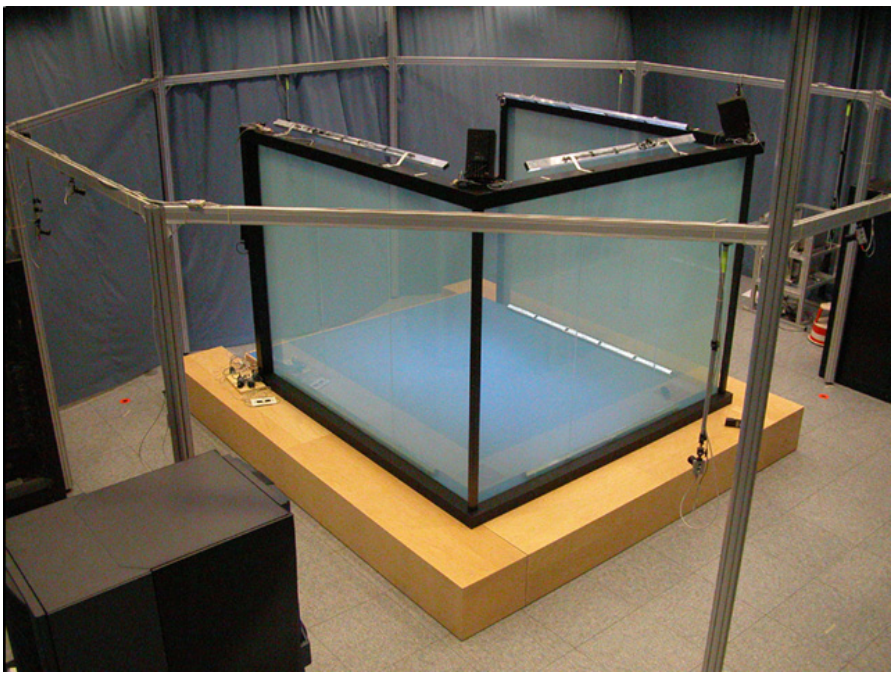


Figure 1.2: The overview over the blue-c "cave" by Gross et al. (2003) shows three panels that are making up the sides of a cube. They allow projection and at the same time capturing the user inside the cube for 3D reconstruction.

while the local user is immersed in the virtual environment shown by the panels. This laboratory experiment is time consuming and costly, so that it is neither sold in the consumer market, nor the business market.

Examples for commercial telepresence solutions targeted for business applications are Lifesize[4] or Cisco's Immersive Telepresence[5], which joins two remote conference rooms into one using big flat screens and spatial audio. The company iRobot offers video collaboration robots[6] such as the iRobot Ava® 500 shown in figure 1.3a, where users can steer the robot in the remote location either manually or with automated navigation and the height adjustable screen allows for seated and standing face-to-face video conversations. While the first two products are not much different from consumer market video calling software, except for being targeted for professional use in conference rooms, the latter product shows the need for mobility and free movement that can increase spatial presence significantly. The problem is that it needs custom hardware – a whole robot – to accomplish this and the mobility is still limited. Lightweight mobile devices not only allow mobility but can literally be taken everywhere a person can go, no matter if that is indoors or outdoors.

[4] http://www.lifesize.com/

[5] http://www.cisco.com/c/en/us/products/collaboration-endpoints/immersive-telePresence/index.html

[6] http://www.irobot.com/For-Business/Ava-500.aspx

(a) The iRobot Ava® 500 video collaboration robot[6]. This is a mobile telepresence robot targetting the business market.

(b) Polly, the parrot smartphone robot in use by Kratz et al. (2014). The remote user controls the view.

Figure 1.3: Two mobile telepresence robots: the commercial robot from iRobot works independently, while the research robot Polly is mounted on another person's back.

1.2 Mobile Telepresence

Smartphones and tablets are pervasive mobile devices. They allow for even more mobility than laptops do. These devices usually have two cameras, one in the back and another in the front. Mobile connectivity is also widespread with 3G, 4G and WiFi. As a result, traditional desktop video calling applications, such as Skype, are now also being used on mobile devices. However, they do not take advantage of the mobile aspect, as – in contrast to a desktop webcam – the cameras in mobile systems can be moved in many different ways as observed by Jones et al. (2015). This together with the sensors present in modern smartphones opens up more possibilities for new and innovative telepresence systems.

There are many applications of mobile telepresence. Its use has been explored in support systems for example, where an expert is helping a user on-site (Gauglitz et al., 2014). Laptops have been used by long distance couples, who carry each other around the house while doing chores as found by Judge and Neustaedter (2010). Many more areas have potential use cases, such as tourism, entertainment, sales, advertising gaming and education to name a few.

The support system example has been investigated in research in various different scenarios such as mining workers (Huang and Alem, 2011) or car mechanics (Gauglitz et al., 2014). Most of the projects still need custom hardware, just as the iRobot telepresence robot in the previous section does. Such telepresence robots have been researched earlier for example by Jouppi (2002). An example with smaller custom hardware is Polly (Kratz et al., 2014), where a robotic arm holding a smartphone is

mounted on the local user's shoulder, while the remote user can steer the view by controlling the robotic arm as shown in figure 1.3b.

Other projects use software to create some sort of spatial representation of the environment in which the remote user can then look around. Most of these still need specialized hardware, such as head mounted displays and cameras (Kasahara and Rekimoto, 2014). Chili (Jo and Hwang, 2013) in contrast simply uses mobile phones and requires no special hardware, but focuses on the interaction rather than the ability to control the view. It supports interaction to view what the users want to view, but does so by directing the other user, instead of allowing independent views as shown in figure 1.4.

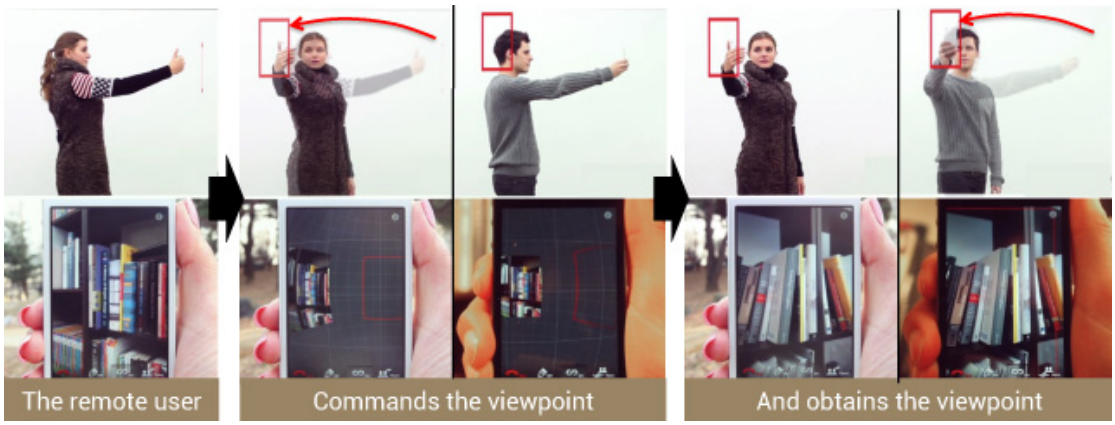


Figure 1.4: Chili (Jo and Hwang, 2013) allows viewpoint control by directing the remote user visualising a spherical grid. Unfortunately the users cannot have independent views.

We argue that there is a lack of systems investigating mobile telepresence. Compared to the large amount of research done in the general field of telepresence, there are still many open and unexplored questions in the field of mobile telepresence. In fact, until now there is no system relying only on mobile hardware that provides the feeling of presence. Smartphones are ubiquitous devices that most people carry around with them all the time and it is therefore possible and important to connect people from one place to another, no matter where they are. The infrastructure and devices provide enough bandwidth, processing power and sensors that can record the surrounding environment. Only the software that makes use of these features to form a sophisticated mobile telepresence system that everyone can use is missing.

1.3 Goals

The goal of this thesis is the creation of a novel mobile telepresence system. We think that it is possible to solve the problems of the previous section, by enabling the user of the telepresence system to look around in the remote environment independently from the remote user and without the need of specialized harware as presented in figure 1.5. Furthermore, we think that current technology is capable enough to do so. The time is

ripe for mobile telepresence systems. The system we want to build for this reason shall work on off-the-shelf and widely available hardware which many people already own, like smartphones and tablets.

For the same reason the system shall be usable in a symmetric fashion in contrast to many examples from the literature, where only one of two communication partners is mobile. The application scenarios hereby are limited to asymmetric ones, such as the expert support system examples, where the expert is still stationary in a room with touch enabled computer screens (Huang and Alem, 2011 or Stafford, Piekarski, and Thomas, 2006). A symmetric system still allows these application scenarios to be implemented. Additionally, it allows symmetric application scenarios, like long distance relationships with partners and family members. These can be at any location, whether it is in the living room showing their new carpet or the peak of Mount Cook in New Zealand showing the amazing view.

This thesis focuses particularly on spatial presence. As already indicated by Yang et al. (2007), a limited view affects the presence and therefore our main goal to achieve spatial presence is an interactive, free and independent view selection. A standard video call always requires the local user to follow the view of the remote user. We argue that a view of the remote location which is independent from the remote user's current view increases the feeling of being at the remote location, hence spatial presence. This shall be determined in a future user study which is out of scope of this thesis.

Next to having an independent viewpoint, physically colocated people can also easily show each other different things. This is done by pointing at them with their fingers in combination with verbal communication. The latter is supported by the simple transmission of audio, which is a common feature of telepresence systems. The former – finger pointing – needs an interface that supports the remote user showing the local user and vice versa. We again predict that this technique improves spatial presence as well as it increases efficiency in collaborative tasks. Again this shall be determined in a future user study outside the scope of this thesis.

For a usable telepresence system, we need an interactive system that feels responsive. We therefore postulate a minimum framerate of 15 frames per second and a latency as low as possible. The application shall work using state-of-the-art networking technology such as wireless LAN, 3G, or 4G mobile networks. Fullfilling these requirements will be the proof that current hardware and infrastructure is capable for the creation of mobile telepresence systems as predicted.

Overall, we want a mobile telepresence system that works on off-the-shelf hardware, has a symmetric architecture, is interactive, allows a free choice of view in either location and supports the users communication with the possibility to point and thereby show the other user around. This system shall be advanced enough to be further used in studies on how it aids presence, especially spatial presence.

(a) Alice's environment (b) Bob's environment

Figure 1.5: The left photo shows how Alice uses the phone to look around in Bob's environment independently. On the right side we can see Bob looking in a different direction in his environment while talking to Alice.

1.4 Results

In this work, we first investigated different available video calling systems to base the work on. We started by defining requirements and then searched for available software while continuously evaluating it. Linphone[7] was one of the systems examined in more detail, but had shortcomings especially with respect to the extensibility of the software. WebRTC (W3C, 2015) turned out to be the best system available with good quality, a big feature set, a bright future, excelent plattform support and a supportive community.

We decided for a panoramic representation of the environment. A cylindrical panorama is generated from the cameras' video streams using the visual tracker PanoMT by Wagner et al. (2010). The panorama is generated in real-time and constantly updated based on the incoming video stream from the camera, resulting in an interactive panorama. There are remaining problems with this approach such as translational movement of the user, an environment not providing enough feature points or a changing environment, so that the visual tracking and panorama generation does not work properly anymore.

After combining the panorama generation and video call, it is then possible to look around in a panorama generated from a remote camera stream. We discuss different architectures that can be used to create a combined system. When two users communicate based on the generated panoramas of their environments, it is crucial to have a congruent coordinate system, which requires synchronization between the two clients.

[7] http://www.linphone.org/

We then evaluated how different mobile phone hardware performs with this system and evaluated the systems with respect to the previously set goals. Figure 1.6 shows the implemented system in use outdoors. You can see that the user is in a different environment on the smartphone than the background shows.

Figure 1.6: Our mobile telepresence application can be easily used anywhere, especially outdoors. The user in this photo is looking at the remote environment of his communication partner.

For a user study, we implemented basic interface techniques targetting spatial awareness, gaze awareness and the ability to point. In the background the application is also highly configurable for different scenarios to be evaluated in the study.

1.5 Contents

The thesis continues with chapter 2 on related work in the fields that are touched by this thesis. Chapter 3 starts with describing requirements for video calling software to build the telepresence system on and then evaluates existing software, finding WebRTC as the most promising candidate. In chapter 4 we investigate possible environmental representations to be used and after deciding for a cylindrical panorama, we get into the implementation details and finish with a short discussion of shortcomings. Putting it all together in chapter 5, we then combine video calls with the environmental representation and evaluate the resulting system. In preparation for a future study, chapter 6 describes the implementation of a basic user interface for the application. Finally, chapter 7 concludes the thesis and gives an overview over the tremendous amount of possible future work.

2 Related Work

The related work can be split into several categories. The history of mobile telepresence started with telepresence robots and similarly bulky hardware and developed into a broader field using more powerful and smaller mobile hardware such as head-mounted displays (Drugge et al., 2004), smartphones (Gauglitz et al., 2014) and tablets (Johnson, Gibson, and Mutlu, 2015). An independent view for the remote user in the telepresence system has been identified as a key feature. The simultaneous localization and mapping (SLAM) algorithm has been used for example by Kasahara and Rekimoto (2014) to enable a exploration of a remote scenery. Panoramas have been used in telepresence systems as well (Norris, Schnädelbach, and Qiu, 2012) and are another potential representation of the environment as panoramas can be generated in real-time on smartphones as shown by Wagner et al. (2010). Last but not least there is a huge body of research focusing on collaboration and interaction with mobile telepresence systems, where usually one user is mobile outdoors and the another stationary indoors with a desktop system, like Stafford, Piekarski, and Thomas (2006) and Sodhi et al. (2013) presented. Interaction techniques investigated include gestures (Huang and Alem, 2011) and drawing (Fussell et al., 2004).

2.1 Mobile Telepresence

The first research in mobile telepresence has been done with bigger machines and robots as the hardware miniaturisation hasn't been advanced enough yet. These teleoperated robots are extensively studied in the context of telepresence, to allow the operators to interact with the remote location just as if they would be there. Contrarily the people in the remote location also should get a feeling that the operator is present and not just a robot. Jouppi (2002) for example investigated a system to allow this mutual immersiveness with a robot displaying the operators head with displays from four sides. While the use of a robot enables many beneficial possibilities for presence, the hardware requirements and costs prohibit the casual use of such a system for many application areas. Zhu, Gedeon, and Taylor (2011) explored different methods for viewpoint control for teleoperated cameras. The three methods investigated are manual, natural interaction, and autonomous tracking. The natural interaction which follows the user's gaze and head movement for zooming showed the best results. Autonomous tracking which followed the user controlled robot arm was perceived slighly worse and the manual method where the user had to control the viewpoint as well as the robot arm manually performed worst.

With the development of powerful enough laptops, Drugge et al. (2004) could conduct early work with wearable mobile telepresence technology. They used a head-mounted display (HMD), a camera, a microphone and a remote control for user input with the computer in a backpack. Three main conclusions to consider for future systems are presented considering the setup time, presence and the appearance and influence on other people. Nowadays smartphones provide enough processing power and are generally accepted, while head-mounted displays are currently controversally discussed in the general public[1]. The miniaturisation of the robots also continued and Kratz et al. (2014) developed Polly, a smartphone mounted on a gimbal with the phone near to the shoulder of the "Guide" called carrier. The remote user is displayed on the screen of the phone and can control his view using the gimbal, allowing him to alternate his view between forward facing and face-to-face contact. The authors expect their system to be socially and physically comfortable especially compared to phones, but don't have enough data to support their claim.

Going away from robots in the field of mobile telepresence, we can find work of Kasahara and Rekimoto (2014) and Gauglitz et al. (2014) who both use the simultaneous localization and mapping (SLAM) algorithm to create a virtual representation of the environment. JackIn is the mobile telepresence system developed by Kasahara and Rekimoto (2014) where the remote user called "Body" wears an HMD with a camera and the stationary user called "Ghost" watches on a screen as shown in figure 2.1. The Ghost's pointing is recognized with a Leap motion controller and the targets are then displayed on the Body's transparent display. The Ghost's out-of-body view is created by the SLAM algorithm in real-time and the system supports a viewpoint control mode next to just following the mobile user. Gauglitz et al. (2014) focus more on the collaborative aspect by using annotations that are displayed in the augmented view of the mobile user's smartphone. The major problem of the SLAM algorithm as discussed by the authors is, that it needs a stereo initialisation step. With a single camera this means, that a sidewards translation is required. Depending on how far objects are away, the distance required for this translation can become quite big in outdoor scenes. A big difference between these two works is that while Kasahara and Rekimoto (2014) use an HMD for the outdoor user, Gauglitz et al. (2014) use a smartphone. Johnson, Gibson, and Mutlu (2015) very recently compared a handheld tablet to a Google Glass head-mounted display in a collaborative construction task in an either static or dynamic setting. The remote helper is using a desktop computer and Google Hangouts is used as video calling software. Accordingly the only mobile aspect was that the camera viewpoint could be easily moved and the hands were either free or not, no other special hard- or software features were added. The results showed different preferences for worker and expert in the dynamic scenario with the worker perceiving the collaboration more successful with the tablet than with the HMD although the task completion time is slightly better with the HMD.

We can see that most of the mobile telepresence research tries to improve spatial presence of the second and usually stationary user. In a more theoretically based study with 189 participants, Horvath and Lombard (2010) investigated the importance of

[1] http://edition.cnn.com/2014/02/19/tech/mobile/google-glasshole

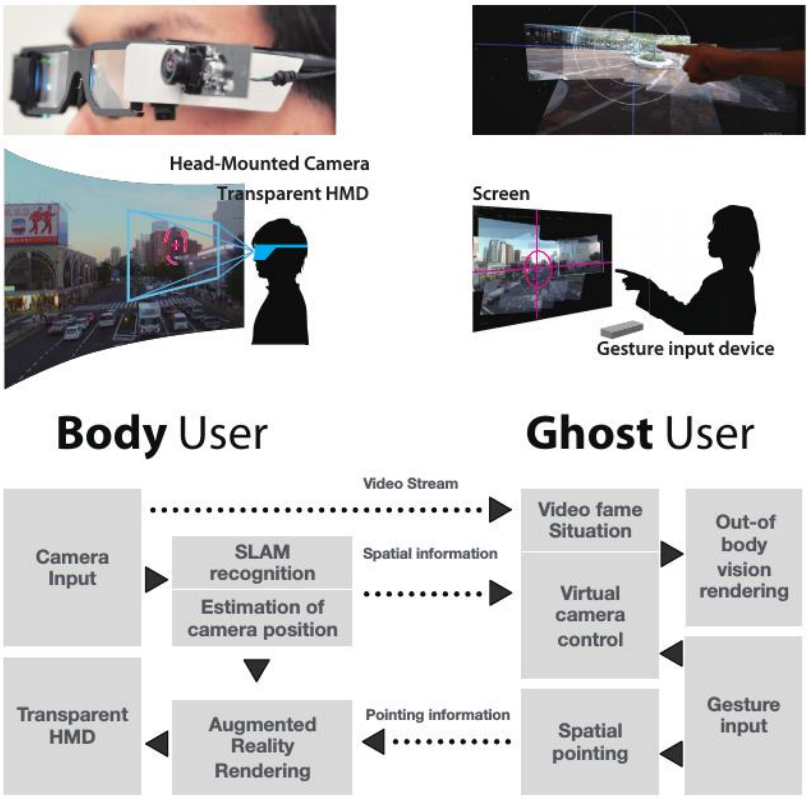


Figure 2.1: In JackIn developed by Kasahara and Rekimoto (2014), the mobile user equiped with an HMD is called "Body". The stationary user called "Ghost" watches the 3D reconstructed environment indoors and can gesture with a gesture input device.

social and spatial cues to result in the corresponding presence. The authors conclude that a focus on a user-centric perspective leading to natural interaction with computers results in a better experience and higher satisfaction and enjoyment, as opposed to a technology-centric perspective. These results were obtained even though the participants were mostly very experienced in computer use and the corresponding technologies. With the PePe project, an automatic location detection system, another study was conducted by Lehikoinen and Kaikkonen (2006) on how location information influences the mobile presence of the participants. An independent view has a strong influence on spatial presence as is shown by the work done in this field by Kratz et al. (2014), Kasahara and Rekimoto (2014) and Gauglitz et al. (2014).

2.2 Panoramas

To allow a free view some kind of visual representation of the environment is needed and panoramas are a good candidate to do so. They have been used in earlier presence research. Dalvandi, Riecke, and Calvert (2011) for example investigated the increased

feeling of presence in video panoramas compared it to regular video and a slide show of static panorama images. The finding was that the feeling of presence is higher in the video panorama environment compared to the others and thus might be worth the effort to create the content if achieving high presence is targeted. Panorama videos have also been used for telepresence as in CamBlend by Norris, Schnädelbach, and Qiu (2012), where three cameras are used to generate a high resolution video panorama which is transmitted to the remote collaboration site with a lower resolution due to bandwidth limitations. Focus areas which are circular areas of higher resolution are transmitted and shown inserted into the blurry low resolution panorama. All communication partners can view and change the position of the focus windows thus allowing an interactive collaboration and guiding of the attention of the communication partners.

To use panoramas in the mobile telepresence context it is necessary to create a panoramic video with just one available mobile camera. The panoramic video textures of Agarwala et al. (2005) are based on a video recorded with a single camera using panning motion and creating a video panorama out of the spatial and temporal data. The algorithmic complexity is rather high to achieve a fully animated panorama out of the limited field of view (FOV) of the recording camera though. Steedly, Pal, and Szeliski (2005) also use a video as input for their panorama generation system optimizing their algorithm by finding key frames based on image-to-image overlap which are then matched to all other keyframes. This allows to optimize the computation time for stitching of intermediate frames to only temporarily neighboring frames and keyframes.

Algorithms creating fully animated video panoramas are too complex for mobile telepresence and algorithms that are fast enough to run in real-time on the mobile processors are required. Envisor as presented by DiVerdi, Wither, and Hollerer (2008) uses vision-based tracking with sensor fusion incorporating a gyroscope and compass for robust and drift free orientation tracking in smartphones. This tracking algorithm is then used to created a cubic environment map on the fly. Automatic gap filling for unobserved areas uses texture diffusion and appears to be mostly useful for small gaps in the observed environment map. Also, Wagner et al. (2010) showed a method for real-time creation of a panorama on mobile phones as shown in figure 2.2. The created panorama is cylindrical and can be used for drift free rotation tracking. The disadvantage of this work is that already mapped areas do not get updated again resulting in a static image, but also allowing very fast stitching of new video frames, as only the newly added information has to be processed into the panorama.

Panoramas, smartphones and telepresence have been combined by Pece et al. (2013). Their software system PanoInserts embeds smartphone videos into a static prerecorded panorama. The video communication is uni-directional from the smartphones to a desktop-based receiver side. It allows a dynamic configuration with phones attaching and detaching from the running session. It can be used in the setting of a spontaneous conference with a remote collaborator. However, the mobile aspect of the smartphones is completely lost in this work, as the phones are just used as cameras.

Figure 2.2: This photo shows the system of Wagner et al. (2010) in use, generating a panorama in real-time on a mobile phone.

2.3 Collaboration

So far we can conclude that mobile telepresence research that an individual view is very important for spatial presence, and it is slowly gaining popularity due to the availability of fast and small hardware. Most of the research applying mobile telepresence techniques focus on collaboration. Jones et al. (2015) conducted an observational study to investigate the mechanics of camera work done in different mobile video collaboration tasks executed by a smartphone user and a desktop user. They conclude that it is important to be able to gesture and draw attention, convey what camera shot is needed and to have a bigger field of view for spatiality and context. Implicit social cues such as pointing should be prefered over explicit user intervention for remote controling the mobile user's camera view. The social awkwardness caused by holding the phone in a typical photo and video shooting pose caused the users to hold the phones in portrait mode which further limited the field of view of the desktop collaborator. Many of the main problems pointed out regarding camera work can be solved by an independent view for the desktop user.

Most of the studies have a desktop and a mobile user and many focus on interaction techniques such as Stafford, Piekarski, and Thomas (2006), who present an interaction technique for the indoor communication partner. The indoor user has a table-top surface showing a virtual representation of the outdoor environment as a map and enabling the user to put physical objects on the table which are 3D reconstructed and displayed at the corresponding GPS coordinates in the augmented reality (AR) view of the remote user. The authors termed their interface as "god-like interaction" as it is possible to guide the outdoor user by just pointing at the destination and the remote

user then sees a huge reconstructed model of a hand in his augmented view. The map shown to the indoor user has been created in advance and not by the outdoor user, so the location has to be prepared. The mobile collaboration system BeThere presented by Sodhi et al. (2013) in contrast utilizes 3D reconstruction on smartphones extended with depth sensors for capturing the scene and gestures (figure 2.3). While the system differentiates between a remote and local user having a front facing kinect for the scene reconstruction and a side facing short range sensor for gestures, it should principally be able to support a symmetric communication if functionality to either merge or switch environments is added. The gesture input is used to not only show the remote user's hand in the augmented view of the local user, but the remote user is also able to manipulate virtual 3D objects placed in the scene.

