A Programmable Display-Layer Architecture for Virtual-Reality Applications

ISBN: 90 6196 553 5
ISBN-13: 978 90 6196 553 4

This research was supported by the Netherlands Organisation for Scientific Research (NWO) under project number 600.643.100.05N08. Title: Quantitative Design of Spatial Interaction Techniques for Desktop Mixed-Reality Environments (QUASID).

A Programmable Display-Layer Architecture for Virtual-Reality Applications

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de
Technische Universtiteit Eindhoven, op gezag van de
rector magnificus, prof.dr.ir. C.J. van Duijn, voor een
commissie aangewezen door het College voor
Promoties in het openbaar te verdedigen
op donderdag 12 november 2009 om 16.00 uur

door

Ferdi Alexander Smit

geboren te Amsterdam

Dit proefschrift is goedgekeurd door de promotor:

prof.dr.ir. R. van Liere

Contents

Preface

Ever since I first owned a computer, I have been trying to program it. My first efforts were in Basic on a Commodore 64, typing over programs from library books. I remember my dad, my brother and me taking turns at typing over page after page of code to complete a simple game. Several years later, with the uprise of the Internet around 1996, I suddenly had access to a wealth of information; the times of scavenging through the local library for scraps of information on programming were over. I immediately became interested in computer graphics, and it was not long before I wrote an optimized triangle renderer in assembly. It was then that I decided I wanted a job related to computer graphics.

A few years later, I started studying computer science at the Vrije Universiteit, Amsterdam. Although I had some doubt whether to pursue a degree in computer science or mathematics, the practical aspect of computer science won out in the end. My studies mostly focused on parallel programming, distributed systems, computer networks and operating systems, none of which had much to do with graphics. When it was time for me to choose a graduation project, I came in contact with Tom van der Schaaf and Henri Bal. The project consisted of research on parallel terrain rendering, bringing me back to computer graphics.

After graduating in 2005, I started looking for a Ph.D. position because I enjoyed the challenges of research. Henri Bal informed me that a colleague of his, Robert van Liere, might have a vacancy for a Ph.D. student in the Visualization and 3D Interfaces group at the Centrum Wiskunde & Informatica (CWI). It is there that I spend the next four years, carrying out the research presented in this thesis. I have learned a lot during my time there, not only about computer science and virtual reality but also about the writing and presenting of papers and the methods of good research. I have come to understand that having an idea is but one part of the equation; being able to convey that idea clearly to others is equally important.

I'd like to take this opportunity to express my thanks towards a number of people who have supported me during the course of my Ph.D. studies. First of all, I would like to thank my supervisor, Robert van Liere, whose advice and encouragement have been invaluable for me to complete this thesis; papers simply turn out better after incorporating Robert's comments and suggestions. I'm looking forward to continuing research together in the upcoming years. In addition, my thanks go out to Bernd Fröhlich for his high quality input and suggestions on projects and his participation in the reading committee. I'm certain there will be opportunities to co-author papers again in the future. I'd also like to thank the other members of the reading committee, Jack van Wijk, Jean-Bernard Martens and Roger Hubbold, for proofreading this thesis. Finally, I thank Arjen van Rhijn and Stephan Beck for co-authoring some papers with me, as well as my other colleagues at the CWI.

Of course, I also owe a great deal of thanks to my close family: my fiancée, Amrita Karunakaran, with whom I hope to spend a long and happy life now that we are finally getting married; my brother and best friend, Michiel Smit, with whom I can converse at length about virtually anything; and my loving parents, Loes Smit and Cees Smit, who gave me the upbringing and means to become who I am today. I am grateful towards all of them for their encouragement, patience and humor.

Ferdi Smit
Amsterdam, 5th October 2009

Chapter 1

Introduction

The virtual-reality vision, as proposed by Sutherland [Sut70], is to treat the display as a window into a virtual world. The image generation should be of sufficient quality such that the displayed images look real and are generated in real time. Furthermore, the user should be able to directly manipulate virtual objects in a natural way. The virtual-reality experience has later been defined as any in which the user is effectively immersed in a responsive virtual world [Bro99]. Immersion is a term with widely varying definitions; however, the underlying concept can be roughly described as follows: when a user is immersed, he or she should feel as if being a participant in a world that behaves and looks like a close approximation of the real world. Therefore, two important technical objectives of VR systems are to provide convincing visuals and natural user interaction.

Research in virtual reality has mostly been devoted to increase the sense of immersion and also to facilitate user interaction. One common technical aspect is the real-time display of realistic 3D images from the perspective of the user. These images are often presented in true 3D on stereoscopic displays in an attempt to increase the sense of perceived realism and immersion. Another aspect is the development of spatial interaction techniques that allow the user to perform interactive tasks in natural and intuitive ways, for effective interaction techniques can increase users' task performance in the virtual world.

Despite the many technological advances, there are still several factors present in today's virtual reality systems that can break the feeling of immersion. Four particularly challenging factors can be identified:

- **End-to-end latency** — An important aspect of interactive VR systems is end-to-end latency, which is defined as the time interval between a user's initiation of an action and the moment when the effect of this action is perceived. In VR this is typically the time interval between moving an interaction device and observing the corresponding change in object pose on the display. It is a well-known fact that high latency has a negative effect on interaction performance [EYAE99]. Furthermore, latency between user motion and the visual representation of this motion can break the illusion of a virtual world [MRWB03, Bro99] and cause motion sickness. Bles and Wertheim report with respect to head-tracking motion that "even with delays as brief as 46 ms, the resulting visual-vestibular mismatch [...] may already be extremely nauseating" [BW00]. Experience shows that latencies of 50 ms or more are already perceptible; however, the latencies of VR systems often extent to hundreds of milliseconds. Minimizing end-to-end latency

is still an open and challenging problem, and constructing a realistic VR system with a latency lower than 50 ms is considered a daunting task.

- **Judder** — The experimental psychology literature describes an effect where motion causes a single object to be perceived as multiple objects [BES95, FPS90]. In the video-processing community this effect is also called judder [Mar01], and it is caused by a repetition of images on the display. If the rendering of a 3D scene is performed at a lower rate than the display frequency, multiple display frames will be displaying a moving object in the same position. When the rendering is complete and a positional update is generated, the object will suddenly jump to this new position. The human visual system has difficulties interpreting these sudden large changes, and the result is that multiple objects are perceived. The presence of judder degrades perceived image quality and causes user fatigue and eye-strain [BW00].

- **Crosstalk** — Stereoscopic display systems suffer from crosstalk, an effect that produces visible ghost images. Crosstalk is generally believed to be undesirable, and experimental studies have suggested that crosstalk can reduce, and at times even inhibit, the ability to perceive depth. For example, Seuntiens et al. [SMI05] examined the influence of crosstalk intensity on the subjective perception of depth, visual strain and image distortion. Their results show that increasing crosstalk affects perceived image distortions negatively. Yeh and Silverstein [YS90] performed a controlled experiment to determine the limits of fusion and depth judgment for stereoscopic displays. It was shown that for extended duration stimulus exposure, the introduction of crosstalk has a significant effect on fusion limits. This indicates that a loss of image quality, in this case due to crosstalk, can have a direct negative effect on immersion.

- **Optical tracking** — A commonly used method for six-degree-of-freedom (6-DOF) user interaction in VR is optical tracking. An inherent problem in optical tracking is that line of sight is required: if the input device is even partially occluded, a pose can often not be found. When interaction techniques start to feel unnaturally clumsy and difficult to use, the user's sense of immersion in the virtual world will quickly degrade [MRWB03]. Furthermore, the user's performance for an interactive task also often depends on the performance of the optical tracking system. Thus, it is important to evaluate and compare the performance of optical trackers. Many aspects must be taken into account, such as the type of interaction task that is performed; the intrinsic and extrinsic camera parameters, such as focal length, resolution, number of cameras and camera placement; environment conditions in the form of lighting and occlusion; and end-to-end latency. Most optical tracker descriptions do not take all these aspects into account when describing the tracker performance.

1.1 Problem statement

Several implementations of VR systems have appeared in the past. Although the exact details may vary, most of these are based on similar underlying design principles. Usually the system consists of four components: a tracking system for interaction, a simulator to update the virtual scene, a rendering system to produce images, and a display device. The user initiates an action using a tracking device, after which the tracker data is used to update a simulation process. Next, a scene graph is constructed or updated according to the simulation output data. This scene graph is then rendered, and the resulting image is sent to the display device.

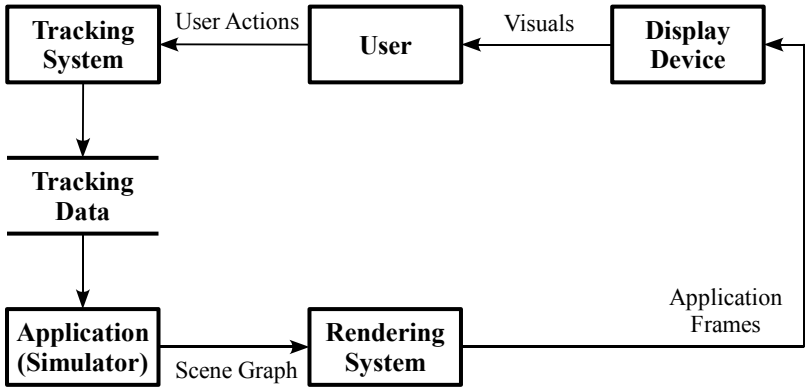

Figure 1.1: Schematic overview of a typical VR architecture. A sequence of four system components is shown, where each component depends on the output from the previous one. The user interacts with the system through a tracking system and sees the results of his actions on the display device. The application updates a scene graph according to the input tracker data, which is then rendered and output to the display device.

Finally, the visual feedback of the user's action is presented on the display device. Figure 1.1 depicts the interaction cycle of such a typical VR-architecture pipeline.

Each of the components in the VR pipeline generates updates at its own rate. For example, tracking devices may be sampled at a rate of 120 Hz, rendering performed at 20 Hz, and the display refresh rate is usually 60 Hz. An important observation is that when these components are coupled in a sequential fashion, the update rate of the entire interaction cycle can only be as high as the slowest component. This is an undesirable property, since the update rate of the interaction cycle corresponds directly to the system's end-to-end latency.

Certain architecture designs, such as the decoupled simulation model (DSM [SGLS93]), succeed in decoupling the interaction, simulation and rendering components by running them in parallel. Each component then operates at its own rate, repeatedly using the same input data until new data is made available by another component. However, these architectures still do not allow for independent display updates. This is due to the fact that standard display hardware is highly rigid and non-programmable: in the absence of rendering updates, the hardware will repeatedly display the last received image. Consequently, updated application data can only be presented to the user at the rate of the rendering system, regardless of potentially faster update rates of either the display device or any of the other system components.

Another often employed method is to make use of static level-of-detail (LOD) approaches, where the number of polygons rendered is reduced to such extent that the rendering can be performed at the same rate as the display refresh rate. The geometric models are decimated by successively removing all those polygons that are considered to have the least visual significance until a target number of polygons is reached for which a high update rate can be guaranteed. However, level-of-detail methods come at the cost of reduced image quality.

The inherent design principles of classic VR architectures, in particular the tight coupling between the dynamic input, the rendering loop and the display system, inhibits those systems from simultaneously addressing all the aforementioned challenging factors of VR systems. For example, scene graph updates can only be perceived by the user after an entire application update has been rendered and displayed; therefore, the end-to-end latency is at least as high as the time required for rendering [OCMB95]. Furthermore, crosstalk is an effect that occurs

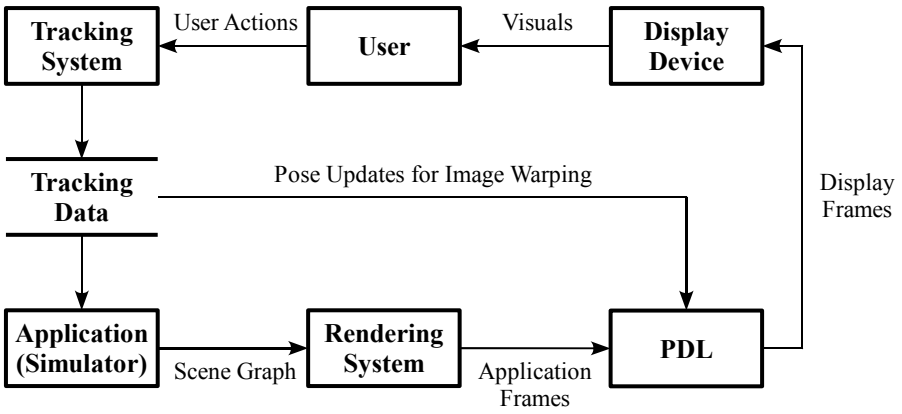

Figure 1.2: Schematic overview of the PDL architecture. An extra component is added to the pipeline between the rendering system and the display device. This allows for independent display device updates at the rate of the display refresh cycle, regardless of the update rate of the rendering system. Rendered application frames are modified and updated using image warping. In this way, updated display frames that incorporate the latest tracker data can be generated at a fast rate.

between consecutive frames on the display, not between application updates [SvLF07b]. Finally, judder is caused by discontinuous object motion due to the differences in application and display update frequencies [BES95]. These problems can be addressed by decoupling the display update cycle from the interaction and rendering cycles in a VR system, allowing for independent display updates.

1.2 PDL architecture

In this thesis, a VR-architecture design is introduced that is based on the addition of a new logical layer: the Programmable Display Layer (PDL). The governing idea is that an extra layer is inserted between the rendering system and the display, as is shown in Figure 1.2. In this way, the display can be updated at a fast rate and in a custom manner independent of the other components in the architecture, including the rendering system.

To generate intermediate display updates at a fast rate, the PDL performs per-pixel depth-image warping by utilizing the application data. Image warping is the process of computing a new image by transforming individual depth-pixels from a closely matching previous image to their updated locations. The PDL's image warping is performed in real-time and in parallel with the application that is busy generating new application frames. Since the PDL runs independently of the application, the generation of new application updates is not postponed. The key differences between the PDL implementation and architectures with similar goals are that the PDL architecture runs in real-time on commodity hardware and imposes only minimal restrictions on the used virtual scenes — dynamic scenes are allowed in particular. Also, the image quality produced by the PDL architecture is generally superior to the quality produced by static level-of-detail methods.

The PDL architecture can be used for a wide range of algorithms and to solve a number of problems that are not easily solved using classic architectures. Three such algorithms have

been implemented:

- An algorithm for fine-grained latency reduction using a shared scene graph and image warping. The minimal amount of latency is achieved if changes in the input state are reflected for every single consecutive image on the display. Using the PDL, it is possible to re-sample or predict the input for every display frame at a fast rate, instead of only once every application update. This results in an environment with reduced latency.

- A non-uniform crosstalk reduction algorithm that can be used to accurately reduce crosstalk over the entire display area. On the PDL architecture, crosstalk reduction can be performed for each consecutive display image, independent of the application update cycle. This results in improved quality of stereoscopic visuals.

- An algorithm for judder reduction and smooth motion. The effect of judder is eliminated by extrapolating object positions for each display frame. This increases the perceived quality of motion.

1.3 Scientific contribution

A central theme throughout this thesis is the automatic detection and quantification of errors and subsequently attempting to resolve these errors. The primary scientific contribution is threefold:

- A quantitative metric for determining errors in 3D image warping. For the image-warping algorithms implemented on the PDL, per-pixel errors are detected in the output images and direct ray tracing is used in an attempt to resolve these errors. This allows for objective image-quality-versus-latency trade-offs.

- Non-uniform crosstalk reduction and its quantitative evaluation. For the crosstalk reduction algorithms, the amount of additional per-pixel intensity is evaluated and the pixels are corrected accordingly. This results in objective metrics to compare various crosstalk correction algorithms.

- A framework for the quantitative evaluation of optical tracking methods. The optical tracker simulator framework allows for the quantification of optical tracker performance and subsequent improvement of the algorithms and setup. This allows optical trackers to be judged according to objective metrics.

