

Design and Development of a Desktop-Based Tele-Immersion Prototype



Knoblauch Daniel

Masterarbeit
Winter 2005/06

Computer Graphics Laboratory
D-INFK
ETH Zürich
Prof. Dr. Markus Gross
Dr. Stephan Würmlin

Institute for Data Analysis and Visualization
Computer Science Department
UC Davis
Prof. Dr. Oliver Stadt

Abstract

Tele-Immersion enables users in different locations to collaborate in a shared, simulated environment as if they were in the same physical room. In a tele-immersive environment, users and objects are tracked and captured to permit them to be projected in realistic, multiple, geographically distributed immersive environments where individuals can interact with each other and with computer generated models. Most tele-immersive systems make use of large spaces and expensive hardware. This master thesis introduces a Desktop-Based Tele-Immersion System that limits the used space to the area in front of the desktop display and uses modules one can buy off the rack.

Based on previous work in the blue-c system we tried to adjust this software to our new requirements. Due to the unnecessary complexity of the networking infrastructure and the large amount of hardware used by this approach we discarded it and designed a new system from scratch. This approach is based on a graphics hardware accelerated disparity-map algorithm that allows us to capture users and objects in an efficient way to render them as 3D objects on an auto-stereoscopic display.

This system has been implemented and tested in its ability of real-time performance. From the results it was apparent that the new design was able to perform in a real-time environment and to construct 3D models. The efficiency of our system can be improved by decreasing the delay between capturing and rendering. This can be done by improving the implementations of the subprojects and their combinations and through parallelizing them as far as possible.

Zusammenfassung

Tele-Immersion ermöglicht die Zusammenarbeit verschiedener, voneinander getrennter Benutzer in einer gemeinsamen, simulierten Umgebung. In einer Tele-Immersion Umgebung werden Benutzer und Objekte erkannt und aufgenommen um sie auf realistische Weise in verschiedenen, geographisch verteilten immersen Umgebungen wiederzugeben. Diese Umgebungen ermöglichen die Interaction mit verschiedenen Benutzern oder mit computergenerierten Objekten. Die meisten Tele-Immersion Systeme benötigen viel Raum und teure Hardwarekomponenten. In dieser Masterarbeit versuchen wir ein Desktop basiertes Tele-Immersion System einzuführen, das sich auf den Raum vor dem Bildschirm beschränkt und mit einfach zugänglichen Modulen auskommt.

Auf der Arbeit im blue-c Projekt aufbauend, versuchten wir diese Software an unsere neuen Anforderungen anzupassen. Auf Grund der unnötig komplexen Netzwerkinfrastruktur und der grossen Anzahl an Hardwarekomponenten haben wir diesen Ansatz verworfen und ein eigenes System von Grund auf entwickelt. Diesen Ansatz bauten wir auf einer Graphikhardware basierten Tiefenbildalgorithmus auf der uns ermöglichte Benutzer und Objekte in effizienter Weise aufzunehmen und als 3D Objekte auf einem auto-stereoskopischen Bildschirm darzustellen.

Dieses System wurde implementiert und auf die Echtzeittauglichkeit getestet. Aus den Resultaten wurde klar, dass wir ein System entwickelt haben das 3D Objekte in einer Echtzeitumgebung aufnehmen und darstellen kann. Die Effizienz unseres Systems kann durch eine Verkleinerung der Zeitverzögerung zwischen Aufnahme und Darstellung verbessert werden. Dies kann durch effizientere Implementierung der einzelnen Systeme und ihrer Kombination sowie der möglichst parallelen Verarbeitung dieser Module verbessert werden kann.

Index of Contents

1	Introduction and Motivation	1
1.1	Project	1
1.2	Motivation	1
1.3	Overview	2
2	Fundamentals and Related Work	3
2.1	Tele-Immersion	3
2.2	Auto-Stereoscopic Displays	6
2.3	Visual-Hull-Based Approach	8
2.3.1	<i>blue-c Design</i>	8
2.3.2	<i>Visual Hull Approach</i>	8
2.3.3	<i>Point-Based Rendering</i>	9
2.3.4	<i>Differential 3D Fragment Operators</i>	10
2.3.5	<i>Active Camera Control</i>	10
2.3.6	<i>CORBA and Audio/Video-Streams</i>	10
2.4	Disparity-Map Based Approach	10
2.4.1	<i>Brick Idea</i>	10
2.4.2	<i>Graphics Hardware based Depth Calculation</i>	11
2.4.3	<i>Rectification and Calibration</i>	12
3	Realization	15
3.1	Auto-Stereoscopic Display	15
3.2	Visual Hull based Approach	17
3.2.1	<i>64 Bit vs. 32 Bit</i>	17
3.2.2	<i>Portation from Audio/Video-Streams to Sockets</i>	17
3.2.3	<i>CORBA, Audio/Video-Streams Upgrade</i>	18
3.2.4	<i>One Camera per Machine to several Cameras</i>	18
3.3	Disparity-Map based Approach	19
3.3.1	<i>Design</i>	19
3.3.2	<i>Capturing</i>	21
3.3.3	<i>Rectification</i>	21
3.3.4	<i>Disparity-Map Calculation</i>	23
3.3.5	<i>Calculation of Point Coordinates</i>	24
4	Results and Conclusion	29
4.1	Auto-Stereoscopic Rendering	29
4.2	Calibration and Rectification	30
4.3	Disparity-Map Results	31
4.4	Stereo Rendering	34
4.5	Overall Capturing Speed	37
4.6	Conclusion	39
4.7	Acknowledgments	41
5	Bibliography	43
A	Changes	45
A.1	Changes in link.bcl	45
A.2	Changes in CORBA and Audio/Video-Streams	45
A.3	Changes in OpenCV Library	49

B How to use prototype	51
B.1 Calibration and Preparation	51
B.2 Capturing and Rendering	51

1

Introduction and Motivation

In this chapter we will introduce you to the main ideas of our project and explain why we considered doing this work. At the end we will give an overview over this report.

1.1 Project

The goal of our project is to develop and design a desktop-based tele-immersion system which enables examination of different ways of virtual interaction between users that are separated in the real world.

Our approach to Tele-Immersion does not require expensive and space-consuming hardware devices but goes into the other direction. With a desktop-based affordable solution we want to provide the opportunity for everyone to use this kind of communication in the near future. This means that we have to find solutions for acquiring images and constructing 3D models of the captured objects that are able to work with off-the-shelf hardware. Beside this constraint we have to assure that our approaches work in real time, to give a real-life impression using the virtual world. This structure also provides us with the advantage that we only have to focus on the space in front of the display when trying to capture the users. Having this constraint we can use simpler approaches that suit our real-time need much better. We also do not want the users to use additional tools like shutter glasses to be able to see the opponents in 3D. This is why we use a auto-stereoscopic display that can show 3D without having additional hardware. Figure 1.1 depicts the current version of our prototype.

The main goal of this master thesis is to design and develop a prototype that provides the possibility of recording users and representing them in 3D on an auto-stereoscopic display. To do this we examined different ways of data capturing that enables us to reconstruct a 3D model of the users, which can be rendered on a auto-stereoscopic display.

1.2 Motivation

With globalization in nearly all parts of our life the need for collaboration between users in different places has grown rapidly. This development has led to several attempts of bringing people together without having to be in the same room. Beginning with the telephone the need



Figure 1.1: Our Desktop-Based Tele-Immersion Prototype.

for visual contact has emerged. This led to video conferencing tools which have never really been accepted as replacement for human-to-human interaction. One of the problems of this approach is the fact that it is not possible to have eye-contact with the opponent. Several studies have shown that eye-contact is an essential part of communication. This is why tele-immersion systems have been introduced. They are not wide spread yet because of the fact that they mostly need a lot of place and are not affordable for everybody. As Tele-Immersion approaches usually also include a 3D representation of the opponent wearing shutter glasses has been necessary for most approaches. But this prevents eye-contact as well.

These are the reasons why we introduce a new approach to Tele-Immersion. We use components that do not need much space. In fact we want to develop a Desktop-Based Tele-Immersion System that builds on components that can be bought off the rack. We use an auto-stereoscopic display that allows us to have a 3D representation without wearing shutter glasses and a minimal amount of cameras to capture the conversational partner and construct a 3D model of him. This allows us to interpolate between the captured pictures and give the opportunity to render the user directly looking at his opponent. It also enables us to render the user in relation to available interactive objects or data that can be manipulated and seen by all the users online.

1.3 Overview

We will discuss in Section 2 the fundamentals and related work for our project. We will introduce you to the subject of Tele-Immersion in Section 2.1. In Section 2.2 we will present our auto-stereoscopic display and explain its functionalities. Section will talk about firewire cameras and how we use them. Section 2.3 and Section 2.4 will introduce you to related work on which we built up our approaches.

In Section 3.1 we will show how we achieved rendering on our auto-stereoscopic display. We acquired our data by two different approaches. The visual hull based approach, building up on the blue-c project is introduced in Section 3.2 and the disparity-map approach in Section 3.3. We will show you some results in Section 4 and give a conclusion about our work in Section 4.6.

2

Fundamentals and Related Work

In this chapter we want to introduce the concept of tele-immersion and present previous projects that have been done on this topic. Out of this knowledge we will present our own approach. Along with this presentation we will introduce you to our special hardware and to the techniques we used to develop our Tele-Immersion prototype.

2.1 Tele-Immersion

There has always been an urge to bring people who are separated from each other together, not only to communicate with each other but also to collaborate. At the beginning the only method of doing so were letters. With advances in technology, telephones and even video conferencing were introduced. But all of these approaches have their disadvantages and problems. With a telephone you are not able to see the conversational partner. With a video conferencing tool you can see your collaboration partner but you are still not able to interact with him like in the real world. One of the reasons for this is the lack of ability to build up eye-contact between the conversational partners, which is an essential part of communication between humans, as can be seen in Figure 2.1. There is also no spatial interaction between the users, which makes it

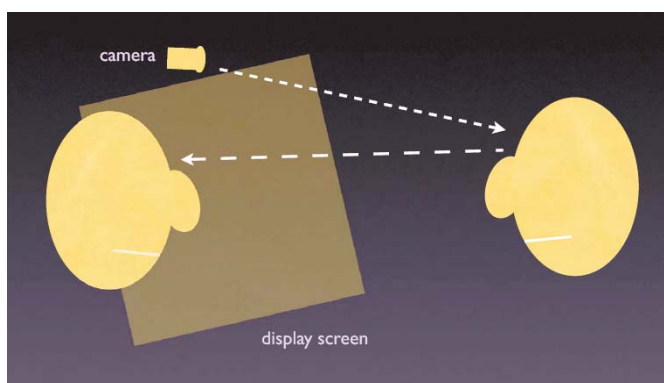


Figure 2.1: Usual constellation in video conferencing tools. Camera and display are not on the same sight line which prevents eye-contact with the conversation partner. (image courtesy of Jaron Lanier).

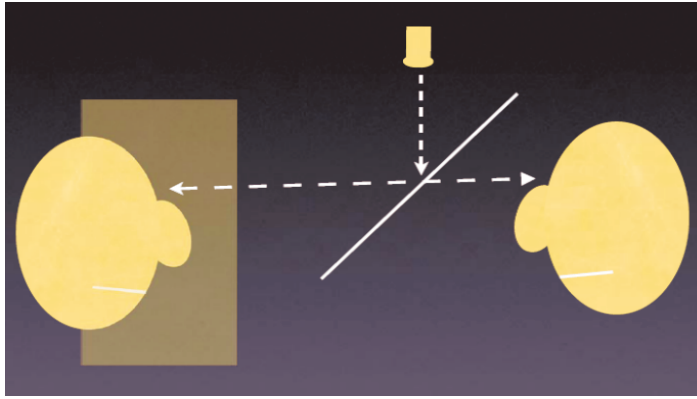


Figure 2.2: The half-silvered mirror allows the camera and the display to be in the same sight line. This results in the possibility to build up eye-contact with your conversation partner. (image courtesy of Jaron Lanier).

difficult to work with each other.

There have been several attempts to solve the eye-contact problem. One of them is to share a single sight line between the two participants. This can be achieved by a half-silvered mirror, as illustrated in Figure 2.2 or with an image-based simulation of the opponent like it was done at University of North Carolina, Chapel Hill (UNC), shown in Figure 2.3. These solutions break down as they only work for interaction between two opponents. As soon as we have more participants it is not possible to build up an eye-sight-line to each user. Considering this problem, the most current strategy is to skew up the images of users so that a correct perspective is not even suggested. We can see an example from Apple's iChat in Figure 2.4. Another approach is



Figure 2.3: We can see an image-based simulation of the opponent suggested at UNC.

reduce resolution so that features like eye and mouth are not ambiguous anymore.

The goal is to bring people that are in different places together to one place, not only as images and audio but as a whole individual. This is why Virtual Reality tools have been used to advance these technologies. If we combine video conferencing with virtual reality we get a tool that allows the communication and interaction of opponents, in different places, in the same virtual place. This approach is called Tele-Immersion.

There have already been several approaches to this idea. The first major Tele-Immersion research project was the National Tele-Immersion Initiative in the 1990's, we can see their approach in Figure 2.5. This approach enables persons in different offices to interact with each other as if they were in the same office. The back- and foreground of the users' representations are static environments that simulate the extension of one's own office. The users are rendered interactively, building up an eye-sight-line with each other. Using several displays the interaction between several collaborators is supported. Another major tele-immersion project was blue-c [1]. This approach enables the interaction and collaboration of several CAVE users.



Figure 2.4: Apple's iChat solves the problem by skewing up the images to not suggest the possibility of eye-contact.



Figure 2.5: The National Tele-Immersion Initiative introduced an approach that enabled several communication partners to meet in one extended office. (image courtesy of [26]).



Figure 2.6: The blue-c project enables several CAVE users to meet in a virtual environment and interact with each other. (image courtesy of [1]).



Figure 2.7: HP's Coliseum project tries to keep a large enough sensor array to capture several users and having a small display, sacrificing the sight line between the users. (image courtesy of [27]).

This creates the opportunity to bring the users into a virtual world. You can see this approach in Figure 2.6. Another approach has been done by Hewlett-Packard's Coliseum. This project is an experiment to keep the sensor array large enough to see several users and to keep the screen small. To achieve, the true sight lines are given up and they do not support any stereo, as you can see in Figure 2.7. Unfortunately none of these approaches can simultaneously support full duplex communication, more than two users, and the normal range human motion while seated. Many of the configurations fail because users have to wear shutter glasses to see their opponents in 3D.

2.2 Auto-Stereoscopic Displays

Wearing shutter glasses in communication environments leads to several problems. As already mentioned in Section 2.1 one of the reasons for 3D representation in Tele-Immersion is to give the ability to make eye contact with the opponent. If we have to wear shutter glasses this is not possible. Besides this, we want to simulate an environment as realistic as possible and wearing shutter glasses does not support that. This is why we focused on auto-stereoscopic displays, specifically on the Synthagram 204 display by Real D [24].