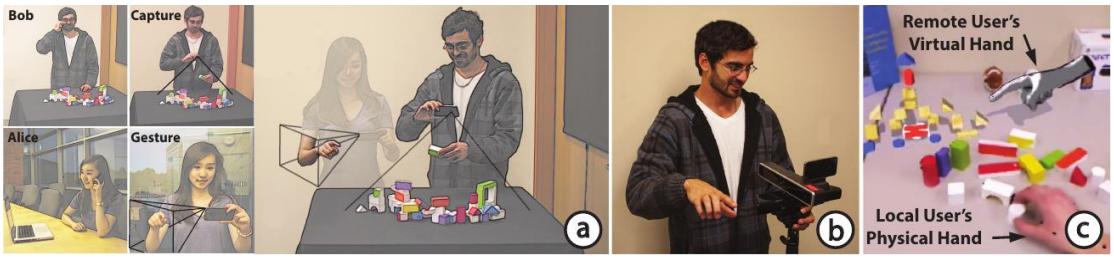


Figure 2.3: Sodhi et al. (2013) developed a mobile collaboration system called BeThere. The concept (a) shows two users, one capturing the shared environment and another gesturing. The environment and scene are captured with depth cameras (b) and the virtual hand is integrated into the camera image of the workspace (c).

Last but not least additional interface techniques for telepresence systems aid collaboration. Next to the independent view, techniques that aid in spatial awareness, gaze awareness and interactions can improve the collaboration. In an extensive user study, Fussell et al. (2004) for example compared a simple pointing gestures against drawings in more complex collaborative assembly tasks. The finding is that simple pointing is not enough for effective collaborative work while drawing enhances the experience not needing actual hand gestures to be shown. Gergle and Clark (2011) ran a user study with their eye tracking system to identify how participants reference objects for gaze coordination in collaborative tasks. A key insight of the study is that there is a difference between mobile and seated participants hinting that this should be kept in mind for the described systems which were mostly asymmetric with one mobile user and another immobile user usually sitting in front of a desktop computer. Mobile users not only changed their position and used simple words for refering such as "it", but also use more feature-based descriptions, compared to the seated participants that used location-based descriptions such as "on the right". Lastly, the work of Huang and Alem (2011) focuses on collaborative work using hand gestures. The remote mine worker wears a helment extended with two displays and a camera, alowing him to look up and see video instructions from the desktop user. The desktop system is based on a large touch enabled screen displaying a camera view, several storage spaces for recorded camera sequences and a pregenerated panorama of the workspace which is used for overview purposes only. The desktop user can then gesture over the video streams received by the remote user. A camera capturing this display and the desktop user's

gestures records a video stream that is then sent back to the worker. This are only three examples of the many different possibilities for interaction techniques. They are not our main focus, but simple interaction techniques will be included in our work.

2.4 Summary

One important work falls into multiple of the described categories. Chili by Jo and Hwang (2013) is a mobile video calling system supporting view control and a drawing interface. The application works on two mobile devices (iPhone or iPad) in a symmetrical fashion with no difference between remote and local user. The user interface used to control the view of the remote partner utilizes a wireframe rendered sphere with the video and a red rectangle projected onto the sphere, guiding the partner where to look at. This however requires the users to build a mental picture of the remote location and allows only approximate targeting. To allow the users to look at each other the view modes can be changed by swiping between remote, face-to-face and local view. Collaboration is furthermore supported by a drawing interface using touch or phone movement as input. Technically Chili incorporates sensor fusion between the phone's gyroscope and feature-tracking based on FAST features.

We can conclude that view control or independent views are very important to increase spatial presence in a mobile telepresence system and a lot of research effort is put into achieving this. We saw that panoramas are a great way to record an environment on-the-fly in any location and we saw that panorama generation is possible in real-time on current mobile devices (Wagner et al., 2010). Many of the problems related to dependent views in mobile collaboration tasks (Jones et al., 2015) could be addressed with independent views and additional interaction techniques like a collaborative drawing interface (Fussell et al., 2004) aid in the interaction between participants. Unfortunately these separated areas have so far never been combined to a superior system and evaluated in real world applications.

3 Video Calls

In this chapter we will discuss the basic infrastructure and video calling framework that is needed for the application. We will start off by defining some criteria and requirements for the video calling software to build on. After explaining available technologies, protocols and standards for video calls, we will discuss available software. Two software projects were inspected in greater detail.

3.1 Requirements

A successful implementation of the application needs several requirements to be fulfilled. Those requirments as shown in figure 3.1 can be further categorized. The feature category (brown) contains requirements that enable the development of a telepresence application. Requirements in the development category (red) are necessary for easy and fast development, while the quality category (blue) contains requirements which improve the system's quality. Lastly, the future category (green) contains requirements that enable future development and future research.

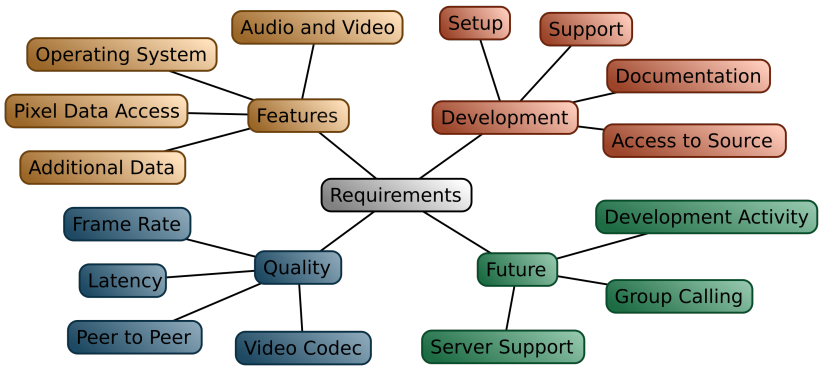


Figure 3.1: The overview of the requirements for the video calling software utilizes colors to separate them into four categories. The features are basic functionality that is required in any case, the development category contains properties that help developing fast, the quality category contains parameters that influence the quality of the resulting output and lastly the future category holds indicators for sustainable support for further research.

Operating System The target platform for this project is Android, so support for Android is mandatory. For future work it will be nice to have support for more platforms, such as for iOS and Windows Mobile as well as Desktop operating systems. The latter can also be used during the development for easier debugging.

Audio and Video Even though this requirement is obvious, it is important to note that audio is also a very important factor of video calls next to video streaming. Thus both, audio and video streaming are required.

Pixel Data Access Visual tracking requires access to the raw image data of both incoming and outgoing video streams. One problem of the pre Android 5.0 camera API is that access to the pixel data is tightly tied to display elements. Therefore another API to access the pixel data is preferable.

Additional Data The transmission of additional data is required to exchange additional information between the clients, such as the pointing information or the information of which location is currently being looked at. Of course it is always possible to open an additional network stream independently of the framework used for video calls, but using available functionality will speed up development and eases connection establishment.

Setup An easy setup helps to increase development speed. It would be possible to use Skype as a video calling framework for example. As there is no API to access and modify the video streams and renderings, it would be necessary to intercept the data on an operating system level. This means that we would have to implement drivers that also communicate to the application code. This is time consuming to implement and thus unreasonable to use.

Support The development and application support is strongly connected the development activity. If a company is developing the framework, the question is if it is possible to get support from the developers. In the case of community-driven development, the bigger the community, the easier it is to get help from others.

Documentation A good documentation can increase development speed a lot in contrast to missing documentation which requires detailed investigation of the API and most likely underlying source code.

Access to Source As a result of the previous requirement it is at least necessary to get access to an API. Having access to the whole source code gives even more benefits. With the whole source code available it is possible to edit any needed software component without having to rely on external help.

Frame Rate The frame rate is the rate at which new images are produced. The producer can be a camera, a display, or a video file or stream for example. It is usually expressed in terms of frames per second (FPS) which is the inverse of the time between two frames. All these come to play in video calls and the overall frame rate of the system is the minimum of the different components. If the camera supplies the video encoder with frames at a higher rate than it is able to process, then frames need to be dropped lowering the overall frame rate. To be able to claim real-time an minimum overall frame rate of 15 FPS is required.

Latency The latency is the time delay between the input and the output of a system. For video calls, the end-to-end delay as shown in figure 3.2 is the time from when the photons enter the camera at the sender side until the photons leave the display on the receiver side. This definition captures the whole processing pipeline from taking the image with the camera, video encoding, network transmission, video decoding and rendering the image. Additionally all processing steps in between add time to the latency. The goal in communication applications is to have the lowest possible latency. This latency is not to be confused with the time between two frames, which is the reciprocal of the frame rate.

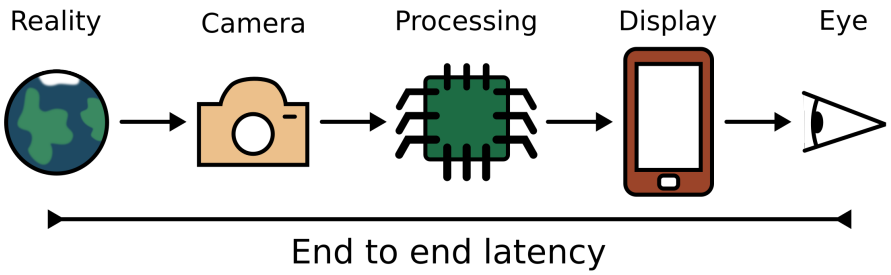


Figure 3.2: End to end latency in common AR systems is the delay from the real world image appearing until the image can be seen on the screen with the eyes of the user.

Peer to Peer For the sake of latency it is important that the clients exchange the data peer-to-peer, which means directly between each other. Routing it through a server instead adds to the latency.

Video Codec The video codec defines how the video stream is compressed for the transmission over the network. This is very important as the available network bandwidth is too low for an uncompressed video. For example, a video with a resolution of 640 × 480 pixels, 15 FPS, and 16 bit per pixel has a bandwidth of 73.728 Mbit/s. In comparison LTE has a maximum theoretical upload speed of 75 Mbit/s[1]. Considering packet overhead and a practical bandwidth, it is not possible to transmit this video with

[1] http://www.3gpp.org/technologies/keywords-acronyms/98-lte

the rather low specification given. Compression and quality are competing properties. The higher the compression, the lower the quality. For vision based tracking it is important to have a high quality so that the tracker can still find features in the video frames. As mentioned before, latency is an important factor for the codec. Since the target is to build on an existing video calling framework, it would be nice to have a choice over the used video codec.

Development Activity The influence of active development of the framework can go either way. On one hand it can help the resolution of problems and fixing of bugs that might be found inside the framework, but on the other hand it can slow down development if the framework is constantly changing, unstable or too early in development and not mature enough. This problem should be considered for future work as a technology or framework that is not future-proof might complicate things later, for example porting the framework to other operating systems or newer versions of Android.

Group Calling Another feature that would be nice to have for future work is group calling support. This can be implemented either through using a server or by creating a network between the clients with a star or mesh architecture. The star architecture is the same as a server architecture, where one of the clients would turn into the server. The mesh architecture in contrast connects all clients to each other.

Server Support Contrary to the previous requirement, it would be nice to have support for servers handling video streams. This can be utilized in future development to offload computationally intensive processing to the server or to handle video conferences for bigger groups.

3.2 Software

The technical challenges for video calls in general are quite high. The software has to be able to initiate video calls with the communicating partners, which at the very least requires some form of identification (ID) that identifies the communication partner. In the conventional telephone network this ID would simply be a telephone number. Over the internet the easiest ID would be an IP (internet protocol) address, however especially in the mobile context IP addresses change all the time or multiple users connected to the same WLAN router have the same IP address. With these scenarios in mind it would be better to have a user-based identification like an e-mail address. This requires a way to establish connections to a specific device based on a user ID, which is usually handled by a server and using a specific protocol called session control or signaling protocol. These servers usually also handle more complex cases where for example a direct connection to users behind a router cannot be established and NAT traversal has to be used. After this “call setup” step, when the connection is established, data streams for audio and video have to be setup. The bandwidth is usually limited,

and it limits the achievable quality and latency. As the bandwidth can vary during the communication, audio and video codecs have to adapt to the changing bandwidth. Another problem rises from the required synchronization of audio and video when they are presented at the receiver's side.

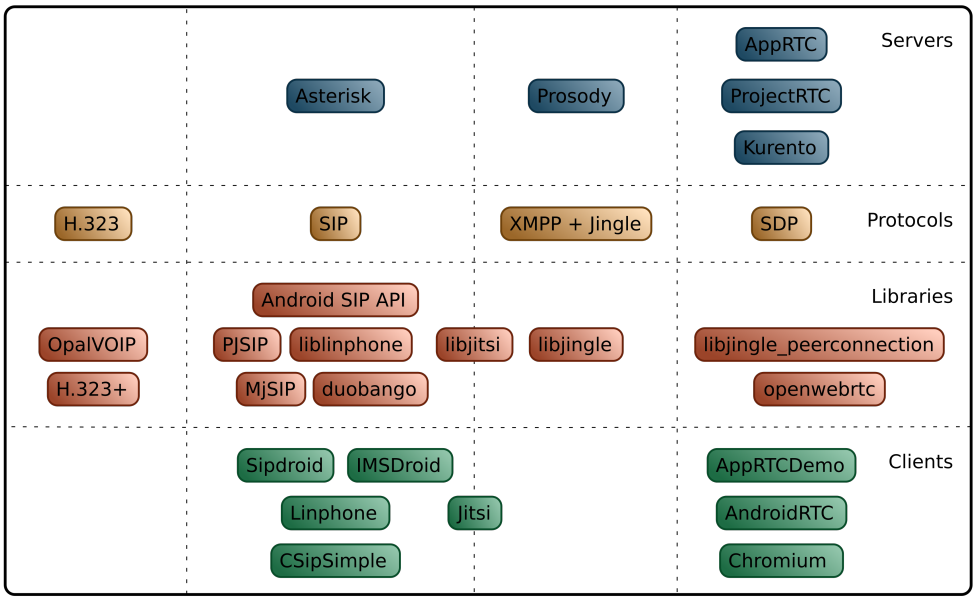


Figure 3.3: This overview diagram over open source video calling software is separated into four columns, with one column per protocol. Jitsi and it's library libjitsi support both SIP and Jingle and are therefore spanning over two columns.

Considering all the technical difficulties, it is more efficient not to start from scratch, but to base the implementation on existing software. To make the required changes, the work can either be based on available libraries that handle the required protocols and codecs, or on an open source application that already implements all the features of a generic video calling application. It would also be possible to use a proprietary application such as Skype and hook into the video stream by writing a driver. However, getting this working on Android may exceed the amount of work required for starting from scratch.

3.2.1 Codecs

At the moment, the probably most popular codec for video streaming is H.264 (AVC, ITU-T, 2005), which is not only used by video calls, but also for online video streaming on platforms such as YouTube[2] and Vimeo[3]. The biggest competitor is VP8[4], which has the advantage of being free from patents and can therefore be utilized by anyone. Both of these codecs have successors, H.265 (HEVC, ITU-T, 2013) and VP9[4], which are

[2] https://www.youtube.com/
[3] https://vimeo.com/
[4] http://www.webmproject.org/

getting established at the moment and will replace their predecessors in the foreseeable future. Nevertheless, they are not yet readily available for video calls with Android.

There is also a big selection of audio codecs. Traditionally audio codecs used for voice communication target very low bandwidth usage and accordingly are low in quality. These codecs include iLBC[5], GSM (Redl et al., 1995) and Silk (Vos, Jensen, and Soerensen, 2009). However, recently the audio codec Opus (Valin, Vos, and Terriberry, 2012) has been released. It has the big advantage that it can scale bandwidth usage and quality further than older codecs. It can be used from very low to very high quality depending on the available bandwidth (Rämö and Toukomaa, 2011).

3.2.2 Protocols

The most popular session control protocols are the ITU-T (ITU Telecommunication Standardization Sector) standard H.323 (Hersent, Gurle, and Petit, 2000), the Session Initiation Protocol (SIP, RFC 3261, Rosenberg et al., 2002) and Jingle (Ludwig et al., 2009) developed by Google and the XMPP (Extensible Messaging and Presence Protocol) Standards Foundation. Each of these protocols are implemented in standalone libraries. SDP (Session Description Protocol, Handley, Perkins, and Jacobson, 2006), is an exceptional case because it is not a full singaling protocol. SIP and H.323 use it to exchange session information used to negotiate codecs, transport protocols, addresses and other meta data.

3.2.3 Libraries

H.323 is supported by H.323+[6] which seems to support only Desktop systems, and OpalVOIP[7], which has Android support only for audio. SIP is directly supported by Android. Third party libraries also exist like PJSIP[8], which has only planned support for video on Android. MjSip[9] is completely written in Java and supports Audio and Video. Jingle is implemented by libjingle[10] and is supported by many applications. The library works on the three major desktop operating systems and Android.

3.2.4 Client Applications

Open source Android apps using the discussed technologies are available as well. Linphone[11] (figure 3.5a) is based on SIP and supports Windows, Mac, Linux, Android and iOS, and has been used as a base for Chili (Jo and Hwang, 2013). However, it doesn't have support for group calls with video as shown in figure 3.5b. The development

[5] http://www.webrtc.org/license-rights/ilbc-freeware
[6] http://www.h323plus.org/
[7] http://www.opalvoip.org/
[8] http://www.pjsip.org/
[9] http://mjsip.org/
[10] https://developers.google.com/talk/libjingle/
[11] http://www.linphone.org/

of the Jitsi Android app[12] is stalled and the application is in an early development phase. Next to SIP it should also support Jingle and group video conferences. While the desktop application works fine (figure 3.4), the Android app fails to run. There are more SIP based applications, such as Sipdroid[13] (figure 3.5c), IMSDroid[14] (figure 3.5d) by Doubango Telecom[15] and CSipSimple[16] (figure 3.5e), which are all Android apps using different SIP implementations. Among these three, only IMSDroid has video calls working. Lastly, browsers are now becoming video call client programs due to the development of the WebRTC API, which allows real time communication (RTC) on websites by using JavaScript. Originally, WebRTC has been developed out of Jingle and libjingle by Google, but now it only uses SDP. The actual transfer of the SDP data is delegated to the developers who use the API. Therefore, it can be used to connect with signaling servers using SIP. An Android client that uses Chrome's native WebRTC implementation is AppRTCDemo (figure 3.5f).

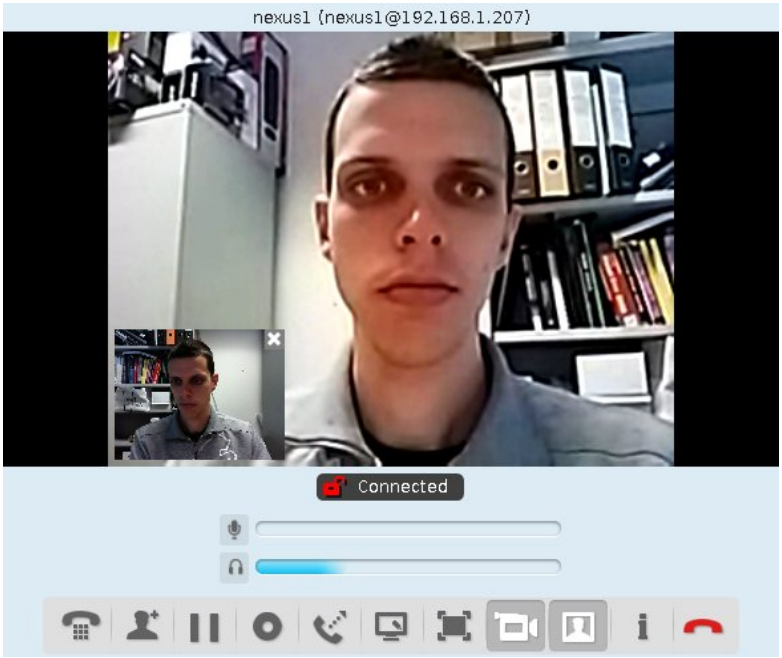
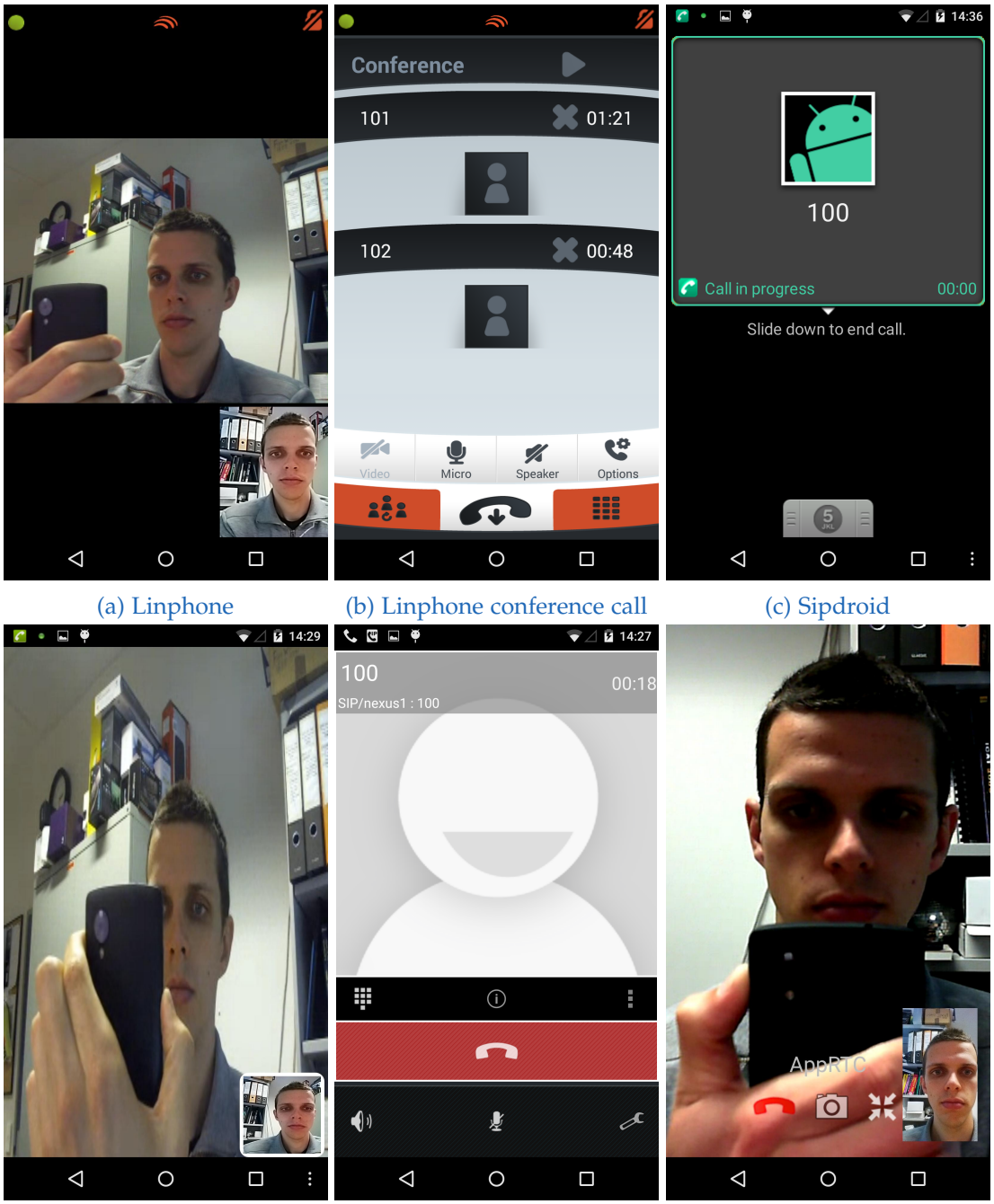


Figure 3.4: Jitsi's desktop application worked fine, while the Android app failed to operate at all during our test runs. Its development is currently inactive and it may not be further developed.

3.2.5 Server Applications

Every session control protocol needs a server to enable two clients to find each other. Otherwise the clients would need to know their respective IP addresses or another

[12] https://github.com/jitsi/jitsi-android
[13] http://sipdroid.org/
[14] https://code.google.com/p/imsdroid/
[15] http://www.doubango.org/
[16] http://www.csipsimple.com



(a) Linphone (b) Linphone conference call (c) Sipdroid

(d) IMSDroid (e) CSipSimple (f) AppRTCDemo

Figure 3.5: Different open source android video calling applications were tested. Only three of these support video calls. Linphone supports group calls, but as soon as the third person joins the call, video is turned off. AppRTCDemo is a demo application using Google's WebRTC implementation. It works very well and the WebRTC API promises to support group calls.

form of address that allows for direct contact. Unfortunately, clients often change IPs and provide no other fixed identifier that allows a direct connection. The solution to this problem is to have a server with a fixed direct address, where clients can register. The server then keeps a record, storing the client's ID and IP address.

Asterisk[17] is a very widely used SIP server and has been used for trying out all SIP client programs. Prosody[18] is an XMPP server with jingle support and has been used to try Jitsi's jingle support. It is important to note that both of these servers are only used for the session initiation. The video and audio streams are directly transmitted between the clients, as soon as the connection between them is set up. Servers that transmit the media streams are generally refered to as video bridges. They can mix streams and handle selective forwarding in video conferencing. To allow clients with different bandwidths, these servers can decrease bandwidth requirements by lowering the frame rate, resolution and/or quality of the stream. Multipoint control units (MCUs) are dedicated hardware devices that are used for this task.

Open source video calling applications for Android support either SIP or XMPP. To try out those applications, a server for these protocols is needed. We used a computer with Arch Linux[19] as server. To connect the phones in the same network with the computer there are several options. One of them is to set up an access point where both computer and phones connect to. The computer then needs another ethernet card or a wireless USB device. We used a wireless USB device and also tried to use the server as access point directly. Unfortunately the wireless USB device was not supported by hostap[20], the Linux access point software.

Asterisk

Asterisk is probably the most famous SIP server for Linux. Others, such as reSIProcate[21], openSIPS[22] or Kamailio[23] would probably work as well. Setting up asterisk is fairly easy, just following the Arch wiki entry[24] about installing asterisk from the arch user repository (AUR)[25]. The server has to be configured in the file `/etc/asterisk/sip.conf` and `/etc/asterisk/extensions.conf`. A problem was that while audio calling was working, video calling needed further configuration according to another website[26].