A common approach in VR is to implement an algorithm and then to evaluate the efficacy of that algorithm afterwards by either subjective, qualitative metrics or quantitative user experiments, after which an updated version of the algorithm may be implemented and the cycle repeated. A different approach is explored in this thesis. The efficacy of various algorithms is evaluated using completely objective and automated quantitative methods. For these evaluation methods, existing algorithms such as the VDP [Dal93] are used. Note that the focus lies not on the implementation of these methods, but on the way in which they are used. Furthermore, errors are dynamically detected in the output using such evaluation methods while the algorithms are running, and a subsequent attempt is made to resolve these errors using different, specialized algorithms. Two different types of quantitative evaluation metrics are required: an accurate off-line method that uses a known error-free reference and

a fast online method that results in a quick approximation of the errors. The former method enables the objective assessment and comparison of the efficacy of various algorithms, while the latter can be used to detect errors dynamically and improve the output.

1.4 Scope

Throughout this thesis, the focus lies on desktop-driven near-field virtual reality environments, such as the Personal Space Station [MvL02] and Fish Tank VR [WAB93]. It is assumed that 3D images are presented by means of active, time-sequential stereoscopic displays. It is further assumed that the generation of images is performed by the rendering of geometry contained in an available scene graph.

The application domain is assumed to be that of scientific visualization, where outside-to-inside viewing conditions are typical. This means that the focus lies on applications where the user interacts with scenes consisting of complex visualization objects in a world-in-hand fashion, in contrast to VR applications that use inside-to-outside viewing, such as walk-throughs in expansive virtual worlds.

1.5 Thesis outline

In Chapter 2 an overview is given of the terminology and concepts used in this thesis. Chapter 3 describes the design and implementation of the PDL architecture in combination with image warping. In Chapter 4, a number of novel contributions are described to detect and resolve errors in image warping and to avoid these errors in the first place using optic flow information. Chapter 5 describes a non-uniform model for crosstalk and the implementation of crosstalk reduction algorithms based on this model. The details of the quantitative methods used to evaluate the quality of the image-warping and crosstalk-reduction implementations are given in Appendix A. Furthermore, as additional motivation for the use of image warping, a comparison between the quality of image warping and a classic level-of-detail mesh decimation method is described in Appendix B. In Chapter 6 a simulation framework for the quantitative evaluation of optical trackers is presented. In this chapter, two optical trackers are examined in particular: GraphTracker and an ARToolKit-based optical tracker. While the implementation of GraphTracker is not part of the main focus of this thesis, a detailed description of this implementation is provided in Appendix C for reference. Finally, conclusions and possible future work are given in Chapter 7.

1.6 Publications from this thesis

This thesis is based on the following peer-reviewed conference and journal publications:

1. F. A. Smit, R. van Liere, and B. Fröhlich. A programmable display layer for virtual reality system architectures. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2009 [in press]. **(Chapters 3 and 4 and Appendix B)**

2. F. A. Smit, R. van Liere, S. Beck, and B. Fröhlich. A shared-scene-graph image-warping architecture for VR: low latency versus image quality. *Elsevier Computers & Graphics*, 2009 [in press]. **(Chapters 3 and 4)**

3. F. A. Smit, R. van Liere, S. Beck, and B. Fröhlich. An image warping architecture for VR: Low latency versus image quality. In *Proc. IEEE Virtual Reality (VR)*, pages 27–34, 2009. **(Chapter 3)**

4. F. A. Smit, R. van Liere, and B. Fröhlich. An image warping VR-architecture: Design, implementation and applications. In *Proc. ACM VRST*, pages 115–122, 2008. **(Chapters 2 and 3)**

5. F. A. Smit, R. van Liere, and B. Fröhlich. The design and implementation of a VR-architecture for smooth motion. In *Proc. ACM VRST*, pages 153–156, 2007. **(Chapter 3)**

6. F. A. Smit, R. van Liere, and B. Fröhlich. Non-uniform crosstalk reduction for dynamic scenes. In *Proc. IEEE Virtual Reality (VR)*, pages 139–146, 2007. **(Chapter 5)**

7. F. A. Smit, R. van Liere, and B. Fröhlich. Three extensions to subtractive crosstalk reduction. In *Proc. EGVE*, pages 85–92, 2007. **(Chapter 5)**

8. F. A. Smit and R. van Liere. A simulator-based approach to evaluating optical trackers. *Elsevier Computers & Graphics*, 33(2):120–129, 2009. **(Chapter 6)**

9. F. A. Smit and R. van Liere. A framework for performance evaluation of model-based optical trackers. In *Proc. EGVE*, pages 33–40, 2008. **(Chapter 6)**

10. F. A. Smit, A. van Rhijn, and R. van Liere. Graphtracker: a topology projection invariant optical tracker. *Elsevier Computers & Graphics*, 31(1):26–38, 2007. **(Appendix C)**

11. F. A. Smit, A. van Rhijn, and R. van Liere. Graphtracker: a topology projection invariant optical tracker. In *Proc. EGVE*, pages 63–70, 2006. **(Appendix C)**

Chapter 2

VR-architectures: an overview

This chapter gives a brief overview of a number of different aspects of VR architectures and their designs. Several techniques that have been proposed in the past are introduced, as well as a number of new concepts and terms that are used throughout this thesis.

2.1 VR-architecture design

As was described in Section 1.1 and shown in Figure 1.1, VR architectures typically consist of a number of components that operate at different rates. A sample VR setup is shown in Figure 2.1. The user is viewing a CT-scanned model of a coral on a stereoscopic display operating at 60 Hz using active-stereo shutter glasses. Head-tracking is provided by an acoustic head-tracker that operates at 50 Hz. The model can be interactively rotated by means of 6-DOF optical tracking that generates reports at 120 Hz. However, the stereoscopic rendering of this complex 17M polygon model is relatively slow and done at a rate of only 6 Hz. Therefore, on a classic architecture, the scene graph update is constrained by the render process, and input data arriving at a higher rate than the application frame rate is ignored, introducing both high latency and judder.

Display systems typically operate at a minimum of 60 Hz for monoscopic viewing, or 120 Hz in the case of active stereoscopic viewing (60 Hz per eye). This implies that consecutive images on the display are only visible for approximately 16.7 ms. These consecutive images on the display are called *display frames*. Graphics APIs and display hardware are highly rigid with respect to display updates. The default behaviour of repeating the same display frame over and over in the absence of application updates can virtually never be changed. The only control the application has over the display is in filling a frame buffer and requesting a buffer swap. The frame buffer is then read from memory and scanned-out to the display whenever the hardware sees fit, usually at the next display refresh signal. Such an update of the display by the application is called an *application frame*. This is shown in Figure 2.2, which corresponds to the VR-latency model proposed by Mine [Min93].

As explained in Chapter 1, an architecture where the temporal coupling between the various components is less strict is desirable. In the past, architectures have been presented that attempt to decouple the various components. Several of these succeed in decoupling the interaction, simulation and rendering components; however, none of them decouple the rendering and display components. A number of these designs are described in Section 2.1.1. A dif-

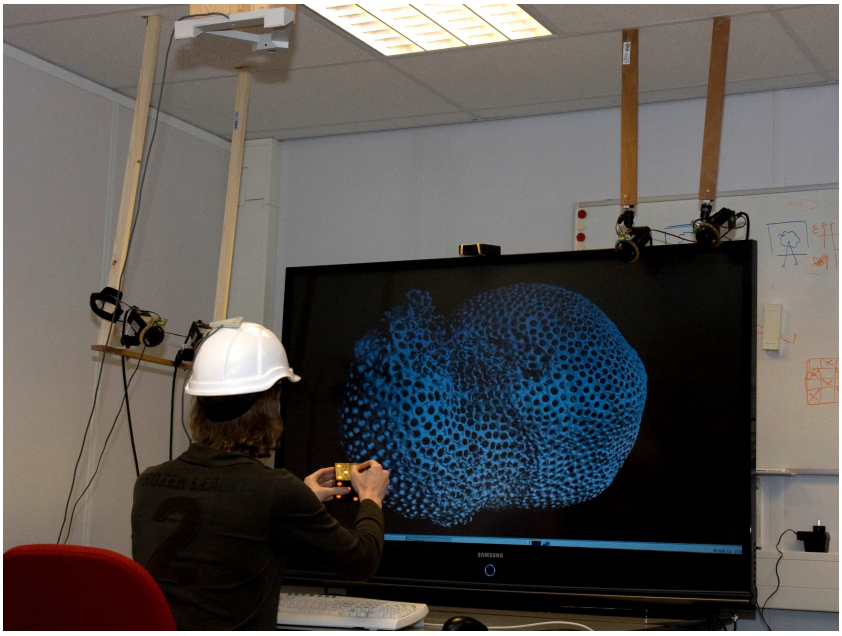

Figure 2.1: Interactive visualization of a 17M polygon model. A user is examining a model of a coral structure on a Samsung HL67A750 60 Hz stereoscopic DLP TV using LCS glasses (for image capturing purposes a monoscopic image is displayed at screen-filling HD resolution). Head-tracking is performed by a Logitech head-tracker running at 50 Hz. The model can be interactively rotated using 6-DOF optical tracking running at 120 Hz. The rendering of the model is done at 6 Hz only. Therefore, on a classic architecture, new application frames are generated at a maximum rate of 6 Hz.

ferent class of architecture design that also succeeds in decoupling the rendering and display components is based on image warping. The proposed PDL architecture falls in this class.

2.1.1 Decoupled designs

Shaw et al. [SGLS93] proposed the Decoupled Simulation Model (DSM), which is an API for general VR architectures. The DSM utilizes four decoupled components for computation, geometry, interaction and presentation. Each of these components can independently generate events. Most other standard APIs (e.g., VR Juggler [BJH+01] and CAVElib [VRC07]) operate using a similar model, although the exact components may be different. All of these APIs share the fact that application frames are repeated because the display frame rate is not the driving rate of presentation.

Olano et al. [OCMB95] noted the need to separate the image generation from the display update rate to combat rendering latency. They propose the SLATS system, which guarantees only one display frame (16.7ms) of latency. This is achieved by insisting that all work for one display frame is finished during the frame immediately before it. The architecture consists of a number of graphics processors, a ring-network, and an additional number of rendering processors. The graphics processors generate rendering primitives in batches, which are then

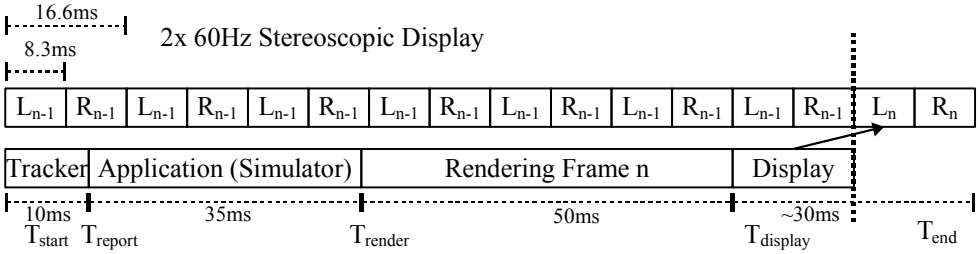

Figure 2.2: (left) Temporal overview of a classic VR-architecture conforming to Figure 1.1. Two processes are running in parallel: the application and the display. The user initiates an action at T_{start}, the tracking system reports a pose at T_{report}, the application completes the generation of a scene graph at T_{render}, the rendering system renders the scene graph, and the resulting application frame is sent to the display at $T_{display}$. The new application frame will be visible at T_{end}. During the time required to process tracking data, generate a scene graph, perform rendering and sending the resulting application frame to the display, several consecutive display frames are presented on the display. Hence, application frames are repeated several times on the display until a new application frame has been generated.

sent over the ring-network to the rendering processors. In turn, the rendering processors are responsible for rendering the primitives and scanning out the resulting images to the display. In this way, the rendering processors operate independently of the graphics processors in updating the display, and a single frame of latency is guaranteed. This method is limited by the constraint on rendering time available to the rendering processors; only a small number of primitives can be rendered, and shading must not be complex and time-consuming.

Several other architectures follow the approach of Olano et al. by attempting to update the display at every display frame; however, different means of achieving this are often used. For example, Kijima and Ojika [KO02] proposed an architecture to reduce latency effects on HMDs. Scenes are first rendered with a greater field-of-view than the HMD provides. Next, the user's head-motion is extrapolated at every new display frame and the image is shifted accordingly on special HMD hardware. Stewart et al. [SBM04] proposed the PixelView architecture. Instead of rendering a single 2D image for a specific viewpoint, they construct a 4D viewpoint-independent buffer. Then, for every display frame, a specific view is extracted from the 4D buffer according to a predicted viewpoint. The architecture requires the entire scene to be subdivided into points, or alternatively into specific types of primitives the system can handle. An implementation is provided using custom-build hardware. The problem with these architectures is that custom hardware is required, that the rendering is constrained to specific primitives, and that they are limited to viewpoint prediction only, with no support for prediction at the object level.

2.1.2 Image-warping designs

Image warping is an image-based technique that, given source images rendered from a certain viewpoint, generates images for new viewpoints by performing a per-pixel re-projection. The basic idea of image warping was first introduced by McMillan and Bishop [MB95]. The central idea to implementing image warping is to first un-project each 2D pixel of a rendered image back to 3D space, then optionally transform the pixel according to some updated state,

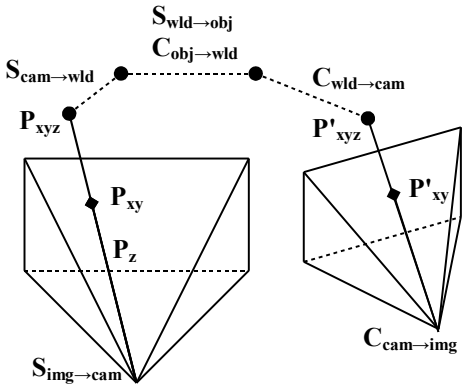

Figure 2.3: Central idea to image warping. A 2D server-side pixel P_{xy} is converted back to a 3D pixel P_{xyz} using its depth P_z and the camera projection matrix $S_{img \rightarrow cam}$. The pixel is then transformed into world space ($S_{cam \rightarrow wld}$) and subsequently passes through object space ($S_{wld \rightarrow obj}$) into the client-side world space ($C_{obj \rightarrow wld}$). These transforms allow the modification of the pixel's 3D position for every display frame. On the client-side, the pixel is transformed back into the client's camera space using $C_{wld \rightarrow cam}$ and is then projected back to 2D (P'_{xy}) using $C_{cam \rightarrow img}$.

and finally to re-project the transformed pixel back to a new 2D camera image. The process is depicted in Figure 2.3 and is described in depth in Section 3.1.4.

Using image warping, rendered application frames can be transformed and re-projected for new object poses and camera viewpoints by a different process in a post-processing step. Since image warping is completely image based and operates on fixed-sized images, this post-processing can be performed at rates independent of the rendered scene's complexity. Image warping comes at the expense of some trade-offs. First, it can have a negative effect on image quality, such as sampling artefacts and occlusion errors due to missing image information. The quality of warped images depends on the distance of the new viewpoints to the ones used to render the original images. Another limitation is that scene translucency, volume rendering and deformable objects are not easily handled using standard image warping and require specialized algorithms. However, as is shown in Appendix B, image warping generally results in better image quality than achieved using static level-of-detail methods.

The benefit of being able to construct images at constant rates for new viewpoints in a post-processing step is that it provides a means to decouple the rendering component from the display component. The governing idea is that the rendering system first renders an image as normal, after which an image-warping process that runs in parallel to the rendering system warps these images to updated viewpoints. The image-warping process is then responsible for updating the display at a constant rate. A schematic overview of how image warping techniques can be used to decouple the rendering from the display is shown in Figure 2.4. Mark et al. were the first to introduce an architecture based on this concept [MMB97, Mar99]; however, the implementation only ran in simulated real-time and focussed on viewpoint changes only.