This display is a normal LCD display with an array of lenticular lenses in front of it that breaks the sight lines of each eye in a way that we see two different pixels with each eye. You can see this in Figure 2.8. By having nine pictures of an object from nine different cameras (real or virtual) on one horizontal aligned camera array, as shown in Figure 2.9, we can interzig the pixels of these nine views so that the display can show the object in 3D based on the lenticular array. We can see this conversion in Figure 2.10. Interzigging is the act of joining the nine views with help of a precomputed interzigging map to one image. The interzigging map contains the information of which pixel from which image has to stay on a certain pixel of the resulting image. The lenses and the corresponding interzigged pixels on the display are ordered in a way that we have several viewing zones. This allows several people to see the 3D images from different angles at the same time. Support for multiple simultaneous users of the Synthagram display is illustrated in Figure 2.11. Each viewing zone consists of nine views which leads to several stereo pairs and stereo-views. This allows us to move the head around in front of the display and see the object from slightly different angles, which is considered as the look around effect. The viewed pixels for one image pixel change not only by moving the head from the left to the right but also if we go fourth and back in front of the display. This gives us the ability to

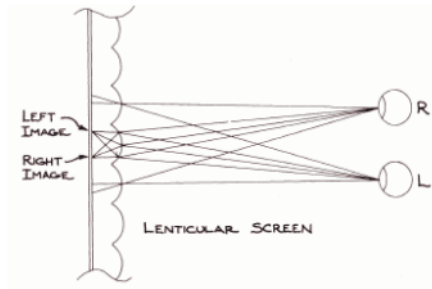


Figure 2.8: Due to the fact that the lenses break the sight lines of the eyes, each eye sees a different pixel on the display. Thanks to the special configuration of the pixels we can see 3D images. (image courtesy of [24]).



Figure 2.9: We need 9 different views from cameras that are aligned on one horizontal line to create our 3D image. (image courtesy of [24]).

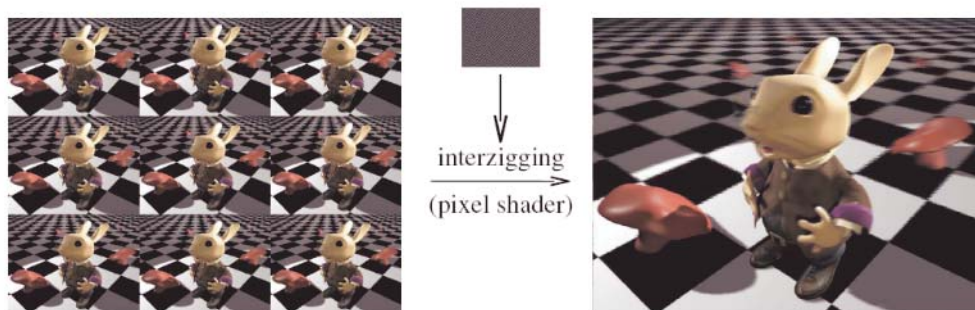


Figure 2.10: On the left-hand side we can see the 9 camera views. On the right hand side we see the interzigged image that shows a 3D image on the auto-stereoscopic display. The interzigging is done by an interzigging map shown in the middle of the figure.

move in front of the display without loosing quality of the 3D image.

By using vertex shaders to implement the interzigging we can achieve real-time render speed, despite the fact that we have to render nine different images for each frame. Firewire Cameras

To capture our users we use Firewire cameras. We decided to use Point Grey's Flea cameras with a resolution of 1024x768 pixels and the corresponding frame rate of 30 frames per second. As we have such a high resolution we can make a color reconstruction by down sampling the images and still have a resolution of 512x384 pixels. The small size of the cameras allows us to place them in nearly every position and combination that could be an advantage for our 3D reconstruction. The large amount of data transmitted by each camera restricts us to a structure

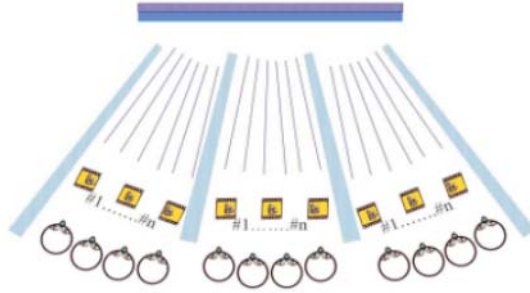


Figure 2.11: There are different viewing zones in front of the display that can see the same 3D pictures. Each of these viewing zones includes several stereo-pair pictures that allow a look-around-effect. (image courtesy of [24]).

where not more than two cameras can be used at one client machine. This restriction is given by the maximum bandwidth 400 Mbit/s of Firewire. If we want to use more than two cameras we will have to use an appropriate number of clusters. Another way of avoiding this restriction is to capture with smaller resolution using Format7, which enables us to capture only parts of the image. But this would lead to less information from each picture and to a more expensive color reconstruction. Using a synchronization unit from Point Grey we can guarantee synchronized capturing with multiple cameras.

2.3 Visual-Hull-Based Approach

As there have already been a large amount of research projects on this topic and the fact that the data of the blue-c project, presented in paper [1], was available we decided to use these sources in the first step and adjust them to our needs. In the following chapter we will give an overview of the blue-c project.

2.3.1 blue-c Design

The blue-c project combines simultaneous acquisition of multiple live video streams with advanced 3D projection technology in a CAVE like environment, creating the impression of total immersion. Out of the multiple video streams a 3D video representation of the user is computed in real time using a visual hull based approach. The resulting representation is rendered using a point-based method [7] with active stereo on the three projection walls, as you can see in Figure 2.6. The communication between the capturing clients and the rendering is based on a CORBA/AV-Streams environment. The constructed 3D video is streamed in real time using a technique introduced by Wuermlin et al. [4].

The blue-c system consists of 16 cameras. Each of these cameras is driven by one cluster which performs all the pre computations like segmentation and contour extraction. All of the resulting image data is streamed to a processing unit where the 3D model is reconstructed using a visual hull approach. The resulting points of the 3D model are then submitted to the renderer where the data is displayed on the CAVE walls in active stereo. You can see this design in Figure 2.12.

2.3.2 Visual Hull Approach

The visual hull approach in blue-c is based on the image-based visual hull method by Matusik et al. [2], which takes advantage of epipolar geometry to build a layered depth-image

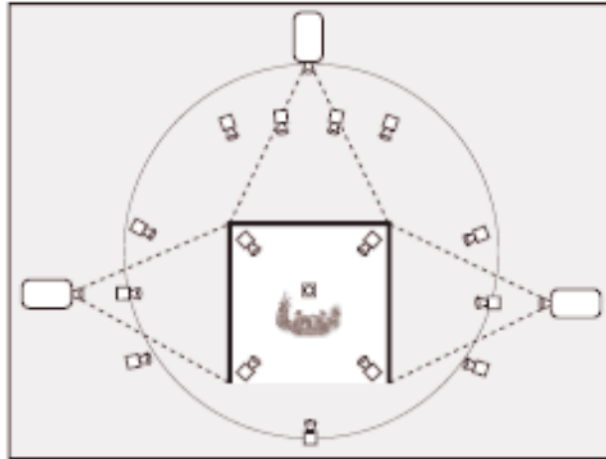


Figure 2.12: The blue-c system consists of 16 cameras, each driven by one cluster and 3 active stereo walls with corresponding projectors. (image courtesy of [1]).

representation, and the polyhedral visual hull method [3], which constructs a triangular surface representation. These approaches have been modified to enable point-based rendering.

The image-based visual hull method was implemented by extracting contours from the foreground objects as shown in [6]. These objects are found by using background extraction. This leads to a reduced amount of points of interest and reveals the object from which the contour has to be extracted.

The intersection of the resulting silhouette cones, calculated from different points of view, define an approximate geometric representation of an object called a visual hull, as shown in

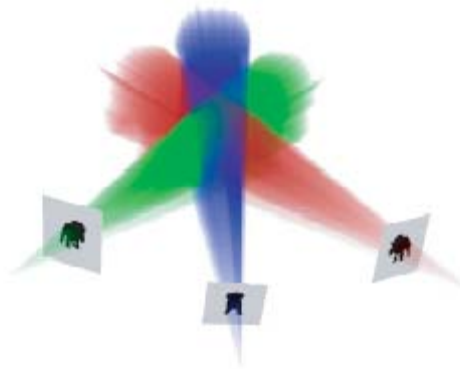


Figure 2.13: The intersection of all silhouette cones leads to the visual hull which is an approximate representation of a 3D object. (image courtesy of [2]).

Figure 2.13. Out of this information we can calculate a 3D point for every pixel of interest in each video stream.

2.3.3 Point-Based Rendering

One way of doing point-based rendering is to use `GL_POINTS` as splats for each 3D point. As these splats cannot adjust to the shape of the object's silhouette in an optimal way, the EWA surface splatting approach from Zwicker et al. [7] is used. This rendering technique is based on a screen space formulation of the elliptical weighted average (EWA) filter which is adapted for irregular point samples. EWA surface splatting provides the system with high-quality images in real-time.

2.3.4 Differential 3D Fragment Operators

blue-c uses a differential update scheme for dynamic point samples as shown in [4]. The basic primitives of this scheme are point samples with different attributes like position, a surface normal vector, and a color. The update scheme is expressed in terms of three different 3D fragment operators. These operators are INSERT, which adds new 3D video fragments, DELETE, which removes 3D video fragments and UPDATE, which corrects appearance and geometry attributes of 3D video fragments. Using this update scheme the data traffic has significantly been reduced.

2.3.5 Active Camera Control

As blue-c is designed for real-time use, the amount of transferred and used data has to be as small as possible without losing quality. This is the reason why active camera control has been introduced to the blue-c system. If the current virtual viewpoint is known, we can figure out which cameras have to transfer which amount of data. This is why the cameras can have three different states in the blue-c system. They either submit intra-frame prediction scheme data as explained in Section 2.3.4 or they submit auxiliary information for the employed 3D reconstruction algorithm or they can even provide no data at all. This approach can be found in paper [8].

2.3.6 CORBA and Audio/Video-Streams

The blue-c environment has its own communication and system layer architecture, described in paper [1]. The data transmission within the blue-c portal, and between blue-c portals, is achieved with the programming model of CORBA architecture and its Audio/Video streaming service. All control and non-real-time information is transmitted using the traditional CORBA environment. The real-time data, however is transmitted via the Audio/Video-Stream implementation from the TAO/ACE framework [9].

Unfortunately the versions of TAO/ACE and the corresponding Audio/Video-Stream versions still used in the blue-c project have had several major releases and fundamental changes in the usage, which caused a lot of problems adopting the blue-c code to our environment, as with recent compilers the old versions did not compile anymore.

2.4 Disparity-Map Based Approach

Our second approach was designed by ourself and adjusted to our special needs. This means that we tried to use only a small number of cameras due to the fact that we only have to cover the space in front of our display to enable our Desktop-Based Tele-Immersion approach. We also tried to have as many cameras as possible on one cluster. As stated in Section there are two cameras for each machine. No extra processing machine should be needed and the amount of cameras should be scalable according to desired quality. We adapted several ideas from previous research projects and added them to a scalable acquisition and rendering system. In the next chapters we will present these ideas.

2.4.1 Brick Idea

Waschbüsch et al. [10] suggested a scalable 3D video approach for dynamic scenes. They introduced the idea of bricks. In this approach bricks are an arrangement of a projector, a texture camera and a stereo camera pair, as seen in Figure 2.14. Each of these bricks calculates a depth map for its own point of view, using the stereo cameras and a structured light pattern from the projector. Out of this information they construct the 3D points in a world coordinate system and

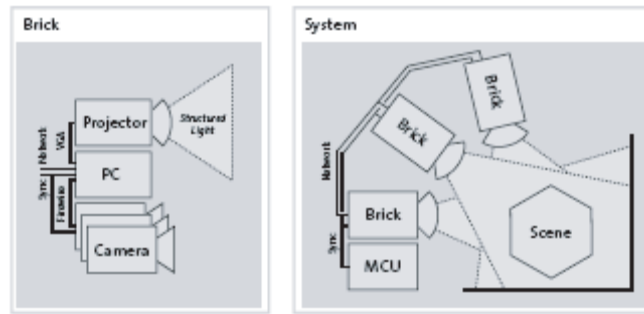


Figure 2.14: Brick consist in Waschbuesch’s approach of a projector for structural light, a texture camera and a stereo camera pair for the 3D reconstruction. Several Bricks can be used to acquire a scene (as seen on the right). (image courtesy of [10]).

color them using texture information from the texture camera. All of the calculated points are sent to the renderer where they are merged with the points from possible additional bricks and represented as one 3D picture. We can scale this approach until we have a point cloud dense and reliable enough to describe our scene.

We adapted the brick idea but changed the arrangement. As we want to use a minimal amount of cameras, we only use the stereo cameras and reconstruct depth and texture from them. We do not include a projector and construct our depths by background extraction and a hardware based disparity-map calculation algorithm presented in the following chapter. That means that we only have bricks including stereo camera pairs.

2.4.2 Graphics Hardware based Depth Calculation

There has been a lot of research on image-based rendering and real-time reconstruction/rendering systems. Beside Image-based Methods such as the visual hull method mentioned in Section 2.3.2, there are also techniques that build on vision-based methods. Depth from stereo seems to be the most widely available technique [11] because of its biological approach and the ease of data acquisition. The main idea is to find within the stereo image pairs the corresponding pixels and identify their disparity. The smaller the disparity, the further away the related 3D point is. This approach is based on the functionality of human eyes. However, to find the corresponding pixels we have to search in the whole image which leads to very expensive computations. These techniques often require special hardware [12,13,14] or search only locally, resulting in a trade off on precision and reliability. Mulligan and Daniilidis proposed a new trinocular stereo algorithm in software [15], where they introduced a number of techniques, such as motion prediction and assembly level instruction optimization to accelerate the computation. However the stereo matching algorithm is still the bottleneck in their method. The Yang et al. [16] approach uses commodity graphics hardware to accelerate the disparity calculation. In this approach they project the images of several cameras on a global sweep-plane, Figure 2.15. On each sweep-plane depth they calculate, with the help of programmable pixel shader technology, the square intensity differences between reference image pixels, and choose the depth of the plane with minimum difference, i.e. the most consistent color for each pixel. This results in a dense disparity-map from the current virtual view point, related to the global sweep-planes. Since this approach builds on several cameras but we only want to use stereo-pairs we looked at a following method introduced by Yang [17]. It is a similar approach but adjusted to stereo image pairs. Instead of projecting the images on a global sweep-plane with different depth values, the two images are moved in reference to each other in horizontal direction for a certain disparity value. This is done for every disparity value in a certain disparity range and the dis-

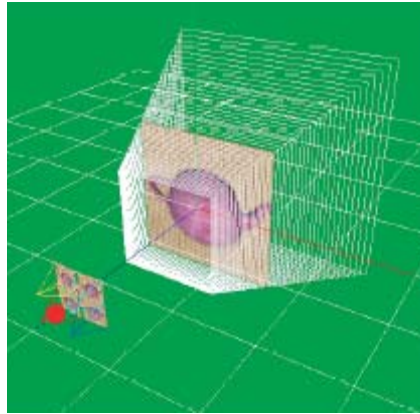


Figure 2.15: The images from the different reference cameras are projected on a sweep-plane parallel to the virtual view point for different depth values. On each of these planes the minimum difference in color per pixel is calculated. The depth where a pixel has the smallest difference is the depth where the corresponding 3D point lies in the real world.(image courtesy of [16]).

parity value with the smallest square intensity difference is considered to be the corresponding disparity value of this pixel. This is done with help of programmable pixel shaders. As we only have the information of two images this leads to a not so robust algorithm. Yang suggests to work not only with these two images but also with different mipmap levels. The disparity range calculation is done for every mipmap level and the best disparity value per pixel is chosen. This leads to a more robust algorithm, which has the disadvantage of a more “blurry” result that is not able to detect fine structures. Cornelis [18] suggested a derivation of this algorithm by adding some filtering techniques. We focus on Yang’s mipmap approach. To reduce the computational costs of the disparity calculation Yang suggests to use rectified images. In the next chapter we will discuss some rectification approaches.