[17] http://www.asterisk.org/
[18] http://prosody.im/
[19] http://archlinux.org/
[20] http://w1.fi/hostapd/
[21] http://www.resiprocate.org/
[22] http://opensips.org/
[23] http://www.kamailio.org/
[24] https://wiki.archlinux.org/index.php/Asterisk
[25] https://aur.archlinux.org/
[26] http://www.voip-info.org/wiki/view/Asterisk+video

Prosody

Prosody is an XMPP server. Fortunately no extra support for the jingle protocol is needed because only clients have to support it and the server is just used for connection establishment. The installation of prosody for Arch linux is also covered on a wiki page but then one has to follow the configuration as well as user setup pages of prosody. We configured the server and added users using the `prosodyctl` command line tool. There was some trouble with this server application as clients couldn't login. This was resolved by installing the `lua51-sec` package and enabling security by generating a certificate with `prosodyctl`.

3.3 Selection

From the description of found and tested client programs, it is clear that only client programs that should support Android and video calls have been investigated. Also only open source software has been considered because commercial client applications are costly and add additional hurdles to get access to the source code.

With respect to the other requirements that could be tested by simply looking at the project website and trying the client application, we found that all of the described client programs communicate peer-to-peer once the connection is established. Moreover, all of them support the Opus audio codec and the H.264 video codec, which currently offers the best quality by using the least bandwidth for video calls on Android. The clients differ in the requirements illustrated in tables 3.1 and 3.2.

Client	Operating System	Support
Linphone	Android, iOS, Desktop	community, commercial
Jitsi	Android, Desktop	community, commercial
Sipdroid	Android	none
IMSDroid	Android	commercial
CSipSimple	Android	none
WebRTC	Android, iOS, Desktop	community, pot. commercial

Table 3.1: Three SIP clients for Android are written specifically for this operating system. They can communicate with other SIP clients on the desktop but don't run on the desktop themselves. Support for these clients is also limited. The other three applications have desktop clients and a better support available.

The remaining requirements (pixel data access, additional data, documentation, latency and frames per second) needed further investigation. The tables 3.1 and 3.2 allow a preselection of which clients will be further investigated. Sipdroid, IMSDroid and CSipSimple are all pure Android clients and have no advantages over the other clients. The Jitsi Android application is not working at the moment. This leaves us with Linphone, which already has successfully been used by Jo and Hwang (2013) for similar research and the relatively new WebRTC.

Client	Setup	Development Activity	Group Calling
Linphone	easy	active	Audio
Jitsi	easy	stalled	Video
Sipdroid	easy	maintenance	Audio
IMSDroid	easy	maintenance	Audio
CSipSimple	easy	maintenance	Audio
WebRTC	difficult	active	Video

Table 3.2: Again the Android-only SIP clients differ from the other three clients. They are not in active development but still maintained. Even though WebRTC is difficult to set up, it is the only API that potentially supports group calling on Android.

3.4 Linphone

Linphone is a mature and open source Voice over IP (VoIP) client that has originally been developed for Linux since 2001. It has been used as a base for Chili (Jo and Hwang, 2013) and has free developer support over a mailing list. The company Belledonne Communications also offers commercial support[27]. The client meanwhile supports not only Linux, but all major platforms including Windows, Mac OS X, Android and iOS. Due to its history of eing used in previous research, we knew that it is possible to be used for development of similar research projects and we decided to try working with it first.

3.4.1 Architecture

Linphone for Android is separated into the Android app Linphone, and the libraries liblinphone and mediastreamer2. Liblinphone handles the SIP protocol related functionality, while mediastreamer2 handles the audio and video en-/decoding, and RTP sending of the packets over the network.

3.4.2 Demo Application

Building the libraries separately from the Linphone Android app needed some fixes for liblinphone to get it to run. Following the tutorials and demo application source code for liblinphone we got registration and audio calls to work relatively easily as well, but this is where the tutorials end and getting video to work became a bit more problematic. However, as we found out, the main problem is that linphone has tied the sending and receiving of the video streams very tightly to the display widgets on Android. For example, even the preview widget of the own video needs to be there to get video streamed to the communication partner. As a consequence of this tight tie between display elements and video streams, it becomes hard to access the video streams independently from the display elements.

[27] http://www.linphone.org/contact.html

3.4.3 Result

Figure 3.6 shows the developed demo application running on Google Glass. The conclusion is that the access to the video streams is possible, but cumbersome from a software architecture view. The documentation of linphone was good up to the end of the tutorials, but further documentation is missing. The performance (frame rate and latency) was subjectively good as the application even worked on comparably weak hardware such as the Google Glass. Lastly, additional data transfer has not been tried out but should be possible by abusing the instant messaging functionality.

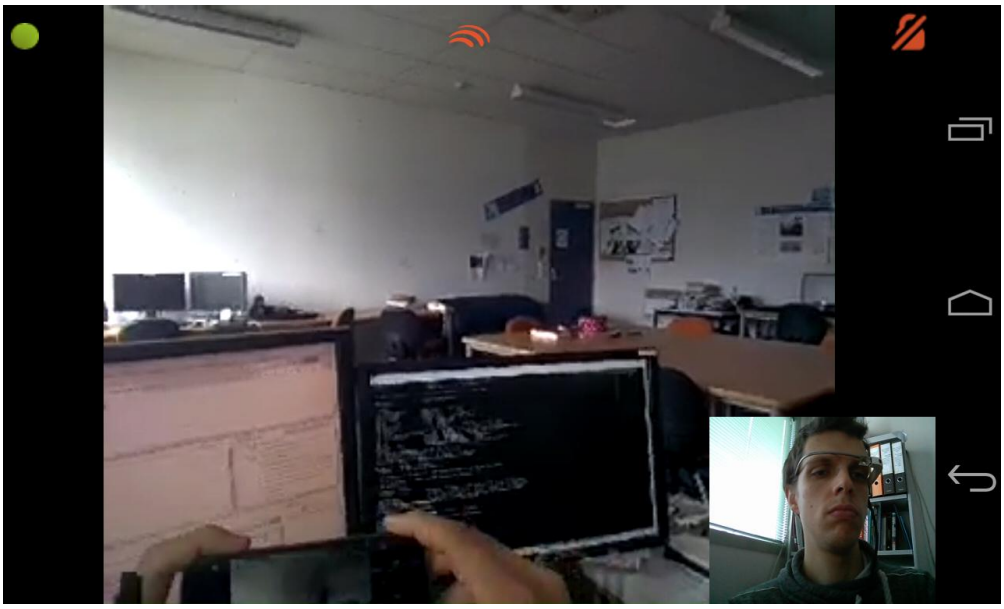


Figure 3.6: We implemented a linphone demo client which runs on a Google Glass and can call a normal Linphone Android client on a Nexus 4 for example as shown in this photo.

3.5 WebRTC

We were not fully convinced that Linphone is the optimal solution, so we decided to try WebRTC as well. It has originally been developed by Global IP Solutions, a company that has been bought by Google in 2010 which is now the driving force behind it. WebRTC is an API specfication to enable real-time communication in web browsers. Since 2011, the World Wide Web Consortium (W3C) is working on its standardization. Three major implementations of WebRTC exist so far: Google's own implementation which is used in Chrome[28], Mozilla's implementation for Firefox[29] and an implementation initiated by Ericsson which is now developed as openwebrtc[30]. All

[28] http://www.webrtc.org/native-code
[29] https://wiki.mozilla.org/Media/Standalone_WebRTC
[30] http://www.openwebrtc.io/

of these implementations are open source. We decided to build upon Google's WebRTC implementation, as Mozilla's implementation cannot be built as a standalone from the browser for Android and openwebrtc doesn't support Windows yet. Moreover, Google's implementation has a very helpful and active developer community which supports developers through a mailing list[31] and bug tracker[32].

3.5.1 Architecture

The code base and nomenclature of Google's WebRTC implementation that have historically grown are still changing and can be quite confusing. The code base has different subfolders including folders named *chrome* which contains parts of Chrome's code base that is shared, *talk* seems to originate from Google Talk source code, and *webrtc* which does not include the complete native implementation of the WebRTC API though.

The Jingle protocol which has originally been used for WebRTC also left its marks in the architecture of WebRTC, as there are many libraries starting with libjingle in their name. The most important library is libjingle_peerconnection which contains the complete native implementation of the WebRTC API written in C/C++. On top of that there is libjingle_peerconnection_so for Android which contains additional JNI interfaces to access the native implementation with Java. In combination with libjingle_peerconnection_jar, which adds a usable Java interface on top by using the JNI layer. With this it is relatively easy to build a standalone Android application which uses WebRTC.

3.5.2 Protocols

WebRTC uses a couple of protocols to accomplish it's tasks and some basic knowledge about these is required for the implementation of a WebRTC application. Most importantly, WebRTC leaves the actual implementation of the signaling to the application developer. It uses SDP, which is basically a text based format to describe session information. The application developer has to make sure to transfer the SDP strings delivered by WebRTC from one client to another. Web applications therefore most commonly use websockets to connect to a webserver that handles this transfer of strings between clients. However there are different methods possible, such as phones directly exchanging the strings via text messages for example.

The next important protocol is the Interactive Connectivity Establishment (ICE) which is necessary to build a peer-to-peer connection between clients. If every device would have an own IP address, it would be easy to directly connect between two clients. However many devices are behind routers in private networks using Network Address Translation (NAT) or might have firewalls preventing connections from outside. ICE therefore figures out a way to connect two clients using Session Traversal Utilities for

[31] https://groups.google.com/forum/#!forum/discuss-webrtc
[32] https://code.google.com/p/webrtc/issues/list

NAT (STUN) servers, which resolve the public IP and help figuring out if the clients sit behind a NAT. If still no peer-to-peer connection can be made, a Traversal Using Relay NAT (TURN) server is used to establish a connection. The TURN server then routes the traffic between the two clients just by forwarding whatever data it gets from one client to another. Although TURN servers are usually fast as they only forward the data like a repeater, they add latency as the connection is not a peer-to-peer connection anymore.

Finally WebRTC uses the Real-time Transport Protocol (RTP) to transfer the audio, video and even data streams usually encoded in Opus and H.264 or VP8 for WebRTC and is the same protocol used by SIP and H.323. By default WebRTC uses the secured profile of RTP, which is consistently called Secure Real-time Transport Protocol (SRTP).

3.5.3 Setup

Setting up the WebRTC code base for development is a bit more challenging than for Linphone. The checkout of the code base also requires a lot of space: more than 13 GB for the pure checkout and more than 15 GB in sum including files that are built during compilation. Furthermore, the access to the code base changes over time as the code base is still in heavy development.

To check out the source code, Google provides its own tools named `depot-tools` which are also under active development. Once `depot-tools` have been installed, the program `fetch` is used for the first checkout and then the program `gclient` is used for updating the source code. The build system used for WebRTC is `gyp` together with `ninja`.

At the moment getting the source code on Arch Linux can be done by using the following command line statements:

```
ln -s $(which python2) ~/bin/python
export PATH=~/bin/python:$PATH
export JAVA_HOME=/usr/lib/jvm/java-7-jdk
export GYP_DEFINES="OS=android"
fetch webrtc_android
```

The first two lines are necessary as WebRTC requires Python 2 instead of Arch Linux' Python 3. The `JAVA_HOME` environment variable makes sure that Oracle's Java 7 JDK is used when multiple Java versions are installed. `GYP_DEFINES` sets the necessary build system parameters to build the WebRTC standalone libraries for Android.

The fetch command will create a directory *src* containing the checkout. Subsequent updates to the code require the same three environment variables set in the above listing and then the commands `git pull` and `gclient sync` have to be executed in this directory.

To build the WebRTC standalone, the same environment variables have to be set again and then the command `ninja -C out/Debug libjingle_peerconnection_jar` builds the standalone libraries as debug version.

3.5.4 Example Applications

Google's native WebRTC implementation comes with two Android demo applications. WebRTCDemo uses the functionalities provided in the *webrtc* folder of the code base and while allowing detailed control over network and codec parameters it is not using the WebRTC API directly and tightly integrated into the build system. As a consequence it is also not easily possible to connect to other WebRTC applications with this demo program. The AppRTCDemo app however uses the Java API provided by libjingle_peerconnection_jar and is the reference implementation for other Android clients.

The AndroidRTC[33] app is also based on Google's libjingle_peerconnection_jar and uses the ProjectRTC[34] server for connection establishment. However when trying these two applications, they didn't work as it's code hasn't been updated with the current development of Google's library.

Another application worth mentioning is the Kurrento[35] server. It is a streaming server that the clients connect to. While this is obviously not a peer-to-peer connection anymore, the server allows for different features like group calls and it also provides OpenCV integration.

3.5.5 Implementation

We decided to build our WebRTC application based on the AppRTCDemo application, but we started with a new application to implement a more lightweight client.

Server

The AppRTCDemo uses a Google App Engine Server for signaling and handles clients connecting to rooms where clients can join and then get connected to each other. As we wanted to keep this part as simple as possible, we implemented an own signaling server in Go[36] using websockets. The server waits for two clients to connect and simply forwards the messages between them. The clients disconnect from the signaling server as soon as they are connected between each other. This results in a behavior similar to Chatroulette[37] where random users are connected. The server also serves a HTML page that allows to connect with browsers using a Java Script based WebRTC client implementation.

[33] https://github.com/pchab/AndroidRTC
[34] https://github.com/pchab/ProjectRTC
[35] http://www.kurento.org/
[36] http://golang.org/
[37] https://www.chatroulette.com/

Client

The documentation of the native WebRTC API[38] and the WebRTC Standard (W3C, 2015) describe how the connection establishment with WebRTC works. Following the example of the AppRTCDemo application, it is relatively straightforward to implement a WebRTC client. A `PeerConnectionFactory` object is used to create all objects. Then a `PeerConnection` object is created and a local `MediaStream` object is added, which itself obtains the necessary objects for audio and video streams attached. Afterwards, the calling client, which in our case is the client that connected to the signaling server first, calls the `createOffer` method of the `PeerConnection` object. The resulting SDP packet gets transmitted to the other client which responds by calling the `createAnswer` method of its own `PeerConnection` object. Additionally ICE candidates, which are possible addresses for the peer-to-peer connection are exchanged via the signaling server. To handle the callbacks from WebRTC, the two observer interfaces `PeerConnection.Observer` and `SdpObserver` have to be implemented.

Additional Data Transfer

WebRTC supports data channels, and we could easily add a data channel to the application by calling the `createDataChannel` method of the `PeerConnection` object and implementing the `DataChannel.Observer` interface. Messages can be sent with the data channel's `send` method and be received with the observer's `onMessage` method.

Pixel Data Access

Ideally we wanted to have pixel data access in native C++ code instead of Java. This was easily possible by simply implementing the `webrtc::VideoRendererInterface` which requires a `RenderFrame` method to be implemented. This method has a single parameter, which is the video frame as `cricket::VideoFrame` pointer. These video frames are stored in buffers for the YUV planes and can be copied with the `CopyToPlanes` method. The format is YUV 4:2:0 with all separate planes.

3.6 Conclusion

In this chapter we systematically went through the process of selecting a proper video calling base for our telepresence application. After defining the requirements for this base we analytically discussed available codecs, protocols, libraries, clients and server applications. The available Android clients were tested and two clients, Linphone and the WebRTC based AppRTCDemo showed the biggest potential. These two clients were then evaluated in more detail regarding our requirements. Linphone allows for additional data transfer and pixel data access, but it is difficult and complex to use this functionality. WebRTC on the other hand provides this functionality with a clean and

[38] http://www.webrtc.org/native-code/native-apis

easy interface. Furthermore the better documentation and demo applications as well as the supportive development community and safe future of being an upcoming web standard make the final decision to select WebRTC for further development easy.

4 Mapping the Environment

In this chapter we will start by discussing possible mapping solutions to represent the environment of the application's users. The chapter continues with a theoretical and mathematical model of the chosen cylindrical panorama mapping and shortly discusses how this has been implemented on the smartphone utilizing the phone's GPU. We conclude with issues of the cylindrical projection and its current implementation.

4.1 Mapping

To enable the users to look around in the remote environment it is necessary to have enough data to create a visual display of the remote environment. We call this data a map and there are many possibilities to represent and generate this map. The probably most simple representation would be a single photo. On the other end is a full 3D representation with a quality that allows for realistic rendering. Of course this is not feasible at the moment due to limited storage and processing capabilities, so the solution is to find a representation somewhere in the middle.

The generation of the map can happen in advance, so 3D models can be created by hand, or quadcopters could scan the area. However, the need of a pregenerated map contradicts the objective of being mobile and able to use the telepresence system everywhere and constrains the system to those locations where a pregenerated map exists. Moreover, a pregenerated map is static and cannot handle any changes or dynamics within the environment, such as people, cars and other moving objects. Therefore, we want an on-the-fly generation of the data.

Many of the other projects in the research literature use a 3D and vision based simultaneous localization and mapping (SLAM) approach. Next to the map generation, as the name already indicates, these techniques include localization which is important for telepresence systems. This can be used to not only have an independent view for the local user, but also to know the location of the remote user.

A general problem of 3D SLAM algorithms is, that they need a translation step for the initialization to create a big enough parallax. The further objects are away, the bigger this initial sideway translation has to be, to get a parallax that is usable for the initialization. As a consequence, it is not very suitable for outdoor applications where most objects are usually further away. Additionally, we argue that the movement is usually limited when using a mobile telepresence application. When people show each other around, the orientation is more important than the location.

Another problem of 3D SLAM is of course that next to requiring more processing power, the result is usually a sparse point cloud that is difficult to render. Approaches that lead to dense point clouds require more memory which is also problematic on mobile phones. Other 3D reconstruction hardware, such as Microsoft's Kinect[1] or Google's Tango[2] also only work for smaller distances, namely indoor environments. Urban environments are already difficult and outdoor environments are even more so.

We therefore use a 2D map of the environment which is based on the orientation. These panoramas can be mapped with different projections. The most common ones are spherical, cubic and cylindrical. Spherical panoramas are usually problematic at the poles of the sphere and the same is true for cubic panoramas at the corners. Cylindrical panoramas are less problematic in this regard, but don't map the whole surroundings as the top and bottom are missing. This however is not a problem as the sky, ceiling and ground are usually not of interest. Another advantage of the cylindrical panorama is that only a single image is needed to store the map and this image shown in plain without any projection is still easily understandable.

4.2 Camera

To generate the map based on camera images, we need to know the intrinsic and extrinsic camera parameters. The intrinsic camera parameters are based on the pinhole camera model, that can be determined via camera calibration and designated for specific cameras. The extrinsic camera parameters are it's location and orientation, and change for our mobile cameras. As we decided for a two dimensional map as panorama around the camera, we only need the camera's orientation and can ignore the location for now.

4.2.1 Camera Calibration

OpenCV[3] offers an Android application that can be used directly for the camera calibration by capturing images of a chessboard pattern. However, this calibration program was not stable enough so we decided to simply take pictures and then used the desktop version of the OpenCV camera calibration program.

4.2.2 Orientation Tracking

For our cylindrical panorama we assume the camera to always be positioned in the center of the cylinder. We therefore don't need the translational part of the external camera parameters and solely concentrate on the orientation. To determine the orientation we investigate using the sensors of the phone and a vision based tracking. We also

[1] http://www.microsoft.com/en-us/kinectforwindows/
[2] https://www.google.com/atap/project-tango/
[3] http://opencv.org/

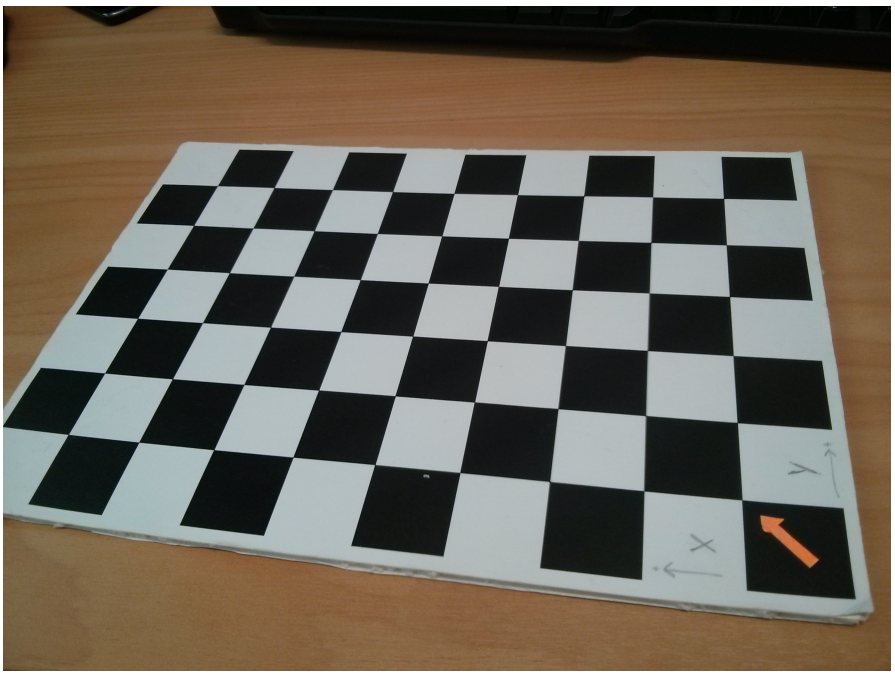


Figure 4.1: A checkerboard pattern with 9×6 inner corners has been used to calibrate the cameras with the OpenCV camera calibration application. This is one of the calibration photos of a Nexus 4 smartphone.

neglect the difference between camera/sensor and screen orientation as the tracking will only be implemented relative to the initial orientation.

Sensor Fusion

One way to track the camera's orientation is to use the phone's built in sensors, namely the accelerometer, the magnetometer and the gyro in an approach usually known as sensor fusion. The magnetometer and accelerometer are used to get an absolute orientation, which however is noisy and slow. Therefore it is used in combination with the gyro, which is precise and quick, but relative and drifts over time. These two orientations are combined by using a low pass filter on the former and a high pass filter on the latter before adding the two signals. We are using an existing implementation[4].

Vision Based Tracking

PanoMT by Wagner et al. (2010) is a vision based tracker, that internally also generates a cylindrical panorama of the environment. It however cannot be used directly as the internal panorama is only built once and not kept up-to-date as it is too slow on the CPU to update the whole panorama. We want a panorama that continuously

[4] http://www.codeproject.com/Articles/729759/Android-Sensor-Fusion-Tutorial

gets updated with the camera stream. However, as PanoMT uses the same cylindrical panorama map and generates it in almost the same way, it's orientation is very accurate and can be used to build an almost perfect panorama. PanoMT empirically works best with a camera resolution of 320×240 and the internal panorama has a resolution of 2048×512, which is the limiting factor for the accuracy of the orientation.

PanoTracker

PanoTracker by Kim, Reitmayr, and Woo (2012) is another visual tracker. As the comparison in figure 4.2 shows, this tracker on average takes 32 milliseconds compared to PanoMT which only takes 7 milliseconds per frame. We therefore decided to use PanoMT as tracker to build the panorama.

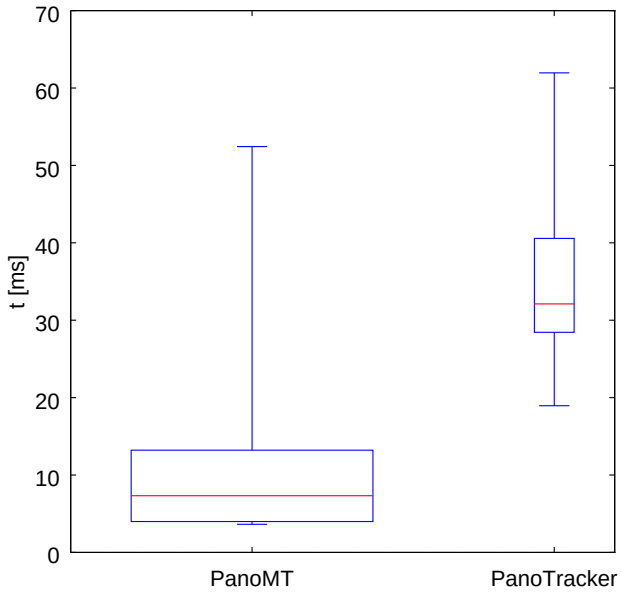


Figure 4.2: The box plot of the time needed to track one camera frame shows that PanoMT is faster than PanoTracker in more than 75 % of all cases.

Sensor Fusion Versus Vision Based Tracking

Figures 4.3 and 4.4 compare two panoramas. The first one is built with a sensor based orientation while the second is built with a vision based orientation. As we can see the sensors are not precise enough to build a nice looking panorama. However for just looking around in a panorama they are sufficient. They are also good when vision based tracking fails, which might be the case for example when there are no features in the camera image, or when the movement of the camera results in camera images with too much deviation from the required homography.

Figure 4.3: This panorama has been generated based on sensor tracking. Next to exposure time color artifacts, we can see that objects are not perfectly stitched together. For example the house and lamp look bent and the people on the right side are doubled.

Figure 4.4: This panorama has been generated with PanoMT. We can again see exposure time and some other artifacts, but in general all objects are stitched together nicely compared to a sensor based panorama.

4.3 Cylindrical Panorama

Our camera is placed in the middle of an open cylinder with a radius of 1 unit, which means it has a circumference of 2π. As the aspect ratio of the panorama is $4:1$, the height of the cylinder therefore is $\frac{\pi}{2}$.