Approaches that are somewhat different in the implementation of image warping but similar in the concept of parallel display updates have also been proposed. For example, Regan and Pose proposed a virtual address recalculation pipeline [RP94, PR94], where for each

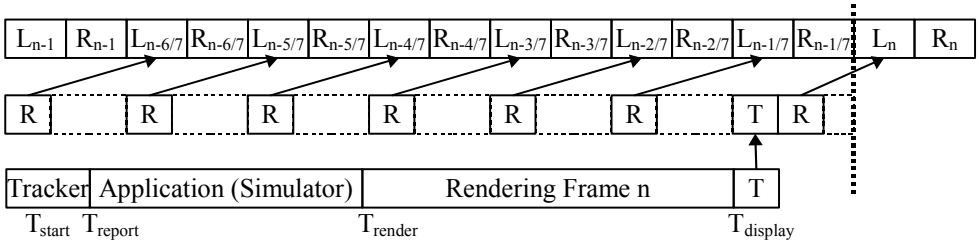

Figure 2.4: (left) Temporal overview of the image-warping PDL architecture conforming to Figure 1.2. To change the default display behavior of repeating frames until an update is available, an extra programmable display layer (PDL) has been added. Three processes are running in parallel: the application, the PDL and the display. New applications frames are sent to the PDL at $T_{display}$ (T). The architecture operates similarly to a classic architecture, except that application frames are sent to the PDL instead of the display. The PDL performs image warping and sends intermediate frames to the display every time a new display frame is required (R).

application frame individual objects are rendered into distinct buffers. All these buffers are then transformed and combined to produce display frames at a fast rate. This approach shows similarities with the Talisman architecture by Torborg et al. [TK96] and, to some extent, layered depth images [SGHS98], which is an approach that combines several consecutive depth slices of the same image into a layered representation to address occlusion artefacts. Popescu et al. [PLAdON98] used layered depth images to accelerate the rendering of architectural walk-throughs. A hybrid geometric-image-based approach was proposed by Hidalgo and Hubbold [HH02] where image warping is performed for geometry that is considered to be far away from the viewer, and regular rendering is used for the geometry close to the viewer in an attempt to reduce warping errors. Finally, in the context of auto-stereoscopic displays, image warping using splatting was used to generate the multiple required shifted view-points from a single rendered view [HZP06].

Drawbacks of these previous image-warping architectures are that they either require special hardware to be used in real-time, impose constraints on the scenes used for rendering, or are test-bed systems that do not operate in real-time for realistic resolution and scenes. Furthermore, the focus of these systems mostly lies on static scenes and viewpoint changes, with no support for moving objects.

2.2 Latency reduction methods

End-to-end latency in VR is defined as the time-delay between an action and its observed effect. As shown in Figure 2.2, a user initiates an action at time T_{start} and the tracking device reports a pose at T_{report}. The application then generates an updated scene graph and starts rendering this at time T_{render}. Once rendering is complete, the newly generated application frame is scanned out to the display at time $T_{display}$. The first updated display frame is completely visible at time T_{end}. For a stereoscopic display it is somewhat difficult to determine exactly when T_{end} occurs, so a reasonable estimate is at the display's second vertical blank signal. End-to-end latency is now defined as $\Delta t = T_{end} - T_{start}$.

Figure 2.5 shows the effect of end-to-end latency on rendering in VR. Suppose an appli-

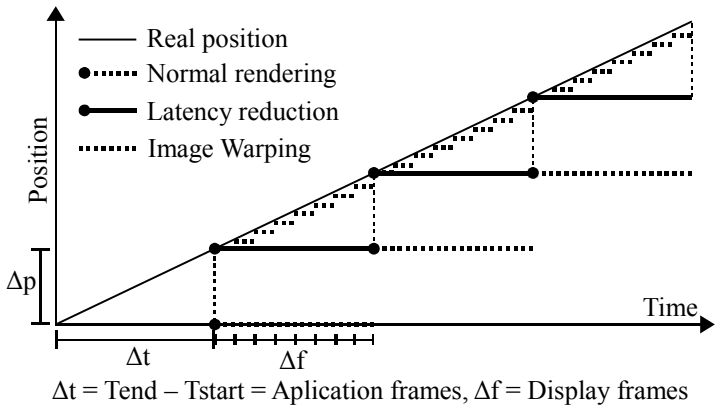

Figure 2.5: Timeline for latency reduction. The real object's position is depicted by the solid line. Rendering an application frame takes an amount of time equal to Δt, during which the object moves by an amount of Δp. Normal rendering samples the object's position at the beginning of rendering and renders it in that location; however, in reality the object has already moved by Δp when the frame is finally displayed. Classic latency reduction predicts the object's position at the time of display instead of the time of rendering. Since new display updates only occur once every application frame, the object remains in the same position for the entire duration Δt. An image-warping architecture is able to predict the object's position every display frame Δf, resulting in more accurate prediction on average.

cation is run where an object tied to an interaction device is moving at a constant velocity over a one-dimensional path. The position of the object set out against the application time is then a straight line. The application starts by sampling the object's position and then renders it. Generating and rendering an updated application frame takes an amount of time shown as Δt. However, during this time Δt the object moves by an amount of Δp; therefore, when the updated application frame is visible on the display, the object is already at a different location using normal rendering. An often-used solution is to predict the position of the object at a time Δt in the future using Kalman filtering [Kal60]. In this way, the object is rendered at the position where it should be when the display is actually updated. This approach is called latency reduction or dead reckoning. Note, the object remains visible at the same location on the display until a new application frame is generated and displayed, regardless of the display frame rate.

Since image-warping architectures can generate predicted, extrapolated display frames, they are well-suited for performing latency reduction. For every new display frame, all the pixels are extrapolated by means of image warping according to a prediction of Δt and the scene's motion information. In this way, the scene is warped to where it should be at the time of display, precisely corresponding to normal latency reduction. Contrary to normal latency reduction, not only a time-step Δt is predicted, but for every display frame an extra time-step Δf is predicted as well. This scenario is also shown in Figure 2.5.

Instead of predicting all motion, it is also possible to obtain additional samples of the interaction device. For every display frame, the interaction device is polled to see if a new pose report is available. If this is the case, the latest known pose is used instead of a prediction, after which the scene is warped accordingly. This allows for a higher sampling rate of the

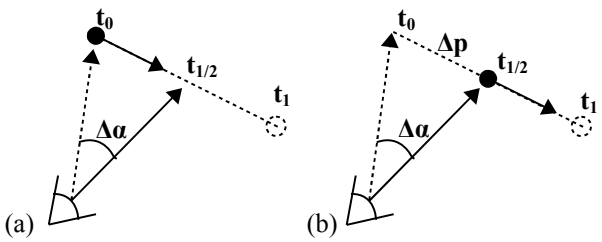

Figure 2.6: Synchronization issues between head-tracker predictions and a simulation. (a) A user is following a moving simulation object from t_0 to t_1. Viewpoint extrapolation predicts the user's viewpoint $\Delta\alpha$ at $t_{1/2}$; however, this does not take into account the object's position, causing the user to miss the object. (b) When object motion is taken into account, the object's position Δp can be predicted at $t_{1/2}$, in addition to the viewpoint $\Delta\alpha$, allowing for the object to be followed correctly.

interaction device, resulting in fewer prediction errors than with regular latency reduction, where the device is sampled only once per application frame. Furthermore, as can be seen from Figure 2.5, the rate of visual feedback is higher. Even with regular latency reduction, the object's position on the display is only updated once every application frame. Using image warping, the position is predicted and possibly sampled every display frame, resulting in a faster observed response from the interaction device.

A problem occurs when only the user's viewpoint is extrapolated, but not the simulation or scene itself. This is an effect that may occur when the scene graph is rendered repeatedly from different viewpoints before being updated by the simulation, which is a typical approach for existing architectures providing viewpoint extrapolation only (e.g., [SBM04, KO02, SGLS93]). An example is given in Figure 2.6. Imagine a head-tracked observer is following a moving simulation object. Simulation updates are generated in the form of new application frames at time t_0 and t_1; therefore, the object jumps from one location to the next at t_1. When predicting head-tracking motion, intermediate display frames will be generated between these two application frames. A sample intermediate frame is shown for time $t_{1/2}$ in Figure 2.6 (a). The user's viewpoint change is predicted by $\Delta\alpha$ and the scene is temporarily updated for that specific viewpoint. However, when the simulation is not updated as well, the object will remain where it was at time t_0 until a new application frame is available. This causes a visual artefact when the user is trying to follow the moving object. Image warping allows motion extrapolation to be performed for the entire scene, including the simulation objects and their predicted motion. Hence, the object's position at time $t_{1/2}$ is also predicted by Δp, and the user can follow the object correctly. This is shown in Figure 2.6 (b). In effect, an image-warping architecture can synchronize the prediction of both simulation, rendering and head-tracking.

2.3 Stereoscopic displays and crosstalk reduction

Stereoscopic display systems allow the user to see three dimensional images in virtual environments. For active stereo with CRT monitors, Liquid Crystal Shutter (LCS) glasses are used in combination with a frame sequential display of left and right images. Stereoscopic displays often suffer from crosstalk or ghosting, which occurs when one eye receives a stim-

ulus that was intended for the other eye only.

Woods and Tan [WT02] studied the various causes and characteristics of crosstalk. Three main sources of crosstalk can be identified: phosphor afterglow, LCS leakage and LCS timing. Typical phosphors used in CRT monitors do not extinguish immediately after excitation by the electron beam, but decay slowly over time. Therefore, some of the pixel intensity in one video frame may still be visible in the subsequent video frame. Also, LCS glasses do not go completely opaque when occluding an eye, but still allow a small percentage of light to leak through. Finally, due to inexact timing and non-instantaneous switching of the LCS glasses, one eye may perceive some of the light intended for the other eye. The combination of these effects causes an undesirable increase in intensity in the image, which is visible as crosstalk. The amount of crosstalk increases drastically towards the bottom of the display area in a non-linear fashion. Hence, crosstalk is non-uniform over the display area. The causes of crosstalk are all inherently related to the sequential display of left- and right-eye display frames, as light leaks from the previous to the current display frame.

To enhance depth perception, the negative perceptual effects of crosstalk should be eliminated or reduced. One way to achieve this is by using better, specialized hardware, such as display devices with faster phosphors. However, phosphor persistence and the LCS shutter glasses are about equal contributors to crosstalk; therefore, the crosstalk problem can not be solved solely by using fast-phosphor display hardware. An alternative solution is to reduce the effect of crosstalk in software by post-processing the images that are to be displayed.

The governing idea of software crosstalk reduction methods is to subtract an amount of intensity from each pixel in the displayed image to compensate for the leakage of intensity from the preceding display frame. This method is called *subtractive crosstalk reduction*. A pre-condition is that the displayed pixels have enough initial intensity to subtract from. If this is not the case, the overall intensity of the image has to be artificially increased, thereby losing contrast. As such, there appears to be a trade-off between loss of contrast and the amount of crosstalk reduction possible.

Lipscomb and Wooten [LW94] were the first to describe a subtractive crosstalk reduction method. First, the background intensity is artificially increased, after which crosstalk is reduced by decreasing pixel intensities according to a specifically constructed function. The screen is divided into sixteen horizontal bands, and the amount of crosstalk reduction is adjusted for each band to account for the non-uniformity of crosstalk over the screen. Although a non-uniform model is used, the difficulty with this method is determining proper function parameters that provide maximum crosstalk reduction for each of the sixteen discrete bands.

Most CRT display devices use phosphors with very similar characteristics, such as spectral response and decay times; however, there is considerable amount of variation in the quality of LCS glasses [WT02]. This indicates that software crosstalk reduction methods need to be calibrated for specific hardware. Such a calibration-based method was proposed by Konrad et al. [KLD00]. The amount of crosstalk caused by an unintended stimulus on a pre-specified intended stimulus is measured by a psychovisual user experiment. The viewer has to match two rectangular regions in the center of the screen in color. One contains crosstalk and the other does not. After matching the color, the actual amount of crosstalk can be determined. The procedure is repeated for several values of intended and unintended stimuli, resulting in a two dimensional look-up table. This look-up table is inverted in a preprocessing stage, after which crosstalk can be reduced by decreasing pixel intensities according to the inverted table values. Optionally, pixel intensities are artificially increased by a contrast reducing mapping to allow for greater possibility of crosstalk reduction. A drawback of this method is that it assumes crosstalk is uniform over the height of the screen.

Klimenko et al. [KFNN03] implemented real-time crosstalk reduction for passive stereo systems. Three separate layers are combined using hardware texture blending functions. The first layer contains the unmodified left or right image frame to be rendered. The second layer is a simple intensity increasing, additive layer to allow for subsequent subtraction. Finally, the left image is rendered onto the right image as a subtractive alpha layer and vice versa. The alpha values are constant but different for each color channel. In effect, this is a linearised version of the subtractive model of Lipscomb and Wooten [LW94]. Although the method works in real-time, it does not take into account the interdependencies between subsequent display frames. Also, with constant alpha values the model is uniform over the screen, and some effort is needed to determine the proper alpha values.

All previous subtractive methods (e.g., [LW94, KLD00, KFNN03]) operate in the RGB color space, and on each of the red, green and blue color channels entirely independently. If the estimated amount of leakage for one of the three color channels is larger than the desired display intensity for that color channel, the best those subtractive reduction methods can do is to set the corresponding color channel to zero. Therefore, previous subtractive reduction methods are unable to reduce crosstalk between different color channels, for example a green object on a red background. The pixel regions where this is the case are said to be *uncorrectable regions*.

2.4 Optical tracking

Tracking in virtual and augmented reality is the process of identifying the pose of an input device in the virtual space. The pose of an interaction device is the 6-DOF orientation and translation of the device. Several tracking methods are in existence, including mechanical, magnetic, gyroscopic and optical. The focus will be on optical tracking as it provides a cheap interface, which doesn't require any cables, and is less susceptible to noise compared to the other methods. Furthermore, given sufficient camera resolution, the accuracy of optical tracking is very good.

A common approach to optical tracking is marker based, where devices are augmented with specific marker patterns recognizable by the tracker. Pose estimation is then performed by detecting the markers in one or more two-dimensional camera images. Optical trackers often make use of infra-red light combined with circular, retro-reflective markers to greatly simplify the required image processing. The markers can then be detected by fast threshold and blob detection algorithms.

Once a device has been augmented by markers, the three dimensional positions of these markers are measured and stored in a database. This database representation of the device is called the model. Optical trackers are now faced with three problems. First, the detected 2D image points have to be matched to their corresponding 3D model points. This is called the point-correspondence problem. Second, the actual 3D positions of the image points have to be determined, resulting in a 3D point cloud. This is referred to as the perspective n-point problem. Finally, a transformation from the detected 3D point cloud to the corresponding 3D model points can be estimated using fitting techniques.

A common and inherent problem in optical tracking methods is that line of sight is required. There are many reasons why a marker might be occluded, such as it being covered by the user's hands, insufficient lighting, or self-occlusion by the device itself. Whenever a marker is occluded there is a chance that the tracker cannot find the correct correspondence any more. Trackers based on stereo correspondences are particularly sensitive to occlusion, as they might detect false matches, and require the same marker to be visible in two cameras

simultaneously.

Many current optical tracking methods make use of stereo correspondences. All candidate 3D positions of the image points are calculated by the use of epipolar geometry in stereo images. Next, the point correspondence problem is solved by the use of an inter-point distance matching process between all possible detected positions and the model. A drawback to stereo correspondences is that every marker must be visible in two camera images to be detected. Also, since markers have no 2D identification, many false matches may occur. There are several implementations of stereo correspondence based optical trackers (e.g., [Dor99, RPF01, vRM05]). Since the focus lies on projective invariant trackers, these are not discussed any further.

More recently, optical trackers have made use of projection invariants. Perspective projections do not preserve angles or distances; however, a projection invariant is a feature that does remain unchanged under perspective projection. Examples of projective invariants are the cross-ratio, certain polynomials, and structural topology. Using this information, the point correspondence problem can be solved entirely in 2D using a single camera image. Invariant methods have a clear advantage over stereo correspondences: there is no need to calculate and match 3D point positions using epipolar geometry, so markers need not be visible in two cameras. This provides a robust way to handle marker occlusion as the cameras can be positioned freely; i.e., they do not need to be positioned closely together to cover the same area of the virtual space, nor do they need to see the same marker.

ARToolkit [KB99] is a widely used framework for projection invariant optical tracking in augmented virtual reality. This system solves pattern correspondence by detecting a square, planar bitmap pattern. Using image processing and correlation techniques the coordinates of the four corners of the square are determined, from which a 6-DOF pose can be estimated. Drawbacks are that ARToolkit cannot handle occlusion and only works with planar markers and four coplanar points. Fiala [Fia05] presented an extended version of this system called ARTag. Marker recognition was made more robust by using an error correcting code. However, the detection of the marker is still based on the detection of a square in the image and, therefore, suffers from similar occlusion problems. Another example of a projection invariant optical tracker is GraphTracker, which is based on projection invariant graph topology and is described in detail in Appendix C.