2.4.3 Rectification and Calibration

Rectification is the transaction that enables us to transform images so that the epipolar lines are aligned horizontally. If this is the case we only have to look for corresponding pixels in one dimension and can solve our disparity problem much more efficiently. Epipolar geometry is explained in Figure 2.16.

There are different ways to achieve this goal. As mentioned in [17] it is possible to perform a 3x3 homography by implementing a projective texture mapping [19] on a GPU. Pollefeys [20] introduces a simple and efficient rectification method for general motion that can be necessary if the epipoles are in or close, to the images. There are also more universal methods like the one from Oram [21] that allows us to make a rectification for any epipolar geometry.

To create an efficient rectification we have to be able to tell where our cameras stand in our space and what parameters they have. This is why we need to calibrate our cameras. Calibration delivers us the extrinsic and intrinsic parameters of each camera. Extrinsic parameters tell us where in correspondence to a arbitrary point in real world our camera stands and the intrinsic parameters reflect the internal attributes of each camera, as focal length, pixel size and so on. As we already knew the calibration toolbox for MATLAB from Mei [22] we decided to use the OpenCV [23] implementation of this algorithms. As the CvCalibFilter of the cvaux library, which is part of the OpenCV project, included the possibility of rectifying our images, we decided to use this algorithm to get our initial results.

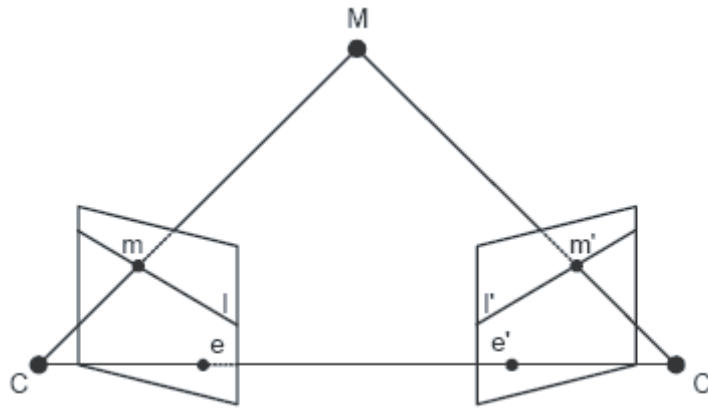


Figure 2.16: Epipolar geometry:

C and C' are the stereo cameras and M is a 3D point. e and e' are the epipoles of the cameras which means that they are the images of the cameras in the corresponding images. l and l' are the epipolar lines these are the mappings of the light line from the point M to the corresponding camera (for l' it is the connection between C and M on the image of camera C'). (image courtesy from [21]).

`CvCalibFilter` uses several images of a moving checkerboard from a stereo camera pair to calculate the position of the cameras relative to a world coordinate null point on this checkerboard. You can see this in Figure 2.17. The algorithm first extracts the edge points of the checkerboard and calculates with these points all the necessary parameters.

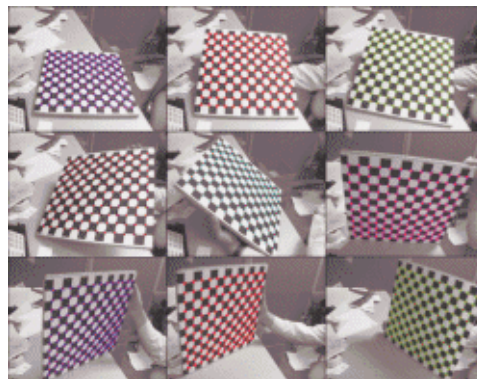


Figure 2.17: The corners of the checkerboard are extracted automatically and used for the calculation of the intrinsic and extrinsic parameters of the cameras. (image courtesy of [22]).

3

Realization

The first part of this chapter describes the auto-stereoscopic display and the corresponding rendering software for our project. Next we will show how we ported the blue-c code to present library versions and adjusted it to our needs. The second part discusses the design and implementation of our system from scratch.

3.1 Auto-Stereoscopic Display

Before beginning with the implementation of our Tele-Immersion system we had to confirm that it is possible to render in real time with our auto-stereoscopic display. To do this we implemented test programs to determine possible frame rates. The display was delivered by the manufacturer with an API that allowed rendering in 3D on a software based manner. We did not want to render based on software but with hardware support, which included vertex and fragment shaders. We used some given examples using hardware based rendering in DirectX to determine if a hardware based implementation is able rendering the needed nine views for our interzigging, explained in Section 2.2. As we wanted to use the auto-stereoscopic display on a Linux machine with OpenGL and not on a Windows machine with DirectX, we had to adjust our code to OpenGL and make sure that it compiled and worked on Linux.

We give a short overview of our implementation and the generic design for our renderer. As already mentioned in Section 2.2 we need nine views from nine cameras, in our case virtual ones, that are aligned on a horizontal line looking toward the scene. These nine views are joined to a nine-tile image on which we can execute our vertex shader to accomplish the interzigging. In the following we show this in a short passage of pseudo code:

```
StereoRenderer::Render()  
{  
    prepare OpenGL for frame;  
    for(all 9 camera views) {  
        glFrustum(...); // adjust the view perspective for this  
                        // camera image  
        glViewport(...); // give the part on the ninetile image  
                        // where this camera image has to be placed  
        gluLookAt(...); // readjust the eye point to the new  
                        // camera and to the object  
    }  
}
```

```

        DrawScene();           // draw the scene corresponding to
                               // the new camera given by previous
                               // instructions
    }
    //now we have ninetile image in color buffer
    read ninetile from color buffer;

    Bind shaders;

    glBegin(GL_QUAD);
        render quad with ninetile image as texture 1
        and the corresponding interzigging mask as texture 0
    glEnd(GL_QUAD);

    Disable shaders;

    glutSwapBuffers();
}

```

As can be seen the whole nine-tile production and interzigging is completely independent of the actual object rendering. This is the reason why we created a render function that takes as input a function that is responsible for the rendering of the objects. This leads to a generic algorithm that can render every scene you want, by providing the actual scene rendering routine. We have to take care that there are no OpenGL functions in this routine which interfere with the rest of the 3D rendering. In general we can say that if we do not clear any buffers there are no problems. Even if we never ran into problems with this small constraint it would be good to examine all the possible functions that can interfere with our algorithm. In the next code sample you can see our generic approach to the rendering function.

```

StereoRenderer::Render(void(*myScene)(void))
{
    prepare OpenGL for frame;
    for(all 9 camera views) {
        glFrustum(...); // adjust the view perspective for this
                        // camera image
        glViewport(...); // give the part on the ninetile image
                        // where this camera image has to be placed
        gluLookAt(...); // readjust the eye point to the new
                        // camera and to the object

        myScene();     // draw the scene corresponding to
                        // the new camera given by previous
                        // instructions
    }
    //now we have ninetile image in color buffer
    read ninetile from color buffer;

    Bind shaders;

    glBegin(GL_QUAD);
        render quad with ninetile image as texture 1
        and the corresponding interzigging mask as texture 0
    glEnd(GL_QUAD);
}

```

```

    Disable shaders;

    glutSwapBuffers();
}

```

3.2 Visual Hull based Approach

In this chapter we show our steps porting the blue-c code to the latest libraries and adjusting it to our system.

3.2.1 64 Bit vs. 32 Bit

After testing and preparing the auto-stereoscopic display we could advance and begin to port the blue-c system to our environment and needs. The first decision we had to make was whether we wanted to run our environment on machines with a 64 Bit or 32 Bit based Scientific Linux. As all the machines in the lab were configured with 64 Bit and we thought that this would improve compatibility for later approaches, we therefore decided to proceed with the 64 Bit version. This did not only mean that we had to compile the blue-c code on 64 Bit, but also all the libraries that were used. We had to do this with the firewire libraries libraw1394, dc1394_control, the Rendering environment Qt, and the communication environment CORBA and ACE/TAO. After investing some time we were able to compile and run them all on 64 Bit machines. As we discovered that there is no way to compile the Intel library libippi on 64 Bits and we knew that this library is an essential part of the blue-c code, we had to go back. The following steps were to install on all machines the 32 Bit version of Scientific Linux and recompile the libraries on it. After these changes we could finally begin to compile the blue-c code. As there have been several major releases on compilers we had to adjust a lot of syntax mainly on templates. Following these changes we found out that there had been some major releases in CORBA and ACE/TAO that included different syntax and exclusions of whole classes which were essential in the blue-c implementation for Audio/Video-Streams. Knowing this and the fact that we did not need such a complex communication environment, we decided to avoid the problem by using an already available socket approach for internal communication in the blue-c portal. This approach had already been used for information exchange between different blue-c portals.

3.2.2 Portation from Audio/Video-Streams to Sockets

The communication in the blue-c system is regulated by CORBA. This means that we run a naming service which runs an ORB where all the clients can register themselves. The ORB connects the right communication partners to each other with a direct Audio/Video-Stream. Since we wanted to use sockets instead of Audio/Video-Streams we had to change the method for building up the connections. We used the pre-existing socket classes in blue-c API but had to modify the process of naming the clients, as we wanted to assure that it is possible to connect several clients on one host to the processing unit as a later step. In the original communication protocol of the blue-c system the clients had the name of the host, since there was only one client per host. We changed that, enabling new names. We used not only the hostname but we also numbered the clients on it. This means that the names of the clients were not only the hostname anymore, but we also appended the number of the client on this host. Thus, we were able to allocate the right socket ports to the right clients regarding the configuration file. As the connections now had to be constructed by the clients themselves we had to include a new row in the configuration file link.bcl. This file is used in the blue-c system to specify all the parameters needed

to build up connections. The new entry in this file told the clients not only on which host they were but also the port and the host to which they had to connect for a connection to the processing unit. This changes had also to be included into the code. We describe the changes to the configuration file in Appendix A.1.

These changes allowed us to communicate between the clients and the processing unit with sockets. We tested the system in this configuration but no data was transmitted. After testing the incoming and out coming data in the used sockets and clients, we found that the connections were open and that there was a small amount of data transferred. A closer look at this data proved that there was a data transfer for each frame, but that it only consisted of headers and no information for the reconstruction. As this change was a fundamental change in the blue-c architecture and we could not tell if we truly achieved to do all the necessary changes, we decided to try the communication with Audio/Video-Streams.

3.2.3 CORBA, Audio/Video-Streams Upgrade

As we wanted to use several cameras on one machine, we did not have to restore all the changes from our socket approach. We continued naming our clients by hostname and the actual numeration on the corresponding host.

As previously mentioned in Section 3.2.2 we had to adjust our code to several major releases in CORBA, but mainly on TAO/ACE's Audio/Video-Streams. These modifications fundamentally changed the usage of these streams by even removing whole classes and functions that were used by the blue-c code. This was not the only trouble, but also the fact that they had changed the syntax of several functions in order, to make it more compatible to several platforms. We list the most important changes in Appendix A.2. Finding out these changes and integrating them into the blue-c code took us several weeks and threw us back in our time schedule. But the hope to have a working system after these changes prompted us to stay working on it. At the end, we finally managed to change the code in a way that all the clients and the processing unit could register at the naming service. This service established the Audio/Video-Streams between the corresponding communication partners. But we had the same result as we already had had with sockets. There was communication between the clients and the processing unit but no relevant data was transferred. Only headers with empty bodies were send around. We could determine now that the error should not be in the communication. Therefore, we tried to find the reason for our difficulties somewhere else. One of the biggest differences in our approach and the one in blue-c was that we were using only two cameras at the time compared to the 16 cameras in blue-c. The visual hull approach used by blue-c had problems working with only two cameras and the active camera control integrated in the system, as explained in Section 2.3.5, had problems with this number of cameras as well. As a result we increased the number of cameras used by our system. We introduced several cameras to one machine to at least double our number of cameras, since we only had a limited amount of machines to use.

3.2.4 One Camera per Machine to several Cameras

As our changes for communication already considered the change from one to several cameras on one machine we only had to change the internal structure of the capturing clients. We did not add cameras to the existing clients but changed the code and structure of the clients in a way that it was possible to start several clients on one machine, because making this changes only affected the clients and not the processing unit, that still could handle the clients in the same way. The changes were mostly the adjustment to the new client names. This led to multiple independent clients on one machine, which allowed us to use the processing unit without any changes as it did not have to care where the clients are or how they are organized. Using

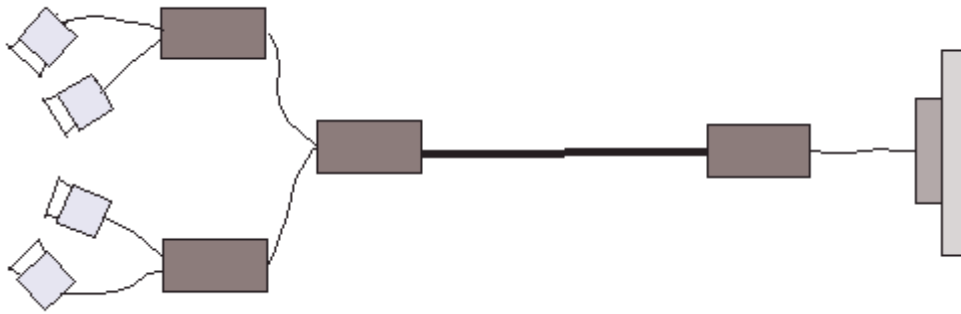


Figure 3.1: This is our four camera system design using the blue-c code. Two cameras are connected to one host. This host performs the precomputation as segmentation and contour extraction and sends the resulting images to the processing unit. Where the images from all four cameras are considered to construct a 3D model. This result is transferred to a renderer where the auto-stereoscopic rendering is performed.

this structure we were able to use two cameras on two capturing machines each, which lead us to a total of four cameras. In Figure 3.1 we can see the four camera system design. However, despite the changes we still had the same problems. We saw possible problems in the active camera control integration and we tried to get rid of it and have all the four cameras activated for textural and structural data transmission. The clients did capture and construct all the needed data but the transmission still did not work out. We were not successful with this four cameras and we could not tell if the problem was still in the transmission or in the amount of cameras or in the activation of all cameras or even in some other place. As the structure needed for the capturing and communication in the blue-c system was too complex for our purposes and it would be possible to design an easier and more efficient way for our usage, we decided, regarding the fact that there was only around one month left for our work, to build everything up from scratch to have a fundamental system at the end of this thesis on which it would be possible to build up successfully in future work.

3.3 Disparity-Map based Approach

In this chapter we will introduce our own tele-immersion approach that bases the 3D reconstruction on a graphics hardware accelerated disparity-map calculation algorithm by Yang as mentioned in Section 2.4.2.