4.3.1 Projection

Recording our map as a cylindrical panorama by using the mobile phone's built in camera, we need to have a way to transform from camera space to map space. The camera space coordinate is given as two dimensional vector $\mathbf{c}$ in $[0,1]$ normalized space and the resulting map space coordinate is also a two dimensional vector $\mathbf{m}$ but in $[-1,1]$ normalized space. This projection needs the intrinsic and extrinsic camera parameters. As intrinsic camera parameters, we only use the linear camera model and thus need the $[0,1]$ normalized principal point $\mathbf{p}$, the resolution $\mathbf{r}$ in pixels as well as the focal length $\mathbf{f}$, also in pixels. The orientation $\mathbf{O}$ of the camera as extrinsic parameters is given as a 3×3 rotation matrix. At this point we want to recall that a rotation matrix is an orthogonal matrix with a determinant of 1. Orthogonality is given when

$$\mathbf{O}\mathbf{O}^T = \mathbf{I} \tag{4.1}$$

and this means, that the inverse of an orthogonal matrix is the same as the matrix transposed.

The first step is to calculate a vector $\mathbf{l}$ pointing from the camera to the direction the camera space coordinate came from in the three dimensional world space by using the intrinsic and extrinsic parameters.

$$\mathbf{l} = \mathbf{O}^T \begin{bmatrix} (\mathbf{c} - \mathbf{p}) \circ \mathbf{r} \circ \begin{bmatrix} \frac{1}{x_\mathbf{f}} \\ \frac{1}{y_\mathbf{f}} \end{bmatrix} \\ 1 \end{bmatrix} \tag{4.2}$$

This vector forms a ray from the origin, where the camera is located, which we will cut with the upright cylinder. The cylinder equation

$$x^2 + z^2 = r^2 \tag{4.3}$$

therefore needs to be fullfilled to get the intersection point $\mathbf{d}$ as

$$\mathbf{d} = \frac{\mathbf{l}}{\sqrt{x_\mathbf{l}^2 + z_\mathbf{l}^2}}. \tag{4.4}$$

Lastly to transform this three dimensional coordinate into a two dimensional and normalized map coordinate, the angle is calculated from the $x_\mathbf{d}$ and $z_\mathbf{d}$ coordinate and normalized together with the $y_\mathbf{d}$ coordinate to finally get

$$\mathbf{m} = \frac{\begin{bmatrix} \tan^{-1}\left(\frac{x_\mathbf{d}}{z_\mathbf{d}}\right) \\ 4y_\mathbf{d} \end{bmatrix}}{\pi}. \tag{4.5}$$

4.3.2 Panorama Size

Based on the projection equations we can calculate an approximation of the panorama's resolution based on the distance between two camera space pixels in map space. We first assume a principal point

$$\mathbf{p} = \begin{bmatrix} \frac{1}{2} \\ \frac{1}{2} \end{bmatrix}, \tag{4.6}$$

which has no influence on the outcome, as well as an orientation of

$$\mathbf{O} = \mathbf{I}, \tag{4.7}$$

which could be problematic as it means looking at the equator of the cylinder. It's upper or lower end in contrast is further away and thus means, we need a higher camera space resolution to cover as many map space pixels as in the center, but we focus on the equatorial area, which in practice will be more important as already discussed. It also doesn't consider rotation around the looking axis, but this should – as we will later see – not influence the result if the focal lengths are nearly the same.

First, we need to define the camera space coordinate based on the pixel coordinates i and j as

$$\mathbf{c} = \begin{bmatrix} \frac{1}{2x_\mathbf{r}} + \frac{i}{x_\mathbf{r}} \\ \frac{1}{2y_\mathbf{r}} + \frac{j}{y_\mathbf{r}} \end{bmatrix}. \tag{4.8}$$

This equation can then be inserted into the first projection equation to get

$$\mathbf{l} = \begin{bmatrix} \frac{\left(\frac{1}{2}+i-\frac{x_\mathbf{r}}{2}\right)}{x_\mathbf{f}} \\ \frac{\left(\frac{1}{2}+j-\frac{y_\mathbf{r}}{2}\right)}{y_\mathbf{f}} \\ 1 \end{bmatrix} = \begin{bmatrix} c_x + \frac{i}{x_\mathbf{f}} \\ c_y + \frac{j}{y_\mathbf{f}} \\ 1 \end{bmatrix}, \tag{4.9}$$

where we defined the constants

$$c_x = \frac{\frac{1}{2} - \frac{x_\mathbf{r}}{2}}{x_\mathbf{f}} \tag{4.10}$$

$$c_y = \frac{\frac{1}{2} - \frac{y_\mathbf{r}}{2}}{y_\mathbf{f}}. \tag{4.11}$$

Following the calculation of the intersection point

$$\mathbf{d} = \frac{\begin{bmatrix} c_x + \frac{i}{x_\mathbf{f}} \\ c_y + \frac{j}{y_\mathbf{f}} \\ 1 \end{bmatrix}}{\sqrt{\left(c_x + \frac{i}{x_\mathbf{f}}\right)^2 + 1}} \tag{4.12}$$

and the transformation into a normalized map coordinate

$$\mathbf{m} = \frac{\begin{bmatrix} \tan^{-1}\left(c_x + \frac{i}{x_\mathbf{f}}\right) \\ 4\frac{c_y + \frac{j}{y_\mathbf{f}}}{\sqrt{\left(c_x + \frac{i}{x_\mathbf{f}}\right)^2 + 1}} \end{bmatrix}}{\pi}, \tag{4.13}$$

we can then start to determine the distance between two pixels in map space. We start by calculating the horizontal distance as

$$\Delta x_\mathbf{m} = \frac{\tan^{-1}\left(c_x + \frac{i}{x_\mathbf{f}}\right) - \tan^{-1}\left(c_x + \frac{i-i}{x_\mathbf{f}}\right)}{\pi}. \tag{4.14}$$

We are interested in the maximum distance between two pixels and therefore linearize the arcus tangens function with the highest slope which is

$$\max \frac{d \tan^{-1}(x)}{dx} = 1 \tag{4.15}$$

at $x = 0$. Using this linearization, we get

$$\max \Delta x_\mathbf{m} = \frac{c_x + \frac{i}{x_\mathbf{f}} - c_x - \frac{i-1}{x_\mathbf{f}}}{\pi} = \frac{1}{\pi x_\mathbf{f}} \tag{4.16}$$

as highest horizontal distance between pixels. For the vertical distance we get

$$\Delta y_\mathbf{m} = \frac{4}{\pi\sqrt{\left(c_x + \frac{i}{x_\mathbf{f}}\right)^2 + 1}} \left(c_y + \frac{j}{y_\mathbf{f}} - c_y - \frac{j-1}{y_\mathbf{f}}\right). \tag{4.17}$$

Looking for the maximum distance again, we can see that the square root term in the denominator gives

$$\max \frac{1}{\sqrt{\left(c_x + \frac{i}{x_\mathbf{f}}\right)^2 + 1}} = 1 \tag{4.18}$$

and therefore we get

$$\max \Delta y_\mathbf{m} = \frac{4}{\pi y_\mathbf{f}}, \tag{4.19}$$

which, considering a $4:1$ aspect ratio is no surprise, given the maximum horizontal distance $\frac{1}{\pi x_\mathbf{f}}$ and assuming an (almost) equal horizontal and vertical focal length.

If we define this maximum distance between pixels to be the same as the distance between pixels in the map, we can now simply calculate the resolution of the panorama as

$$\mathbf{r}_m = \begin{bmatrix} \frac{2}{\max \Delta x_\mathbf{m}} \\ \frac{2}{\max \Delta y_\mathbf{m}} \end{bmatrix} = \begin{bmatrix} 2\pi x_\mathbf{f} \\ \frac{1}{2}\pi y_\mathbf{f} \end{bmatrix}, \tag{4.20}$$

where the factor two stems from the map space coordinates being in the range $[-1, 1]$. As the maps resolution depends linearly on the focal length in pixels which for a given camera depends linearly on the resolution, we can conclude that doubling the resolution of the camera image, we also should double the resolution of the panorama.

Let's assume a resolution of 320×240 pixels for the camera and a focal length $f = x_\mathbf{f} = y_\mathbf{f} = 282$. We get a map resolution of 1772×443, which we can round up to the next power of two as 2048×512. Figure 4.5 shows a plot of this scenario with a projected camera image onto the map where each point represents a pixel. We can see that the blue camera image pixels have a similar resolution as the red map pixels are showing.

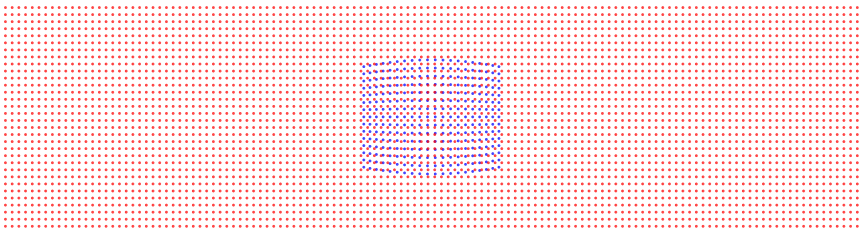


Figure 4.5: A 320×240 pixel camera image (blue) with a focal length of $f = 282$ mapped on a 2048×512 pixel panorama (red) has about the same pixel size as the panorama. Pixels in every 16^{th} row and column are represented by a dot in this plot.

4.4 Implementation

We implemented an Android application with a graphical pipeline starting with the camera image and outputting a rendered view of the generated panorama. The pipeline has been implemented in C++11 native code and consists of two major classes, the `FrameReceiver` class which gets frames from the camera and executes the tracker, and the `Renderer` class which uses shaders implemented in GLSL to generate a panorama based on the tracked camera frames and renders a view of this panorama to the screen.

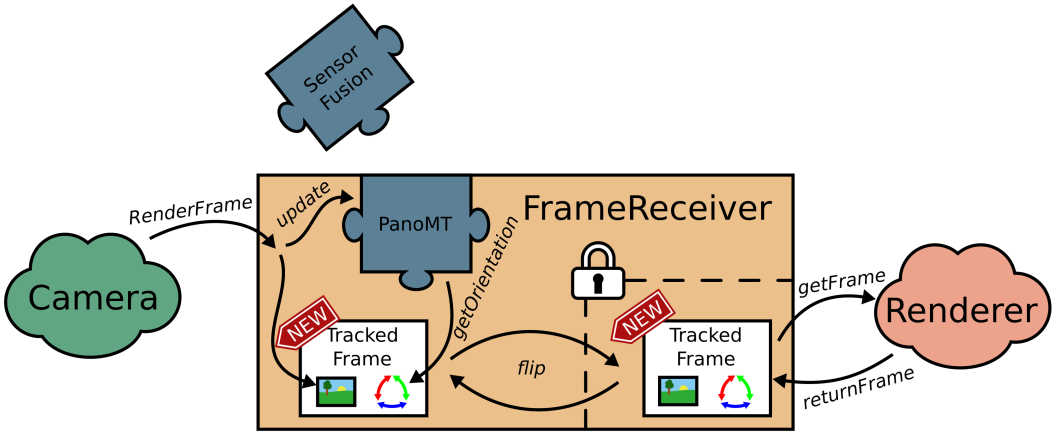


Figure 4.6: The FrameReceiver class handles the tracking of incoming camera frames. When a new camera image comes in, it is tracked with either PanoMT or with the sensors. It is stored as internal TrackedFrame displayed on the left, which is then marked as new. This internal TrackedFrame can be flipped with the external, when the Renderer is not accessing it at the moment. This is guarded by a mutex to ensure thread safety, as the Renderer runs in a different thread.

4.4.1 Tracking

Figure 4.6 shows the `FrameReceiver` class that handles the incoming image data, tracking and buffers the frames for the renderer. As soon as a camera frame arrives, it is sent to the tracker to get the orientation corresponding to the camera image, which is stored together with the frame in an internal `TrackedFrame` object.

Next to the internal, the `FrameReceiver` handles a second `TrackedFrame` object. This external frame is accessible from outside and can be – secured by a mutex – accessed from the rendering thread. After retrieval of the frame, it has to be returned to the `FrameReceiver` before a new frame can be made accessible. This is done with the an internal `flip` method that exchanges internal and external thread whenever possible and a new internal frame is available. The frames therefore have a `new` flag, that marks the internal frame as new when tracked and ready to be swapped with the external one. It is unflagged when the external frame has been returned by the renderer.

The `TrackedFrame` also stores flags for whether the tracker has been reset and if the current frame was successfully tracked. If the tracking fails or if comparably slow visual tracking is not needed, as the frames will not be used to update a panorama, the `FrameReceiver` is able to use a sensor fusion based tracker instead of the visual tracker.

4.4.2 Shader Implementation

A template class named `Shader` implements generic shader functionality that all shaders need. The simplest shader implementation classes, which do not need uniforms only

have to derive from this class using the curiously recurring template pattern (CRTP) and set some constant class member variables. CRTP is only used to be able to access these constant class members from the `Shader` class. Shaders using uniforms simply need to add one or more methods to set their uniform values.

The GLSL shader sources are implemented as two constant class member strings. To aid the development, a shell script can be run before compilation that updates those strings in the C++ source files based on shader source files. These have the same file name as the C++ files and the file endings `vs` for the vertex shader or `fs` for the fragment shader. Additionally the shell script allows those shader sources to have `#include` statements that simply replaces this line with the content of the included file. These include files usually have a `glsl` file ending and cannot further include other files.

4.4.3 Panorama Update

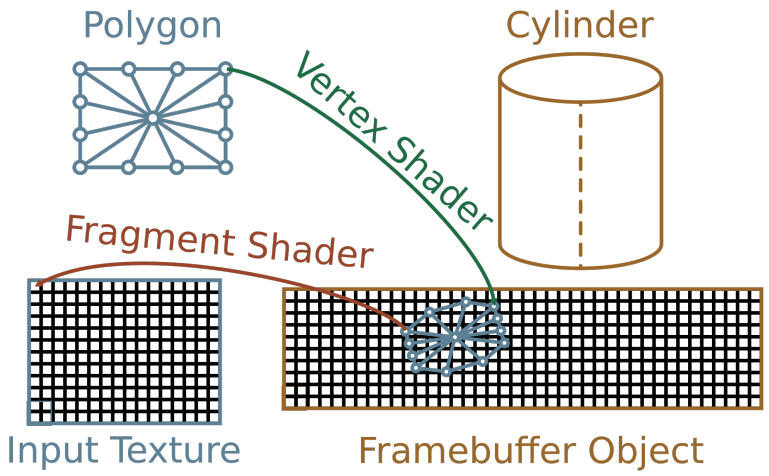


Figure 4.7: The panorama update shader first transforms a rectangular polygon from camera space to map space. The fragment shader then does the inverse transformation to look up the color to draw to the panorama in the camera image that gets drawn onto the panorama.

The process of updating the panorama is shown in figure 4.7. To update the panorama map with a given camera frame and camera parameters, a simple polygon with camera space coordinates is rendered as a triangle fan. The rendering target is a frame buffer targetting the panorama texture. The vertex shader transforms the polygon's vertices from camera space to map space. This deforms the originally rectangular polygon to a convex polygon. The polygon has a central vertex, which should prevent problems with a slightly concave polygon, which could happen for example if non-linear intrinsic camera calibration is added. The fragment shader then transforms the input map space coordinates back to camera coordinates and uses them for a texture lookup on the camera frame, which is subsequently drawn onto the map.

4.4.4 Panorama Rendering

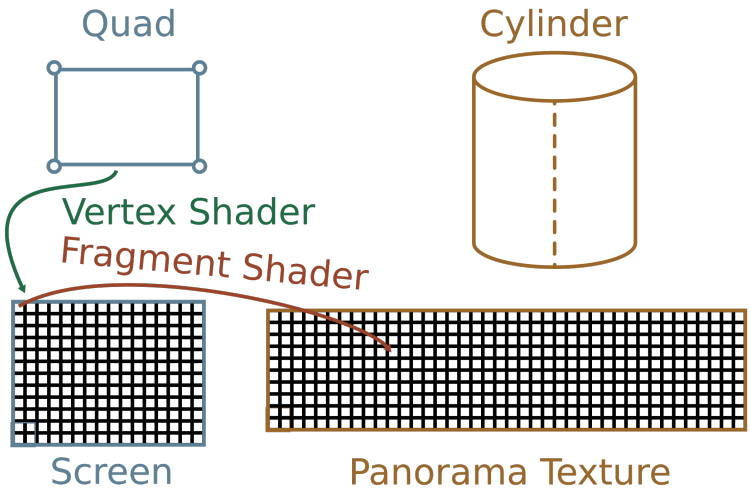


Figure 4.8: The panorama rendering shader draws a quad that completely fills the screen space. The fragment shader transforms from screen to map space to look up the pixel to draw from the panorama.

Figure 4.8 shows the process of rendering to the screen based on the panorama map. A simple screen filling quad is rendered with the vertex shader doing no transformation at all. The fragment shader then transforms the camera space coordinate into map space for a texture lookup into the panorama map.

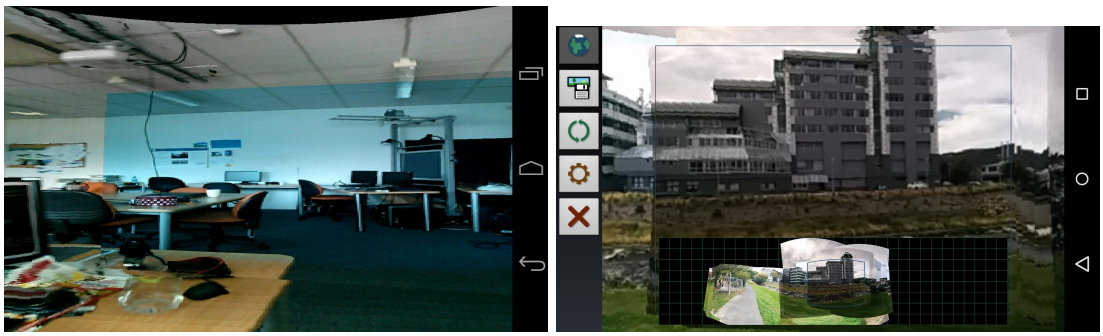
4.5 Issues

Clearly even though a cylindrical panorama has many advantages, it also has its shortcomings, which we will discuss briefly here.

4.5.1 Adaptive Cameras

The cameras in current smartphones have to be easy to use by average users, so they handle focus, exposure time and white balance by themselves. Professional photographers control these settings manually to get the best out of their pictures. In our panorama generation case, we would like to have more control over these settings as well, but the hardware automatically changes the settings without us being able to influence it.

As a result automatic white balancing creates nice single pictures, but when multiple pictures are combined together it is easy to spot the difference as can be seen in figure 4.9a for example. Automatic exposure time setting has a similar effect as can be seen in figure 4.9b.



(a) auto white balancing artifacts (b) auto exposure artifacts

Figure 4.9: Auto white balancing and auto exposure create similar gradients and edges during panorama generation when the camera automatically changes the settings. These issues can be corrected by fixing the white balancing and exposure settings.

The solution of this problem is relatively easy. Current trends are to go from fully automated hardware to APIs that allow developers to control the settings of the camera, such as shown with the frankencamera by Adams et al. (2010). Android 5 has a new camera API[5] that when fully supported by the device, supports setting the camera parameters to fixed values which then result in better quality panoramas. The Nexus 4 phone only has legacy support for this API and doesn't allow such settings, while the Nexus 5 has full support for the new API.

4.5.2 Translation

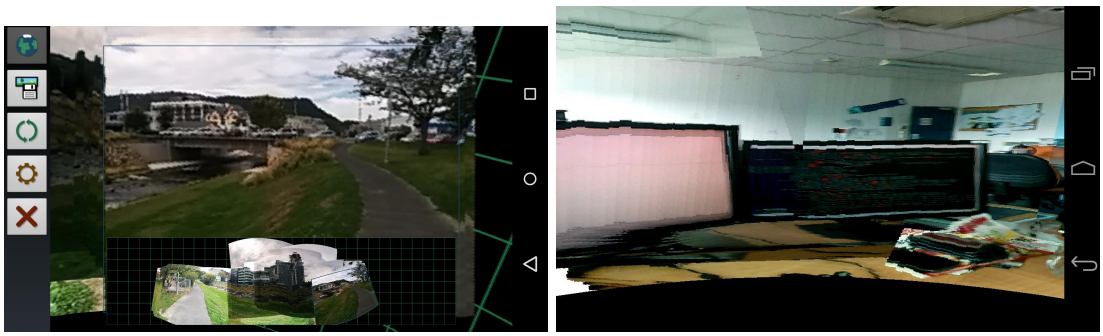
As we only consider the orientation as extrinsic camera parameters and no translation, a translation results in problems during the generation of the panorama as can be seen in figure 4.10a. Rotating the smartphone perfectly around the center of the camera would not result in any problems. Even when the user stands still and looks around this inflicts a translation as the center of rotation is the center of the user's body, about an arm length away from the camera's center. DiVerdi, Wither, and Hollerer (2008) have shown that for this translation objects have to be at least 5 meters far away so that the resulting error after a full rotation (360°) is less than 0.1°. Closer objects cause problems as shown in figure 4.10b.

4.5.3 Panorama Tracking and Generation

No matter how long it takes, a static scene will end up in a similar panorama. Our environment is not static however, and moving objects such as people and cars can result in erroneous panoramas like in figure 4.11.

Moreover the PanoMT tracker internally uses a static panorama, that once created is never updated again. As a result, changing environments can lead to failure of the

[5] https://developer.android.com/reference/android/hardware/camera2/package-summary.html



(a) movement during outdoor use (b) movement during indoor use

Figure 4.10: Movement causes issues as the assumption that the camera is only rotated but not translated is invalidated. The error caused is inversely proportional to the distance of the objects in the scene. In outdoor scenes issues are caused when the user walks during panorama generation. Indoors this can happen by simply turning around the center of the own body instead of turning around the center of the camera as objects are usually much closer.

Figure 4.11: A changing environment can easily transform a normal car into a stretch limousine during panorama generation.

tracker. This could of course be fixed by updating the internal panorama as well for most of the cases. Big and quick changes might still lead to failure as no correlation between the current panorama and the changed environment can be found anymore. This can be prevented by making sure that the camera movement and changes are slow enough in comparison to the frame rate with which the panorama is updated.

Other reasons why the tracker might fail are looking at the caps of the cylinder or a lack of features to match, such as on big homogenously colored surfaces. The tracker needs these features to compare the current camera image with the internal panorama and determine the camera's orientation based on matching features.

5 Telepresence System

In this chapter we will now merge the work of the previous two chapters, the video calling and the map generation to create a novel telepresence application. In this context we will talk about two users being in two environments and usually call them either the local or remote user or environment. Of course this description is interchangeable. One user is in the local environment and sees the other user as remote user being in the remote environment, while for this user it is the opposite.

The following section discusses possible system architectures regarding to the network model, whether a server is used or not and which kind of data is transfered. After this theoretical part, the practical part of merging the two systems will be discussed starting with an overview of the software architecture, describing some issues to be considered on the way and important implementation details. We conclude the chapter with a brief evaluation of the system.

5.1 System Architecture

There are many possible architectures for the application regarding which data is transfered and as a result which data has to be processed on which device. We will start with an overview of which data can be transmitted and then we will have a look at possible architectures that minimize data transfer and required processing. We will investigate the architectures based on the assumption that the local user is looking around in the remote environment and this happens symmetrically. This means that both users look around in the other user's environment. How environments are switched and synchronized so that both users are in the same environment will be described later in this chapter.

5.1.1 Data to Transfer

Apart from audio which is not altered or processed in any way, we can transmit the recorded camera images, the tracked orientation and the resulting panoramas. To understand the implications of these different types of data, we have to split the panorama generation into two processing steps. The first one is the orientation tracking and the second one is the panorama stitching. The camera frame is needed for the visual tracking that results in the orientation. The orientation and the camera frame are then used to stitch the panorama. It is possible to transfer data from one device to another at any stage during the processing. Figure 5.1 shows the symbols used for the different types of data in the following diagrams.

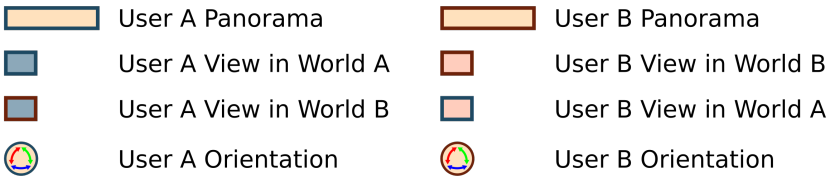
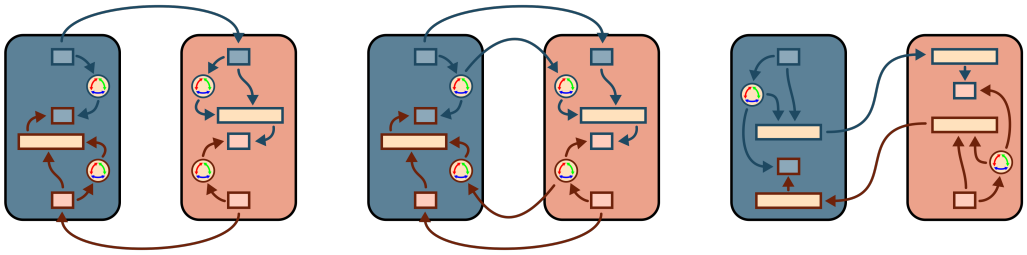


Figure 5.1: This diagram shows the symbols used in the following diagrams. The colored borders show which user the data comes from, where blue represents User A data and red represents User B data. The wider rectangles represent panoramas, the smaller ones camera images and the round objects the orientation of a user. The camera frames have a colored filling which tells which user's view it is. Therefore, a rectangle with a blue border and a red filling is User B's view into User A's environment.