2.4.1 Tracker performance evaluation

An important issue in optical tracking is an objective way of measuring performance. The user's performance for an interactive task often depends on the performance of the optical tracking system. Tracker accuracy puts a direct upper-bound on the level of accuracy the task can be performed with. Particularly in cases where the tracker cannot detect the input device, for example due to occlusion, interaction performance is reduced significantly. Therefore, many aspects must be taken into account when evaluating the performance of an optical tracker. These aspects include the type of interaction task that is performed; the intrinsic and extrinsic camera parameters, such as focal length, resolution, number of cameras and camera placement; environment conditions in the form of lighting and occlusion; and end-to-end latency. Furthermore, performance can be expressed in a number of different ways, such as positional accuracy, orientation accuracy, hit:miss ratio, percentage of outliers and critical accuracy, among others. Most optical tracker descriptions do not take all these aspects into account when describing the tracker performance.

Van Liere and van Rhijn [vLvR04] examined the effects of erroneous intrinsic camera

parameters on the accuracy of a model-based optical tracker. They recorded a real, interactive task and subsequently ran three different optical tracking algorithms on these images, providing them with varying intrinsic camera parameters to simulate errors in the camera calibration process. They showed how these parameters affect the accuracy, robustness and latency of the tested optical tracking algorithms. The framework presented in this thesis is more general in the sense that many more parameters can be studied than just the intrinsic camera calibration. Since virtual camera images are generated, it is possible to realistically study effects such as lighting conditions, occlusion and varying camera placements.

In the past, several techniques have been proposed to study the properties of multiple camera setups and camera placements for general optical tracking. Two examples are the Pandora system by State, Welch and Ilie [SWI06] and the work by Chen [Che00] on camera placement for robust motion capturing. The Pandora system [SWI06] allows the user to set varying extrinsic and intrinsic camera parameters and projects a visualization of these parameters on a virtual scene. Every virtual camera projects a resolution grid on the scene using shadow mapping. In this way, the user can explore the virtual scene and examine, for example, which parts of the scene are visible and at which resolutions for different camera placements. Chen [Che00] proposed a quantitative metric to evaluate the quality of a multi-camera configuration in terms of resolution and occlusion. A probabilistic occlusion model is used and the virtual space is sampled to determine optimal camera placements. The focus lies on finding a robust multi-camera placement for general motion capturing systems.

Chapter 3

Design and Implementation of the PDL Architecture

The PDL architecture updates and renders a scene graph at the display refresh rate. This is achieved using a parallel client and server process, which access a shared scene graph (see Figure 3.1). The client is responsible for generating new application frames at its own frame rate depending on the scene complexity. The server performs constant-frame-rate image warping of the most recent application frames, based on the latest state of the scene graph. The image warping itself is primarily based on a combination of earlier proposed image-warping techniques. In particular, the core image-warping equations consist of a modified version of the image-warping equations proposed by McMillan, Mark and Bishop [MB95, MMB97] adapted for modern graphics hardware and dynamic scenes, and written in matrix form.

An important aspect of this architecture design is the shared scene graph. Both the client and the server have full access to the complete scene graph state and geometry. Each pixel in the client's rendering output is tagged with an object identification number from the scene graph. This enables the server to use the latest available pose information from the corresponding object in the scene graph for warping. It further allows for parts of the geometry to be rendered directly on the server. Previous image-warping architectures did not share the scene graph. The use of object IDs by themselves to detect discontinuities in the warped output has been proposed [Mar99] but was not widely adopted.

Compared to previous image-warping methods, there are a number of benefits to the approach presented here using a shared scene graph and per-pixel object IDs:

- The latest pose for objects as well as cameras can be re-sampled from the input devices or animations in the scene graph. Consequently, late input device re-sampling is no longer restricted to viewpoint updates alone and allows for latency reduction at the object level and for dynamic scenes. This enhances the flexibility of the image warping architecture and improves the interactivity for all input devices, not just the head tracker.

- Certain scene graph updates, such as object selection, can be performed directly at 60 Hz by the server performing image warping, without waiting for the next client frame. This facilitates user interaction.

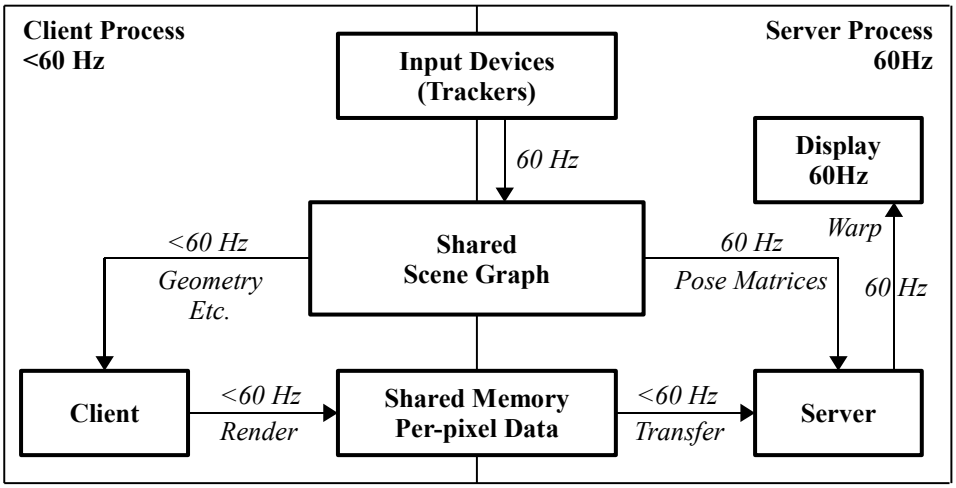

Figure 3.1: Overview of the proposed real-time image-warping architecture. A client and server process run in parallel, where the client generates new application frames at a rate lower than 60 Hz and the server produces new display frames at 60 Hz. The scene graph is shared to allow the server access to geometry and the latest pose information.

- The server can directly render parts of the geometry contained in the shared scene graph, instead of only performing image warping. In this way, errors in the image-warping output can be detected and the corresponding geometry can be re-rendered at these locations by the server. This increases the quality of the output.
- The implementation of the PDL architecture is realized using commodity components only; no special hardware is required. Furthermore, there are no limitations to the number or types of primitives rendered. Because of this, and the fact that the image warping equations are formulated in a matrix form standard to VR, existing applications require only minimal modifications to make use of the PDL architecture.

The combination of these benefits indicates that the PDL architecture can provide realistic latency and judder reduction, for head-tracking as well as for general 6-DOF input devices, in common, practical VR-environments.

3.1 Implementation

The PDL architecture has been implemented on both a single- and a multi-GPU system. Most of the details are the same for both implementations, with the exception of the data transfers described in Section 3.1.3. An overview of the architecture's hardware implementation for a multi-GPU implementation is given in Figure 3.2. The GPUs are connected over the PCIe bus and communicate using a large segment of shared system memory. The first GPU, which is called the *client*, is responsible for rendering application frames. These frames contain a fixed number of rendered scenes from various viewpoints. The viewpoints used for rendering are not necessarily equal to the user's viewpoint. All application frame data is transferred over the PCIe bus to shared system memory. The second GPU, which is called the *server*, is responsible for generating intermediate display frames and updating the display device.

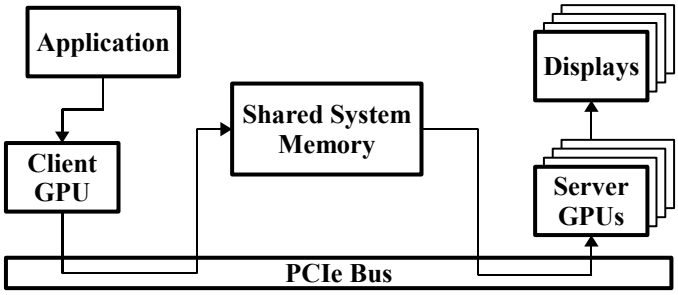

Figure 3.2: Hardware overview of the PDL image-warping architecture as a multi-GPU implementation. The application runs on a client GPU and sends rendered data to shared system memory over the PCIe bus. Server GPUs use this data to generate updated display frames through image warping. An implementation could support multiple server GPUs in order to drive several displays, such as tiled displays or CAVE-like setups. The current PDL implementation uses a single server GPU to drive a single desktop-VR display.

Whenever the display needs to be refreshed, image warping is used to generate an updated display frame from the latest client data. For a single-GPU implementation the distinction between the client and server is similar, except that both run on the same physical GPU and share its memory instead of system memory.

3.1.1 Client

The purpose of the client is to render images from different viewpoints, in such a way that the server can use this data to perform effective image warping. The data generated by the client for this purpose is called a *client frame*. To generate the required data, the client renders the scene from multiple viewpoints. The current implementation uses two viewpoints per client frame, but depending on application requirements more can be added. At the start of a client frame, a buffer slot is acquired for data storage (see Section 3.1.3). Next, the input devices are sampled to determine the latest camera and object poses. Since stereoscopic displays are assumed, a left- and a right-eye view are rendered for the obtained camera pose. To avoid clipping at the borders of the screen after warping, the cameras use an enlarged field-of-view. The eye-separation is also set to be larger than normal in an attempt to avoid some warping occlusion artefacts (see Section 4.2 for more details on client-side camera-placement strategies). For each viewpoint, the inverse of the corresponding camera's projection and modelview matrices are stored in the buffer slot as $C_{img \to cam}$ and $C_{cam \to wld}$, respectively. An overview of the mathematical symbols that are used to describe the image-warping implementation is given in Table 3.1.

Next, the scene graph is rendered for each of the viewpoints. For every geometric object that is to be rendered, a static 16-bit object ID i is assigned to that object and the inverse of the corresponding object matrix is stored in an array in the buffer slot as $C^i_{wld \to obj}$. The scene is rendered using a custom shader program that outputs per-pixel color, normal, depth and object IDs. The pixel's RGB color is stored in a 4-component 8-bit BGRA format, where the alpha component is used for the low eight bits of the 16-bit object ID, which is split into two 8-bit parts for storage and retrieval efficiency. The normal is converted to the $[0,1]$ interval for each of three components and stored as 16-bit fixed point. No shading is performed since

$C_{x\to y}$	Client-side 4x4 transformation matrix, transforming from space x to space y. The coordinate spaces used are the 3D world space (wld), 3D object space (obj), 3D camera space (cam) and 2D projected image space (img).
$S_{x\to y}$	Server-side 4x4 transformation matrix, transforming from space x to space y
$C^i_{x\to y}$	Client-side 4x4 object transformation matrix for the object with ID i. Only applicable to $obj \to wld$ and $wld \to obj$ transforms.
$S^i_{x\to y}$	Server-side 4x4 object transformation matrix for the object with ID i.
M^i	4x4 warping matrix for the object with ID i, mapping a corresponding pixel from client-side image space to server-side image space.
M^i_n	The n-th column of the 4x4 matrix M^i
M^i_{nm}	The element at row m and column n of the 4x4 matrix M^i
N^i	4x4 normal matrix for the object with ID i, mapping a corresponding normal from client-side camera space to server-side camera space.
P_{xyzw}	Client-side post-projection pixel represented as homogeneous 4-vector. The subscript indicates a combination of individual elements. x, y and z represent the 3D spatial coordinates and w the homogeneous coordinate.
P'_{xyzw}	Pixel warped from source pixel P_{xyzw} to a server-side post-projected pixel as 4-vector
n	Client-side 4-vector normal with homogeneous component set to zero.
n'	Server-side warped 4-vector normal with homogeneous component set to zero.
S	Size of a pixel in post-projective space.
s	Warped pixel's splat size computed from the Jacobian.

Table 3.1: A summary of the mathematical symbols used for image warping.

deferred shading will be done on the server using the stored normal. Post-projection depth information is converted to $(0,1)$ and stored in a 16-bit fixed point integer format as $2^{16} \cdot z/w$. Finally, the high eight bits of the 16-bit object ID are stored in a single 8-bit component. This results in a total required storage space of 13 bytes per pixel (an 8-bit 4-vector, a 16-bit 4-vector and a single 8-bit value).

The client optionally generates a 3D motion field corresponding to the optic flow of the scene. This motion field can be used to determine good client-side camera placements, which is described in Section 4.2. This motion field can be used to perform image warping as well [SvLF08], but a shared scene graph approach appears to be superior; however, motion-field-based image warping may still be of use when a shared scene graph is not available. For every geometric object that is to be rendered, the client stores its transformation matrix for the previous application frame M_{prev} and its current transformation matrix M_{cur} in a hardware vertex program. Then the object is sent to the GPU for rendering using a vertex and fragment program. For each vertex V_i, the vertex program calculates the previous position $P_{prev} = M_{prev} \cdot V_i$ and the current position $P_{cur} = M_{cur} \cdot V_i$. Both vectors are then passed to the fragment program, where they are automatically interpolated per-pixel over the triangle by the hardware. For every pixel, the fragment program calculates the value $\Delta P = P_{cur} - P_{prev}$, resulting in a 3D motion vector per pixel. Since vertices are transformed into camera space,

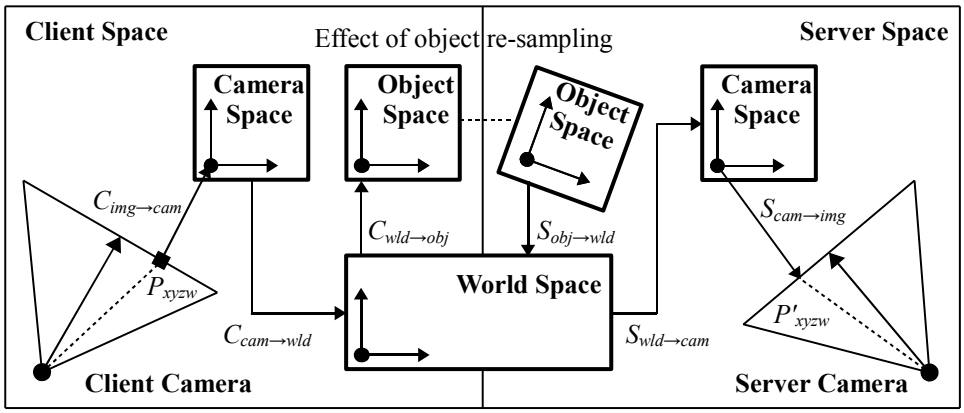

Figure 3.3: Schematic overview of warping a client pixel P_{xyzw} to the server pixel P'_{xyzw}. Resampling of object poses is essentially achieved through the server's object-to-world transform. All six required matrices are concatenated into a single warping matrix for each different object. Also see Figure 2.3.

the motion vector ΔP also resides in camera space. This approach is similar to the ones proposed by Wallach et al. [WKC94] and Cheng et al. [CBPZ04].

3.1.2 Server

The server starts by polling the client in a non-blocking fashion to determine if a new client frame is available. Details of different approaches to achieve this are given in Section 3.1.3. If a new frame is available, it is used as the source for warping; if not, the server continues using the previously received frame. A ratio is kept that indicates the number of frames the server renders until a new client frame becomes available; for example, if the server renders three frames for every client frame, this ratio equals 3:1. Next, a Δt value can be computed according to the number of times a new client frame was not available since the last received frame and the kept ratio. This value is useful for motion extrapolation, which is described below. Finally, the server generates a left- and right-eye stereoscopic view from a newly sampled head-tracker pose $S_{wld\to cam}$ by warping the two client viewpoints for each view individually.

Let P_{xyzw} be the 3D homogeneous coordinates of a pixel in the client's post-projection space and i the pixel's corresponding object ID. It is now possible to warp the pixel to the server's new viewpoint by first unprojecting the pixel, transforming it back to world-space, applying object transforms and finally reprojecting it using the server's camera: $P'_{xyzw} = M^i \cdot P_{xyzw}$ with

$$M^i = S_{cam\to img} \cdot S_{wld\to cam} \cdot S^i_{obj\to wld} \cdot C^i_{wld\to obj} \cdot C_{cam\to wld} \cdot C_{img\to cam}$$

The matrices denoted by S are the server's projection, camera and object matrices, and the ones denoted by C are the client's corresponding matrices. The procedure is depicted in Figure 3.3. The per-pixel normals are stored in the client's camera space. For the purpose of deferred shading, the normals need to be transformed to the server's updated camera space. This is achieved by computing the following normal-matrix:

$$N^i = ((S_{wld\to cam} \cdot S^i_{obj\to wld} \cdot C^i_{wld\to obj} \cdot C_{cam\to wld})^{-1})^T$$

The matrices M^i and N^i are calculated for each object once per server view, and are then uploaded to the GPU. In this way, a GPU image warping algorithm can warp a pixel or a normal by a single 4x4 matrix multiply, where the required matrix is pre-computed per-object instead of being computed for each pixel. Furthermore, all the warping equations are expressed in terms of general 4x4 homogeneous matrix computations, allowing for arbitrary transforms to be easily inserted in the warping pipeline and effortless integration with the standard OpenGL rendering pipeline.