3.3.1 Design

We split our approach into two main parts. The first part is the *ClientViewer*. This project is responsible for the capturing, rectification, disparity-map calculation and finally for the conversion from reference camera points to world coordinate points that are transmitted to the second part of the system, the *StereoRenderer*. The *StereoRenderer* project is responsible for the reception of the points calculated by the *ClientViewer* and the representation on the auto-stereoscopic display. You can see this design in Figure 3.2.

ClientViewer uses three different tools. First we developed our own camera classes that perform the capturing of raw data by the firewire cameras and reconstruct the color values by downsampling the captured raw data. The background is distracted from the object we want to reconstruct based on the segmentation used in the blue-c code. This project delivers also the

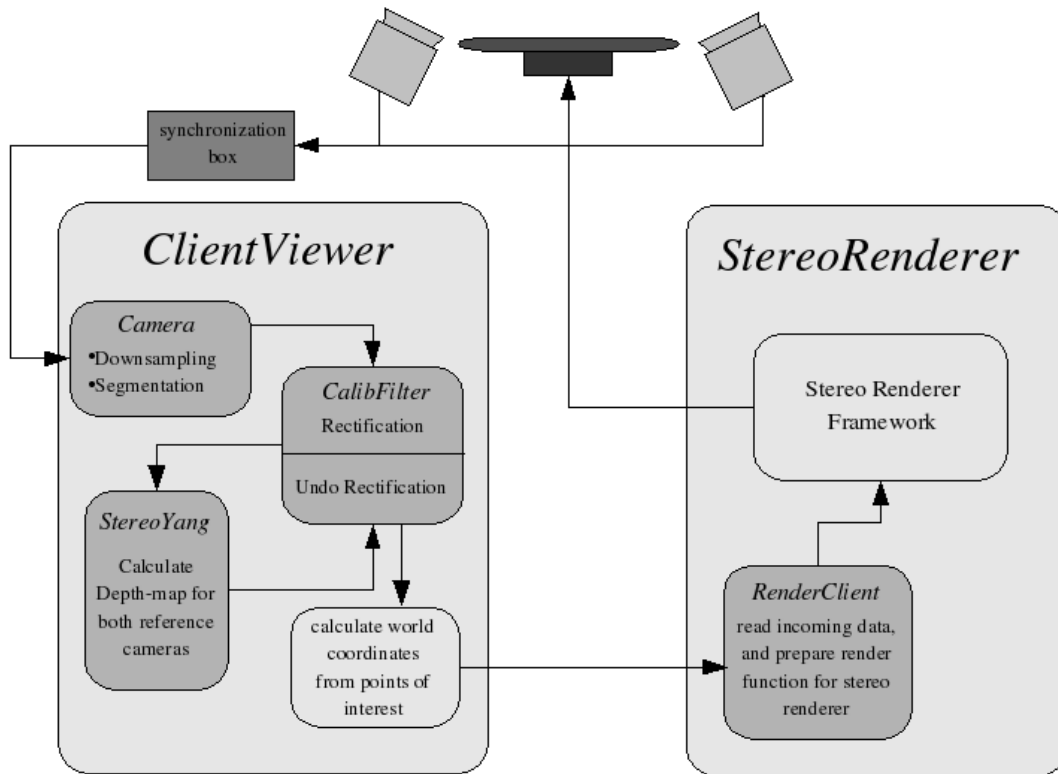


Figure 3.2: Our system consists of two main parts. The ClientViewer, responsible for capturing and 3D model construction, and the StereoRenderer responsible for rendering on the auto-stereoscopic display

ability to write images into files, which we needed for several tests and result generations. After preparing the images it rectifies them using the *CalibFilter* class, which has been derived from the *CvCalibFilter* class in OpenCV. The next step is to calculate the disparity-maps with the *StereoYang* class for the reference camera. For this calculation we used the code delivered by Yang and adjusted it to our needs. Out of these disparity-map the points of interest are calculated for the 3D world coordinates in order to submit them to the renderer. A further part of this project is the communication between the *ClientViewer* and the *StereoRenderer*. At this time we solve that problem by writing all the interesting points calculated by *ClientViewer* into a file, from which the *StereoRenderer* can retrieve them when needed. This is only a temporary solution that should be replaced later on by a socket communication.

The *StereoRenderer* consist of the rendering framework developed in Section 3.1 and a *RenderClient* that reads the points from the file and delivers a render function. This render function has to be given to the framework for auto-stereoscopic rendering.

If we set this approach in reference to the hardware the *ClientViewer* controls two firewire cameras and calculates the 3D points of interest in world coordinates for a reference camera. It is possible to use several *ClientViewers*, as bricks, because the only connection between the *StereoRenderer* and the *ClientViewer* are the transmitted 3D points in world coordinates. The *StereoRenderer* receives all the points from the *ClientViewer(s)* in world coordinates and renders them without any further knowledge about the scene on the auto-stereoscopic display.

3.3.2 Capturing

We implemented our own camera class. Because we already had our experience with the blue-c camera class we tried to adopt as much as possible from this code. It is possible to initialize a camera by providing the firewire port number of the camera and a name for the camera. According to this name we have a configuration file where the needed parameters are indicated. This file has to be saved in the configuration file directory with the name of the camera plus the ending “.param”. It has to include the following parameters for the camera:

- Brightness
- Exposure
- WhiteBalance_Blue
- WhileBalance_Red
- Gamma
- Shutter
- Gain

and the five coefficients needed for the blue-c segmentation:

- Coef1 (traditional backsegmentation threshold: -the bigger – the less foreground pixels)
- Coef2 (shadow processing: the smaller – the more detection of shadows)
- Coef3 (filter coefficient)
- Coef4 (number of times to dilate)
- Coef5 (box filter mask size)

The camera class is also responsible for the color reconstruction which is done based on code from the Coriander project [25]. As the resolution, we achieve through our cameras, is far to high for a real-time disparity-map algorithm we decided to use downsampling for color reconstruction. This algorithm is the fastest way to perform a bayer color reconstruction but it also reduces the image resolution. This reduction comes from the fact that each pixel is calculated by merging all neighboring pixels into one pixel. In our case this is not a disadvantage.

After the color reconstruction we apply the segmentation to the captured images to extract the needed foreground objects. This leads to a mask that indicates which pixels of the image are part of the foreground and which ones are not. This segmentation leads to a smaller amount of pixels that have to be considered in the following calculations. It also reduces the noise in our resulting disparity-maps as the background pixels do not interfere anymore with the foreground objects in the disparity-map calculation. In Figure 3.3 you can see a segmentation.

For several testing and result acquiring purposes we also needed the ability to write images into files. This is the reason why we also integrated a function that is able to write a PPM image file given RGB data.

3.3.3 Rectification

We use the CvCalibFilter from the cvaux library that belongs to the OpenCV project. This filter is designed to take checkerboard pictures taken by a stereo camera pair and calculate out of the extracted corners of the board the extrinsic and intrinsic parameters of the participating cameras. In Figure 3.4 we can see the detected corners in two samples of stereo camera pair images. We can decide how many checkerboard stereo images we want to feed to the algorithm. As soon as the checkerboard pictures are fed to the filter he can calculate the extrinsic and intrinsic



Figure 3.3: Background extraction with blue-c segmentation algorithm.

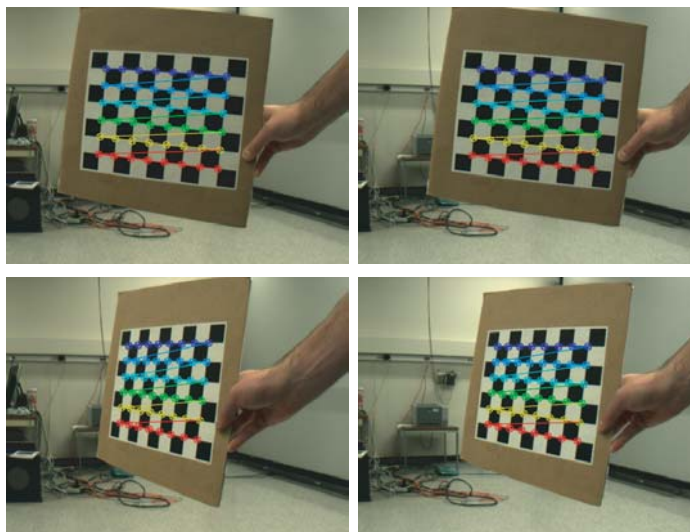


Figure 3.4: We can see the detected corners in both stereo cameras in two different frames. On the left side we have the images from the left camera and on the right side the ones from the right camera.

parameters of our cameras. The calculated parameters can be saved to a file. This means that we only have to calibrate our cameras once and we can reuse these parameters until we change the location of our cameras. But we have to mention that systems involving cameras and calibration are very unstable and the smallest movement of the cameras can cause big errors, which leads to the need of frequent calibrations. As our approach enables use to do calibration automatically and fast, this is no big disadvantage.

Nevertheless we decided to use this calibration and the associated rectification. Mostly because the rectified images looked good, the recalibration was very fast and it was easy and not time consuming to integrate it into our system. We had to change some minor things in the OpenCV library to be able to compile and run the filter, but we could find these changes in newsgroups. You can see these changes in Appendix A.3. Figure 3.5 shows a rectified image pair. Having this rectification it was possible to feed Yang's graphics hardware based disparity-map algorithm. We use the resulting disparity-map to calculate the world coordinates of the 3D points. But before we can do this we have to unrectify the disparity-map again. As there is no function in the CvCalibFilter that gives us this feature we implemented our own. We wanted to have a fast unrectification that did not need expensive calculations. This is the reason why we decided to use an approach with lookup images. These lookup images contain for each rectified

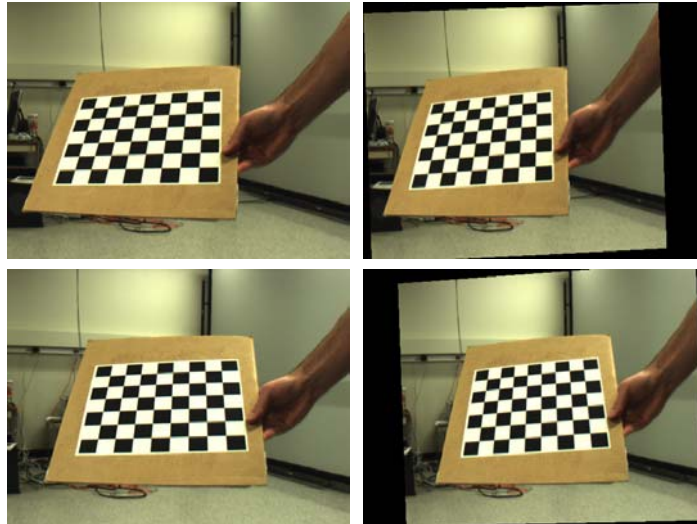


Figure 3.5: On the left side we see the original images. On the right side the rectified images. The top line corresponds to the left camera and the bottom line to the right camera.

pixel the information where it was before the rectification. To receive this lookup images we introduce two artificial images. One of them has as pixel value the row number of this pixel and the other has the column number. If we rectify these two images this results in two lookup images that have the information where each pixel was before the rectification. As we only have to calculate this images ones, this leads to a very fast unrectification. We can see a sample in Figure 3.6. As we can see, we have holes in the unrectified image. This results from the fact that

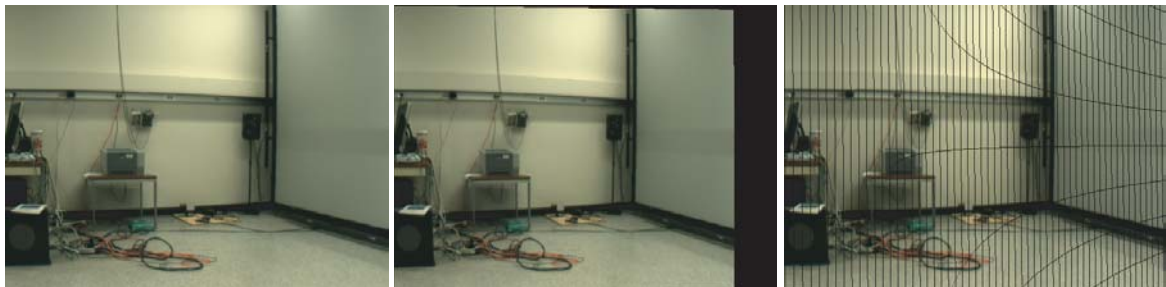


Figure 3.6: On the left we have the original images. In the middle the rectified image and on the right the unrectified image.

with the rectification we loose information and we cannot regain that information back by unrectifying. We tested this back rectification with our system and we did not have any problems as a result of the missing information in the unrectified image. This can be explained by the usage of a smoothing filter over the disparity-map before constructing the 3D model. We discuss this in Section 3.3.5.

3.3.4 Disparity-Map Calculation

The main part of our implementation is Yang's disparity-map calculation algorithm, which we introduced in Section 2.4.2. If we have the depth of the pixels in the images we can calculate their 3D camera coordinates and later on the 3D world coordinates. The independence of the resulting 3D points from the camera allows us to construct our 3D model with help of accumulated 3D points from several stereo camera pairs.

As we decided to use Yang’s algorithm we started to look for implementations and we found an implementation from Yang himself. It is provided on his homepage and can be used by reading in two rectified RGB images. It calculates then for these two images the disparity-map. It is possible to interactively change the mipmap levels and the disparity range can be given at the beginning as a console parameter. As we wanted to use the code not only for one image pair but for every frame’s stereo image pair we had to change some parts of the code. This was not the only difference, but also the fact that we do not read in the images from a file. We already have the RGB data ready and want to feed it directly to the disparity-map algorithm. This is why we introduced a new function that allowed us to feed Yang’s algorithm with our image data. In the original code the images are given to the functions as values. That is no problem, if you only load the images once and never change them again. However, we have to load the images every frame and as we have high resolution images this result in high amount of data transfer. This is why we changed the functions to take the image data as reference. This allowed us to feed the images much faster to the algorithm. In Yang’s original code they only calculate the disparity-map from one reference image, but we want to be able to calculate the both disparity-maps of each reference camera to acquire more depth data and be able to improve our 3D model by having a denser point cloud. This is an advantage in places where the information from one camera is not dense enough. But this also leads to the double amount of points from which most are duplicates. In future work it should be considered to filter out all the double points and therefore optimize the amount of points. To do the disparity-map calculation for both cameras without having to load the images twice we had to adjust Yang’s code again. All these changes worked out but they are certainly not as efficient as they have the potential to be. We only changed the code already available for our purposes, which lead to some unpleasant hacks. In future work the reimplementaion of this algorithm should be considered.

Nevertheless we achieved the integration of the code into our system and receive very fast good disparity-maps. To be precise, we receive disparity maps with a value range from 0 to 255. The values resulting from the calculation with a certain disparity range are scaled to this range. As the depth can be calculate from the disparity by dividing one through the scaled disparity d it is no problem to get the depth values. The needed calculation is shown in the following equation:

$$depth = \frac{1}{d/255} \quad (3.1)$$

We are also able to interactively change the mipmap levels and the disparity range we work with. This leads to a more universal system that can be adjusted to different depth ranges. You can see different used parameters for different depth ranges and the results in Section 4.3. By using higher mipmap levels the diparity-map becomes more blurry, which results in disparity values corresponding to pixels that are not part of the foreground and hence not interesting for us. This is the reason why we apply our segmentation mask mentioned in Section 3.3.2 on our disparity-map again. In Figure 3.7 we can see the different results with different mipmap levels.