5.1.2 Serverless Architectures

The first and most simple solution regarding modification of the video call signal is to simply transmit camera frames only as shown in figure 5.2a. With this condition we will look at which processing has to be done: For the remote camera frames we need to do all the processing, that means tracking and stitching to get the remote panorama. A local panorama is not needed, but the local orientation is needed to look into the remote panorama, so for the local camera frames we have to do the tracking as well or alternatively use sensor based tracking. On one hand the disadvantage of this approach is clear: both communication partners have to do the tracking for both camera streams in case of pure visual tracking. This means a lot of processing and thus a strong enough device is required. On the other hand the advantage is, that no special data has to be transmitted, which allows an easy implementation and even supports different clients that do not do any of these steps.



(a) camera transfer only (b) tracked frame transfer (c) panorama transfer

Figure 5.2: The serverless system architectures shown in this diagram each transfer different data requiring different processing steps on each phone.

Minimizing the processing, there are two possibilities. For the local camera frames we either do tracking and then transfer the orientation next to the camera frames, or stitch the panorama and transfer the panorama directly. In the former method shown in figure 5.2b we would now track the local camera frames and send them with the

resulting orientation and receive the remote camera frame and orientation and stitch that into the remote panorama. The big advantage of this approach is the minimum of processing and data transfer required for each device. The disadvantage is that the orientation has to be transfered linked to the camera frames which depending on the implementation is more difficult to achieve and eventually makes the application incompatible with other clients.

The latter approach as depicted in figure 5.2c has the stitching on the local site. This means that the local panorama is calculated, which is not really necessary. The big disadvantage of this solution is that the transmission of the panorama requires more data to be transfered than the camera frame itself. On one hand transfering the whole panorama results in a lot of redundance. On the other hand, transfering only updated parts of the panorama makes the process highly complicated, as different amounts of data would need to be transfered for every frame. Other clients are supported in the first case, but this time the whole panorama will be displayed instead of the camera frames.

5.1.3 Server Architectures

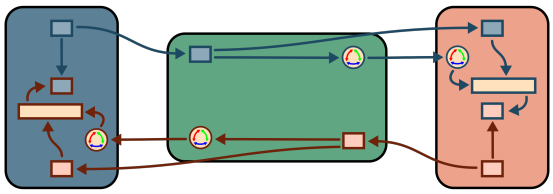
Introducing a server into the architecture on one hand has the general advantage of being able to offload processing from the clients. This allows less powerful clients and even for those that are powerful enough allows them to save on energy, which is important considering the short battery life of current smartphones. On the other hand the server adds latency in any real world scenario, which brings the big disadvantage of a worse user experience.

We also have to take in consideration another processing step. So far the views of a device into the remote panorama have been assumed to be only executed on the local device. With a server we can consider moving this processing step to the server as well. This processing step could of course be done on the remote device as well, but this would mean that it needs to process the local panorama instead of the remote one. In contrast to the second scenario of the serverless architectures this approach only adds latency as the orientation has to be sent to the remote partner. Only then the view of the panorama with this orientation can be returned. In this case the local view could be sent instead of the orientation, but this would not only double the amount of video data to be transfered, but also add more redundant processing, as another tracking step is required.

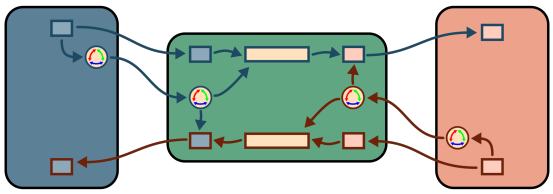
With three devices, two clients and a server, and four processing steps for one direction, there are $3^4 = 81$ possible ways to have the processing steps spread over the devices. Most of them don't make sense as the data would need to be transfered too often or in a direction opposite of the sender-server-receiver direction. We of course also rule out any configuration where the server doesn't have any processing step. Considering the data stream in the other direction, which has the same processing pipeline, we see that the tracking step can be shared, so instead of eigth processing steps, it's possible to merge the tracking steps, resulting in six overall processing steps. Ruling out configurations that don't merge those tracking steps from the remaining configurations, we end

up with six possible configurations. Three of these transfer the panorama, which is inefficient or overly complex, as discussed in the previous section and are therefore not further investiagted.

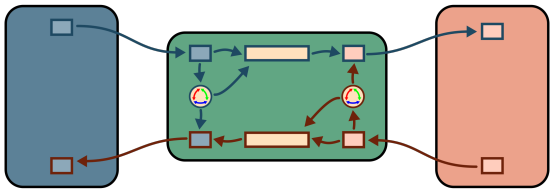
This leaves us with three possibly advantageous configurations which are shown in figure 5.3. One in which the devices handle the stitching and panorama generation (5.3a), while the server does the tracking. The inverse of this configuration, where the devices do tracking and the server does the rest (5.3b), and finally the server doing all of the processing steps (5.3c). Which of the first two configurations is better, depends on which processing steps require more time. Overall the solution where the server does all the work should be the best, as it allows very thin clients that don't have to bring much processing power or spend a lot of energy on the processing. In the end the additional latency of the server has to be compensated with a lower processing time to balance out the increase in overall latency.



(a) serverside tracking



(b) clientside tracking



(c) thin clients

Figure 5.3: The server handles some or all of the processing steps from the phones.

5.1.4 Asymmetric Architectures

Of course it would be possible to do an asymmetric setup, like doing all the processing on one of the two clients. This makes sense, when one of the clients is a lot more powerful than the other. However this would require a dynamic system that negotiates which device gets how much to do. Such a system is difficult to implement and not the goal of this work, so this is left open for future work.

5.1.5 Synchronization Problems

Next too the so far theoretical view of the architecture, there is also a practical problem to be considered, which is synchronization. Transfering the orientation linked to the video frame is of course technically possible, but not necessarily simple to accomplish. In practice there are some possibilities to link the orientation data to the video frames. First it is possible to directly encode them in the video data, but the video compression makes it very hard to restore the data without errors at the same time as not changing the video noticably for the viewer. The next possiblity is to change the protocol with which the video is transmitted and add the orientation information there. The big disadvantage here is, that this makes the application incompatible with other applications using the same original protocol.

The most simple solution is to have a data channel separately from the video, over which the orientation gets transmitted. This however leads to a new problem, which is synchronization. It can be that either the video frame or the orientation arrives first, it could even be that two orientation packages arrive before the video frame for the first one, and even worse, video frames can be dropped in case the network is overloaded.

All these problems can of course be avoided if orientation and video frames don't have to be transmitted together. Leaving those configurations where only both video frames and orientation or panoramas get transmitted, we end up with only one possibility, either serverless or with a server. We decided to begin with the very first serverless configuration, where the video frames only get transmitted. This configuration has been easy to implement and has been proven to work as we will see shortly.

5.2 Combining Video Call and Panorama Generation

In this section we will discuss the combination of video calls and panorama generation. Some problems and design possibilities that arise with this combination will be investigated in detail.

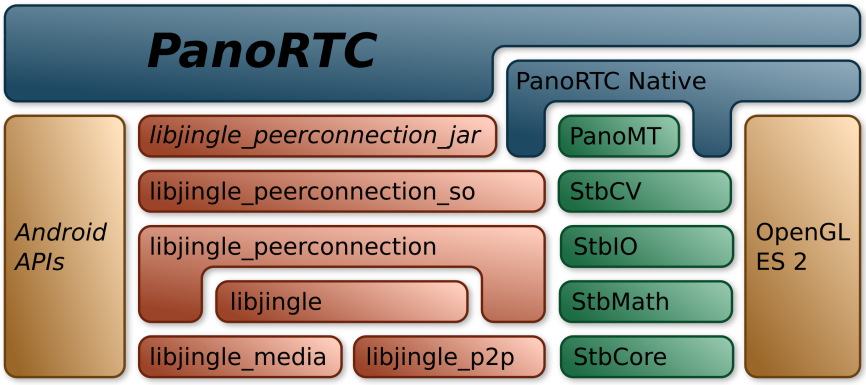


Figure 5.4: The software architecture of our Android application uses standard Android APIs (brown), Google's native WebRTC implementation (red) as well as PanoMT (Wagner et al., 2010, green). The two blue components have been written by us. Cursive font shows that a component is written in Java, while all others are written natively in C/C++.

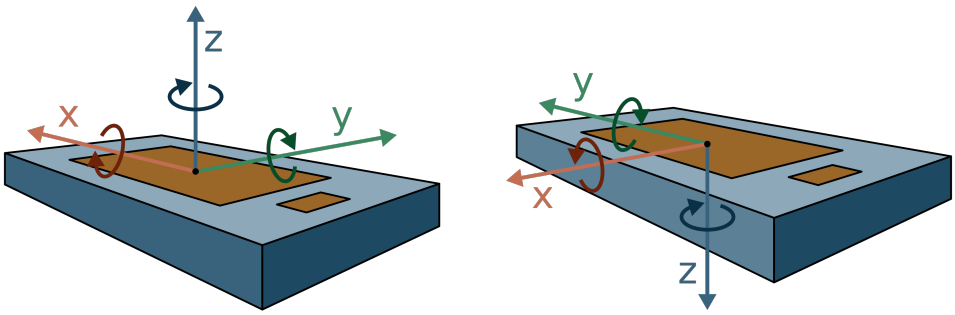
5.2.1 Software Architecture

Figure 5.4 shows the used libraries in the implemented Android application. *Cursive* font shows parts that are implemented in Java, while the remaining parts are implemented natively in C/C++. The red blocks are libraries from WebRTC, while green blocks are PanoMT libraries, the libraries needed for the visual tracking. The two yellow blocks are APIs that are available on Android by default. That is OpenGL ES 2 which is used for GPU accelerated rendering on the phone and the Android Java APIs for the main App development APIs, UI elements and the sensor based tracking. Finally, the blue blocks show the parts of the Android application we implemented. This is an Android activity implemented in Java and the image processing and rendering is implemented natively. Except for the camera frame rendering, all WebRTC functionality is accessed within Java.

5.2.2 Sensors and Visual Tracking

In chapter 4 we found that visual tracking is needed for panorama generation. When looking around in the panorama, the less processing intensive sensor based tracking is good enough. This can be used when looking around in the remote panorama. When looking around in the local panorama, we can still use the remote user's orientation to show where the remote user is looking right now, as will be discussed later. For this case, the remote user's orientation is also based on sensors and transmitted over a WebRTC data channel.

To combine both orientations – visual and sensor based – it is important to use a common coordinate system. Figure 5.5a shows the left handed coordinate system of PanoMT, while figure 5.5b shows the right handed coordinate system of the sensors.



(a) PanoMT (b) sensors

Figure 5.5: PanoMT and the sensor fusion code return their tracking results in different coordinate systems. PanoMT's coordinate system is left-handed and has different axes than the right-handed coordinate system of the sensor based tracker.

To transform any vector $\mathbf{v}_l$ in the left handed coordinate system of PanoMT to a vector $\mathbf{v}_r$ in the sensor based coordinate system, we can use the transformation matrix $\mathbf{T}$:

$$\mathbf{v}_r = \mathbf{T}\mathbf{v}_l \tag{5.1}$$

Based on the axes shown in the figures we determine $\mathbf{T}$ to be

$$\mathbf{T} = \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}. \tag{5.2}$$

A vector $\mathbf{a}$ is rotated by the tracker's orientation to a vector $\mathbf{b}$ with help of the visual tracker's orientation matrix $\mathbf{O}_l$ via

$$\mathbf{b}_l = \mathbf{O}_l\mathbf{a}_l, \tag{5.3}$$

or with the sensor based orientation matrix $\mathbf{O}_r$ via

$$\mathbf{b}_r = \mathbf{O}_r\mathbf{a}_r. \tag{5.4}$$

Using equation 5.1 for $\mathbf{a}_r$ and $\mathbf{b}_r$ we get

$$\mathbf{T}\mathbf{b}_l = \mathbf{O}_r\mathbf{T}\mathbf{a}_l, \tag{5.5}$$

where we can multiply with $\mathbf{T}^{-1}$ from the left to get

$$\mathbf{b}_l = \mathbf{T}^{-1}\mathbf{O}_r\mathbf{T}\mathbf{a}_l. \tag{5.6}$$

Comparing this equation to equation 5.3 we can see that we can transform the right handed orientation matrix $\mathbf{O}_r$ to the left handed orientation matrix $\mathbf{O}_l$ using $\mathbf{T}$

$$\mathbf{O}_l = \mathbf{T}^{-1}\mathbf{O}_r\mathbf{T}. \tag{5.7}$$

As the sensor based orientation is given as ZXY Euler angles, the transformation to PanoMT's left handed coordinate system can be calculated more directly. Based on the axes' orientations as shown in figure 5.5, we can see that the sensor's z-axis rotates in the same direction as PanoMT's. PanoMT's x-axis corresponds to the sensor's y-axis and is rotating in the opposite direction, which means that this angle has to be negated. Finally, PanoMT's y-axis corresponds to the sensor's x-axis and is rotating in the same directon. To use the sensor's rotation angles, we consequently use the angles as ZYX Euler angles negating the third angle.

5.2.3 Synchronization

The orientation resulting from the visual tracking is relative to the orientation of the first tracked frame, while the orientation of the sensors is relative to the magnetic north pole meassured by the magnetometer and the gravitation meassured by the accelerometer. These orientations need to be synchronized and therefore the sensors' orientation is synchronized to the visual orientation by storing the sensor based orientation whenever the visual tracking is started and further calculate the difference between these two orientations.

Another problem that requires synchronization is the difference between the local and the remote user's orientations, as the visual trackers might start with different frames. This is not a problem as long as the users don't interact. As we want to be able to see the remote user's orientation and further interactions we need to synchronize the orientation to actually be in the same environment.

Frame Relative Orientation

One possible solution is to transmit all orientation related data relative to the own orientation and camera frame, which then requires synchronization to the camera frame. The data channel however is not synchronized with the video channel. An investigation of the frames sent over the video channel shows two time related values that potentially could be used for synchronization, which are also shown in log entries produced by WebRTC. Unfortunately further investigation showed that for local frames both attributes are simply set to the local clock value and for remote frames `elapsed_time` is starting at 0 with the first frame and `time_stamp` is also set as local time and not as remote time. This has been tested by setting the date wrong on one of the two phones to clearly see a difference in the timestamp values.

Remote Relative Orientation

To allow the users to exchange their current orientations the difference between their initial orientation must be known so that an actual difference between their orientations can be determined for gaze awareness and pointing. To do this it is necessary to transmit the tracked remote orientation back to the remote user so that he can compare it to his own local orientation. This however would again need synchronization to the frame which is not easily possible as shown.

We solved this problem differently by sending a reset message whenever the tracking is initialized to also trigger a reinitialization on the remote side. This results in an equal initial orientation assuming that the initialization really happens at the same frame. We argue that in case the initialization is not exactly on the same frame, that due to only small differences between two consecutive frames, the difference of the initial orientation then is negligible.

Another advantage of this is, additionally assuming a lossless transmission regarding quality and dropped frames, that the panoramas generated on both sides are equal as well. Last but not least a reset needs to be triggered on connection establishment to initially synchronize the orientations, also resulting in both users looking in the same direction initially which allows them to interact immediately without one having to turn around first for example.

5.2.4 Implementation

The `FrameReceiver` class has been mostly described in section 4.4.1. It implements the `webrtc::VideoRendererInterface` interface and is set as a renderer for WebRTC's frames. One instance handles the local camera image and another one the incoming remote camera images. The external orientation either comes from the local sensors directly or from the remote sensors streamed via a WebRTC data channel.

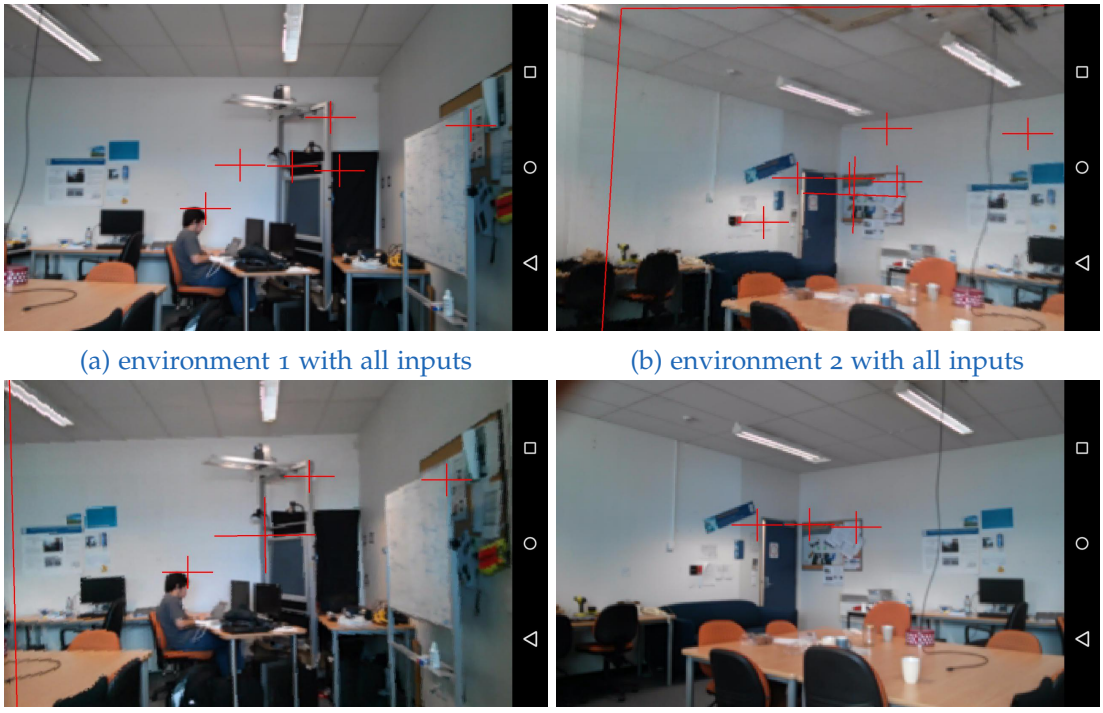
When the visual tracking is bad due to the different tracking problems such as close objects, no features or fast movement, the implementation automatically falls back to the sensor based orientation in case the user is looking around in the local panorama, while panorama updates are disabled until visual tracking works again.

5.2.5 Interface

A thorough discussion of the implemented user interface follows in chapter 6. Here we will briefly discuss three important implementation details to be considered in the creation of the system.

Touch Events

As a very first test, we implemented touch events which add small plus signs to the screen as can be seen in figure 5.6. The input events are projected from the screen to map space and get stored and transfered to the communication partner. Without considering the environment the local user is in, the signs would appear in both environments. As a result, we implemented them to only appear in either the local or remote environment as can be seen in the lower row of figure 5.6.



(a) environment 1 with all inputs (b) environment 2 with all inputs

(c) environment 1 with own inputs (d) environment 2 with own inputs

Figure 5.6: The rendering of touch events has to consider which environment they happened inside. When all touch events are shown in any environment this could easily lead to confusion of the users. The small plus signs in these screenshots show the positions where the users touched the screen.

Switching Worlds

To switch environments a button has been added, so that the users can choose which environment they are in. This switch can also be synchronized so that both users are always in the same environment, which for many use cases makes sense. This could be an employee from a company setting up a machine at a remote construction site, calling a local expert to help. An example of a use case where no synchronization is needed is when two friends show each others their flat, or a couple wants to be in each other's room. Unsynchronized environments have more issues with respect to

privacy concerns, when the user doesn't know if the other user is looking at the own environment or not.

Resetting the Panorama

Next to switching environments, resetting the panorama is a crucial feature to synchronize the environments as described. Additonally to the automatic reset on connection establishment, we also added a button to manually trigger a reset that is of course synchronized via the data channel. Optionally an automatic reset can be enabled that resets the panorama when the environment is switched.

5.3 Evaluation

In this evaluation, we discuss the performance of the system with different available smartphone models. The technical evaluation looks closer at the typical measurements frame rate and latency which are important indicators for the system's interactivity. However, we conclude that the system has different aspects with tracking and map generation on one side and the interactivity of the interface and rendering on the other. The latter is the more important factor for interactivity. Lastly we have a brief look at the panoramas generated on the local and remote phones to see how good these match and allow collaboration in a shared environment.

5.3.1 Hardware

Development started with Google Nexus 4 and Samsung Galaxy S3 phones with Android 4.4. Both models were released in 2012 and are shown in figure 5.7. While implementing the panorama tracking, we noticed a big difference regarding the phones' camera frame rates. According to the the specification of the phones they can record videos at higher frame rates. Access to the camera's frames via the available APIs is not that fast. The Nexus 4 only has a frame rate of around 10 frames per second, while the Galaxy S3 provides the requested 30 frames per second. Consequently we evaluated different Android devices for their performance regarding camera frame and vision based tracking. Figure 5.8 shows the achieved frame rates and most of the tested devices have around 30 frames per second. The frame rate is not correlated with the age of the phones as the minimal difference between Galaxy S3 and S5 shows with the older model having a slightly better frame rate. It is surprising however that quite similar models can show big differences. The Nexus 5[1] and LG G2[2] are both manufactured by LG and have the same CPU, GPU, RAM and resolution. The camera in the LG G2 is better according to the specification, allowing to record 60 frames per second videos at a resolution of 1920×1080. In our test accessing the frames via the

[1] http://www.gsmarena.com/lg_nexus_5-5705.php
[2] http://www.gsmarena.com/lg_g2-5543.php

Android camera API however, the LG G2 is the second slowest at about 18 frames per second, while the Nexus 5 delivers precisely 30 frames per second.

(a) Galaxy S3 (b) Nexus 4 (c) Nexus 5 (d) One M8

Figure 5.7: Four different Android smartphone models have been used during development. We started with the Samsung Galaxy S3 and the Nexus 4 and later we also developed on a Nexus 5 and an HTC One M8.

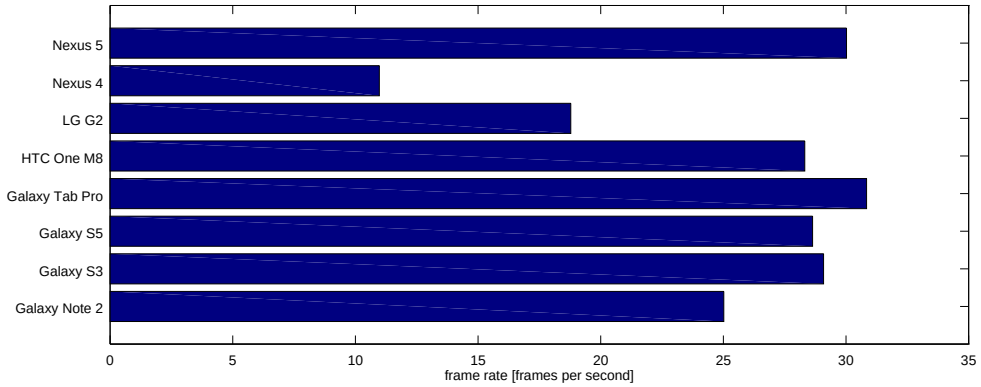


Figure 5.8: In a preliminary test, we tried the camera of different Android devices. This plot shows the camera frame rate of these smartphones and one tablet. The new camera API on the Nexus 5 ensured constant delivery of the requested 30 frames per second, while all the other phones could not deliver the frames fast enough.

5.3.2 Frame Rate and Latency

Figure 5.9 shows the difference between frame time and latency. The frame rate f measured in frames per second (FPS) is the inverse of the time between two frames which we call frame time. In contrast the latency Δt_l is the time from taking the image (shown with a blue pin) to displaying it (shown as a red pin), which can be shorter or longer than the time between two frames.

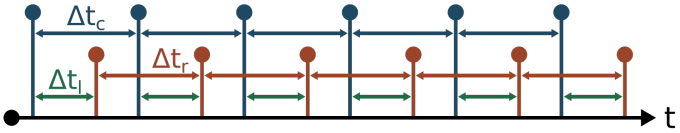


Figure 5.9: The blue pins in this diagram show the moment a frame is taken with the camera and the red pins show the moments a frame is rendered on the display. The time between two camera frames and two rendered frames is the same ($f_c = f_r$) and the frame rate is the inverse of this time. The latency Δt_l in this example is the time it takes from the camera to take the photo until it is rendered to the screen.

In the previous section we measured the camera frame rate f_c based on the camera frame time Δt_c. In general the rendering frame rate f_r is higher than the camera frame rate, which means that the rendering frame time Δt_r is smaller. Thus we have a system with two different frame rates

$$f_c = \frac{1}{\Delta t_c} \tag{5.8}$$

$$f_r = \frac{1}{\Delta t_r}, \tag{5.9}$$

which means that the latency in our system is not a constant, but variable as shown in figure 5.10.

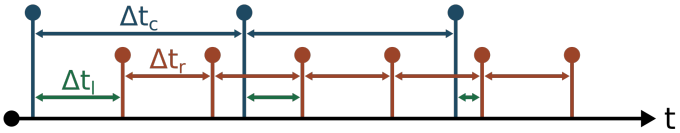


Figure 5.10: In contrast to figure 5.9, the camera and renderer have two different frame rates. As a result, the latency is variable depending on the exact timing of the camera and renderer.