So far the significance of the server's object matrices $S^i_{obj \to wld}$ has not yet been mentioned, nor how they are calculated. If the server's object matrix for any particular object is equal to the client's object matrix for that object, the object will only appear to change pose whenever a new client frame becomes available. This results in non-smooth motion, or judder, and high latency. However, using image warping, the server's object matrices as well as its camera matrix can be modified freely to reflect their latest pose. Examples of achieving this are sampling the newest pose from a 6-DOF interaction device, updating the matrix from an animation in the scene graph, or extrapolating object motion based on the previous pose. In order to extrapolate object motion, the server keeps the client's object matrices for the previous client frame as well as for the current client frame. Using these two object poses and the previously calculated Δt value, the pose can be extrapolated by performing a quaternion spherical linear extrapolation on the rotational part and a regular linear extrapolation for the translational part. In effect, this is a linear prediction. Note, no such prediction is performed for object poses that can be updated in the scene graph at 60 Hz.

3.1.3 Data transfers and synchronization

A mechanism is required to move data from the client to the server and to synchronize these transfers. The most efficient way to do this depends heavily on whether the architecture is implemented on a multi-GPU or on a single-GPU system. In the former case, the data needs to be transferred from the client to the server GPU over the PCIe bus, while in the latter case buffer data can be left in the video RAM of the single GPU and need not be explicitly transferred. For either case, buffer synchronization is required. Due to this dependence on the underlying system, different implementations for data transfers and synchronization are used on a multi- and single-GPU system.

A straightforward implementation would use direct, synchronous data transfers; however, the time required for synchronous transfers turns out to be a bottleneck. A way to avoid this bottleneck is to make use of indirect, asynchronous data transfers. The architecture runs four threads: a client rendering thread, a server warping thread, a client frame downloading thread and a server frame uploading thread. Each of these threads are executed in parallel on a quad-core CPU. The client rendering thread and the server warping thread each open a separate OpenGL context on a different GPU and share this OpenGL context with either their corresponding down- or uploading thread, respectively. Client frames are written to and read from a circular producer-consumer buffer containing a number of slots where client frames can be placed. With such thread-synchronized buffers, a request can either be made for a new slot to write to, or a new slot to read from, optionally in non-blocking fashion. Since separate GPUs can not share memory directly, three instances of such circular buffers are needed: one on the video RAM of the client GPU, one in system shared memory and one in the video RAM of the server GPU. The slots in shared system memory can be accessed by data pointers, while the slots in video RAM are accessed by OpenGL buffer object IDs. These producer-consumer buffers are shared between the threads to allow communication;

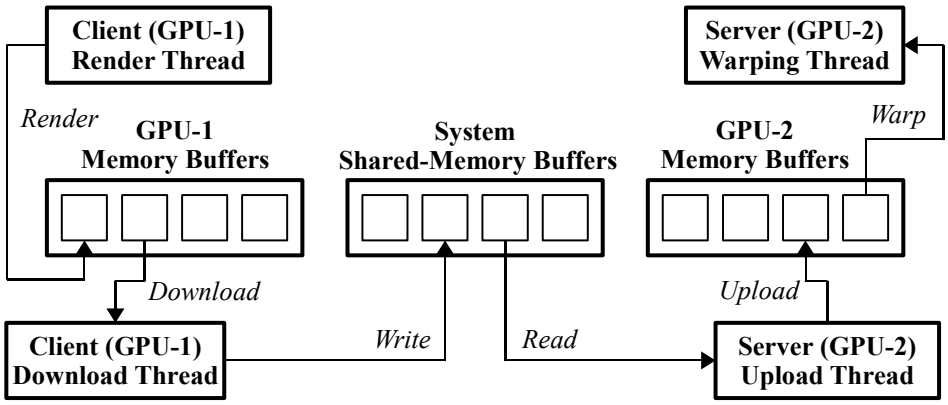

Figure 3.4: Schematic overview of the implementation of asynchronous data transfers for image warping. A situation is shown where every thread is working on a different buffer slot. In practice, other situations may occur; for example, the server will often be using the previous slot used for client rendering since data transfers are relatively fast and client rendering relatively slow.

the client's GPU buffer is shared by the client's render and download threads, the shared system memory buffer by the client's download and the server's upload threads, and the server's GPU buffer by the server's upload and the server's warping threads. This is shown schematically in Figure 3.4.

The transfer of client frames to the server GPU now proceeds as follows. First, the client render thread acquires a free slot to write to from its producer-consumer buffer in client GPU space, after which it renders the frame to that GPU buffer. The client's download thread polls for newly rendered buffers and downloads them to system shared memory. Next, the server's upload thread will detect a new buffer in shared memory and proceeds to upload it to the server GPU. Finally, the server warping thread detects that a new frame is available in its GPU memory, and switches to that GPU buffer as the source for image warping. All these threads operate asynchronously and data transfers are performed using the OpenGL PBO extension's DMA transfers. The effect of transferring data in this fashion is that the image-warping server does not need to spend time on uploading data to the GPU; however, new client frames will arrive a short time later due to background transfers. An important observation is that updates received by the server of newly rendered client frames were already infrequent (say every 200ms), so an extra delay of a few milliseconds is hardly significant with respect to the total update time between new client frames; on the other hand, since only 16ms are available for image warping, had this delay of a few milliseconds occurred then, even such a small delay would have a significant impact on image-warping performance, taking up a large percentage of the available 16ms. With asynchronous transfers, the server does receive new client updates with some delay, but it can still continue to perform image warping in the meanwhile; therefore, the delay slightly reduces image quality due to the larger warping distance between slower client updates, but it has minimal impact on warping performance. The resulting latency is unaffected because the server still manages to perform image warping for every display frame.

For the single-GPU implementation, no data transfers are required; therefore, it suffices to synchronize multiple buffers that are kept in the GPU's video RAM. Two threads are now

used: one for the client's rendering and one for the server's warping. Both threads open an OpenGL context on the same GPU, which is shared to allow accesses to all resources from either thread without copying data. The same circular producer-consumer buffer as described previously is used; however, only a single instance is now required that manages the frames in the single GPU's video RAM.

3.1.4 Image warping

Image warping is performed entirely on the GPU by custom vertex, geometry and fragment shaders. Rendering begins by drawing a static, screen-aligned grid consisting of a number of evenly spaced vertices equal to the client's resolution. There is a one-to-one correspondence between vertices in the grid and pixel-centers in the client texture. Therefore, per-pixel image warping can be achieved by transforming the individual vertices in the grid. Each vertex contains the post-projection 2D coordinates $P_{xy} \in [-1,1] \times [-1,1]$ of the corresponding pixel's center. Other per-vertex information that is available from the buffer generated by the client is the post-projection depth $P_z \in (0,1)$, the object ID i, the normal n and the pixel's color. Furthermore, it is assumed that the homogeneous coordinate $P_w = 1$. Each vertex is warped to its new location in a vertex shader. First, the pre-constructed warping matrix M^i, corresponding to the pixel's object ID, is fetched from GPU memory with four texture reads. Since the combination of the pixel's 2D coordinates and depth results in a valid vertex in post-projection space, it can be warped easily by pre-multiplying it by the warping matrix: $P'_{xyzw} = M^i \cdot P_{xyzw}$. Homogeneous or perspective division will occur later in the OpenGL pipeline, so this P'_{xyzw} can be sent directly to the fragment shader.

Next, the projected size of the warped pixel needs to be determined. Suppose that the depth for a given pixel with coordinates (x,y) is given by a two-dimensional function $z = f(x,y)$. In that case, the warp can be seen as a general 2D coordinate transform: $P'_{xyzw} = M^i_4 + M^i_1 x + M^i_2 y + M^i_3 f(x,y)$, where M^i_n is the n-th column of M^i. This follows directly from $M^i \cdot P_{xyzw}$. The expansion factor of this transform is given by the Jacobian, which is the determinant of the Jacobian matrix of partial derivatives, after homogeneous division:

$$J = \begin{pmatrix} \partial(P'_x/P'_w)/\partial x & \partial(P'_y/P'_w)/\partial x \\ \partial(P'_x/P'_w)/\partial y & \partial(P'_y/P'_w)/\partial y \end{pmatrix}$$

After simplification of the derivatives and substitution by the elements of P'_{xyzw}, it is found that:

$$J_{11} = \frac{P'_w M^i_{11} - P'_x M^i_{41} + (P'_w M^i_{13} - P'_x M^i_{43}) \cdot \partial f(x,y)/\partial x}{(P'_w)^2}$$

The other three elements are found through very similar equations. The partial depth derivatives for $f(x,y)$ can be approximated by using the depth buffer gradients: $\partial f(x,y)/\partial x = (f(x+1,y) - f(x-1,y))/2$, and so on. However, a more robust depth derivative can be computed from the pixel's original normal n as $(Sn_{xy})/n_z$ where S is the size of a pixel in post-projective space, which is $(2/w, 2/h)$ for a resolution of wxh. This final form is equivalent to the warping equations used by McMillan [MB95]; however, the ones here are written in general 4x4 homogeneous matrix form for easy concatenation of transforms. In this way, the Jacobian matrix can be computed quite efficiently in the vertex shader, given that P'_{xyzw} is already computed. Finally, the pixel's normal must be warped to camera space for the purpose of deferred shading. This is achieved by pre-multiplying the normal by the pre-computed normal matrix $n' = N^i \cdot n$.

Figure 3.5: A quality comparison of a 60x60 pixel close-up of a 10M polygon statue model. From left to right: point-splat, quad-splat and mesh-based image-warping methods and the directly rendered reference. Some noise appears for the splat-based methods due to overlapping splats; however, this is generally not very noticeable when viewed from a distance. The mesh-based result shows slightly too much blurring.

Three different warping algorithms that use the same warping equation but vary in the way the warped pixels are rendered have been implemented: a screen aligned point splat, a general quadrilateral splat and a mesh-based reconstruction. The point-splat method computes the Jacobian matrix and then determines a per-pixel splat size $s = ceil(max(J_{11} + J_{12}, J_{21} + J_{22}))$. This size is set in the vertex shader on a per-pixel basis using the *gl_PointSize* variable; hence, this method can only render screen aligned squares of size *sxs* pixels. The splat size is ceiled to avoid hole artefacts resulting from discrete splat sizes that are too small to cover the surface entirely. A drawback is that this generates slightly thicker edges and overlaps individual splats. A more flexible approach is the use of a general quadrilateral splat. In this case, the geometry shader inputs a pixel to be warped and outputs a single quad. The four vertices of this quad are computed by post-multiplying the Jacobian matrix by half-pixel offsets $(x \pm 0.5, y \pm 0.5)$ of the warped pixel's position. This results in less overlap for individual splats. Note that the previously described discrete point splat size S is equal to the smallest square completely covering this quad. Finally, a mesh-based reconstruction similar to the one proposed by Mark and McMillan [MMB97] treats the grid of pixels as a connected triangle mesh. Each vertex in this grid mesh is warped, and the fragment shader renders connected triangles accordingly. Because the grid is completely connected, there is no need to calculate a splat size and the Jacobian matrix need not be computed.

Sample output of these three reconstruction methods is given in Figure 3.5. In general, the perceived quality of point splats and quad splats was found to be roughly equal — with quad splats showing slightly better quality in smooth shaded regions. The quality of the mesh-based approach appeared to be better than either point or quad splats, although it resulted in slightly too much image blurring. The additional blur may have given a false sense of better image quality, which is not equal to the reference.

3.2 Performance

In this section, the run-time performance and the observed latency are reported for the implementation of the PDL architecture. Both a single- and a multi-GPU implementation are evaluated.

Multi-GPU								
	640×480		**800×600**		**1024×768**		**1280×960**	
	Warp	**Update**	**Warp**	**Update**	**Warp**	**Update**	**Warp**	**Update**
Point	5.8	212	9.0	214	14.7	218	23.1	222
Quad	10.2	212	15.9	214	25.9	218	41.0	222
Mesh	15.7	212	24.5	214	40.3	218	63.2	222
Single-GPU								
Point	9.2	900	11.8	1150	16.3	1586	25.9	2325
Quad	14.5	1426	20.1	1883	30.3	2857	44.5	4319
Mesh	48.7	654	76.0	655	123.5	659	193	672
Regular	207.0		207.1		207.4		207.9	

Table 3.2: Frame times for single- and multi-GPU implementations in milliseconds (60 Hz corresponds to 16.6ms of available processing time). Two client views are warped to produce a stereoscopic image; therefore, image warping needs to performed four times. The bottom row shows the time required for a regular stereoscopic renderer to produce the same images. For each resolution, the first column shows the time required for warping on the server, while the second column represents the time between fresh buffer updates arriving at the server from the client. The pipeline of a single GPU is quickly congested under heavy warping loads. In combination with the scheduling overhead, this leaves little time for new client updates to be rendered. (Intel Core2 Quad Q9950 2.83Ghz + 2x NVidia GeForce GTX 260)

3.2.1 Frame rate

The performance of the multi- and single-GPU implementations of the warping architecture has been evaluated on a system consisting of of an Intel Core2 Quad Q9950 2.83Ghz and two NVidia GeForce GTX 260 video cards connected over a PCIe 2.0 16x bus (see Appendix A). For the single-GPU implementation one of these video cards was disabled. The test scene consisted of the 28M polygon Stanford Lucy angel model orbited by twelve torus knots. Table 3.2 shows that a regular stereoscopic renderer would take 207 ms to render this scene in stereo. The output of this regular renderer is used as reference images to compare the quality of the output produced by image warping.

For each of the three reconstruction methods, point-, quad- and mesh-based, the server's image-warping frame rates are given in milliseconds per frame in Table 3.2 for various resolutions, both for single- and multi-GPU implementations. In all cases stereoscopic image warping using two client views is performed. Note, to achieve a 60 Hz frame rate, image warping must be performed in less than 16.6ms. From these data it can be deduced that mesh-based reconstruction is too slow to be practically used currently, as is quad-based splatting for higher resolutions. The performance of point splatting is higher than the other reconstruction methods, while resulting in almost the same image quality. Therefore, point splatting is used exclusively throughout this thesis; although, in the future, the other techniques may become feasible. Using point splatting, a 60 Hz frame rate can be guaranteed for stereoscopic resolutions up to 1024x768, both for single- and multi-GPU implementations. Performance was found to be linear in the number of warped pixels.

Every second column in Table 3.2 gives the time between consecutive client frame updates that arrive at the server. While there is no strict performance threshold here (such as 16.6ms for the server), the longer it takes for new client frames to arrive as a source for image warping, the larger the image warping errors become. For the multi-GPU implementation

	Reference			Multi-GPU			Single-GPU		
Faces	**latency**	σ	**time**	**latency**	σ	**time**	**latency**	σ	**time**
1M	48.5	3.5	9.6	71.9	5.1	19.7	95.0	4.4	27.1
2M	68.5	0.5	18.8	74.0	3.5	19.8	89.9	3.5	27.3
4M	113	5.3	37.0	65.4	5.4	16.9	95.0	3.1	27.1
8M	220	8.4	72.9	58.6	2.8	15.4	96.2	2.3	27.4
16M	409	12.3	145	62.2	2.4	14.6	94.4	4.6	27.4
32M	780	14.2	289	61.1	2.5	14.3	95.6	4.8	27.6
64M	x	x	577	58.9	2.6	14.1	93.8	4.0	27.6

Table 3.3: Latency, standard deviation of latency and time required to warp, all in milliseconds, for a reference renderer and both the single- and multi-GPU image-warping implementations. In all cases a stereoscopic image is produced from two client views and no hole filling is performed. The reference latency for 64M faces could not be measured because it was larger than half the period of the pendulum used for measuring, introducing aliasing frequencies. (Intel Core2 Quad Q6600 2.4Ghz + Nvidia GeForce 8800GTX + Nvidia Quadro FX5600 + Iiyama Vision Master Pro 512 @ 60 Hz stereo + Polhemus Fastrak @ 120 Hz)

these updates times are stable and are equal to the rendering time (207 ms) plus the time taken to perform asynchronous, background data transfers. However, for the single-GPU implementation the update times vary greatly. This is due to the sharing of a single GPU for both warping and client rendering and the implied scheduling and context switching. Since only a short time is available for client rendering, the relative impact of the overhead of the context switch becomes larger. It can be seen that the larger the server's warping load becomes, and the more closely it fills the allotted 16.6ms per frame, the less time available to render new client frames. Therefore, the ratio between server and client frames is increased, resulting in an increase of warping errors.