Having the depths of the points and knowing the location of our cameras we can calculate the 3D points in the world coordinate system.

3.3.5 Calculation of Point Coordinates

Thanks to the calibration, described in Section 3.3.3, and the depth values for the reference cameras we can calculate the 3D points in world coordinates. To do this we take a closer look at the intrinsic and extrinsic parameters of the camera calibration.



Figure 3.7: In the top row we can see the segmented object. In the middle row we can see from left to right the disparity-map with 0,1,2 and 4 mipmap levels. The bottom image represents the disparity-map with 4 mipmap levels cut out by the segmentation mask.

If we consider that we have an ideal perspective camera then it is possible to transform a point P_0 in the world coordinates to a point P in the camera coordinate system by,

$$P = RP_0 + T \quad (3.2)$$

where R is the rotation and T is the translation given by the extrinsic parameters. Knowing P we can calculate the pixel coordinates in the image plane.

$$p = \begin{bmatrix} x \\ y \end{bmatrix} = \frac{f}{Z} \begin{bmatrix} X \\ Y \end{bmatrix} \quad (3.3)$$

If $P = [X, Y, Z]^T$, f is the focal length of our camera and p is the point in the camera plane. In the following we will assume that f is 1 for our cameras. These equations are true for ideal perspective cameras. But a camera like that does not exist. This is the reason why we also have to consider the intrinsic parameters of our cameras. The intrinsic matrix converts our camera plane coordinates into image plane coordinates, regarding the pixel size and the actual principal point in the camera plane. You can see this transformation in Figure 3.8.

This transformation can be expressed as,

$$p' = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = K_s p = \begin{bmatrix} s_x & s_\theta & o_x \\ 0 & s_y & o_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \quad (3.4)$$

Where p' is the resulting point on the image plane and K_s is the transformation matrix formed by the intrinsic values.

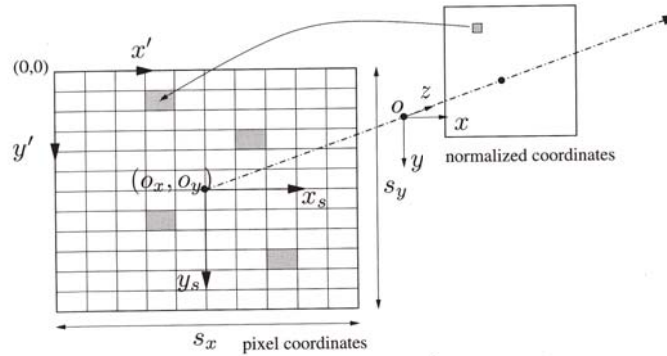


Figure 3.8: We can see in this figure the transformation from camera coordinates to image coordinates, which is done by intrinsic parameters.

- o_x : x-coordinate of the principal point in pixels
- o_y : y-coordinate of the principal point in pixels
- s_x : size of unit length in horizontal pixels
- s_y : size of unit length in vertical pixels
- s_θ : skew of the pixel, often close to zero

We now know how to calculate the pixel of an image corresponding to a 3D world point considering the extrinsic and intrinsic parameters. It is obvious that we lose information with this transformation since in the image plane we do not have any knowledge of the depth of the point. This is the reason why we have to calculate the depth values to be able to reconstruct the 3D point out of our images. This means that for our back propagation we have to follow the following steps. First we have to transform the image plane pixel p' to a camera plane pixel p .

$$p = K_s^{-1} p' \quad (3.5)$$

We do this by multiplying the inverse from K_s with p' . Now we have to expand our camera image point by the depth value z calculated in Equation 3.1.

$$P = \begin{bmatrix} x_{tmp} \times z \\ y_{tmp} \times z \\ z \end{bmatrix} \quad (3.6)$$

Where P is now the 3D point in the reference camera coordinate system. As we want to have the world coordinate values for P we have to undo Equation 3.2 as well. This is done by the following step.

$$P_0 = R^{-1}(p - T) \quad (3.7)$$

Where the resulting point P_0 is in the world coordinates and R^{-1} is the inverse of R the Rotation from the extrinsic parameters and T is the corresponding translation.

Knowing now how to calculate the world coordinates for every point of interest in the camera images we can create the 3D model constructing a point cloud. By doing this we smooth out the depth values with a Gaussian filter to minimize the influence of outliers in our disparity-map to our 3D object. This Gaussian filter also helps us to minimize the influence of the missing infor-

mation resulting from the unrectification, mentioned in Section 3.3.3. We use a Gaussian filter with the mask seen in Figure 3.9.

1	2	1
2	4	2
1	2	1

Figure 3.9: Gaussian filter mask with the weights for the corresponding neighbors.

The transfer of the data is currently done by writing the resulting points to a file and reading them out when needed. This is not a good solution for a real-time application, but it allowed us to advance faster to the main part of 3D model construction and representation.

The renderer displays the resulting 3D points of the objects using the `GL_POINTS` primitives from OpenGL on the auto-stereoscopic display, as described in Section 3.1. In future work the rendering should be extended to a splatting based rendering technique as used in the blue-c system showed in Section 2.3.3.

4

Results and Conclusion

In this chapter we will present our results and discuss the different parameters and the possible approaches to change or improve them. All the results are obtained on a Intel Pentium 4, 3.2 GHz Processor with 1 GB RAM and a nVidia GeForce 6800 PCIExpress graphic card with 256MB RAM. At the end of the chapter we will present our conclusion.

4.1 Auto-Stereoscopic Rendering

In this chapter we look at the time needed to render a large amount of GL_POINTS on our auto-stereoscopic display set in comparison with the rendering time on a standard display. As mentioned in Section 3.1 we have to render nine images and combine them with each other for

Comparison between Rendering 2D/3D

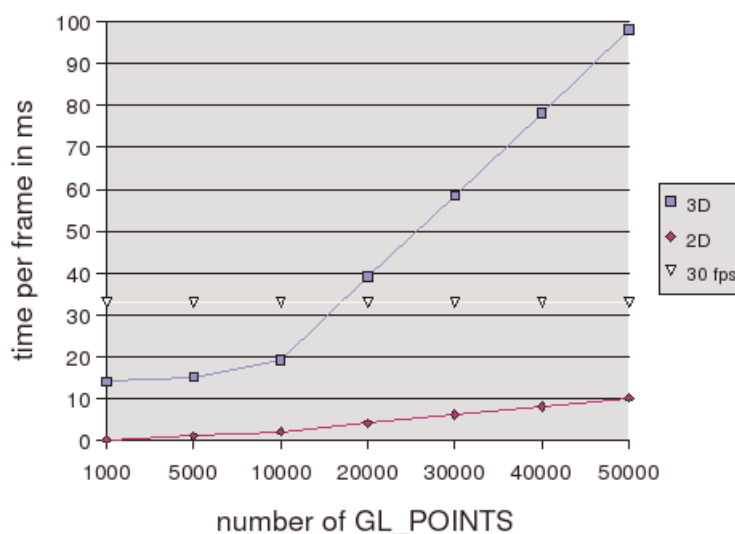


Figure 4.1: Comparison of the rendering time for a large number of points for a standard display (2D) and our auto-stereoscopic display (3D).

our auto-stereoscopic display. We can see this comparison in Figure 4.1. We can also see in this plot that the combination of the nine images takes around 14 ms and is for all the numbers of `GL_POINTS` the same. This is the reason why the ratio between time measures converges to a factor of nine with higher numbers of `GL_POINTS`. We can also see that the rendering on the auto-stereoscopic display passes the 30 frames per second threshold with around 18500 `GL_POINTS`. In our system we have to count on around 38000 `GL_POINTS` for one disparity-map if we have a scene similar to the one in Figure 4.7 with a mipmap level of four. If we consider one disparity-map we have a rendering time per frame of around 75 ms which leads to around 13.5 frames per second. If we consider the 4.5 frames per second that can be captured and processed by the *ClientViewer*, discussed in Section 4.5, we still do not slow down the system.

4.2 Calibration and Rectification

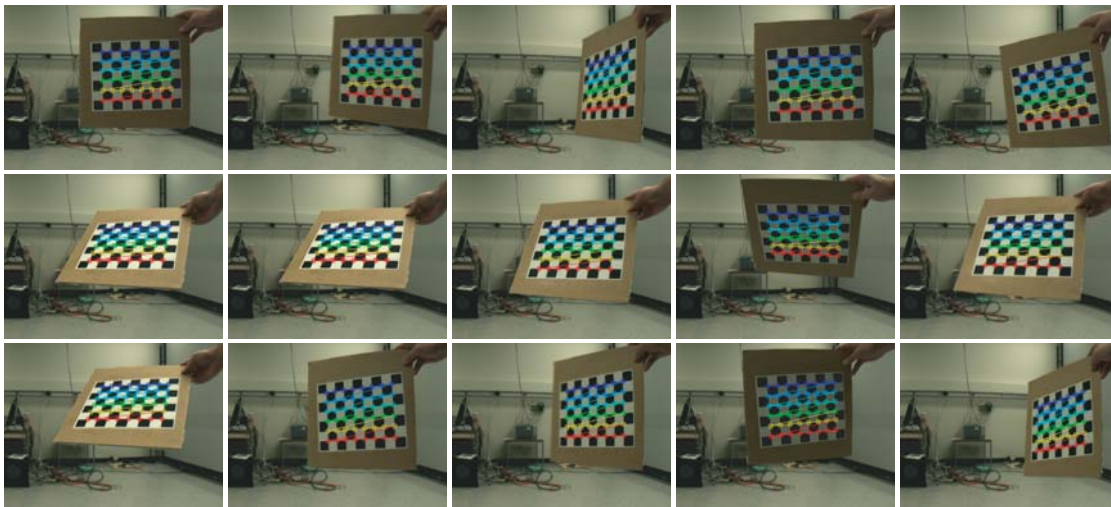


Figure 4.2: 15 reference images of the checkerboard taken by the right camera. The corners of the checkerboard are extracted and used for the calibration.

Before we can calculate the depths of the different points of interest we have to prepare the images for our algorithm. This means that we have to rectify the reference images captured by our stereo camera pair. Before we are able to do so, we have to calibrate our cameras. As explained in Section 3.3.3 we capture images for that from a moving checkerboard and extract the corners of this checkerboard automatically. You can see 15 reference images from the right camera and the extracted corners for one of our test runs in Figure 4.2. We tried to have as many different positions of the checkerboard as possible. It is important that we have, if possible, samples for each possible arrangement of the checkerboard. If the samples are all similar like in Figure 4.3 the probability of bad calibration and bad rectification is higher, even if with the previous example we still had good looking results, as seen in Figure 4.5. The result from the calibration from the images in Figure 4.2 is seen in Figure 4.4.

The calibration and rectification by the `CvCalibFilter` seems to be robust. However, we do not want to focus only on optical correctness, but we also want to know how accurate our rectifications are in reference to the pixel size. Unfortunately there is no way to find out this error directly in `CvCalibFilter`. Therefore, we had a closer look at the calibration code to find out how we could determine the accuracy of the calibration. In this code there is an optimization loop that either stops after 30 iterations or when the resulting change of the optimization is smaller than a threshold. The change is the difference of the positions of the reprojected corners in two

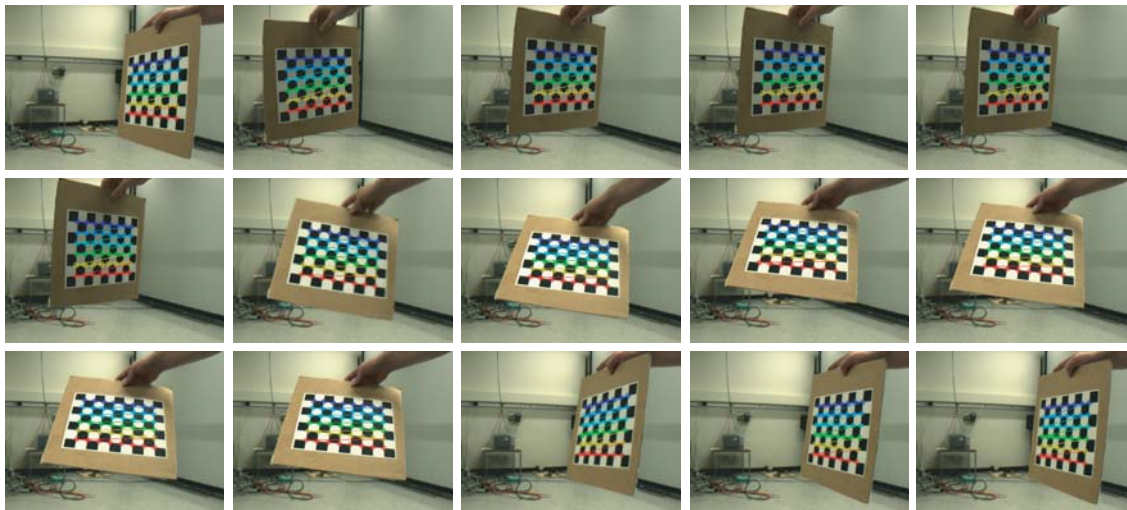


Figure 4.3: This 15 reference images are similar and do not cover the hole position space of a checkerboard.

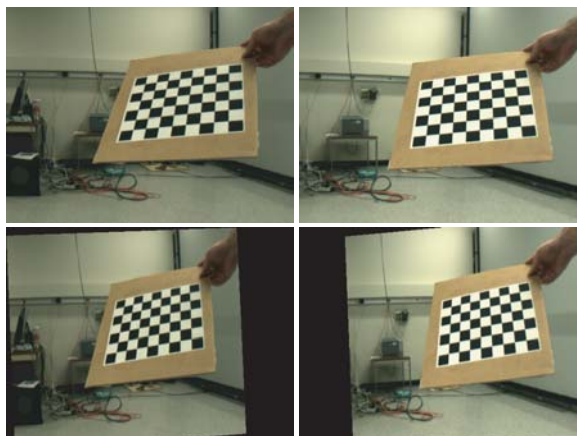


Figure 4.4: Rectification resulting from calibration from reference frames in Figure 4.2.

following iterations of the optimization. Ideally the reprojection of the corners should end up exactly on the corners of the reference image. In most cases the loop was left because the change was smaller than `FLT_EPSILON`, which has the value of $1,1921 \cdot e^{-7}$. This means that the difference is much smaller than a pixel. In the example shown in Figure 4.5 the loop was left after 30 iterations with a final change of 0,0102 which indicates that the optimization did not find a good solution, even if the first optical impression made it seem so.