5.3.3 Tracking Performance

Next to a preliminary evaluation of the camera frame rate, we also measured the time needed for tracking on the devices we had access to for a quick test. Figure 5.11 shows the results of this evaluation. The times shown are a part of the processing pipeline and therefore limit the latency and frame rate of the overall system. At time $t = 0$ the callback function is called with a new camera frame. The following steps are copying the frame to our own memory buffer (copy), tracking the frame with PanoMT (track), waiting for the rendering thread to pick up the tracked frame (pickup) and updating

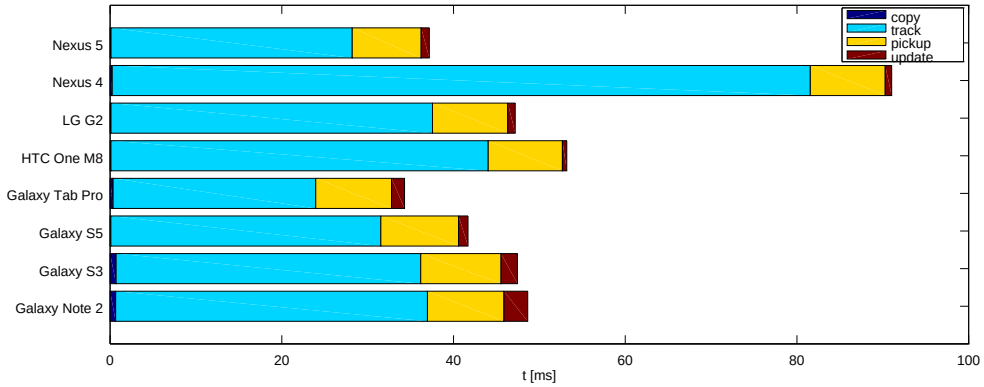


Figure 5.11: During the preliminary hardware test, we also checked the panorama rendering pipeline. This plot shows the average latency from the moment when the callback gets the camera frame until the renderer updated the panorama. Most of this time is spent tracking (track). The pickup time is the time needed until the renderer picks up the tracked frame. This time is similar on all devices. The remaining parts of the pipeline are negligible.

the GPU panorama (update). The first and last of these four operations contribute only a minor part of the time needed.

The biggest part is the tracking which depends primarily on the speed of the CPU, but interestingly also on the camera frame rate. The higher the camera frame rate is, the smaller is the difference between two consecutive frames. This improves the tracking speed as it is easier for the tracker to find matching features if the difference between frames is small. This can easily be seen when looking at the tracking time needed for the Nexus 4, which has the lowest camera frame rate.

Last but not least the pickup time is a time where no actual processing is done, but the already tracked frame waits to be picked up by the rendering thread. As shown in figure 5.10 this time can be variable between 0 and Δt_r. Considering a uniform random distribution in this interval, the means shown in figure 5.11 will give a pickup time of $\frac{1}{2}\Delta t_r$. The pickup time is on average between 8 and 9 ms for all phones, which means they have a rendering frame rate of around 60 FPS.

A more detailed result is shown in figure 5.12, where three phases are shown for each phone. The meassurement starts with the app launching. The first 10 frames are then treated as initialization (init) as the load on the CPU is high in the beginning initializing the camera, user interface and tracker. Then the phone tracks and is rotated slowly to build the panorama with good tracking results (good). Lastly the device is shaken wildly resulting in bad tracking results and then the application is quit. The first frame that has a bad tracking result is the beginning of the last phase (bad), which contains some frames marked as tracked good by the tracker. The colors in the bar plots show the same processing phases as in figure 5.11. As the panorama is not updated when the tracking result is bad, there is no time spent on updating in the bad phase. The only other difference between good and bad phase are the tracking time, which shows

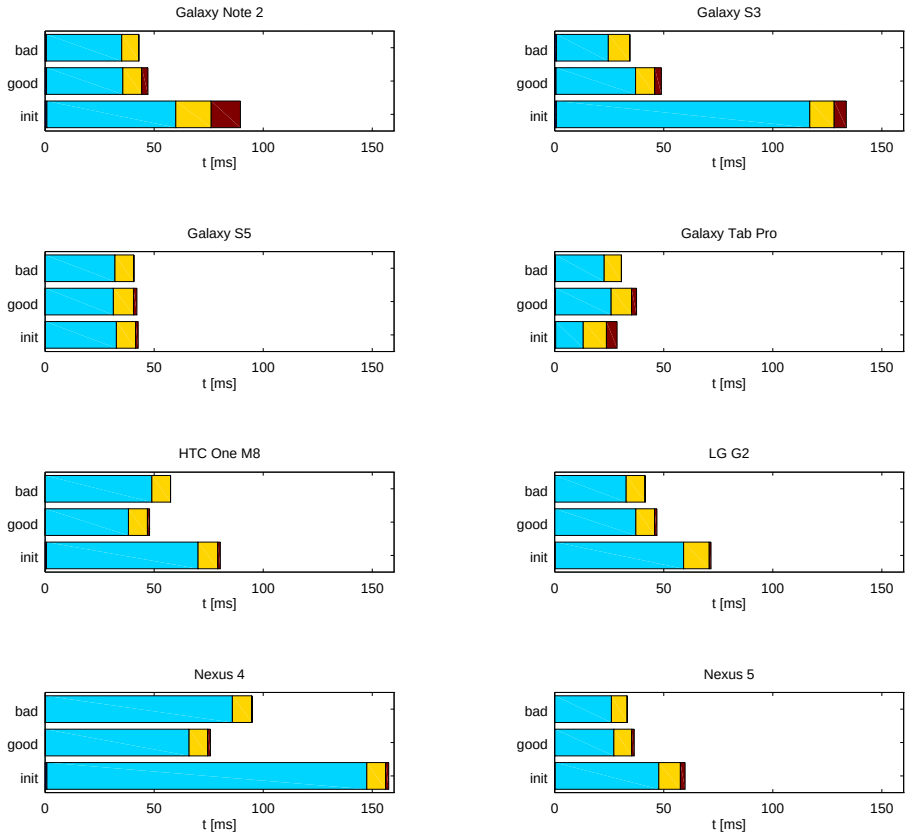


Figure 5.12: This diagram shows the tracking latency from figure 5.11 in more detail. It shows the same processing steps but during different phases of the application running. For each device we measured the average speed of the first 10 frames (init), a steady movement with panorama generation (good) and a jerky movement with bad tracking results (bad).

no direct correlation to the tracking result. The biggest difference is shown with the Nexus 4 where a lost tracking results in more time spent on trying to track again.

5.3.4 Rendering Performance

After tracking, rendering is the last step in the processing pipeline. Rendering here includes all operations that are done in the callback of the rendering thread. These operations include the update of the panorama that was already part of the evaluation in the previous section, rendering a view of the panorama to the screen and all user interface elements in the main view that will be described in the following chapter.

The results in figure 5.13 show two different times. The render time is the time spent in the callback function, while the frame time is the time that passes between two

consecutive calls of the callback and thus is always bigger than the render time. We can see that the phones are usually faster when not yet connected to a communication partner. The HTC One M8 shows a different behavior which can be explained that the graph mixes frames where panorama updates are happening and frames where no updates are happening. Updates require more processing power and were more frequent in the test of the single phone resulting in bigger times while for the other two phones the frequency of updates were approximately the same whether unconnected or connected.

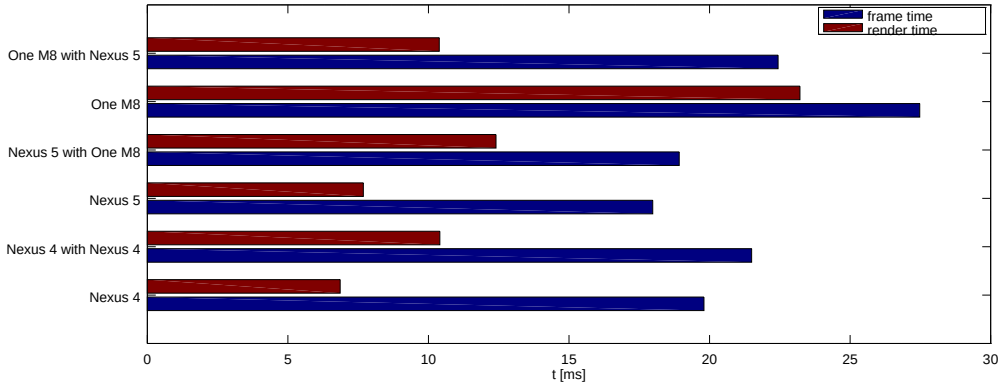


Figure 5.13: This plot shows the time needed in the rendering callback as render time and the time between two consecutive calls as frame time. The phones have been tested without a video call connection and with a video call connection to one of the other phones.

Figure 5.14 therefore splits up the data into frames where panorama updates happened and frames where no updates happened. Here we can see that the update takes most of the rendering time. When phones are connected to another the rendering time is slightly higher because of more user interface elements to be displayed. The frame time doesn't change much at all whether connected or not for frames without updates. With updates the frame time increases when connected due to the higher GPU load as next to rendering the GPUs are usually also used to decode incoming video frames. As the updates cause the render time to rise significantly compared to the low processing load without, the frame time is also increased accordingly and is not independent of the render time anymore lowering the rendering frame rate. Again the unconnected One M8 shows some interesting behavior, as the render time is quite big for the no update case. Further investigation of the data shows that the high render times don't always correspond with whether the frames update the panorama or not. This happened with all measurements but the effects are most visible for this case. The implications are that the render time for updates is slightly underestimated and slightly overestimated for the no update case. The most likely cause for this is that the video frames are not always encoded and decoded at the same time the panorama is updated.

The most important result is, that the frame rate in any case is bigger than 30 frames per second on average even for slower phones where updates cause the frame time to be bigger. Frames without any panorama update compensate for the updating

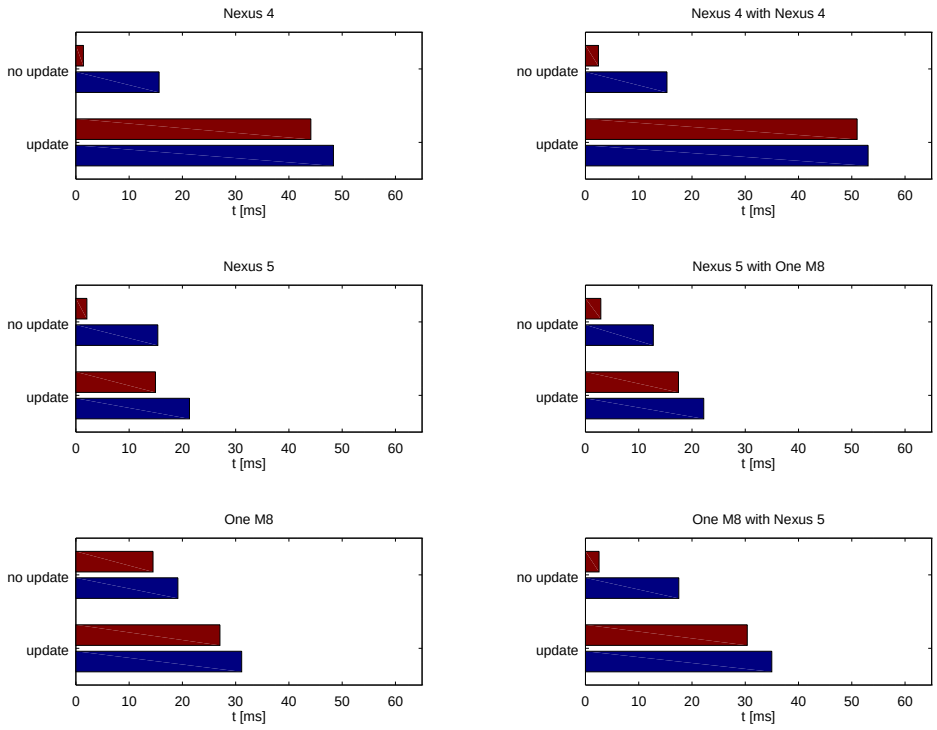


Figure 5.14: Like figure 5.13, this plot shows the time spent in the rendering callback and the time between two consecutive calls. The measurements are split up into frames where the panorama was updated and frames where it was not updated. We can clearly see that the panorama update strains the GPU more, especially for the Nexus 4. This could possibly be caused by framebuffer rendering not being as fast on older mobile GPUs.

frames and keep the frame rate high which in turn should keep the application feeling interactive.

5.3.5 Latency

We measured the overall latency of the system with the phone with the slowest camera, the Nexus 4, to get a worst case result for real applications. To do so, we placed a phone displaying a running clock. A phone running the telepresence application recorded this timer and a second phone was connected to it with both phones displaying the timer. A photo of all three phones then shows the timer and the difference between the displayed timers gives the overall latency.

Some photos of this measurement can be seen in figure 5.15. Due to the fast changing timer and the long exposure time of the phones' cameras, the accuracy of the timer is only in the 0.1 second range. The resulting latency of the local phone is between 0.2 and 0.3 seconds. For the remote phone the latency increases to between 0.7 and 0.8

seconds, so about 0.5 seconds higher. This is the latency that is added by WebRTC and has been measured to be between 300 and 500 milliseconds[3].

Figure 5.15: These photos are used for end-to-end latency measurment. The upper left phone shows a running timer, while the lower phones are running the telepresence application. The left of these two phones is the local phone and the right one the remote phone.

From figures 5.11 and 5.13 we know that the Nexus 4 has a latency of about 0.1 seconds for the pipeline processing the camera frames until rendering is done. The remaining 0.1 to 0.2 seconds are spent aquiring the camera image and passing it to our callback function and when the rendering code is done until the result is shown on the phone's display.

[3] https://www.facebook.com/WebRTC/posts/726032664137227

5.3.6 Panorama Quality

Figure 5.16 shows a locally and a remotely generated panorama. Apart from the panorama generation issues already discussed in chapter 4, we can see that the quality of the local and remote panorama is pretty much the same. The quality of the remote panorama can get worse when the network connection is bad and WebRTC switches to a lower quality video encoding. This can also potentially affect the tracking as the compression artifacts introduced can destroy or alter features used for tracking. The same problems arise when frames are dropped due to limited processing power on the receiving side.

(a) local

(b) remote

Figure 5.16: The two panoramas were generated on two communicating phones. The lower remote panorama has been initialized just a little further to the right than the local one. Apart from that the visual appearance of the panoramas is almost indistinguishable. A closer look reveals that there are slight differences as some of the frames seem to have been dropped on the receiver side. This however didn't affect the tracking and panorama generation much.

Another issue we see in the two panoramas is related to the synchronization. When the connection between the two phones is established, all trackers are reset and the panorama generation starts from scratch. When the phone is moved too much during the not perfectly synchronized reset, a bias is introduced between the two panoramas. In the given example we can see that the rendered part of the remote panorama is a bit further to the right, than the local one. So far this bias could not be noticed explicitly during use of the system, but further evaluations during a study are required to assess whether this problem needs to be addressed further.

6 User Interface

This chapter describes the developed user interface features of our application. Figure 6.1 shows a schematic of the interface elements on the screen. The following sections are divided by the goal that each interface element is trying to achieve: spatial awareness, gaze awareness and pointing. The chapter ends with remaining interface elements such as buttons and options that are not present in the main view.

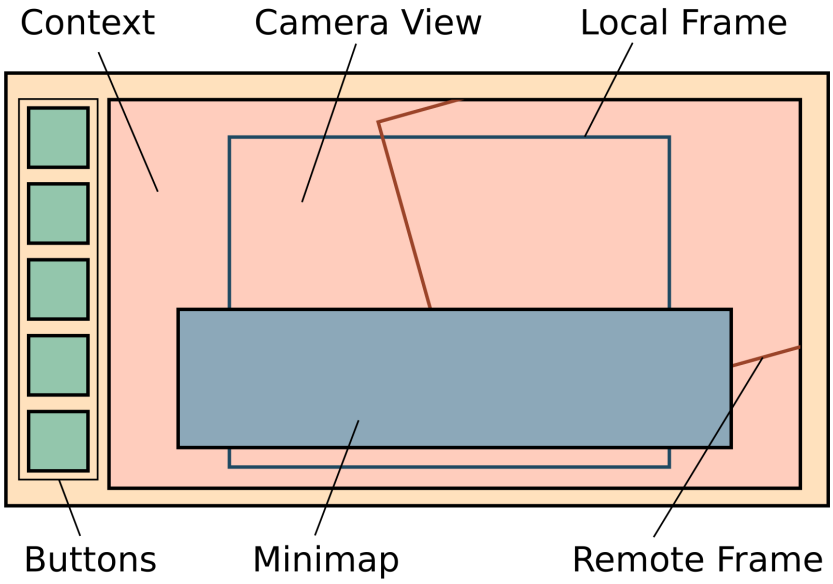


Figure 6.1: This schematic of the implemented user interface shows the major UI elements which will be discussed in this chapter. Except for the buttons on the side all elements are rendered in a maximized OpenGL view.

6.1 Spatial Awareness

We use several methods to increase spatial awareness and help the user to navigate the local or remote panorama. Showing a context around the current camera image, which means showing more of the panorama surrounding the current camera image, allows the user to get a wider field of view and perceive surrounding areas such as parts of the panorama that haven't been recorded yet or parts the other user is looking at. Another method to aid in spatial awareness is a minimap that shows the full panorama

and the current views of the users, which shall allow for a quick assesment of the orientation of both, own and remote orientation.

6.1.1 Incomplete Panorama

To aid in the distinction between parts of the panorama that still need recording the panorama is initalized with a green grid on a black background as shown in figure 6.2. The poles which are not covered by the panorama are simply rendered in black. The green grid on one hand shall help the user to orient himself in the remote panorama and on the other hand motivate the local user to record the panorama in these areas.

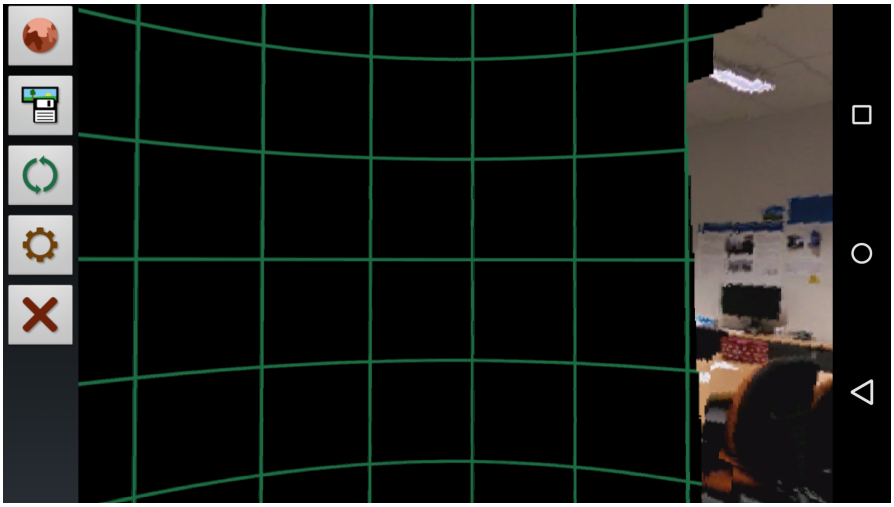
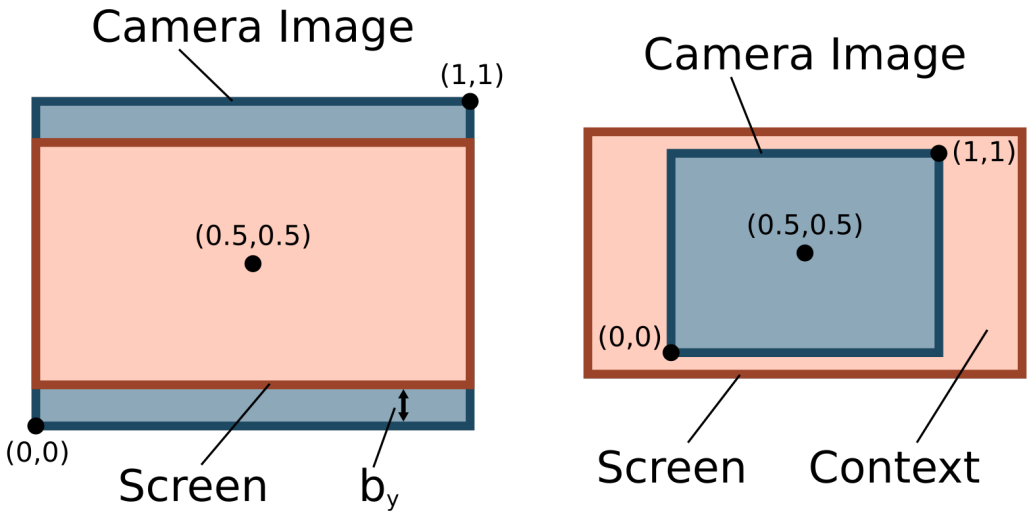


Figure 6.2: A green grid is displayed on parts of the panorama that haven't been recorded yet.

6.1.2 Context

The process of rendering a current camera view of the panorama has been described in section 4.4.4. Additionally to the transformations made it is important to consider the different aspect ratios of the screen and camera image. While the screen might have a 16 : 9 aspect ratio, the camera image is 4 : 3. To counteract stretching of the rendered image, a border is cut from the sides of the camera image on either the top and bottom or the left and right side, depending on whether the aspect ratio is smaller or bigger. Figure 6.3a shows the display inserted in the camera image and the necessary borders to be added. To calculate the necessary border the equations



(a) without context

(b) with context

Figure 6.3: Rendering the camera image, while keeping the screen's aspect ratio, we need to cut a border of the camera image to still be able to fill the screen. Additionally when we render with context, we scale the camera image to be smaller than the screen with a context factor $f_c > 1$.

$$b_x = \begin{cases} \frac{1-\frac{a_d}{a_c}}{2}, & \text{if } a_c > a_d \\ 0 & \text{otherwise} \end{cases} \tag{6.1}$$

$$b_y = \begin{cases} \frac{1-\frac{a_c}{a_d}}{2}, & \text{if } a_c < a_d \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

are used, where b_x and b_y are the borders for the side and top and a_c and a_d are the camera and display aspect ratios.

To show a bigger context around the camera we introduce a context factor f_c which scales the borders of the camera space around it's center $[0.5\ 0.5]^T$ as shown in figure 6.3b. While the border can simply be scaled by this value the minimum and maximum value in the camera space change from 0 and 1 to

$$\min c = 0.5 - 0.5 f_c \tag{6.3}$$

$$\max c = 0.5 + 0.5 f_c. \tag{6.4}$$

we therefore see, that a context factor of $f_c = 1$ keeps the original configuration, while bigger values appear to zoom out and smaller values appear to zoom in to the camera view. It is important to note that the projection equations are able to handle values outside the $[0, 1]$ input range.

We empirically set the context factor to $f_c = 1.3$ shown in figure 6.4, as this value gives an appropriately sized border which is neither too big nor too small around the camera view. Next to being able to see more of the panorama around the current camera view, the bigger context also has the advantage to improve the quality of the image as a higher resolution image is rendered on the screen, which has a bigger resolution than the camera image.

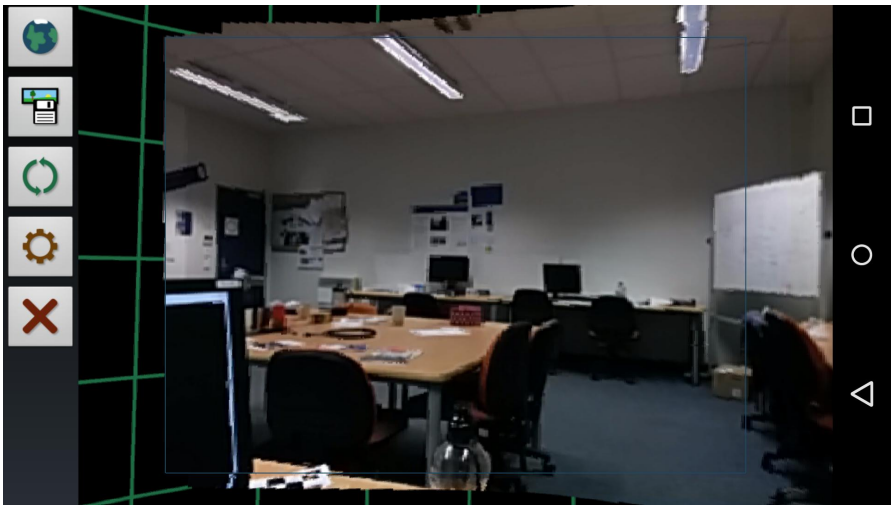


Figure 6.4: The camera view in this screenshot is surrounded by a blue frame and everything around it is context rendered from the panorama. The right side has been viewed already and contains an image, while the left side hasn't been viewed yet and therefore still shows the green grid.