3.2.2 Latency

Latency has been evaluated for both single- and multi-GPU image-warping implementations and a stand-alone regular renderer for reference using the method described in Appendix A. The hardware consisted of an Intel Q6600 2.4 Ghz quad-core processor system using an Nvidia GeForce 8800 GTX for the client GPU and a stereo-enabled Nvidia Quadro FX5600 for the server GPU (see Appendix A). In the case of a single-GPU, both client and server used the NVidia Quadro FX5600. Both the image-warping server and the reference system rendered the scene as normal for the left and right eye and then cleared the display in order to render a small sphere at the position of the input device for latency measurements. The reference system sampled the input device just before rendering the scene, while the image-warping server re-sampled the input device prior to warping using the previously described algorithms. The scene consisted of various numbers of polygons.

The acquired results are listed in Table 3.3. This shows the average latency acquired over several sampling runs of the experiment, as well as the standard deviation over those samples. It can be seen that the latency for the image-warping server is low and almost constant, regardless of the number of polygons. The single-GPU's latency is somewhat higher due to longer warping times but still constant. Both low and constant latency are desirable properties for an interactive system and enable the use of further predictive filtering methods [OCMB95]. The reference renderer's latency is much higher and depends on the number

of polygons in the scene and the frame rate. Furthermore, the standard deviation of the reference is also much higher, indicating non-constant latency that would be hard to predict. When the number of polygons is reduced to about 4M and lower, the multi-GPU latency increases slightly. This is because client frames are updated almost every frame now, and the frequent asynchronous data transfers and context switches between the warp and upload thread start affecting warping performance. This has no effect on a single-GPU because no data is transferred. For small scenes it is probably better to use direct rendering; however, the data shows that when more than two million polygons are to be rendered, image-warping methods can reduce the latency.

Chapter 4

Reducing Image-warping Errors

In this chapter, a number of extensions are described to the basic image-warping implementation on the PDL architecture, with the purpose of avoiding or reducing errors in the warped output. These extensions are beneficial in a number of ways:

- A method is introduced to dynamically detect errors in the output at run-time. Once detected, an attempt is made to resolve these errors by directly ray tracing parts of the shared scene graph. This significantly improves image quality at the expense of longer processing times.
- Intelligent client-side camera placements provide the means to avoid errors in the first place. The camera-placement strategy can optionally be based on the amount of dynamically detected errors. This directly improves image quality and increases performance because fewer errors need to be resolved.
- The method used to detect errors can be modified to function as an off-line quantitative evaluation method for image-warping algorithms. In this way, different algorithms can be compared in an objective fashion, and cases where the algorithms require improvement can be identified.

4.1 Dealing with errors

Image-warping techniques generally introduce a number of errors in the output images. The magnitude of errors depends on the distance between the warped source view and the resulting target view and, consequently, the client's frame rate. Two types of errors can be distinguished:

- **Sampling errors** are caused by the use of image warping to reconstruct continuous surfaces using only discrete pixel information. Generally, these sampling errors manifest as small errors in shading and slightly thicker edges at sharp depth boundaries. When the distance between the source and target images is large, a loss of resolution can also be distinguished. However, in practice these degradations of image quality are not very noticeable, especially not for animated scenes; therefore, the focus lies on occlusion errors.

- **Occlusion errors** are visible as holes in the image, which are caused by occluded geometry in the warped source images. Figure 4.2 depicts what these errors look like in a practical application. Experience indicates that occlusion errors are very disturbing, especially in animated scenes, and should be minimized as much as possible.

In the past, a number of hole-filling strategies have been proposed [Mar99]. A post-processing step can be applied after warping to first find hole pixels and then resolve or fill them. The problem with previous methods is that not all hole pixels are found correctly, and the methods for resolving holes that are found do not work well for dynamic scenes and a GPU implementation. Therefore, a novel way of both detecting and resolving holes in dynamic scenes has been implemented.

4.1.1 Detecting occlusion errors

To detect hole pixels, it must be defined what exactly constitutes a hole pixel. Three types of holes can be distinguished:

- Level-0 holes are those destination pixels where no pixels warps to; i.e., pixels that are left blank in the final image. These are the classic holes found by methods as described by Mark et al.[Mar99].

- Level-1 holes are pixels where an object is visible, but not the correct object. This case occurs frequently in dynamic scenes, or when using multiple client views, when a part of geometry is missing in the warped image due to an occlusion error and part of some different geometry warps into this gap, behind the geometry that should have been there.

- Level-2 holes are pixels that do show the correct object, but not the correct part of it. For rigid object transforms this case occurs infrequently.

Level-0 holes can easily be detected by scanning the warped output image and marking every pixel that does not have some flag set to indicate it is the destination of a warped pixel. Detecting level-1 and level-2 holes requires more effort. The governing idea is to compare a warp map consisting of object IDs and depth of warped pixels to a reference map of which IDs and depth should be at those locations. If the object IDs in the two maps do not match, then the pixel is flagged as a level-1 hole. If they do match, but the warped pixel is further away than the depth of the bounding box, up to a threshold, a level-2 hole has been found. In this way, level-0 holes will always be found because their object ID of zero does not occur in the scene graph. A schematic overview of a number of different cases that can be encountered when detecting occlusion artefacts is given in Figure 4.1.

The current implementation outputs per-pixel object IDs and depth for every warped pixel into a separate buffer making up the warp map. Per-pixel object IDs are directly available from the regular shared-scene-graph warping, so this step is trivial. Next, the reference map is constructed. To obtain a perfect reference map, the entire scene would have to be rendered, so an approximation is used instead. To obtain this approximation, a kd-tree is precomputed for every object in the scene. This kd-tree represents a convex-hull bounding-box hierarchy of increasing resolution, with leaf nodes consisting of small subsets of the geometry. A convex-hull approximation is now constructed for each object by rendering all the bounding boxes at a specific resolution in the corresponding kd-tree. Rendering several thousand of these bounding boxes can be achieved in about one or two milliseconds using hardware instancing,

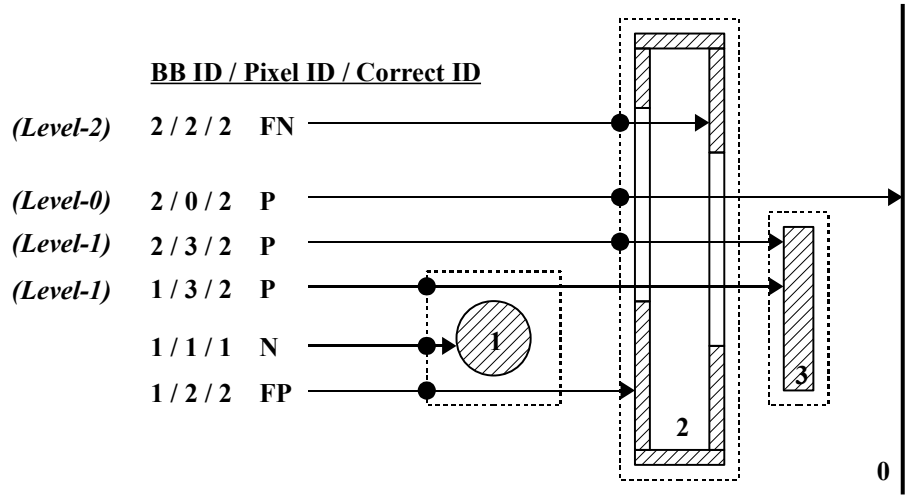

Figure 4.1: A number of cases that can occur when detecting occlusion artefacts. Three objects are shown with object IDs of one, two and three. Pixels where nothing is rendered receive object ID zero. Two occlusion artefacts are schematically shown in object number two. While this particular configuration of occlusion artefacts might not be possible in a real image-warping system, it serves as a good example to illustrate the different cases. The dashed outlines around the objects represent the bounding boxes in the kd-tree. The table on the left gives three different object IDs: the ones detected from the bounding boxes by shooting rays into the kd-tree, the ID of the warped pixel displayed, and the ID of the pixel that should have been displayed. A hole is detected when the former two IDs are not equal. A few cases are depicted for level-0, level-1 and level-2 holes: false negatives (FN) where an existing hole is not detected, positives (P) where a hole is correctly identified, negatives (N) where the absence of a hole is correctly detected, and false positives (FP) where a non-existing hole is erroneously detected.

so this quickly gives an approximated reference map of object IDs and depth. Because of the nature of image warping and occlusions, hole pixels always have a larger depth value (further away from the viewer) than the correct geometry would have. Therefore, holes can be detected by comparing the warp map to the approximated reference map made up of convex-hull bounding boxes.

As an optimization, the entire background quad is always redrawn after warping to solve many common holes. A hole-detection optimization is also possible here. If the approximated reference map indicates that a pixel is a background pixel, then this must be so due to the convex hull nature of the kd-tree bounding boxes. Therefore, if the warped pixel also turns out to be a hole, it can be safely skipped since it must be a background hole that will be filled anyway by redrawing the background quad. This reduces the number of candidate holes found significantly. The same approach is not possible when only scanning for level-0 holes, since there is no distinction between holes and background then.

A problem with the described hole detection implementation is that it will detect some amount of false-positives; i.e., pixels that are flagged to be holes, but in reality are not. This happens most frequently at the edges of objects for the level-1 holes, and at sharp depth edges for level-2 holes. The reason is that the bounding boxes cover a larger portion of screen-space

than the actual geometry, and these pixels receive incorrect object IDs and depth from the approximated reference map. Therefore, for performance reasons, the algorithm is restricted to finding level-1 holes only and to comparing object IDs only, as this reduces the amount of false-positives. Note, however, that detecting false positives only costs performance, and not image quality, after trying to resolve these holes.

A current limitation is that level-2 holes will not be detected correctly and result in false negatives. A possible solution is to subdivide objects into multiple object sections with different object IDs; however, a drawback to this approach is that it results in more false positives.

4.1.2 Resolving occlusion errors

In the previous section it was described how to find a bitmap of candidate hole pixels. Now the found holes will be resolved by directly rendering in the missing geometry on the image-warping server using the shared-scene-graph geometry. The idea is to shoot rays through the hole pixels, trace those rays through the pre-computed kd-tree to find the intersected leaf nodes, and to re-render these complete leaf nodes that form a super-set of the intersected geometry. Rendering entire leaf nodes, which in the current implementation consist of approximately 1K polygons, avoids the need to test every face in the node to find an exact intersection, which is too slow in practice. However, it does cause more polygons to be redrawn, but this can usually be done much quicker on a GPU.

To resolve holes, the hole map needs to be downloaded to the CPU. To reduce the size of data, blocks of 4×4 pixels are tested for holes on the GPU, and the output of these 16 tests is combined into a single 16 bit unsigned integer texture. The hole map now becomes so small that it can be downloaded to the CPU in a fraction of a millisecond. Next, the 16-bit integer hole map is iterated over on the CPU, searching for integers that are not equal to zero, indicating at least one of the pixels in the corresponding 4×4 block is a hole pixel. This saves many comparisons for the generally sparse hole map. A straightforward approach would be to test every pixel in the block and trace a ray through it if that pixel's bit is set, indicating it to be a hole; however, this requires many rays to be traced, degrading performance. Therefore, only a single ray is traced through the center of the 4×4 block of pixels. The corresponding rays for the other potential hole pixels in the block are so close to this center ray that they usually intersect the same leaf nodes. Since these leaf nodes are to be re-rendered anyway, it usually suffices to trace only the single center ray. This approach may cause the right geometry to be missed, leaving the hole unresolved, but it greatly improves performance. Next, the leaf nodes in the kd-tree that the center ray intersects are determined, and the indices of these leaf nodes are stored in a unique set, ensuring that the same leaf node is never re-rendered twice.

Due to the structure of the kd-tree, intersected leaf nodes are found in front-to-back order along the ray; however, this does not guarantee that the first-found intersected leaf node contains the correct geometry to resolve the hole. It is possible for the ray to hit the bounding box for the geometry contained in the leaf node, but not to hit the actual geometry itself. This is often the case at object boundaries and sharp depth edges. In that case, other geometry in leaf nodes further away is actually visible in the place of the hole. To be absolutely certain the right geometry is re-rendered, all the intersected leaf nodes along the ray would need to be re-rendered. Another optimization is to only re-render the first few hit leaf nodes. Again, this may cause the correct geometry to be missed, failing to resolve the hole. Re-rendering the first three hit leaf nodes seems to provides a good quality-performance trade-off.

4.1.3 Results

For the evaluation of the image quality produced by the PDL architecture, a sample scene is used that consists of the 28M polygon Stanford Lucy angel model, orbited by twelve torus knots. All objects rotate about their own Y-axis at 60 deg/s, where each torus knot rotates in opposite direction. The torus knots rotate about the angel at 20 deg/s. The camera is tilted upwards by ten degrees. A sample frame of this animation is shown in Figure 4.2. The client's frame rate is artificially fixed to 6 Hz, resulting in a 10:1 server:client frame ratio; i.e., ten frames are warped for every client frame update. This is possible because off-line rendering is used in combination with a special synchronized mode where the client and server both wait for each other according to a given fixed ratio. In all cases point splatting was used as the image-warping algorithm, and two client views were warped to the left-eye image of a stereoscopic pair.

An example frame where the detected occlusion errors are shown in red and the sampling errors in blue is shown in Figure 4.2. Quantitative data is plotted in Figure 4.3. The off-line method used to detect both types of errors from reference images is described in detail in Appendix A. The chosen sample frame #42, and this complete 360 frame scene animation, is used for the rest of this results section. Note, this is a particularly bad frame that shows near-maximum occlusion errors.

To evaluate the hole-finding method, the pixels that were identified as holes are compared to the map of reference holes. This reference map is constructed in the way described in Appendix A, using a reference and warped depth map. Figure 4.4 shows the hole maps found for frame #42. The left image depicts the reference holes in a warped image where no hole filling is applied. To better illustrate the results, the background quad was not redrawn here, so holes in the background are also visible. The middle image depicts the holes found by a level-0 hole finding method, which has been a popular method of detecting holes in the past [Mar99]. This method only finds holes where no pixels have been warped to. Since any pixel that is not reached by a fragment is necessarily a hole, the method never finds false-positives. However, the method fails to find any level-1 type of hole, which occur frequently in the animation. The right image depicts the holes identified by the level-1 hole finding method. Correctly identified holes are shown in green, where dark green indicates the subset of pixels that is classified as a background hole. Since the background quad is re-drawn anyway, it is not required to ray trace these hole pixels. A few background holes remain that are not classified as such. False-positives, that is non-hole pixels identified as holes, are depicted in yellow and occur most frequently at the edges of objects, as expected.

The percentage of non-background holes correctly identified by the two methods is plotted in Figure 4.5 for the 360 animation frames. The level-1 method is capable of finding most of the holes in the animation, except for a small number of level-2 holes. The level-0 method performs poorly in identifying holes. Although the level-1 method finds more holes and a number of false-positives, it actually scans fewer pixels than the level-0 method due to the background-hole optimization. This is because the level-0 method has no way to distinguish between background holes and other types of holes, so it has to scan all of them. The level-1 method shows superior performance in identifying holes, and is used as the algorithm of choice.