4.3 Disparity-Map Results

In this chapter we want to discuss the disparity-map results from our calculation with Yang’s graphics hardware based approach. First we want to have a look at the differing results with different mipmap levels. As mentioned in Section 3.3.4 we can choose during our calculation to work with one, tow or four mipmap levels. In Figure 4.6 we can see the results with all three choices and a disparity range of 64 with cameras standing in a middle range, distance of around 13 cm, from each other. As we already stated, the higher the mipmap level the more blurry but stable the results are. We can see that the lower mipmap levels have problems with larger uniform colored patches in the images. This is a well-known problem. However we can see that the

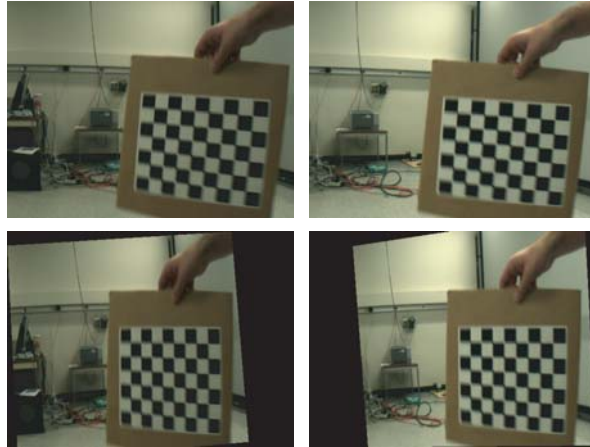


Figure 4.5: Rectification resulting from calibration from reference frames in Figure 4.3.

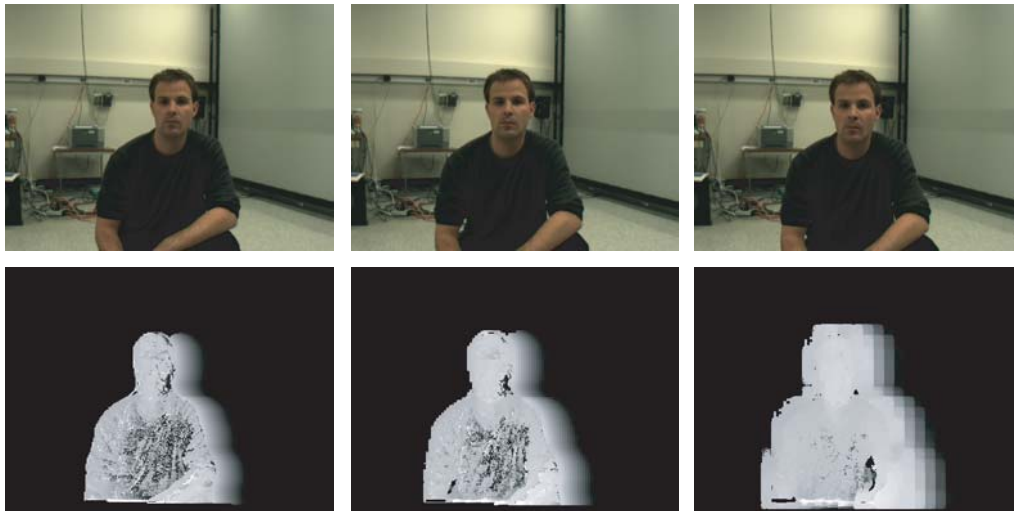


Figure 4.6: Rectified disparity-map results for mipmap level 1,2 or 4 with the corresponding reference image.

level four results can handle this problem pretty well sacrificing the ability to detect thin attributes of the object. This trade-off has to be considered when one decides with how many mipmap levels the disparity-map should be calculated. You can see in Section 4.5 the measured times calculating the one and the four mipmap level results. In the following test we will calculate all our disparity-maps with level four.

We will now have a look at the relationship between the used disparity range, the distance of the cameras to each other and the distance of the foreground object in relation to the cameras. We will see that the higher the disparity range the better the results for near objects and cameras that are further away from each other, but the computational costs increase as well.

In Figure 4.7 we can see the results for the cameras standing near one another - 8.5 cm distance measured from the middle of the cameras, and a disparity range of 32. We can see that for a nearby object, in the top of the image, the disparity-map is not robust and reliable. We can explain this by the fact that most of the corresponding pixels have a disparity of more than 32 pixels. You can see this in Figure 4.8 on the left side. The farther away the objects are the less the disparities of the pixels and we get better results. This means that for near objects we need a higher disparity range. We can see the results with the same arrangement but with a disparity range of 64 pixels in Figure 4.9. The result for a nearby object is now much better, but we can

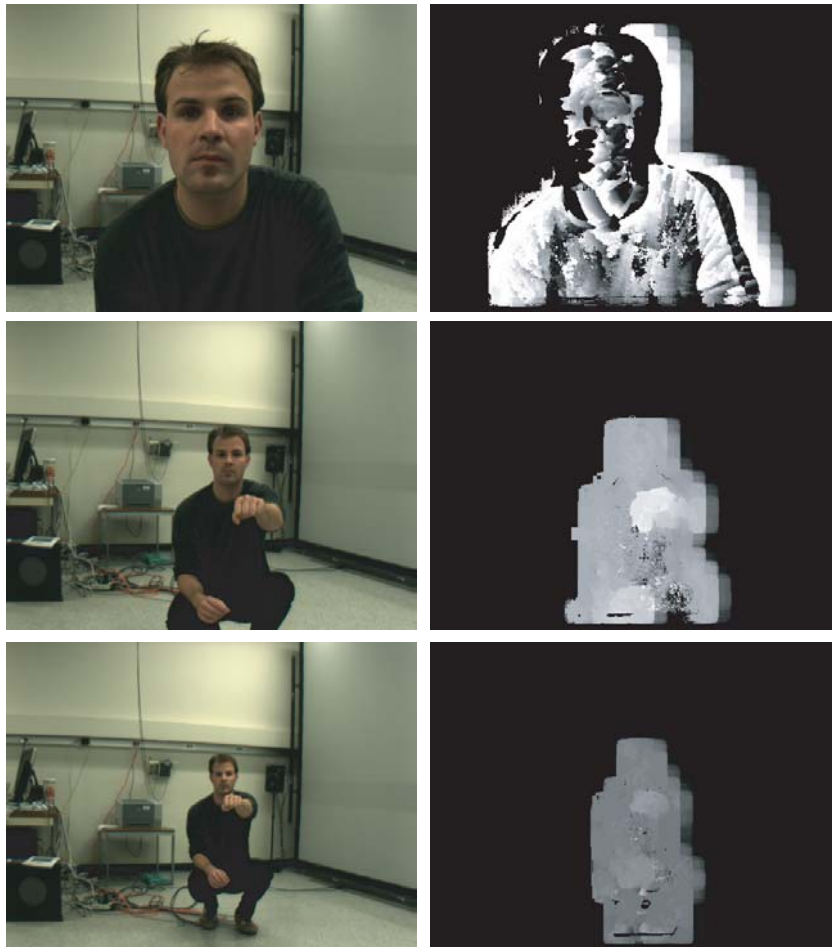


Figure 4.7: Disparity-map of object in different distances to the cameras, that are standing near to each other and disparity range 32.



Figure 4.8: The basic rectified images for cameras near to each other with an near object, on the left and a far object in the right.

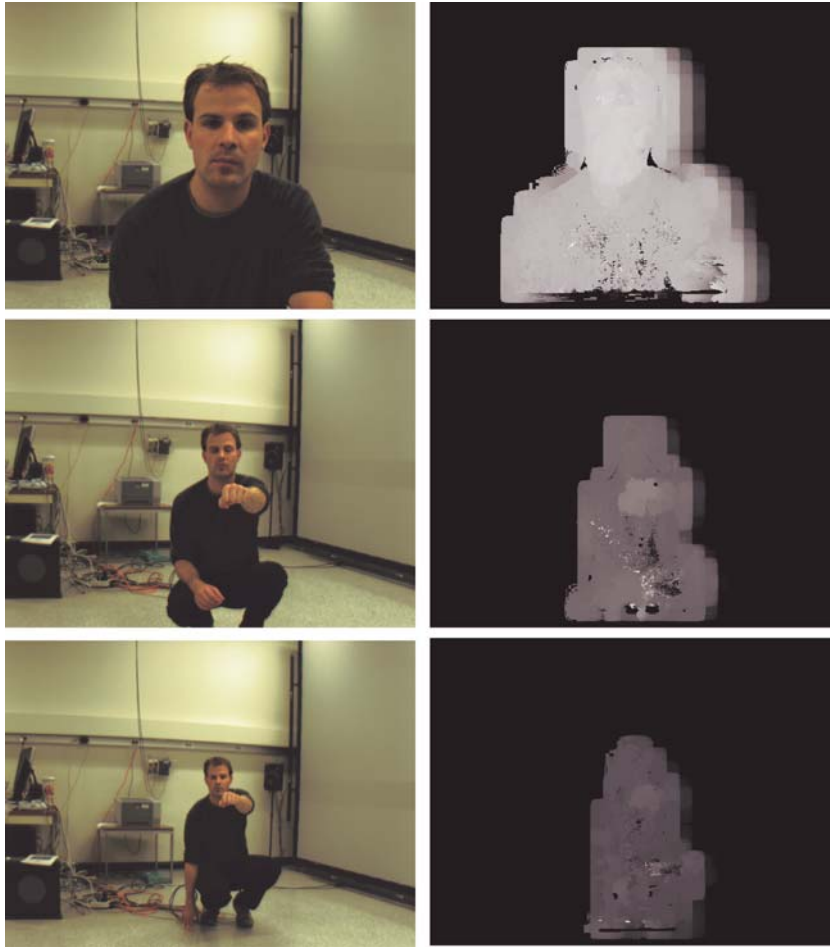


Figure 4.9: In these images we can see the results for near cameras and a disparity range of 64 pixels.

see that the far away object is now considered to be very far away. We can explain this with the fact that the disparity values of the pixels are scaled to range of 0 to 255. The bigger our disparity range is, the less influence has on pixel of disparity, on the disparity-value. This shows that the disparity range is in general only optimal for a certain object distance and the corresponding camera distance. We can see in Figure 4.10 the results with the cameras standing far, approximately 35 cm, from each other. We can see that the results are more fuzzy but in the middle and far object range we can still obtain a reasonable disparity-map. If we increase the disparity range to 128, we can still obtain good results for near objects as seen in Figure 4.11. The pixel values that look like shadows are a side effect of the large distance of cameras and the high disparity range. But we do not have to worry about them because in our system we only use the disparity-map values that are not segmented out by our segmentation mask. We can see in the near object some holes in the area of the nose, which can be explained by the fact, that these pixels are not seen by both cameras because of the high distance between them and the different viewing angles. To get rid of this holes we have to introduce more stereo camera bricks that are able to calculate the depth of these points.

4.4 Stereo Rendering

In the previous chapters we saw all the necessary components that lead to rendering on our auto-stereoscopic display. After obtaining the rectification and the depths we can calculate the

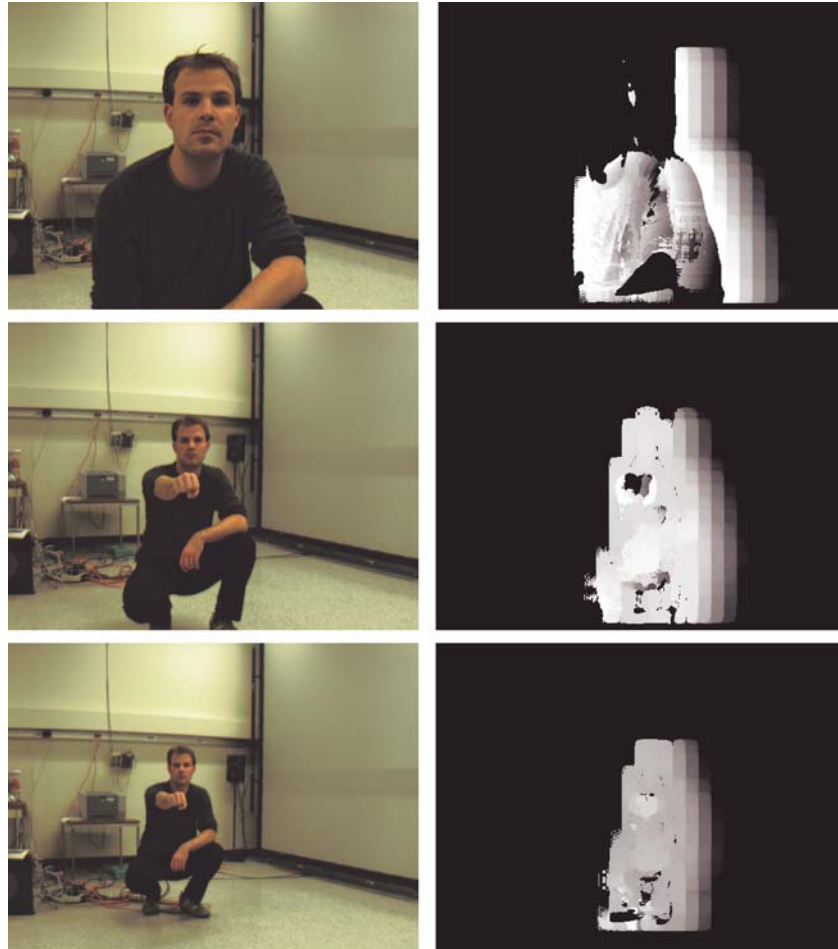


Figure 4.10: In these images we can see the results for cameras that are far away from each other and a disparity range of 64 pixels.

world coordinates of all the points of interest and render them. In this stage of the project we decided to render them as `GL_POINTS`. In Figure 4.12 we can see the results with `GL_POINTS` of size 1.0. On the top row of the figure we can see the resulting images if the user looks directly into the cameras and the virtual cameras are aligned in a way to construct a sight line from user to the stereo camera pair. On the bottom we can see the results when the user looks directly into the display and the virtual cameras are aligned in a way that a sight line between user and display is produced. This leads to possible eye-contact between two users. On the left we see the image of the camera in the middle of the virtual cameras and on the right we see the resulting interzigged image for the auto-stereoscopic display. We can see on the bottom row that there are a lot of holes around the chin and the parts under the chin. This results from the fact that our cameras are situated on top of the display and look down on the user. You can see this arrangement in Figure 1.1. In Figure 4.13 you can see the segmented views from the stereo camera pair. As we only have one stereo camera pair these points can not be reconstructed because we do not have any information about them. This could be avoided by adding more camera pairs. On the top row in Figure 4.12 we can see the reconstructed user looking into the cameras. Here we have holes on the left side of the face. These holes arise out of the same reason. The left camera cannot see the ear and its surroundings and this is why we do not have enough information to reconstruct parts of the face dense enough. There are also holes that result from the interpolation of the real camera positions and the virtual camera positions. These holes can be prevented by adding more points. We could do so by adding more camera pairs. The holes can also be pre-

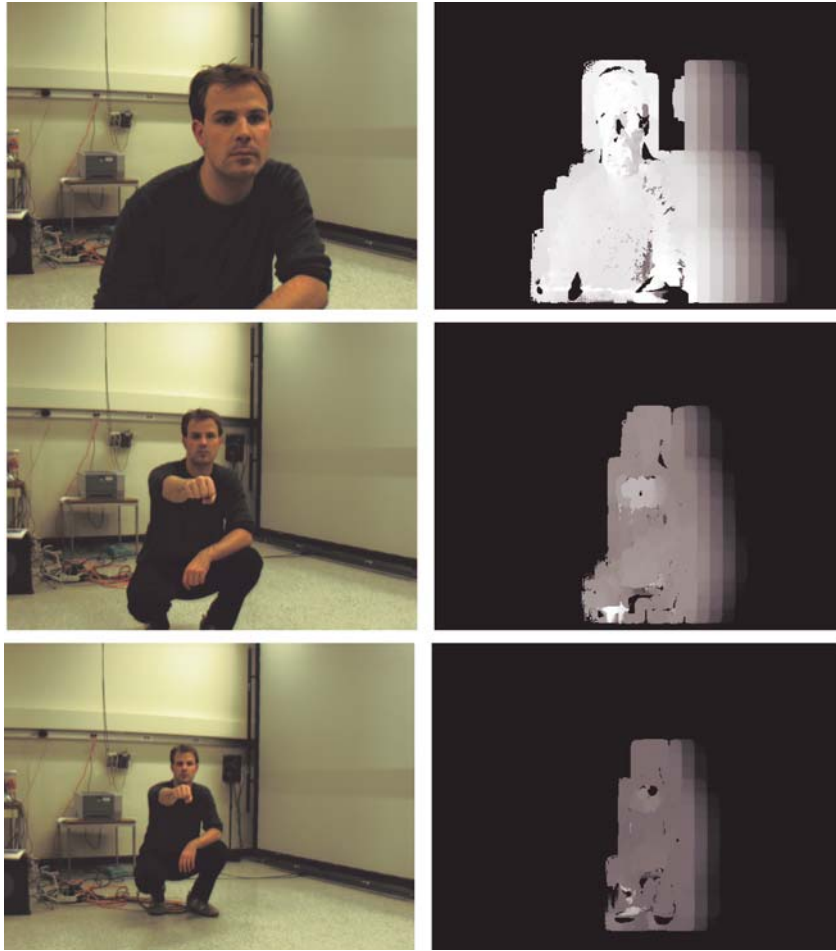


Figure 4.11: Results having the cameras standing far from each other with a disparity range of 128.