6.1.3 Minimap

To position and render the minimap on the screen we define the variables a, b, c and d as shown in figure 6.5. We again have to calculate these distances for two different coordinate systems, while keeping the aspect ratio. The original coordinate system is denoted with an prime ($'$) and is the display coordinate system meassured in pixels. The width and height are for example $w' = 1280$ and $h' = 768$ for the Nexus 4 display. The target coordinate system is OpenGL's $[-1, 1]$ normalized coordinate system in which the vertices used to render the minimap's rectangle have to be placed. The width and height in this coordinate system therefore are simply $w = h = 2$. The aspect ratio r is defined as

$$r' = \frac{w'}{h'},\ r = \frac{w}{h}, \tag{6.5}$$

and axis parallel lengths x and y can simply be transformed linearly with either the ratio between widths or heights from one coordinate system to another with the equations

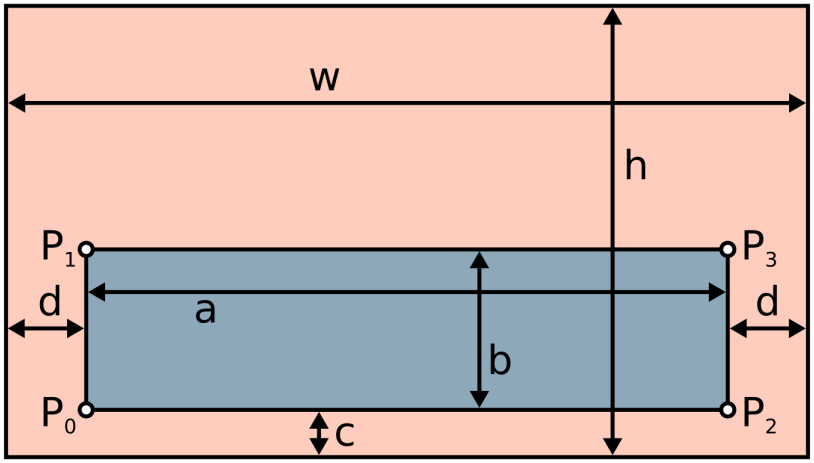


Figure 6.5: This schematic shows the positioning of the minimap (blue) on the screen (red). The variables a to d represent the width and height as well as the distance from the screen edges of the minimap. These values are then used to calculate the four vertices P_0 to P_3 to render the minimap.

$$y = \frac{h}{h'}y', \; x = \frac{w}{w'}x' \tag{6.6}$$

$$y' = \frac{h'}{h}y, \; x' = \frac{w'}{w}x. \tag{6.7}$$

The panorama's aspect ratio is $R' = 4$ in the display pixel coordinate system and can now be related to the target coordinate system using the equations above as

$$R' = 4 = \frac{a'}{b'} = \frac{\frac{w'}{w}a}{\frac{h'}{h}b} = \frac{w'}{h'}\frac{h}{w}\frac{a}{b} = \frac{r'}{r}\frac{a}{b}, \tag{6.8}$$

which gives us a relation between a and b as

$$b = \frac{r'}{r}\frac{a}{R'} \tag{6.9}$$

in the target coordinate system. Now defining d' and c' and then transforming them using equation 6.6, we can then calculate a as

$$a = w - 2d \tag{6.10}$$

and b to finally calculate the positions of the four vertices P_0 to P_3 as shown in figure 6.5 to render the panorama as minimap. The final implementation of the minimap in the

application can be seen in figure 6.6. It also shows two frames that correspond to the current views of the users which aids in gaze awareness as will be discussed in the following section.

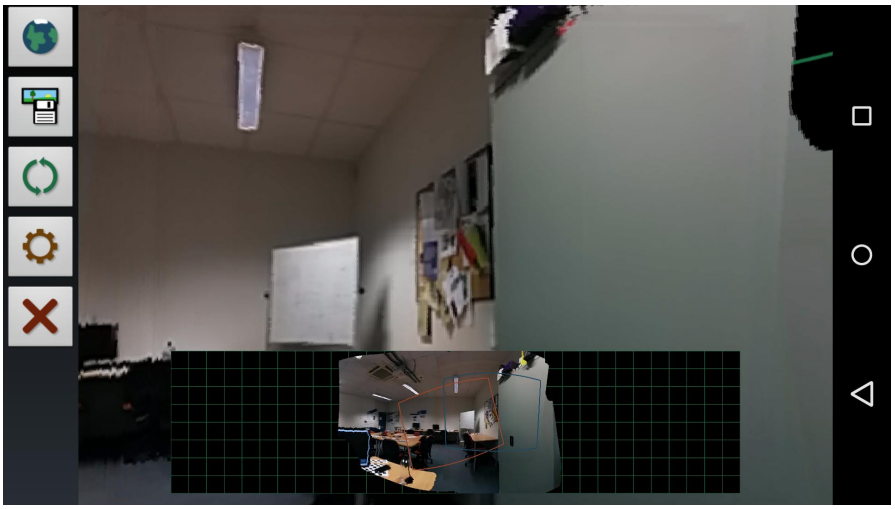


Figure 6.6: The minimap in this screenshot shows the panorama and additionally two frames, which enable the users to orient themselves according to each other.

6.2 Gaze Awareness

To enable the users to see where the other user is looking, we decided to simply render a frame around the current view of each user's camera image. The frame is simply rendered as the outline of a rectangular polygon as can be seen in figure 6.7. For the remote user, the shader transforms the vertices from the camera space of the remote user to map space and then to the camera space of the local user using the orientations of both users.

The local frame is static in relation to the display and would be useless if it was just rendered on the edge of the display, however in cooperation with the context factor, which is used to show more than the camera frame, the frame now shows the border of the camera frame and intiutively helps the user to differentiate between the camera frame and the surrounding context. The context also helps to see where the remote user is looking when the users are looking exactly in the same direction, as the remote user's frame would hardly be visible on the edge of the display otherwise.

The frames are also rendered to the minimap using only the projection transformation from camera to map space and then applying the same transformations to the vertices as in section 6.1.3. This enables the users not only to orient themselves inside the panorama, but moreover to still maintain awareness of the other user's gaze even when it is outside the contextual view.

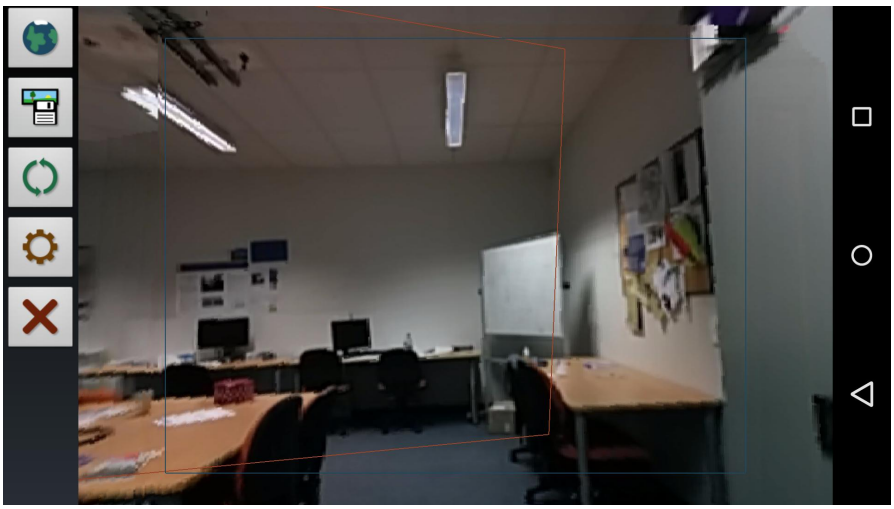


Figure 6.7: In our application, the blue frame shows the local user's view, while the red frame shows the remote user's view.

Simply always rendering the remote frame results in display errors, when it is off screen. To not render it in this case, we need to determine when it is off screen, based on the angle between the two user's orientations. This angle is compared to the field of view angle. As a simplification we assume that a two-dimensional approximation of the problem using the horizontal field of view α_x is a sufficient to solve the problem.

The horizontal field of view α_x is determined based on the horizontal resolution $x_\mathbf{r}$ and focal lenght $x_\mathbf{f}$ via the trigonometric equation

$$\tan\left(\frac{\alpha_x}{2}\right) = \frac{x_\mathbf{r}/2}{x_\mathbf{f}}, \tag{6.11}$$

which can be reordered to

$$\alpha_x = 2\tan^{-1}\left(\frac{x_\mathbf{r}}{2x_\mathbf{f}}\right). \tag{6.12}$$

The cameras are looking in the negative z direction, so from the two orientation matrices $\mathbf{O}_l$ and $\mathbf{O}_r$ we get the look at vector $\mathbf{a}$ as

$$\mathbf{a} = \mathbf{O}\mathbf{e}_z = \mathbf{O}\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}. \tag{6.13}$$

The angle between the two vectors can then be determined through the vector equation

$$\cos\left(\alpha_{\mathbf{v}_1\mathbf{v}_2}\right) = \frac{\mathbf{v}_1 \cdot \mathbf{v}_2}{||\mathbf{v}_1||\ ||\mathbf{v}_2||}. \tag{6.14}$$

As the look at vectors are already normalized (a rotation matrix doesn't change the length of a vector and the unity vector $\mathbf{e}_z$ has a length of 1), the angle between the two look at vectors can be determined as

$$\alpha_{lr} = \alpha_{\mathbf{a}_l\mathbf{a}_r} = \cos^{-1}\left(\mathbf{a}_l \cdot \mathbf{a}_r\right) \tag{6.15}$$

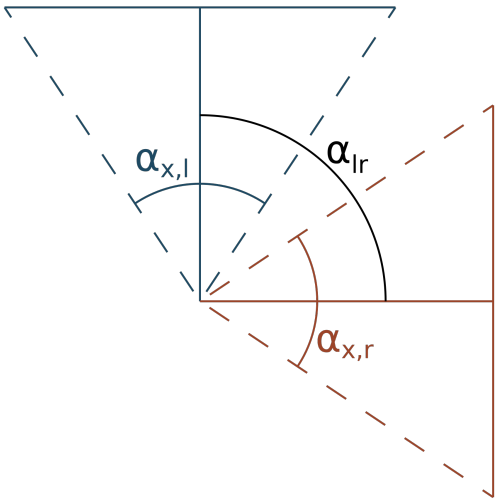


Figure 6.8: This schematic shows the horizontal field of view as seen from the top. The horizontal field of view of the local ($\alpha_{x,l}$) and remote ($\alpha_{x,r}$) user and the angle α_{lr} between their viewing directions allows to determine whether the views are overlapping.

Now as can be seen in figure 6.8, the frame of the remote user is off screen when

$$\alpha_{lr} > \frac{\alpha_{x,l} + \alpha_{x,r}}{2}. \tag{6.16}$$

This has been implemented and successfully tested to resolve the problem.

6.3 Pointing

People working together in an environment usually talk about objects in a scene and gaze awareness is an important factor for one party to figure out what the other is talking about. Sometimes however this is not enough and people use pointing to aid their verbal explanations. We added a simple method to enable both parties to point in the environment.

Without any additions the environment's local user can point with just his finger as long as the camera is able to capture it as shown in figure 6.9. Of course this approach has some problems as the user has to keep in mind the different perspectives. In the real world a person would move his head close to the other person's head, so that when pointing at objects further away their perspectives are similar enough. With this approach the user similarly has to make sure that the finger is pointing at the correct object, which he can make sure by observing the pointing on the phone's screen and correcting accordingly. Additionally the finger might influence the tracking and consistency of the panorama as the hand is cut off at the border of the camera image.

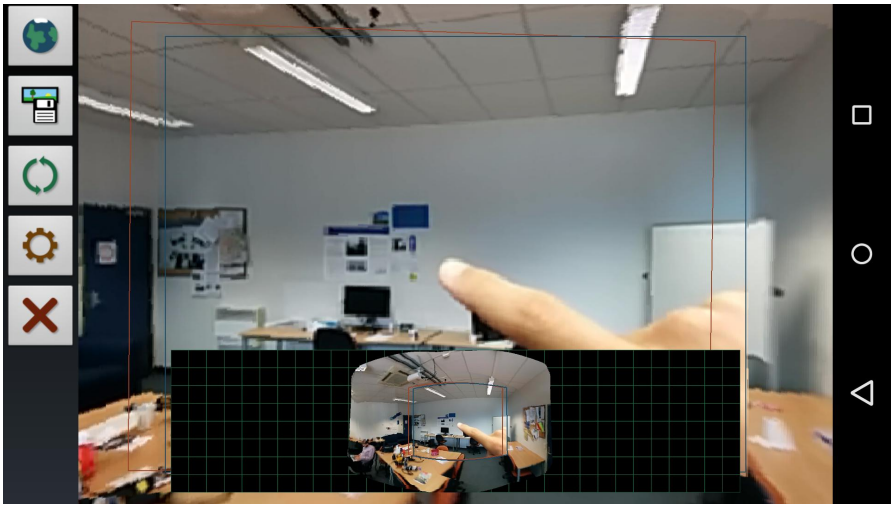


Figure 6.9: The local user can simply point at the poster in this scene by using his finger as usual. However, he has to make sure that the pointing looks correct on the phone's screen instead of from his normal perspective, which makes this form of pointing a bit cumbersome.

To enable both users to point at objects, we implemented a simple touch based interface. When the user touches the screen the touch position gets transformed from display space to map space by the projection transformations. This map position is then transfered to the other user, transformed to the other user's screen space and displayed as a filled circle fading out towards the circle's edge as can be seen in figure 6.10. Of course this approach has some problems too such as the finger and it's circular representation hiding the actual object that is pointed to. On the other hand, this problematic is known with touch interfaces for a while already as "fat finger problem" (Siek, Rogers, and Connelly, 2005) and users are adapted usually. When the remote user stops touching the screen, the circle as a whole starts to fade out, which allows the local user to gradually start seeing the object behind the finger while still knowing where the other user pointed to, which we claim improves the interaction as opposed to a suddenly disappearing circle.

Next to a single circle, it is also possible to enable the display of a finger trail. When the finger is moved across the scene, the circle just doesn't move, but more circles are added leaving a trail as shown in figure 6.11a. The trail slowly fades out as shown in figure 6.11b, just as the single circle does when the finger stops touching. This interface

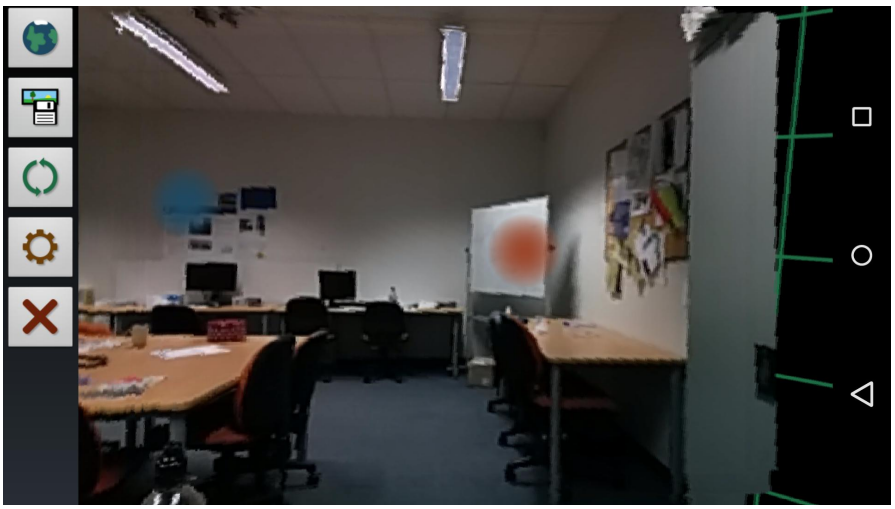
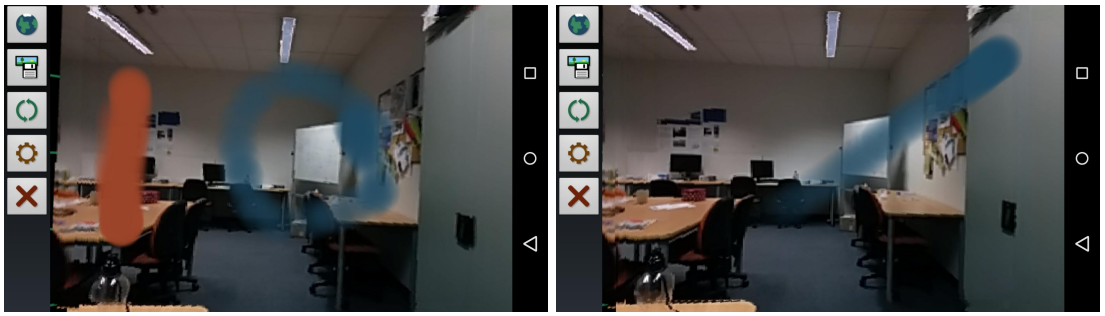


Figure 6.10: The remote and the local user are pointing at different objects in this scene using the touch pointing interface on the phones' screens. The blue local user is pointing at the poster, while the red remote user is pointing at the whiteboard.

method enables a simple form of drawing such as it is also implemented in Chili (Jo and Hwang, 2013) and is a first method of advanced interaction techniques.



(a) both users drawing (b) drawing fading out

Figure 6.11: Remote and local user can draw in the scene as can be seen in the left screenshot. The finger in the right screenshot has been moved from the lower left corner to the upper right corner leaving a trail that is slowly fading out.

6.4 Interface Elements

While the main interaction techniques that aid the users have been discussed, we now want to discuess standard interface elements. These are buttons on the side of the screen for immediate access by the end users of the application and an options menu mainly targeting the experimenter in a study for the behavioral changes of the application.

6.4.1 Side Buttons

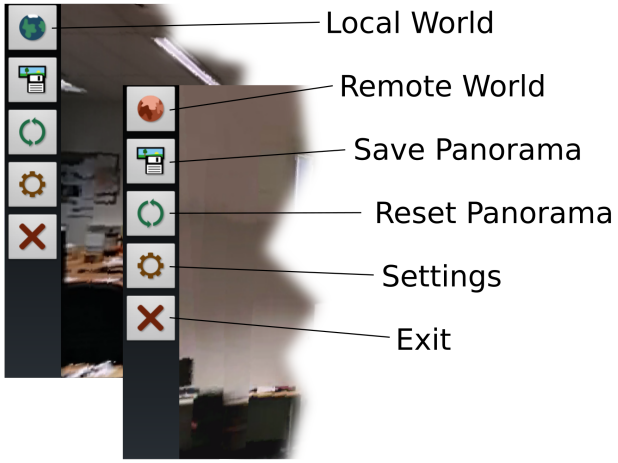


Figure 6.12: Interface buttons on the left side of the screen allow quick access to the application's additional functions.

Figure 6.12 shows the side buttons used in the application. From top to bottom these have the following functionalities:

1. World switch: switches between local and remote environment.
2. Save: saves the current panorama as image on the phone.
3. Reset: resets the panorama starting with a new orientation for both users.
4. Settings: opens the settings menu.
5. Quit: exits the application.

To help the user differentiate between remote and local view, we introduced different colors. The color blue is used for the local and the color red is used for the remote environment. This color is then used for the frames, the finger pointing and the button used to switch the environment. The latter shows a blue planet while in the local and a red planet wihle in the remote environment.

6.4.2 Options

The options menu as shown in figure 6.13 uses the standard Android API to edit and store preferences. The full list of options is:

- Server address: the address of the signaling server.
- Synchronize world: enables the synchronized switching of both users' environment.
- Reset on world switch: resets the panorama when the environment is switched.

- Current world visual tracking: use visual tracking for the current environment, sensors otherwise.
- Other world visual tracking: use visual tracking for the other environment, sensors otherwise.
- Update both panoramas: enables update of both panoramas, when visual tracking is used.
- Backup with sensors: use sensors for viewpoint control when visual tracking fails.
- Audio: use the normal earphone, speakers or no audio at all.
- Local frame: enables the display of the local user's frame.
- Remote frame: enables the display of the remote user's frame.
- Local color: the color of the local user, which defaults to blue.
- Remote color: the color of the remote users, which defaults to red.
- Frame width: the width of the frames in the main view in pixels.
- Context factor: the context factor f_c as described in section 6.1.2.
- Minimap: enables the display of the minimap of the current environment.
- Other minimap: for debugging purposes this option enables a second minimap at the top of the screen which displays the other environment's minimap.
- Minimap frame width: the width of the frames on the minimap in pixels.
- Grid color: the color of the grid shown in the empty panorama parts, which defaults to green.
- Grid width: the width of the grid, also in pixels, but in map pixel space in contrast to display pixel space as for the frame width options.
- Finger trail: enables the finger trail, which is described in section 6.3.
- Finger size: the size of the finger displayed on the screen.

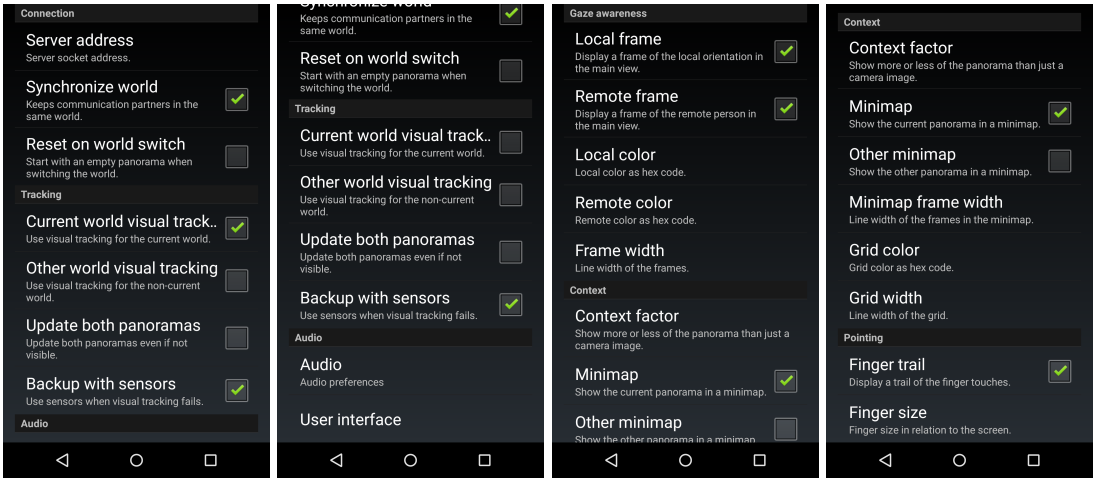


Figure 6.13: The left two screenshots show the main settings menu and the right two the User Interface submenu of the application's preferences.

7 Conclusion

We implemented a novel mobile telepresence system, delivering the first truly handheld telepresence. We are the first relying only on mobile hardware with a symmetrical system architecture that only needs mobile phones or tablets for the communication partners. Telepresence is supported in a mobile to mobile scenario in indoor and outdoor environments without the need for any prior preparation, 3D scanners, robots or any other special hardware. Placing the display of the handheld device close in front of the eyes, it looks like being at the remote location. This free choice of view aims to improve the spatial presence.

Figure 7.1 shows the system being used outdoors with the user being in the local world calling with the person shown in figure 7.2 who is indoors.

Figure 7.1: The main building of the University of Otago serves as object of interest in this photo of the final system in action. The user looks at his local environment on the phone and is communicating with another user using our application shown in figure 7.2.

Looking back at our goals from section 1.3, we achieved all major points and the system is now ready for the proposed user study on its influence on spatial presence which

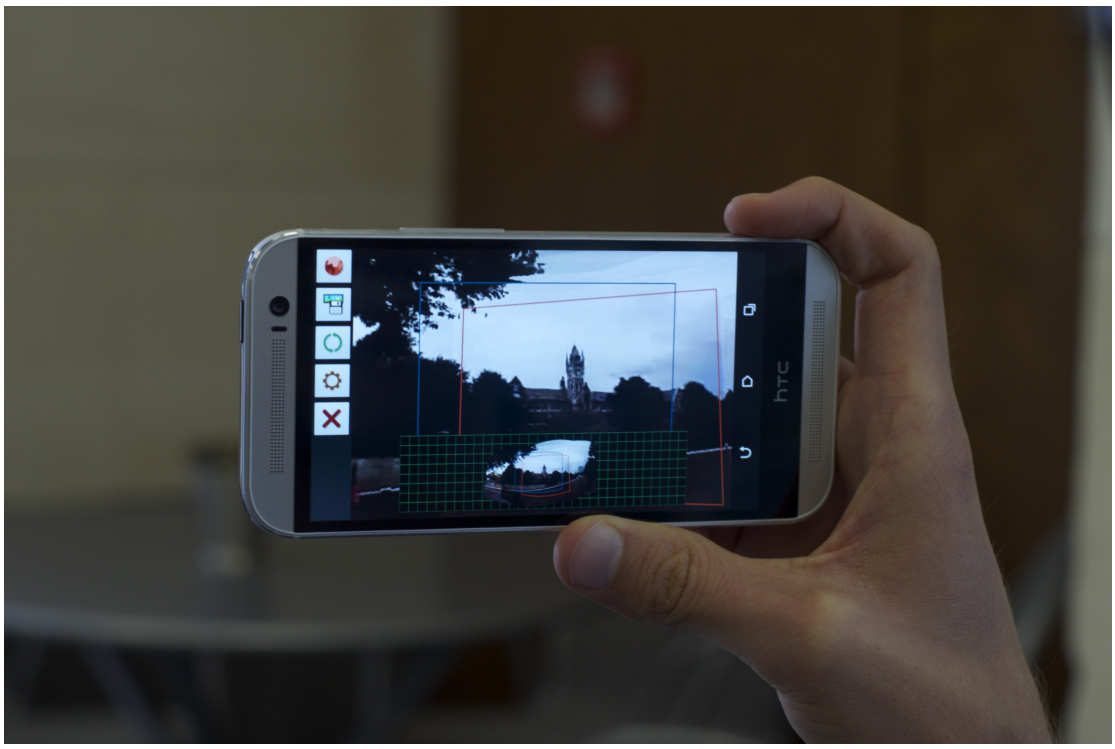


Figure 7.2: This user is seated inside a cafe, but enjoys the beautiful architecture of the University of Otago by communicating with the user in figure 7.1.

we derived from pervious work in this area. The application provides an independent view of any environment for all users. It works on any recent Android device and in a symmetric fashion, requiring no server or desktop computer. A server is solely used for connection establishment.

As evaluated in chapter 5, the system runs on many Android devices including tablets and smartphones from 2012. The biggest drawback of the hardware is usually the camera and its access, which is not directly age dependent, but newer phones tend to achieve better results. Even though the camera frame rate is comparably low and the latency quite high, the system still feels very interactive, as the rendering frame rate is still high and the sensor based interaction has a very quick response. The slower updates of the panorama potentially do not harm the interactivity. When the user turns, the view follows without any noticeable latency as will be evaluated in a user study.

The screenshot of the implemented application in figure 7.3 shows most of the interface's capabilities. It supports gaze awareness for the local and remote user by using frames, a minimap and showing a bigger field of view than the camera would. Next to audio the interaction between the users is also supported by pointing and possibly even drawing on the screen. The drawing stays in place while moving and has different colors to distinguish who drew. Buttons on the side allow easy access to frequently used features and an extensive settings menu has been implemented to support a detailed evaluation of the application in a future user study.