Next, the number of detected holes that can be resolved correctly is evaluated, as well as the number of polygons that need to be re-rendered to achieve this. For the hole map, the output of the level-1 hole finding method is used. Figure 4.6 shows the sample frame #42 with hole filling applied for the first, the first three and all intersected leaf nodes. The larger the number of intersected leaf nodes, the larger the number of polygons that need to be

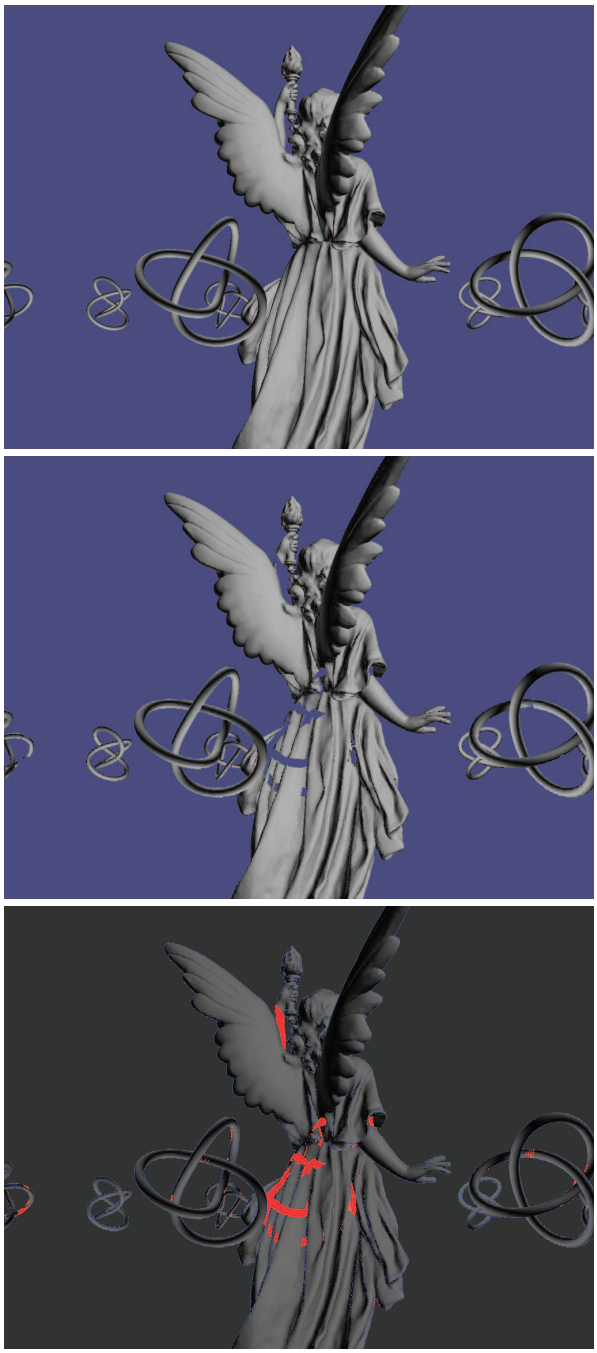

Figure 4.2: (top) A reference image produced in approximately 207 ms by a regular stereoscopic render for a 28M polygon scene. Only the left-eye view is shown. (center) The same scene produced by image warping in only 14 ms. (bottom) Errors in the warped image. Occlusion errors, or holes, are depicted in red and sampling errors in blue.

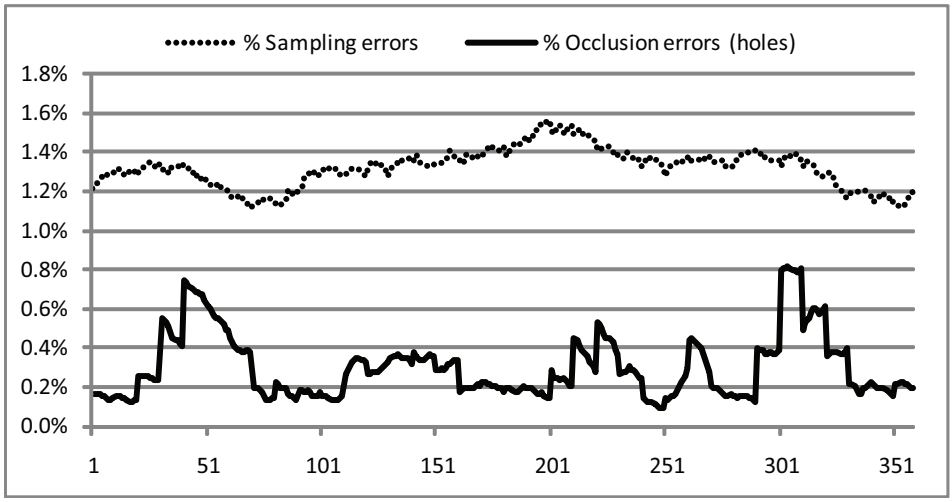

Figure 4.3: The percentage of pixels that constitute occlusion or sampling errors for each frame of a 360 frame animation of the 28M polygon scene. Figure 4.2 depicts frame number 42, which exhibits a near-maximum amount of occlusion errors in the sequence. This frame is used as a sample frame throughout this chapter.

re-rendered. This is plotted in Figure 4.7. The percentages of hole pixels that were correctly resolved are plotted in Figure 4.8. It can be seen that re-rendering the first three leaf node intersection along the ray results in a reasonable trade-off between the number of resolved holes and the number of re-rendered polygons; most of the holes are resolved, while re-rendering only a small percentage of the original 28M polygons. For illustration, the number of resolved holes using a level-0 hole finding method is also plotted in Figure 4.8. This shows the importance of good hole finding, as it forms a baseline for the number of reference holes than can be resolved: holes that are not found can not be resolved, no matter how many leaf node intersections are re-rendered. Finally, these results show that it is possible to create very high quality images using image-warping techniques and good hole filling, depending on the available time.

The average performance on a multi-GPU system for stereoscopic warping using these hole-filling methods are approximately 65 ms, 78 ms and 92 ms, for the re-rendering of the first hit leaf, the first three hits and all hits, respectively. There are a number of reasons for this lower performance. The kd-tree is generated using a simple mid-point split, which is known to result in sub-optimal ray tracing performance. A tree build using a surface area heuristic would increase ray tracing performance significantly. Furthermore, ray tracing is done sequentially on just a single CPU core, and for each pixel independently. Using multiple cores or a GPU ray tracer, in combination with amortized or packet ray tracing, would further improve performance (a good overview of ray tracing techniques is given by Wald et al. [WMG+07]). One final optimization could be to start ray tracing at the depth found in the occlusion map. Due to the convex hull nature of the rendered bounding boxes, no intersections can possibly occur in front of this depth. One issue that remains is the amount of geometry that needs to be re-rendered. If this amount is too high, a 60 Hz frame rate can not be achieved without falling back to level-of-detail methods for the re-rendered geometry, which may be a viable alternative. Another option would be to implement a completely GPU-

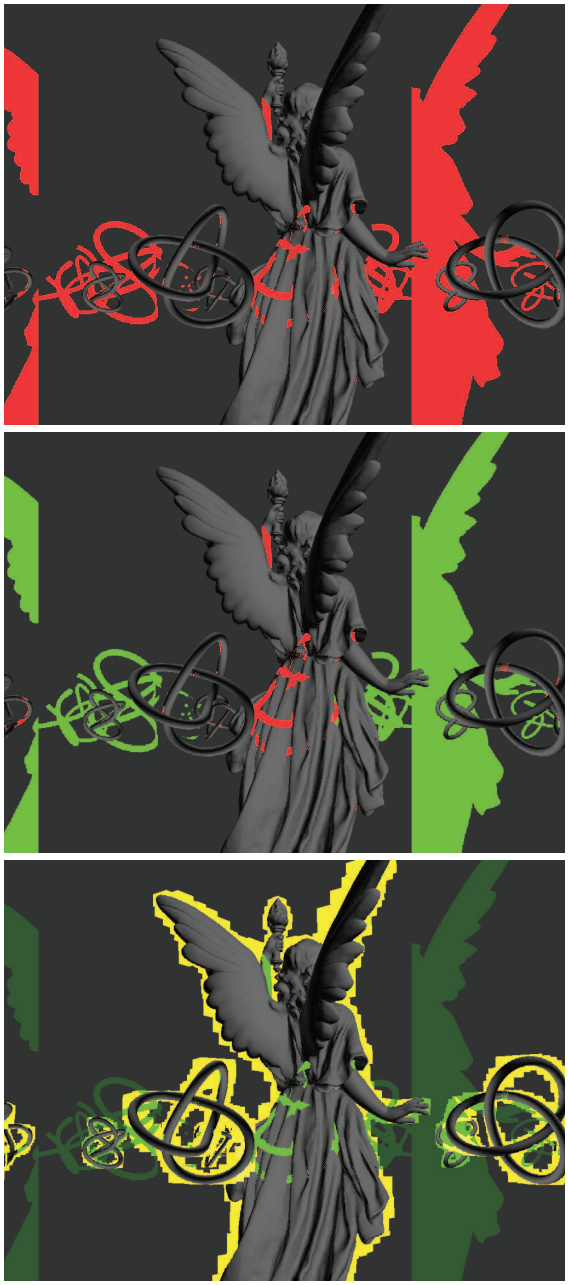

Figure 4.4: (top) Reference holes in red for a warped image where no hole filling is applied, including background holes. (center) Holes correctly detected by a level-0 hole finding algorithm are shown in bright green, while holes that were not found are shown in red. (bottom) Holes found by the level-1 hole finding algorithm. Green indicates a hole is correctly found, where the dark green sub set represents pixels correctly identified as background holes. False-positives are depicted in yellow. Holes that were not found are depicted in red.

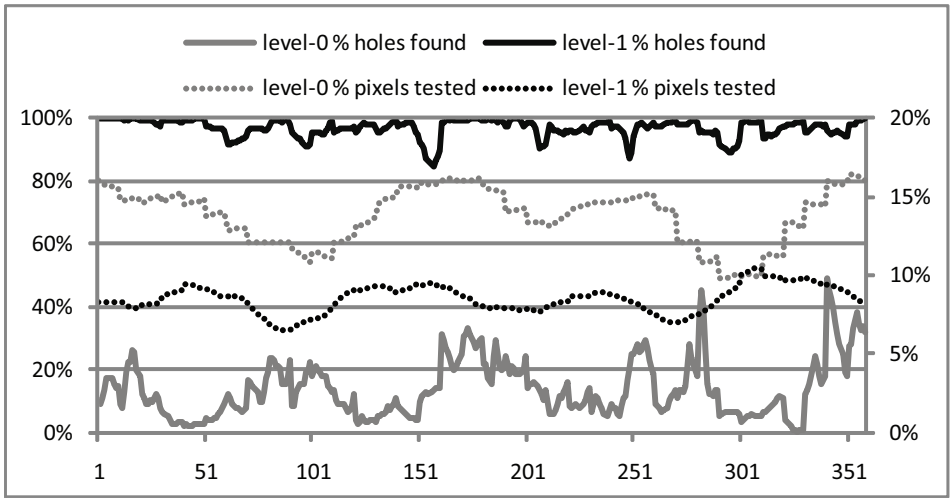

Figure 4.5: (left axis) Percentage of holes found correctly for the level-0 and level-1 hole finding methods, excluding background holes. The level-0 method fails to find many holes, while the level-1 method finds most of the holes. (right axis) The percentage of pixels identified as candidate non-background holes. While the level-1 method finds more holes, it actually scans fewer pixels than the level-0 method.

based ray tracer that can quickly re-render individual pixels, avoiding the need to re-render entire leaf nodes. However, while currently a 60 Hz frame rate can not be maintained when using the described hole-filling methods, an almost-error-free stereoscopic image can still be produced about three times faster than with a regular stereoscopic renderer (70 ms versus 200 ms, see Table 3.2). This result by itself may justify the use of image-warping techniques in some situations.

A trade-off occurs for the presented method of detecting and resolving occlusion artefacts. Resolving a larger number of occlusion errors correctly also required a larger number of polygons to be redrawn, in turn reducing performance. In this way, image quality can be increased depending on the available processing time. Another factor is the size of the leaf nodes in the kd-tree. Larger leaf nodes contain more geometry and require more polygons to be redrawn; however, smaller leaf nodes require more expensive ray tracing. A similar trade-off exists for hole detection; with bounding boxes at a higher resolution in the kd-tree, fewer false positives are detected, and fewer rays need to be traced, but more bounding boxes need to be rendered to construct the reference map. These problems may be partially remedied by using adaptive, dynamic resolutions for the leaf nodes and bounding boxes, but this has not been investigated.

4.2 Client-side camera configurations

The PDL architecture performs image warping to generate a transformed and re-projected server view using images generated by the client. The implementation is free to choose the number of images generated by the client for each frame, as well as the client viewpoints used for this purpose. However, as can be seen in Figure 4.9, a minimum of two client views are generally required to produce a single server view without serious occlusion artefacts

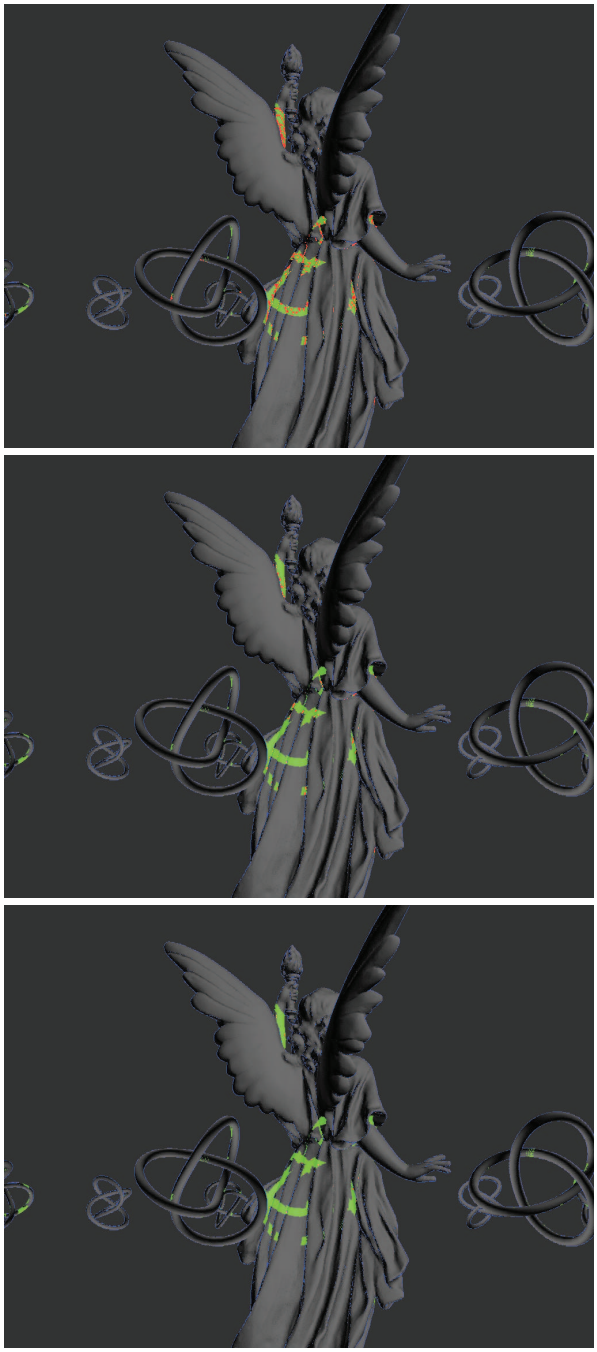

Figure 4.6: Example of hole-filling. Green pixels depict correctly filled holes, while red pixels depict holes that were not filled. Blue represents remaining sampling errors. (top) Resolving holes by re-rendering the first leaf node intersection; some holes remain. (center) Re-rendering the first three intersection resolves most holes (bottom) Re-rendering all intersection resolves all hole pixels that were correctly identified.

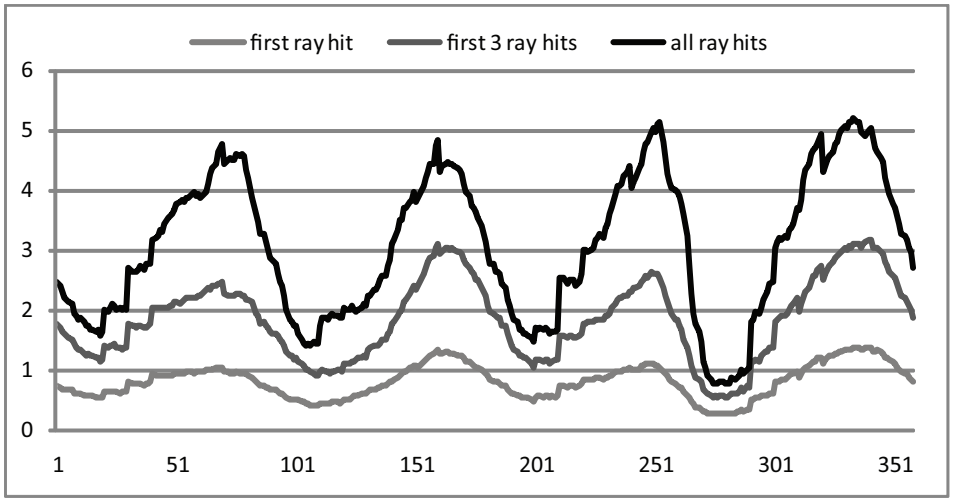

Figure 4.7: Millions of polygons re-rendered by the three hole-filling strategies for each frame of the animation. the original scene consists of 28M polygons. Re-rendering the first three leaf node intersections provides a good trade-off between achieved quality and the number of polygons re-rendered.