Figure 4.12: Rendering results with `GL_POINTS` of size 1.0. On the left side we can see the view of the virtual camera in the middle. On the right side the inter-zigged image for the auto-stereoscopic display. On top we look directly into our stereo camera pair and have the virtual cameras in the same spot. On the bottom we look directly into the display and the virtual cameras are aligned in a manner that a sight line to the conversational partner is built up.

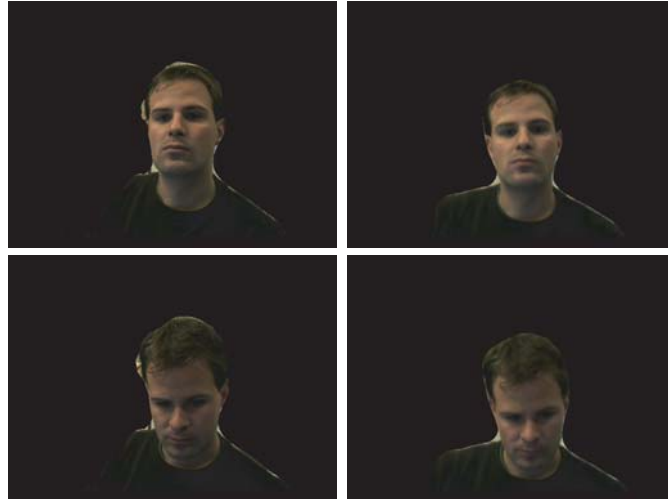


Figure 4.13: Here you can see the base images that are fed to the algorithm to calculate the depth-values. On the top we can see the user looking into the cameras and on the bottom looking directly into the display.

vented by making the points bigger. In Figure 4.14 you can see the results with `GL_POINTS` of size 1.5. We can see that the number of holes is decreased, however as a side effect the whole image is more blurry. The face contours are not so smooth anymore and that leads to small distortions of the mouth and eyes. We can also see in the bottom row images that the chin now looks like a part of the black sweater. This results out of outliers of the black sweater, which were moved to these points by the interpolation to this viewpoint. On the auto-stereoscopic display you can see that they are nearer to the user than the rest of the face. We do not see them with point size 1.0. In general we can say that the bigger the points are the more influence the outliers have on the representation. Small points in wrong places are filtered out by the human brain, but if they achieve a certain size they are considered to be real and needed. It would be better to render the `GL_POINTS` with sizes corresponding to the density of the neighborhood. This is why a splatt based rendering approach like the one introduced in Section 2.3.3 should be used in future work.

4.5 Overall Capturing Speed

In this chapter we will examine the amount of time used for the subtasks. We decided to separate the overall work flow in the *ClientViewer* into the following subtasks:

- Capturing (images acquisition by camera and color reconstruction by downsampling)
- Segmentation (background extraction for both camera images)
- Rectification (for both camera images)
- Disparity-map calculation
- World coordinates (construction of the 3D object, calculating the world coordinates of the relevant points)
- Transmitting points (writing all the points of interest into a file)

We performed the time measuring for the following approaches:

- Disparity-map extraction with mipmap level four
- Disparity-map extraction with mipmap level one



Figure 4.14: Rendering results with `GL_POINTS` of size 1.5. On the left side we can see the view of the virtual camera in the middle. On the right side the inter-zigged image for the auto-stereoscopic display. On top we look directly into our stereo camera pair and have the virtual cameras in the same spot. On the bottom we look directly into the display and the virtual cameras are aligned in a manner that a sight line to the conversational partner is built up.

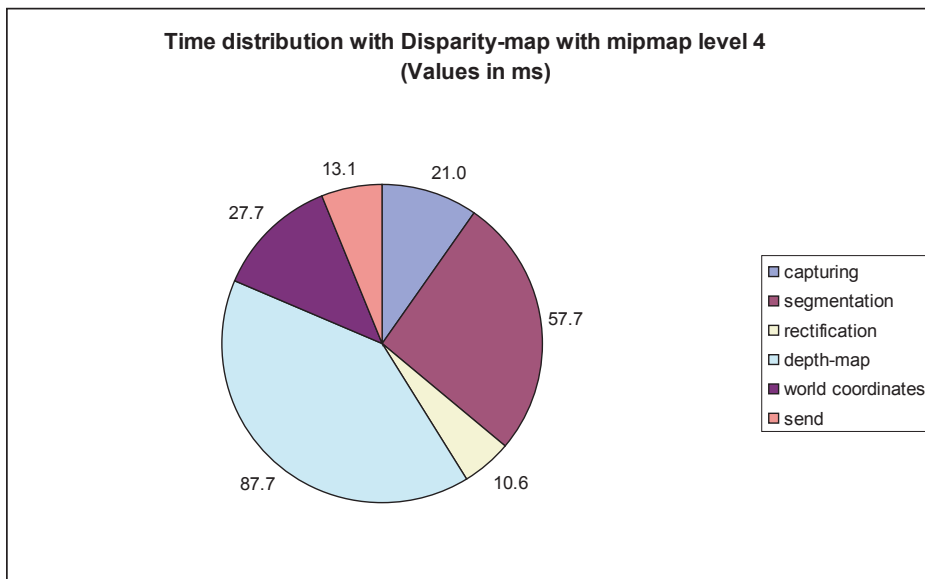


Figure 4.15: Measured values for mipmap level four.

The difference in these two approaches consists of the different mipmap levels. This means that we do the disparity calculation for each mipmap level once. In the first approach this leads to four passes of this calculation and to one pass in the approach with mipmap level one. In Figure 4.15 we can see the measured time values for mipmap level four. We can see that as expected the disparity-map calculation takes the longest. The entire process from capturing to the end of sending the world coordinate points to the *StereoRenderer* takes 218.4 ms which leads to 4.5 frames per second working with images of a resolution of 512x384 pixels. Each of these disparity-maps delivers in average 37964 3D points.

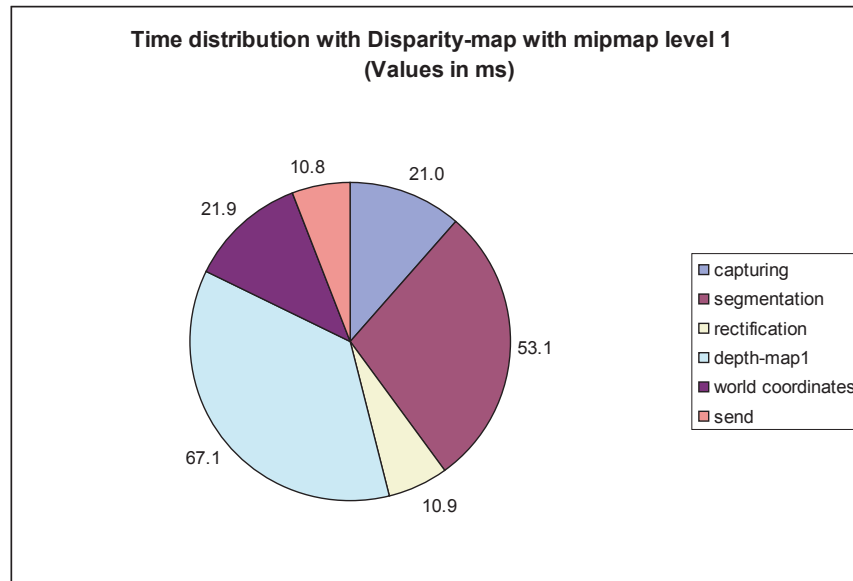


Figure 4.16: Measured values for mipmap level one.

Now we want to examine the same measurements but only with mipmap level one. In Figure 4.16 we can see the results for one disparity-map. The time used for disparity-map calculation is now 67.1 ms compared to 87.7ms with mipmap level four. This leads to an accumulated frame time on the *ClientViewer* of 184.8 ms which leads to 5.4 frames per second with a resolution of 512x384 pixels per reference image. An object of in average 31226 3D points results from one of these disparity-maps. We can say that the disparity-map calculation with mipmap level one is faster but it also delivers more noisy disparity-map images than the level four approach as you can see in Section 4.3

Now that we know all the needed times for the different subtasks we can have a look at the delay in between the capturing of the stereo images and the representation on the auto-stereoscopic display. For this examination we communicate between the *ClientViewer* and the *StereoRenderer* with a file. We know that this leads to a higher delay but at the present state of the project no other communication is available yet. We will have a look at the delay with a mipmap level four approach. As found out before the whole calculation on the *ClientViewer* needs 218.4ms for one frame. The rendering on the auto-stereoscopic display for an object with 37964 3D points needs 72 ms. Reading out the 3D data from the file takes in average 10 ms. This means that we need 82 ms to render one frame. This adds up to an expected delay of around 300ms. In fact we have a delay of around a second. Which can be explained by the fact that the rendering and the disparity-map calculation are made in the GPU using the same graphic card and have to share the resources.

4.6 Conclusion

We designed in our project a tele-immersion system that is able to reconstruct 3D objects out of one or several bricks containing a stereo camera pair. To do this we considered different image based 3D reconstruction approaches and included a graphics hardware based disparity-map calculation algorithm into our system. Out of this disparity-map we reconstruct 3D models in world coordinates and render them on an auto-stereoscopic display. The disparity-map is calculated from two images captured by a stereo camera pair. These images are first segmented, to extract the background and then rectified. Thanks to the background extraction we can concentrate on the pixels of the foreground object and prevent noise in the disparity-map caused by the

influence of background pixels. The rectification enables us to use the graphics hardware based disparity-map calculation efficiently by reducing the search range for corresponding pixels from two dimensions to one dimension. This results in a system that allows us to render the user in a way that the conversational partner can see him as if he were looking in his direction despite the fact that the cameras are not directly aligned to this sight-line.

One of the goals in the master-thesis description was the integration of a 3D video user representation with existing visualization applications. We decided to leave this task to future work as we wanted to concentrate on the design and implementation of a reliable tele-immersion system on which these kind of tasks could be integrated later on without having to rebuilt the whole system.

We used for our system various pre-existing implementations as the CvCalibFilter and Yang's disparity-map calculation. The integration of this code in our system led to different adjustments that could probably be solved more efficiently by implementing them by considering peculiarities of our system and normalizing the data structures.

Besides this the calibration of the cameras should be considered to be implemented in a more general way. The CvCalibrationFilter is able to calibrate stereo camera pairs but could have problems calibrating different stereo camera pairs to one reference point. This fact and the lack of integrated error evaluation argues for another approach if we want to expand the system by several bricks safely.

Our system renders objects resulting from 3D points with GL_POINTS. As these points cannot adjust smoothly to the object contours and because they cannot adjust to different densities of points in the object, which can result into holes or blurry results, we suggest that in future work the rendering is done by a splat based rendering algorithm which allows to render the objects in high quality, adjusting to the density and the arrangement of the 3D points.

The communication between the different parts of our system is at the time done by writing the world coordinate 3D points of interest into a file and reading them out when needed. This is not an approach that meets our real-time efforts. This is the reason why we suggest that in future work the communication should be reimplemented considering the approach used in the blue-c system mentioned in Section 2.3.4. This would lead to less data transfer and to a smaller delay from capturing to rendering.

As already stated in Section 4.5 our delay is at the time to large for a reliable tele-immersion system. This is why we suggest to parallelize as many subtasks as possible. We already looked at this problem and would suggest to try an approach similar to the one shown in Figure 4.17. The capturing and the preprocessing of the .

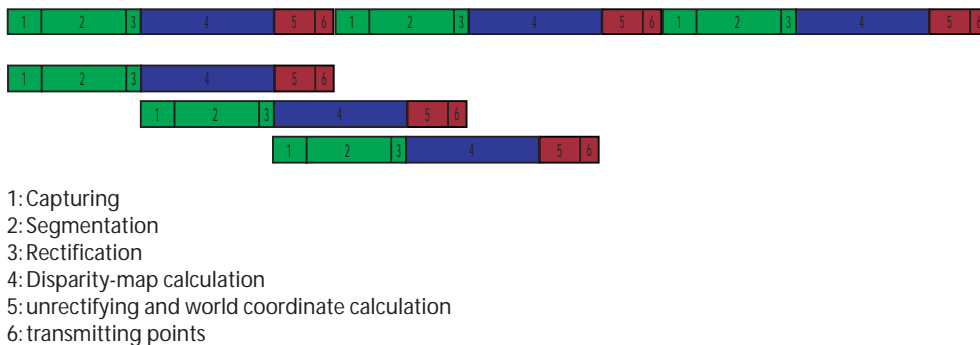


Figure 4.17: Segmentation of the subtask in the ClientViewer to three units which could be parallelized to improve the speed and delay of our system.

images could built one unit. The second unit could consist of the disparity-map calculation and the third one by the world coordinate calculation and the transmission of the 3D points. Parallelizing these units could lead, considering the fact that unit one works on the processor and unit two on the graphics processor, to an acceleration of our approach and a reduction of the delay

4.7 Acknowledgments

I want to thank Prof. M. Gross and Prof. O. Staadt for giving me the opportunity to write my master thesis at UC Davis and assisting me during my stay. I also want to thank Dr. S. Würmlin for helping and tutoring me from ETH in Zürich. Further thanks go to the IDAV members for integrating me into their group and helping me out with my problems. Without my proof-readers Jilan and Farhad Badri I would not have been able to accomplish this report, thanks. Last but not least I want to thank my parents for helping me finance my stay in California and encouraging me during my entire studies and my friends for helping me through all the not so successful and nice times during the last few years.