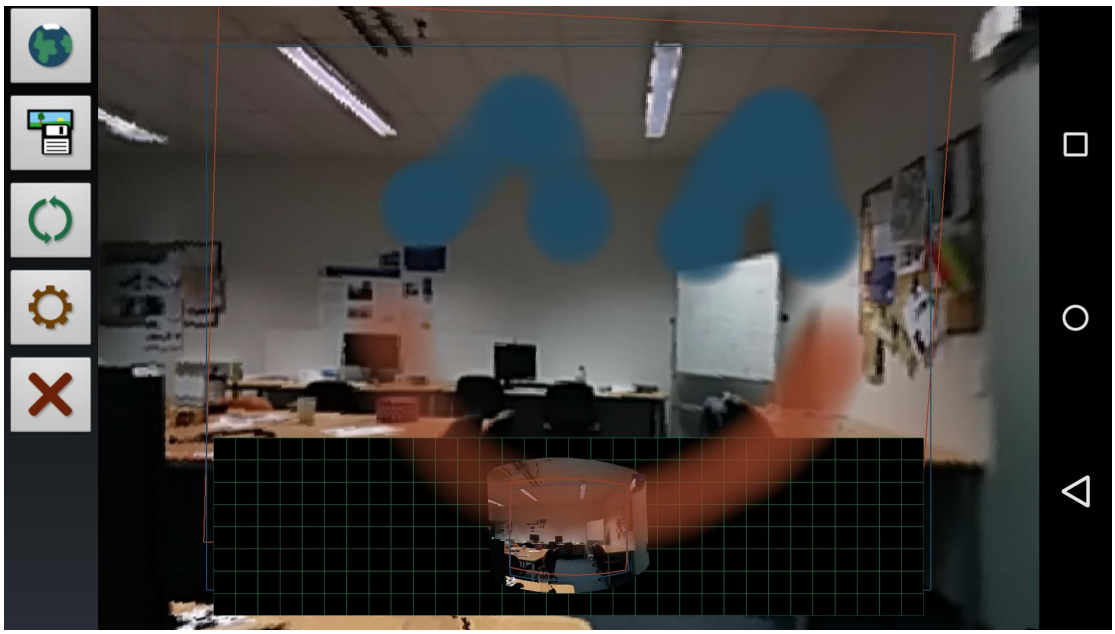


Figure 7.3: This screenshot of the finished implementation of the mobile telepresence application shows most user interface features and the drawing function is used to express the developer's happiness.

In the following sections we will discuss the limitations of the application in its current state and possibilities on how to overcome them. Finally, a broader outlook discusses many different paths to follow in future work.

7.1 Limitations and Possible Solutions

Some of the limitations of the application were already discussed in section 4.5, but we will repeat some of them here and talk about possible solutions.

Movement is potentially the biggest issue. The movement of the user is problematic, as the trackers work with orientation only to generate the panorama. On one hand this problem can be addressed with a different map and tracking approach by using a hybrid between panorama and keyframe based SLAM as proposed by Pirchheim, Schmalstieg, and Reitmayr (2013). On the other hand, it is likely that the users of the application usually stand still when using it, similar to the movement pattern when using an augmented reality browser as found by Grubert, Langlotz, and Grasset (2011). In this case users switch to a typical phone posture, holding the phone to the ear while moving. However, this mostly depends on the application and task at hand. Sensors, such as the accelerometer, the GPS and barometer for height can be used to easily detect movement and switch between different modes of the application. When the visual tracking fails for a longer period of time, an automatic reset – so that a new panorama will be generated – could be another solution for this problem and other

tracking based issues. Inconsistencies in the panorama created by moving objects for example cannot simply be prevented. More complex computer vision algorithms could be used to detect moving objects in the scene and handle them, but this would require a lot more processing power.

Usually mobile video calling applications let the user choose between the front and back camera of their devices. Most of these applications use the front camera by default, showing the communication partners to each other, like the webcam of desktop video calling applications usually do. At the moment the application does not provide this functionality. It could be simply added by adding a possibilty to switch between the back camera based panorama mode and the front camera based face mode. A more advanced implementation would incorporate the front camera picture in the panorama. For the local panorama the remote user's face could even be segmented and displayed in the panorama as well. In collaborative tasks, Jones et al. (2015) showed that the back camera is prefered in most cases and the front camera showing the face of the collaborator is not very helpful to work on tasks.

Telepresence robots can deliver live size representations of the remote user and show a life size image of their face on a screen. This can usually not be done with mobile telepresence applications on a smartphone, because the screen is too small. Holding the phone and looking at its small display, the screen might feel like looking through a tiny window into another world, instead of feeling more immersed in this world. This results in a lower presence than would be possible. This problem could be addressed by using the phone as a display in a head mount like Google's cardboard[1]. This would also help with the limitation that while using a handheld device, tasks that need two hands cannot be executed.

A minor issue of the application is, that the phone has to be held horizontally when the panorama generation starts so that the panoramic cylinder is rotated in a way that the panorama goes around the horizon and not over the head of the user. This could be handled by using the sensor based tracking, which gives an absolute orientation relative to the magnetic north pole and gravitational force. After the horizontal initialization the phone can be held in portrait mode though and the application is fully usable, except that the user interface (buttons and minimap) don't change the orientation yet. Jones et al. (2015) suggests that portrait mode is the prefered way of holding the phone due to social and ergonomic reasons.

The resolution of the panorama is quite low at the moment and can be increased if the accuracy of the vision based tracker is increased accordingly. Otherwise the quality of the generated panorama will suffer from imprecise stitching. Jones et al. (2015) also noted this limitation, stating that the mobile user has to get closer to objects to be detailed enough with the low resolution of the phones. This would not be a problem if the other collaborator could be physically there. Getting closer to objects for detailed views is not possible with the panoramic map approach as movment causes issues. Other techniques like super resolution image reconstruction (S. C. Park, M. K. Park, and Kang, 2003) could be used to increase the resolution.

[1] https://www.google.com/get/cardboard/

Lastly, a stable and fast enough network connection is crucial for the application. When the network quality decreases, the connectivity gets lost or WebRTC lowers the quality of the video stream. This in turn lowers the quality of the panorama and interaction in general. A lower quality video stream not only worsens the visual quality of the panorama, but also can potentially result in impossible visual tracking as the visual features used are harmed considerably by high video compression or low quality videos.

7.2 Outlook

Possiblities for future work are abundant. Many aspects of the system can and will be evaluated in one or more user studies, especially the system's ability to improve the spatial presence. The evaluation of the feeling of interactivity and the usefulness of the implemented user interface and their influence on spatial presence have already been proposed. Studies in different application areas such as tourism, entertainment, sales, advertising, gaming, education or more generally defined collaborative tasks can evaluate the performance of the system. An observational study is a opportunity for new findings and further areas to explore as the field is relatively young and unexplored.

Using a more advanced mapping mechanism has been discussed already. Hybridization can be used together with specialized hardware like Google Tango. Google Tango works only indoors, so it cannot just replace the panorama. As the panorama has difficulties with close objects as they are indoors, a hybridization with Google Tango would be optimal, if this technology gets widespread and readily available in mobile devices. A higher resolution panorama would also increase the quality of the panorama and if the video stream is not simply used to replace the corresponding area in the panorama, but gets combined with the already available data, an even higher resolution will be possible too allow even zooming into the panorama.

An implementation that uses a server for data processing has been discussed in section 5.1.3. With this architecture the performance requirements of the phones could be even lower and they would consequently save battery as well. The higher processing power of a server can also be used for even more advanced features and higher quality mapping mechanisms. The disadvantage of course is a higher latency that gets introduced by adding the server into the system.

At the moment the application works for two users only. Group calling with three or more people is an interesting extension that would open up many new possibilities and questions. For example questions such as which environment users are in, whether users always hear each other or if users can be in different environments and then only hear people there. The implementation would require a proper system to setup group calls and a corresponding architecture to stream the data to all clients.

Next to the possibility of being in a user's current environment, other environments could be made accessible. Using already existing panoramas is just one simple example. Instead of a static preexisting panorama, virtual worlds can have dynamic content and

could be interactive. Trying to achieve a better spatial presence by colocation is another direction that can be investigated, simply by asking the question, how the panoramas or environments of the users can be joined.

There are many possible application scenarios and many possible extensions for the application on the scenario. The already available tracking makes the application a perfect base for augmented reality applications. Combined with the GPS sensors, location based information can be added to the display and panoramas can be stored and reused depending on their location. Gaming applications are another area that can be explored.

The interface of the application is useful, but still quite basic and more improvements are possible in this area. Currently the application renders the view from the panorama even if the user is in the own world. This ensures congruity with the view of the remote users. If the user puts his finger in front of the camera – as a consequence of the failing tracking – the view is not updated and the user has no way to see this problem. This could be solved by showing whether the tracking is working, but then it requires the user to know technical details about the application. The user does not care if the tracking is vision or sensor based, and contrary might even want to steer the view based on touch input.

The rudimentary interface implemented to become aware of where users are looking and pointing at, can be exchanged with other possible techniques. Additionally, an active notification would notify the local user, when remote users are looking at parts of the panorama that haven't been generated yet. For events, which are happening outside the current view, a notification can happen based not only on visual cues such as arrows on the side of the screen, but also on acoustic or haptic cues like audio notifications or vibration. Tonnis and Klinker (2006), for example, combine a 3D visual arrow with 3D audio cues in a car driving context to steer attention. Focus could also be steered by bluring the visuals out of focus.

While the local user is able to use normal pointing with the finger, the remote user has to use the touch interface. To enable remote users to use the same intuitive mechanism, a finger or hand segmentation algorithm could be used and the pointing of the remote user could then be integrated in the panorama.

More advanced touch input based interfaces, that are more complex than pointing, like adding annotations or an extension of the already implemented drawing could be added. The functionality to freeze the current view could greatly aid in this situation as users don't always have to target the affected area where they want to work on with their phones and hold their devices in a more comfortable position for the task.

Appendix

Bibliography

Adams, Andrew et al. (2010). "The Frankencamera: An Experimental Platform for Computational Photography." In: *ACM SIGGRAPH 2010 Papers*. Vol. 29. SIGGRAPH '10 4. Los Angeles, California: ACM, 29:1–29:12. ISBN: 978-1-4503-0210-4. DOI: 10.1145/1833349.1778766. URL: http://doi.acm.org/10.1145/1833349.1778766 (cit. on p. 47).

Agarwala, Aseem et al. (2005). "Panoramic video textures." In: *ACM Transactions on Graphics* 24.3, p. 821. ISSN: 0730-0301. DOI: 10.1145/1073204.1073268. URL: http://dl.acm.org/citation.cfm?id=1073204.1073268 (cit. on p. 12).

Biocca, Frank, Chad Harms, and Judee K. Burgoon (2003). "Toward a More Robust Theory and Measure of Social Presence: Review and Suggested Criteria." In: *Presence: Teleoperators and Virtual Environments* 12.5, pp. 456–480. ISSN: 1054-7460. DOI: 10.1162/105474603322761270. URL: http://www.mitpressjournals.org/doi/abs/10.1162/105474603322761270 (cit. on p. 2).

Dalvandi, Arefe, Bernhard E Riecke, and Tom Calvert (2011). "Panoramic video techniques for improving presence in virtual environments." In: *Proceedings of the 17th Eurographics conference on Virtual Environments & Third Joint Virtual Reality*. Eurographics Association, pp. 103–110. DOI: 10.2312/EGVE/JVRC11/103-110. URL: http://dl.acm.org/citation.cfm?id=2386101 (cit. on p. 11).

DiVerdi, Stephen, Jason Wither, and Tobias Hollerer (2008). "Envisor: Online Environment Map Construction for Mixed Reality." In: *2008 IEEE Virtual Reality Conference*. IEEE, pp. 19–26. ISBN: 978-1-4244-1971-5. DOI: 10.1109/vr.2008.4480745. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4480745 (cit. on pp. 12, 47).

Drugge, Mikael et al. (2004). "Experiences of using wearable computers for ambient telepresence and remote interaction." In: *Proceedings of the 2004 ACM SIGMM workshop on Effective telepresence - ETP '04*. New York, New York, USA: ACM Press, p. 2. ISBN: 1581139330. DOI: 10.1145/1026776.1026780. URL: http://dl.acm.org/citation.cfm?id=1026776.1026780 (cit. on pp. 9, 10).

Fussell, Susan et al. (2004). "Gestures Over Video Streams to Support Remote Collaboration on Physical Tasks." In: *Human-Computer Interaction* 19.3, pp. 273–309. ISSN: 0737-0024. DOI: 10.1207/s15327051hci1903_3. URL: http://dl.acm.org/citation.cfm?id=1466555.1466558 (cit. on pp. 9, 14, 15).

Gauglitz, Steffen et al. (2014). "World-stabilized annotations and virtual scene navigation for remote collaboration." In: *Proceedings of the 27th annual ACM symposium on User interface software and technology*. ACM. New York, New York, USA: ACM Press, pp. 449–459. URL: http://dl.acm.org/citation.cfm?id=2642918.2647372 (cit. on pp. 4, 9–11).

Gemmell, Jim et al. (2000). "Gaze awareness for video-conferencing: A software approach." In: *IEEE Multimedia* 7.4, pp. 26–35. URL: http://www.computer.org/csdl/mags/mu/2000/04/u4026.pdf (cit. on p. 2).

Gergle, Darren and Alan T. Clark (2011). "See what i'm saying?" In: *Proceedings of the ACM 2011 conference on Computer supported cooperative work - CSCW '11*. New York, New York, USA: ACM Press, p. 435. ISBN: 9781450305563. DOI: 10.1145/1958824.1958892. URL: http://dl.acm.org/citation.cfm?id=1958824.1958892 (cit. on p. 14).

Gross, Markus et al. (2003). "blue-c." In: *ACM Transactions on Graphics* 22.3, p. 819. ISSN: 0730-0301. DOI: 10.1145/882262.882350. URL: http://dl.acm.org/citation.cfm?id=882262.882350 (cit. on pp. 2, 3).

Grubert, Jens, Tobias Langlotz, and Raphaël Grasset (2011). *Augmented reality browser survey*. Tech. rep. 1101. Institute for Computer Graphics and Vision, University of Technology Graz. URL: http://www.icg.tu-graz.ac.at/publications/augmented-reality-browser-survey-1 (cit. on p. 83).

Handley, Mark, Colin Perkins, and Van Jacobson (2006). *SDP: session description protocol*. Tech. rep. Telecommunication Standardization Sector Recommendation. URL: http://tools.ietf.org/html/rfc2327 (cit. on p. 22).

Hersent, Olivier, David Gurle, and Jean-Pierre Petit (2000). *IP Telephony: Packet-based multimedia communications systems*. Addison-Wesley Professional (cit. on p. 22).

Horvath, Karl and Matthew Lombard (2010). "Social and Spatial Presence: An Application to Optimize Human-Computer Interaction." In: *PsychNology Journal* 8.1, pp. 85–114. URL: http://www.psychnology.org/File/PNJ8(1)/PSYCHNOLOGY%5C_JOURNAL%5C_8%5C_1%5C_HORVATH.pdf (cit. on p. 10).

Huang, Weidong and Leila Alem (2011). "Supporting hand gestures in mobile remote collaboration: a usability evaluation." In: *Proceedings of the 25th BCS Conference on Human-Computer Interaction*. British Computer Society, pp. 211–216. URL: http://dl.acm.org/citation.cfm?id=2305316.2305356 (cit. on pp. 4, 6, 9, 14).

ITU-T (2005). *H. 264, Advanced video coding for generic audiovisual services*. Tech. rep. Telecommunication Standardization Sector Recommendation H. 264-ISO/IEC 14496-10 AVC. URL: http://www.itu.int/rec/T-REC-H.264 (cit. on p. 21).

ITU-T (2013). *H. 265: High Efficiency Video Coding*. Tech. rep. Telecommunication Standardization Sector Recommendation. URL: http://www.itu.int/rec/T-REC-H.265/en (cit. on p. 21).

Jo, Hyungeun and Sungjae Hwang (2013). "Chili: viewpoint control and on-video drawing for mobile video calls." In: *CHI'13 Extended Abstracts on Human Factors in Computing Systems*. ACM. New York, New York, USA: ACM Press, pp. 1425–1430. URL: http://dl.acm.org/citation.cfm?id=2468356.2468610 (cit. on pp. 5, 15, 22, 26, 27, 78).

Johnson, Steven, Madeleine Gibson, and Bilge Mutlu (2015). "Handheld or Handsfree?" In: *Proceedings of the 2015 conference on Computer supported cooperative work*. ACM Press. ISBN: 9781450329224. DOI: 10.1145/2675133.2675176. URL: http://dx.doi.org/10.1145/2675133.2675176 (cit. on pp. 9, 10).

Jones, Brennan et al. (2015). "Mechanics of Camera Work in Mobile Video Collaboration." In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems - CHI '15*. ACM Press. ISBN: 9781450331456. DOI: 10.1145/

2702123.2702345. URL: http://dx.doi.org/10.1145/2702123.2702345 (cit. on pp. 4, 13, 15, 84).

Jouppi, Norman P. (2002). "First steps towards mutually-immersive mobile telepresence." In: *Proceedings of the 2002 ACM conference on Computer supported cooperative work - CSCW '02*. New York, New York, USA: ACM Press, p. 354. ISBN: 1581135602. DOI: 10.1145/587078.587128. URL: http://dl.acm.org/citation.cfm?id=587078.587128 (cit. on pp. 4, 9).

Judge, Tejinder K. and Carman Neustaedter (2010). "Sharing conversation and sharing life." In: *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*. ACM Press. ISBN: http://id.crossref.org/isbn/9781605589299. DOI: 10.1145/1753326.1753422. URL: http://dx.doi.org/10.1145/1753326.1753422 (cit. on p. 4).

Kasahara, Shunichi and Jun Rekimoto (2014). "JackIn." In: *Proceedings of the 5th Augmented Human International Conference on - AH '14*. New York, New York, USA: ACM Press, pp. 1–8. ISBN: 9781450327619. DOI: 10.1145/2582051.2582097. URL: http://dl.acm.org/citation.cfm?id=2582051.2582097 (cit. on pp. 5, 9–11).

Kim, Hyejin, Gerhard Reitmayr, and Woontack Woo (2012). "IMAF: in situ indoor modeling and annotation framework on mobile phones." In: *Personal and Ubiquitous Computing* 17.3, pp. 571–582. ISSN: 1617-4917. DOI: 10.1007/s00779-012-0516-3. URL: http://dx.doi.org/10.1007/s00779-012-0516-3 (cit. on p. 38).

Kratz, Sven et al. (2014). "Polly." In: *Proceedings of the 16th international conference on Human-computer interaction with mobile devices & services - MobileHCI '14*. New York, New York, USA: ACM Press, pp. 625–630. ISBN: 9781450330046. DOI: 10.1145/2628363.2628430. URL: http://dl.acm.org/citation.cfm?id=2628363.2628430 (cit. on pp. 4, 10, 11).

Lehikoinen, Jaakko T. and Anne Kaikkonen (2006). "PePe field study." In: *Proceedings of the 8th conference on Human-computer interaction with mobile devices and services - MobileHCI '06*. New York, New York, USA: ACM Press, p. 53. ISBN: 1595933905. DOI: 10.1145/1152215.1152228. URL: http://dl.acm.org/citation.cfm?id=1152215.1152228 (cit. on p. 11).

Ludwig, Scott et al. (2009). *Xep-0166: Jingle*. Tech. rep. XMPP Standards Foundation. URL: http://xmpp.org/extensions/xep-0166.html (cit. on p. 22).

Norris, James, Holger Schnädelbach, and Guoping Qiu (2012). "CamBlend." In: *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems - CHI '12*. New York, New York, USA: ACM Press, p. 627. ISBN: 9781450310154. DOI: 10.1145/2207676.2207765. URL: http://dl.acm.org/citation.cfm?id=2207676.2207765 (cit. on pp. 9, 12).

Park, Sung Cheol, Min Kyu Park, and Moon Gi Kang (2003). "Super-resolution image reconstruction: a technical overview." In: *IEEE Signal Processing Magazine* 20.3, pp. 21–36. ISSN: 1053-5888. DOI: 10.1109/msp.2003.1203207. URL: http://dx.doi.org/10.1109/MSP.2003.1203207 (cit. on p. 84).

Pece, Fabrizio et al. (2013). "Panoinserts." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems - CHI '13*. New York, New York, USA: ACM Press, p. 1319. ISBN: 9781450318990. DOI: 10.1145/2470654.2466173. URL: http://dl.acm.org/citation.cfm?id=2470654.2466173 (cit. on p. 12).

Pirchheim, Christian, Dieter Schmalstieg, and Gerhard Reitmayr (2013). "Handling pure camera rotation in keyframe-based SLAM." In: *Mixed and Augmented Reality (ISMAR), 2013 IEEE International Symposium on*. IEEE, pp. 229–238. ISBN: 978-1-4799-2869-9. DOI: 10.1109/ismar.2013.6671783. URL: http://dx.doi.org/10.1109/ISMAR.2013.6671783 (cit. on p. 83).

Rämö, Anssi and Henri Toukomaa (2011). "Voice Quality Characterization of IETF Opus Codec." In: *INTERSPEECH*, pp. 2541–2544 (cit. on p. 22).

Redl, S.M. et al. (1995). *An Introduction to GSM (Artech House Mobile Communication Series)*. Artech House Publishers. ISBN: 0890067856. URL: https://books.google.at/books?id=9Ud4QgAACAAJ (cit. on p. 22).

Rosenberg, Jonathan et al. (2002). *SIP: session initiation protocol*. Tech. rep. Telecommunication Standardization Sector Recommendation. URL: https://tools.ietf.org/html/rfc3261 (cit. on p. 22).

Schubert, Thomas, Frank Friedmann, and Holger Regenbrecht (2001). "The Experience of Presence: Factor Analytic Insights." In: *Presence: Teleoperators and Virtual Environments* 10.3, pp. 266–281. ISSN: 1531-3263. DOI: 10.1162/105474601300343603. URL: http://www.mitpressjournals.org/doi/abs/10.1162/105474601300343603 (cit. on p. 2).

Siek, Katie A, Yvonne Rogers, and Kay H Connelly (2005). "Fat finger worries: how older and younger users physically interact with PDAs." In: *Human-Computer Interaction-INTERACT 2005*. Springer, pp. 267–280 (cit. on p. 77).

Sodhi, Rajinder S et al. (2013). "BeThere: 3D mobile collaboration with spatial input." In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. New York, New York, USA: ACM Press, pp. 179–188. URL: http://dl.acm.org/citation.cfm?id=2470654.2470679 (cit. on pp. 9, 14).

Stafford, Aaron, Wayne Piekarski, and Bruce Thomas (2006). "Implementation of god-like interaction techniques for supporting collaboration between outdoor AR and indoor tabletop users." In: *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*. IEEE, pp. 165–172. ISBN: 1-4244-0650-1. DOI: 10.1109/ismar.2006.297809. URL: http://dl.acm.org/citation.cfm?id=1514201.1514231 (cit. on pp. 6, 9, 13).

Steedly, D., C. Pal, and R. Szeliski (2005). "Efficiently registering video into panoramic mosaics." In: *Tenth IEEE International Conference on Computer Vision (ICCV'05) Volume 1*. Vol. 2. IEEE, 1300–1307 Vol. 2. ISBN: 0-7695-2334-X. DOI: 10.1109/iccv.2005.86. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1544870 (cit. on p. 12).

Tonnis, Marcus and Gudrun Klinker (2006). "Effective control of a car driver's attention for visual and acoustic guidance towards the direction of imminent dangers." In: *2006 IEEE/ACM International Symposium on Mixed and Augmented Reality*. IEEE, pp. 13–22. ISBN: 1-4244-0650-1. DOI: 10.1109/ISMAR.2006.297789. URL: http://dl.acm.org/citation.cfm?id=1514201.1514211 (cit. on p. 86).

Valin, Jean-Marc, Koen Vos, and T Terriberry (2012). *Definition of the opus audio codec*. Tech. rep. Internet Engineering Task Force (IETF). URL: https://tools.ietf.org/html/rfc6716 (cit. on p. 22).

Vos, Koen, Soeren Jensen, and Karsten Soerensen (2009). *Silk speech codec*. Tech. rep. IETF Standards Track Internet-Draft. URL: https://tools.ietf.org/html/draft-vos-silk-02 (cit. on p. 22).

W3C (2015). *WebRTC 1.0: Real-time Communication Between Browsers*. URL: http://w3c.github.io/webrtc-pc/ (cit. on pp. 7, 32).

Wagner, Daniel et al. (2010). "Real-time panoramic mapping and tracking on mobile phones." In: *2010 IEEE Virtual Reality Conference (VR)*. IEEE, pp. 211–218. ISBN: 978-1-4244-6237-7. DOI: 10.1109/vr.2010.5444786. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5444786 (cit. on pp. 7, 9, 12, 13, 15, 37, 54).

Yang, Ruigang et al. (2007). "Immersive Video Teleconferencing with User-Steerable Views." In: *Presence: Teleoperators and Virtual Environments* 16.2, pp. 188–205. ISSN: 1054-7460. DOI: 10.1162/pres.16.2.188. URL: http://www.mitpressjournals.org/doi/abs/10.1162/pres.16.2.188 (cit. on p. 6).

Zhu, Dingyun, Tom Gedeon, and Ken Taylor (2011). "Exploring camera viewpoint control models for a multi-tasking setting in teleoperation." In: *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11*. New York, New York, USA: ACM Press, p. 53. ISBN: 9781450302289. DOI: 10.1145/1978942.1978952. URL: http://dl.acm.org/citation.cfm?id=1978942.1978952 (cit. on p. 9).