5

Bibliography

- [1] Gross Markus, Würmlin Stephan, Naef Martin, Lamboray Edouard, Spagno Christian, Kunz Andreas, Koller-Meier Esther, Svoboda Tomas, Van Gool Luc, Lang Silke, Strehlke Kai, Vande Moere Andrew, Staadt Oliver. "blue-c: A Spatially Immersive Display and 3D Video Portal for Telepresence". Proceedings of ACM SIGGRAPH 2003, pp. 819-827, 2003.
- [2] Matusik, W., Buehler, C., Raskar, R., Gortler, S. J. and McMillan, L. Image-based visual hulls. In Proceedings of SIGGRAPH 2000, pages 369-374. ACM Press / ACM SIGGRAPH / New York, 2000.
- [3] Matusik, W., Buehler, C. and McMillan, L. Polyhedral visual hulls for real-time rendering. Proceedings of Twelfth Eurographics Workshop on Rendering, pages 115-125. Springer, 2001.
- [4] S. Würmlin, E. Lamboray, M. Waschbüch, M. Gross. Dynamic Point Samples for Free-Viewpoint Video. Proceedings of the Picture Coding Symposium 2004, San Francisco, December 15-17, 2004.
- [6] S. Würmlin, E. Lamboray, O. G. Staadt, M. H. Gross. 3D Video Recorder. Proceedings of Pacific Graphics '02, IEEE Computer Society Press, pp. 325-334, 2002.
- [7] M. Zwicker, H. Pfister, J. Van Baar, M. Gross. EWA Splatting. IEEE Transactions on Visualization and Computer Graphics, July-September 2002, Volume 8, Number 3, pp. 223-238, 2002.
- [8] S. Würmlin, E. Lamboray, M. Gross. 3D Video Fragments: Dynamic Point Samples for Real-time Free-Viewpoint Video. Computers & Graphics 28 (1), pp. 3-14, 2004.
- [9] SCHMIDT, D. Real-time CORBA with TAO (The ACE ORB). <http://www.cs.wustl.edu/~schmidt/TAO.html>.
- [10] M. Waschbüsch, S. Würmlin, D. Cotting, F. Sadlo, M. Gross. Scalable 3D Video of Dynamic Scenes. The Visual Computer, p. 629-638, 2005.
- [11] U. Dhond and J. Aggrawal. Structure from stereo: a review. IEEE Transactions on Systems, Man, and Cybernetics, 19(6):14891510, 1989.
- [12] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A Stereo Engine for Video-rate Dense Depth Mapping and Its New Applications. Proceedings of Conference on Computer Vision and Pattern Recognition , pages 196–202, June 1996.

- [13] JohnWoodfill and Brian Von Herzen. Real-time stereo vision on the PARTS reconfigurable computer. IEEE Symposium on FPGAs for Custom Computing Machines, pages 201–210, Los Alamitos, CA, 1997.
- [14] K. Konolige. Small Vision Systems: Hardware and Implementation. In Proceedings of the 8th International Symposium in Robotic Research , page 203-212. Springer-Verlag, 1998.
- [15] J. Mulligan, V. Isler, and K. Daniilidis. Trinocular stereo: A new algorithm and its evaluation. International Journal of Computer Vision, Special Issue on Multi-baseline Stereo , 2002.
- [16] R. Yang, G. Welch, and G. Bisop. Real-Time Consensus-Based Scene Reconstruction Using Commodity Graphics Hardware. Proceedings of Pacific Graphics 2002.
- [17] Ruigang Yang, Marc Pollefeys. A versatile stereo implementation on commodity graphics hardware. Real-Time Imaging 11(1): 7-18, 2005.
- [18] Cornelis, N., Van Gool, L.J. Real-Time Connectivity Constrained Depth Map Computation Using Programmable Graphics Hardware. CVPR05(I: 1099-1104), 2005.
- [19] C. Everitt. Projective Texture Mapping. <http://developer.nvidia.com/attach/6549>.
- [20] Pollefeys M. , Koch R. , and Van Gool L. . A Simple and Efficient Rectification Method for General Motion. Proceedings of International Conference on Computer Vision (ICCV), pages 496–501, Corfu, Greece, 1999.
- [21] Oram, D. Rectification for any Epipolar Geometry. 12th British Machine Vision Conference (BMVC 2001), September 2001.
- [22] Mei C. Camera Calibration Toolbox for Matlab. http://www.vision.caltech.edu/bouguetj/calib_doc/.
- [23] OpenCV. <http://sourceforge.net/projects/opencvlibrary/>.
- [24] StereoGraphics: The SynthaGram Handbook. February 2003 StereoGraphics Corporation.
- [25] Coriander Project. <http://damien.douxchamps.net/ieee1394/coriander/index.php>.
- [26] National Tele-Immersion Initiative. <http://www.advanced.org/tele-immersion/>
- [27] Baker, H. Harlyn; Tanguay, Donald; Sobel, Irwin; Gelb, Dan; Goss, Michael E.; Culbertson, W. Bruce; Malzbender, Thomas. The Coliseum Immersive Teleconferencing System.

A

Changes

A.1 Changes in link.bcl

We introduced two new rows to the link.bcl to enable the clients and the processing unit to build up a socket connection:

- DataHostName: Gives the name of the host to which the client has to connect on the port given by LepId.
- AcknolegmentsName: Gives the name of the host to which the processing unit has to connect on the port given by LepId

You can find a example link.bcl-socket in the Conf directory of the modified bc3dvidpar project.

A.2 Changes in CORBA and Audio/Video-Streams

The most important changes were in the files, bclLEP.cpp, bclCommClient.cpp and bclOrb.cpp.

Here you can see part of the initial bclLEP.cpp the most important parts are written in bold:

```
bcl::LEP::LEP(LEPId id,
              int argc,
              char **argv,
              TAO_ORB_Manager *om,
              bcl::ORBThread *orb,
              bcl::Client *client)
: id_(id), argc_(argc), client_(0), ownOrb_(false),
  ownClient_(false),
  allowRsvId_(true)
// allowRsvId_(false)
{
.
.
.
if(strcmp(bcl::LINK_CONFIGURATION::instance()->beName(id), "TxBE") == 0)
{
    // TxBufferEngine
    ACE_DECLARE_NEW_CORBA_ENV;
```

```

ACE_TRY
{
    // ORB manager -----
    if(om == 0)
    {
        TAO_AV_CORE::instance()->init(argc_, argv_, ACE_TRY_ENV);
        ACE_TRY_CHECK;
        om_ = TAO_AV_CORE::instance()->orb_manager();
    }
.
.
.
}

```

The function :

```
om_ = TAO_AV_CORE::instance()->orb_manager();
```

Does not exist any more and the TAO_AV_CORE instance is not used in newer examples anymore. That is why we changed this code to the following:

```

bcl::LEP::LEP(LEPId id,
              int argc,
              char **argv,
              TAO_ORB_Manager *om,
              bcl::ORBThread *orb,
              bcl::Client *client)
: id_(id), argc_(argc), client_(0), ownOrb_(false),
  ownClient_(false),
  allowRsvId_(true)
// allowRsvId_(false)
{
.
.
.
    if(strcmp(bcl::LINK_CONFIGURATION::instance()->beName(id), "Tx-
BE") == 0)
    {
        // TxBufferEngine
        ACE_DECLARE_NEW_CORBA_ENV;
        ACE_TRY
        {
            // ORB manager -----
            if(om == 0)
            {
                om_ = bcl::COM_CLIENT::instance()->getOrbManager();
            }
.
.
.
}

```

To be able to do this we had to change the bclCommClient from this code:

A.2 CHANGES IN CORBA AND AUDIO/VIDEO-STREAMS

```
bcl::ComClient_i::ComClient_i()
: om_(0), orb_(0), client_(0)
{
  ACE_DECLARE_NEW_CORBA_ENV;
  ACE_TRY
  {
    om_ = TAO_AV_CORE::instance()->orb_manager();
    orb_ = new bcl::ORBThread(om_);
    orb_->init(ACE_TRY_ENV);
    ACE_TRY_CHECK;
    orb_->activate();
    ACE_DEBUG((LM_DEBUG, "bcl::ComClient_i(): ORB activated"));
    client_ = new Client(om_);
    client_->init(ACE_TRY_ENV);
    ACE_DEBUG((LM_DEBUG, "bcl::ComClient_i(): CLIENT initialized"));
  }
  ACE_CATCHANY;
  {
    client_ = 0;
    ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,
                        "\nCaught in bcl::ComClient()\n");
  }
  ACE_ENDTRY;
}
```

To the following:

```
bcl::ComClient_i::ComClient_i()
: om_(0), orb_(0), client_(0)
{
  ACE_DECLARE_NEW_CORBA_ENV;
  ACE_TRY
  {
    om_ = ORB::instance()->manager();

    ACE_TRY_CHECK;

    orb_ = new bcl::ORBThread(om_);
    orb_->init(ACE_ENV_SINGLE_ARG_PARAMETER);
    ACE_TRY_CHECK;
    orb_->activate();
    ACE_DEBUG((LM_DEBUG, "bcl::ComClient_i(): ORB activated"));
    client_ = new Client(om_);
    client_->init(ACE_ENV_SINGLE_ARG_PARAMETER);
    ACE_TRY_CHECK;
    ACE_DEBUG((LM_DEBUG, "bcl::ComClient_i(): CLIENT initialized"));
  }
  ACE_CATCHANY;
  {
    client_ = 0;
    ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,
                        "\nCaught in bcl::ComClient()\n");
  }
  ACE_ENDTRY;
}
```

To do this we had to change the `bclOrb.cpp` file that was in the project but not used. As there were already some changes to newer versions, we assume that somebody already tried to update the code to a newer version. We continued by changing this code:

```
bcl::Orb_i::Orb_i()
{
    bcl::Conf conf("main.bcl");
    conf.read();

    ACE_DEBUG((LM_DEBUG, "bcl::Orb_i::Orb_i()\n"));

    ACE_DECLARE_NEW_CORBA_ENV;
    ACE_TRY
    {
        manager_ = new TAO_ORB_Manager;
        manager_->init(conf.argc(), conf.argv(), ACE_TRY_ENV);
        ACE_TRY_CHECK;
        //      ACE_DEBUG((LM_DEBUG, "\tbcl::Orb_i: running ...\n"));
        //      manager_->run(ACE_TRY_ENV);
        ACE_TRY_CHECK;
    }
    ACE_CATCHANY
    {
        ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,
                           "\nCaught in bcl::Orb_i::Orb_i()\n");
    }
    ACE_ENDTRY;
}

```

To the following:

```
bcl::Orb_i::Orb_i()
{
    bcl::Conf conf("main.bcl");
    conf.read();

    ACE_DEBUG((LM_DEBUG, "bcl::Orb_i::Orb_i()\n"));

    ACE_DECLARE_NEW_CORBA_ENV;
    ACE_TRY
    {
        CORBA::ORB_var orb =
        CORBA::ORB_init (conf.argc(), conf.argv(),
                       0,ACE_ENV_ARG_PARAMETER);

        CORBA::Object_var obj
        = orb->resolve_initial_references ("RootPOA"
                                         ACE_ENV_ARG_PARAMETER);

        ACE_TRY_CHECK;

        // Get the POA_var object from Object_var
        PortableServer::POA_var root_poa
        = PortableServer::POA::_narrow (obj.in ()
                                         ACE_ENV_ARG_PARAMETER);

        ACE_TRY_CHECK;
    }
}

```

```

    PortableServer::POAManager_var mgr
        = root_poa->
            the_POAManager(ACE_ENV_SINGLE_ARG_PARAMETER);
    ACE_TRY_CHECK;
    mgr->activate (ACE_ENV_SINGLE_ARG_PARAMETER);
    ACE_TRY_CHECK;
    // Initialize the AV Stream components.
    TAO_AV_CORE::instance ()->init (orb.in (),
                                    root_poa.in ()
                                    ACE_ENV_ARG_PARAMETER);

    ACE_TRY_CHECK;
    manager_ = new TAO_ORB_Manager(orb, root_poa, mgr);

    ACE_TRY_CHECK;

    ACE_DEBUG((LM_DEBUG, "\tbcl::Orb_i: running ...\n"));
}
ACE_CATCHANY
{
    ACE_PRINT_EXCEPTION(ACE_ANY_EXCEPTION,
                      "\nCaught in bcl::Orb_i::Orb_i()\n");
}
ACE_ENDTRY;
}

```

This was the biggest change and gave us the opportunity of working with the rest of the Audio/Video-Streams implementation. In this changes you already can see the syntax changes that have occurred to the new libraries. Basically the ACE_TRY_ENV macro was changed. Here you can see the rules for this changes:

- If the ACE_TRY_ENV is part of the function declaration then it is replaced by ACE_ENV_SINGLE_ARG_DECL if it is the only argument. If there are more arguments in the function it is replaced by ACE_ENV_ARG_DECL. If ACE_TRY_ENV had a default value it is replaced by either ACE_ENV_SINGLE_ARG_DECL_WITH_DEFAULT (if only one argument) or ACE_ENV_ARG_DECL_WITH_DEFAULT (if several arguments).
- If the ACE_ENV_TRY is used when calling a function it is replaced by ACE_ENV_SINGLE_ARG_PARAMETER if it is the only argument and by ACE_ENV_ARG_PARAMETER if there are several arguments. In this case a default value does change the used macro.

A.3 Changes in OpenCV Library

When we tried to run the CvCalibFilter we always had the following error:

```
MKL ERROR : Parameter 8 was incorrect on entry to SGEMM
```

We found in the yahoo OpenCV news group somebody who had the same error. It seems to be a error that only occurs under linux at the time. This changes were suggested:

```

cxcore/src/cxswitcher.cpp,
...
CV_IMPL int
cvUseOptimized()
{
...
const char* mkl_suffix = arch == CV_PROC_IA32_GENERIC ?
    (cpu_info->model >= CV_PROC_IA32_P4 ? "p4" :
     cpu_info->model >= CV_PROC_IA32_PIII ? "p3" : "def") :
    arch == CV_PROC_IA64_ITANIUM ? "itp" : "";

...
}

```

change it to

```

...
const char* mkl_suffix = "";
...

```

We did it and did not have any problems anymore.

B

How to use prototype

B.1 Calibration and Preparation

To calibrate the system and calculate the average background for the segmentation you have to start the coTest project. This executable will first calculate the background images. This means everything that is now in the view of the cameras will count as background later on. After this calculations the system changes to the calibration. It will be looking for Etalons (checkerboards) depending on how many you specify in the code it will continue searching until it has enough accepted checkerboard pairs. This means that you have to move the checkerboard in front of the cameras in a way that both cameras can see the checkerboard. If it has enough data, it will calculate the extrinsic and intrinsic parameters. Afterward that it will capture 100 frames and rectify this images and write them into the /work/pics/ directory, which has to be created before use. You can use this images as a optical test for the quality of the rectification.

After running coTest you will have to copy the following files from the coTest/Cfg directory to the coClientViewer/Cfg directory:

- params (extrinsic and intrinsic parameters)
- {name-of-cameras}.seg (Configuration for the background segmentation needed by the blue-c segmentation algorithm)

B.2 Capturing and Rendering

To start the cameras you will have to start coClientViewer and adjust with 'm' or 'n' the disparity range. 'm' increases the range and 'n' decreases it. The default value is 32. With the keys '1', '2', '4' you can change in between the different mipmap levels. The mipmap level three is not properly supported by Yang's implementation and leads to a crash.

Now you can start the coStereoRenderer to start the rendering on the auto-stereoscopic display. You can zoom in and out by using the keys ',' or '.'. The reconstructed object should be centered at the beginning.