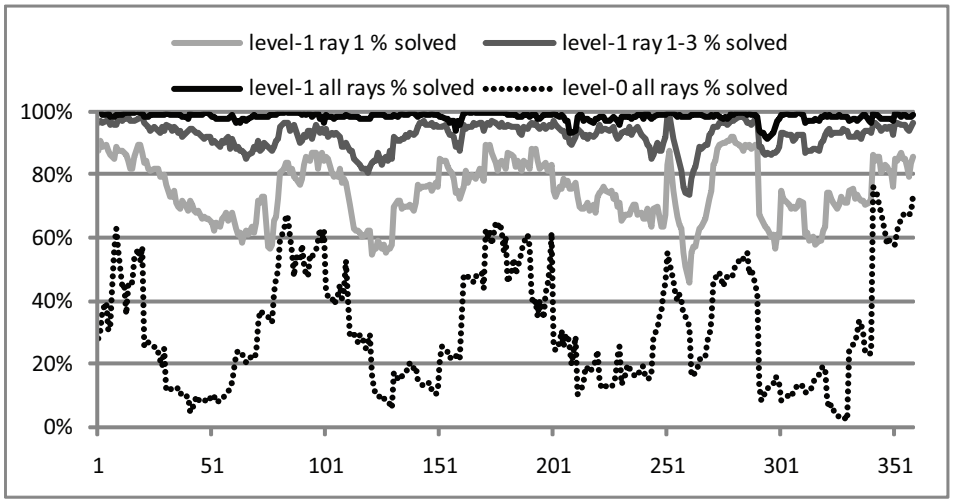

Figure 4.8: Percentage of holes resolved for the three hole-filling strategies for each frame of the animation. Many pixels identified as holes are not correctly filled when only re-rendering the first leaf node intersection found. However, re-rendering the first three intersections resolves most of the holes. When re-rendering all the intersections, every identified hole is resolved and only the holes that were not found are left. In comparison, the dotted line shows resolved holes when using a level-0 hole finding method that re-renders all intersections.

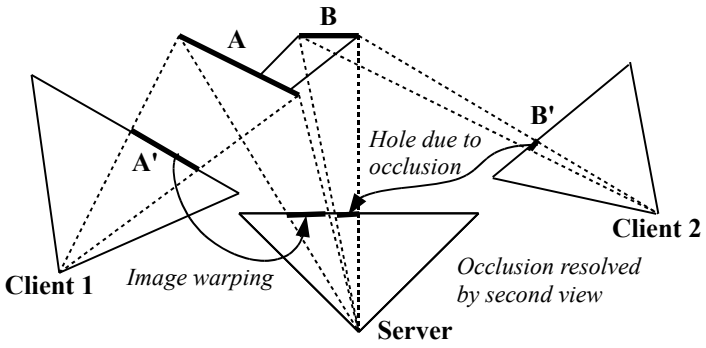

Figure 4.9: Using a second client view in order to reduce occlusion artefacts in the form of holes. Objects A and B are visible in the server's projection; however, object B is occluded by object A in the first client view's projection, resulting in a hole. By adding a second client view the occlusion issue can be resolved.

[Mar99].

The challenge is to determine how many client-side cameras to use and how to choose the pose of these cameras in order to achieve the maximum warped image quality at the server side. The most notable and disturbing type of error in image warping is due to occlusion. Therefore, client viewpoints should be chosen in such a way as to minimize potential occlusions during warping. Another aspect is that of viewport clipping; geometry outside the client's view frustum can not be warped into the server's frustum because it is never rendered. For this reason, the field-of-view (FOV) of the client cameras should, in general, be larger than the server's FOV to avoid viewport clipping. However, these approaches may result in loss of image quality. Hence, the camera-placement strategy should also maximize image quality.

Three types of client-side camera-placement strategies are explored: static, predictive and optic flow based. These strategies are schematically depicted in Figure 4.10. A static strategy simply places the cameras in fixed positions relative to the latest known sensor data. Other strategies have been developed to efficiently place client cameras based on sensor prediction [Mar99]. It was shown that reasonable results can be obtained by using two client-side views: one rendered from the camera viewpoint in the current frame, and one from the predicted future camera viewpoint in the next frame. These strategies assume static scenes where the camera is the sole moving entity. For static scenes, once an image has been rendered from a specific viewpoint it will remain valid throughout the life-time of the application. Therefore, this strategy can reuse old imagery and the client only renders and transmits a single, predicted viewpoint for each frame. However, for dynamic scenes good client-side camera-placement is more challenging. Since objects are moving, client viewpoints should not be determined based solely on camera movement. What is needed is an intelligent camera-placement algorithm that is based on the motion of individual objects in the scene. This may be possible by making use of per-pixel optic flow. Instead of placing client cameras according to the predicted motion of the server camera, they are placed according to the optic flow of the scene at that time. In this way, object and camera movement both implicitly contribute to camera placements.

An important problem that arises when warping two client-side viewpoints to the same server viewpoint is that pixels from both viewpoints may be warped to the same target pixel.

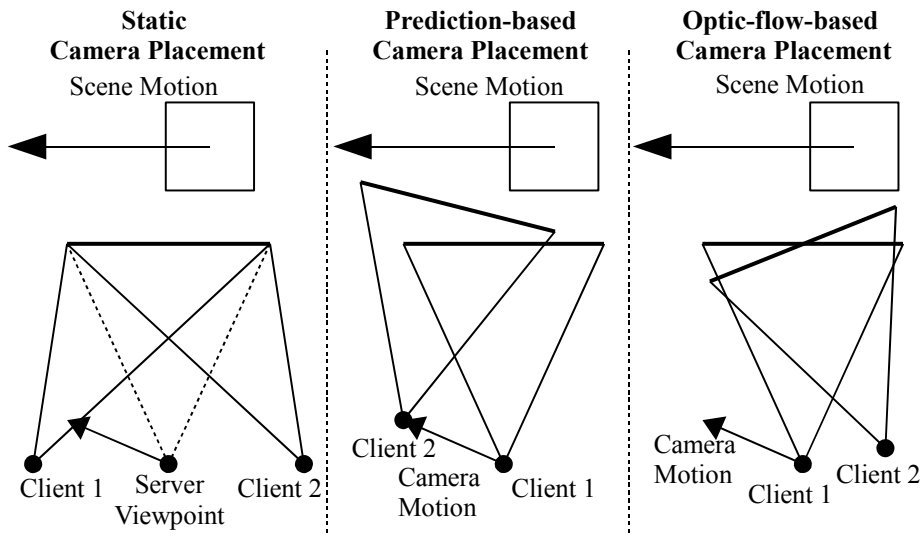

Figure 4.10: Schematic overview of the three different client-side camera-placement strategies. The static strategy always places the client cameras in fixed positions relative to the server viewpoint. The prediction-based strategy places the second camera according to the predicted pose of the camera in the next frame, ignoring scene motion. The optic-flow-based strategy implicitly takes the combined scene and camera motion into account and places the second camera according to averaged per-pixel optic flow. For the latter two strategies, the first camera is always placed at the server's viewpoint.

The depth buffer correctly handles the situation when these pixels belong to different parts of the geometry. However, if the depth values are nearly the same, this indicates that the corresponding geometry is the same and that one of the pixels needs to be discarded in favour of the other. The current implementation tries to discard the lowest quality pixel. Pixel quality is estimated according to the pixel's splat size, which is calculated using the expansion factor of the projective image-warping transformation. Generally, a pixel with a smaller splat size, or smaller warping distance, is a better quality pixel.

To evaluate the effect of client-side camera placements, three different camera-placement strategies have been implemented. In all three cases, two client viewpoints are used. Consequently, for stereoscopic rendering, the server needs to perform image warping four times: twice for each of the left- and right-eye server views. The first is a static placement strategy, where the two cameras are always positioned at a fixed offset relative to the latest known head-tracker pose. This is implemented by using a stereoscopic camera setup with larger-than-normal eye-separation and increased FOV that is centered at the head-tracker pose. The second approach is a prediction-based strategy. The first camera is positioned at the latest head-tracker pose, while the second camera is placed according to the predicted future head-tracker pose. The current implementation predicts three application frames ahead instead of one, since this results in better overall image quality. Furthermore, the FOV of the first camera is equal to that of the server camera, while the second camera is set to increased FOV. The third strategy is based on optic flow. It is similar to the prediction-based method, except that the position of the second camera is determined according to the per-pixel optic flow on

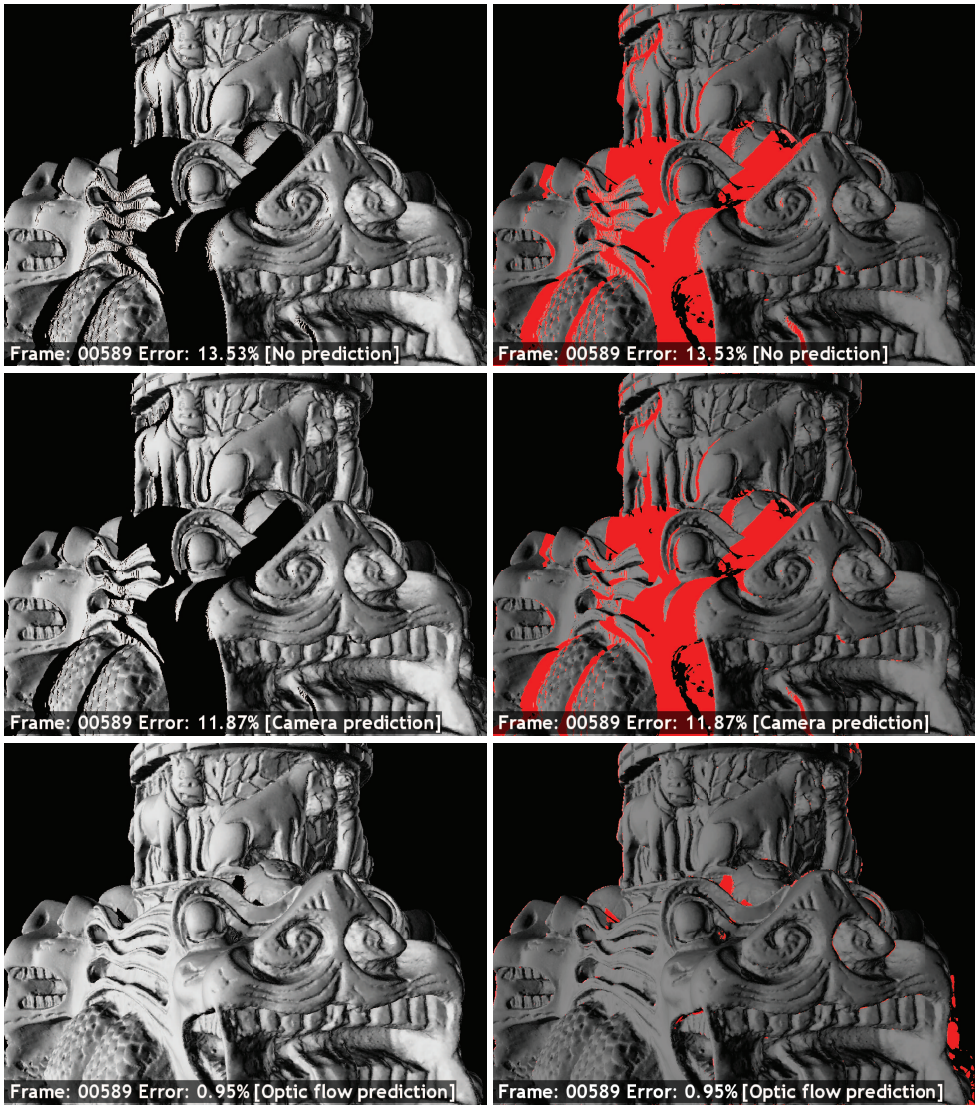

Figure 4.11: The importance of good client-side camera placements. From top to bottom warped images of the dynamic statue scene are shown using no prediction, camera-only prediction, and prediction based on optic flow. The right column depicts the amount of error. Using no prediction results in high error due to the relatively large scene velocity. Since the scene is dynamic, with a rotating object, the camera-only prediction method is unable to accurately predict the scene's motion and also results in high error. The optic-flow-based prediction method is able to correctly estimate the scene motion and results in an almost error-free image. This shows that with good camera placements, image warping can produce very high quality images.

	Statue		Coral	
Dynamic Scene	**Error**	**Stdev**	**Error**	**Stdev**
Static	1.48	1.39	2.16	2.08
Prediction	1.15	1.10	2.01	1.18
Optic flow	0.72	0.61	2.10	1.31
	Statue		**Coral**	
Static Scene	**Error**	**Stdev**	**Error**	**Stdev**
Static	0.95	1.15	2.01	2.11
Prediction	0.63	0.35	1.90	1.03
Optic flow	0.58	0.41	1.80	1.08

Table 4.1: Overview of the average errors and standard deviations for different camera-placement strategies. Error comparisons can only be made vertically between equal scene/model combinations because different animations were used for each.

the client-side. First, the three-dimensional motion field that is generated by the client for the first camera is projected onto the image plane. Next, the projected motion vectors are averaged into a single vector. The second camera is now rotated about the focal point of the first camera, where the opposite averaged motion vector is used to determine the magnitude and angle of rotation. The camera is placed according to the opposite direction of the optic flow because this strategy minimizes the introduction of occlusion artifacts. As before, only the second camera is set to increased FOV.

To compare image quality for the three camera-placement strategies, the same type of animation sequences are recorded as described in Appendix B for a dynamic and a static scene using the coral and statue models. In the dynamic scene the models rotate about their Y-axis, while for the static scene they remain still. Both scenes contain user-controlled camera movements. The results are tabulated in Table 4.1. Figure 4.11 shows the importance of good client-side camera placements.

First, the statue scene is examined. In the dynamic case, the average error is lowest for the optic-flow-based strategy, followed by that for the prediction-based and static strategies. The same is true for the standard deviations of the errors, which give an indication to the number of occlusion artefacts. The static strategy apparently has difficulties with the motion in the scene and results in many occlusion artefacts. The prediction strategy works better because it manages to reduce occlusion artefacts by predicting camera motion. However, object motion in the dynamic scene is not detected by the prediction strategy. On the other hand, the optic flow strategy is able to combine camera and dynamic object motion and results in even less error. In the static case, the results for the prediction and optic flow strategies are nearly equal. This is due to the fact that camera prediction alone almost perfectly predicts the motion in a static scene. Again, both methods result in less error than the static strategy.

The results for the coral scene are somewhat different. First, the average errors for all three methods are very close to each other. This can be attributed to the high-frequency nature of the model; many image-warping errors are made at edges, hiding the impact of occlusion errors. Second, while the standard deviations are lower for the prediction and optic flow methods, there seems to be little difference between a static and a dynamic scene. This can be explained by the fact that the coral model is nearly convex with the exception of many deep holes in the surface at arbitrary orientations. The prediction methods succeed in dealing with large self-occlusions of the model, but almost always fail in handling occlusion

for the surface holes due to their arbitrary directions. The motion of the surface of the coral is not a good indicator to handle all types of occlusion artefacts in this case. An interesting property of both the prediction-based and optic-flow-based camera-placement strategies is that for scenes with little motion, the image quality converges to that of a directly rendered reference image. This is due to the fact that the setup of the first camera is equivalent to that of the server camera. This allows the user to inspect a nearly still scene at a quality level similar to that of direct rendering.

A trade-off exists for using client-side camera placements to reduce occlusion artefacts. To avoid clipping objects that enter the camera space, the client view should have a larger field-of-view than the server camera. However, increasing the FOV results in smaller rendered object sizes and fewer warped pixels per object. This effect is almost equivalent to reducing the client resolution and reduces the overall quality of warping. Another good way to reduce occlusion errors is to warp multiple client views. This can be achieved by placing the client cameras in such a way that a large part of the scene is visible from different directions. However, this could result in the client views to be far away from the server view, resulting in large required warping distances and, consequently, larger errors. Using multiple client views also reduces performance due to an increase of pixels to be warped and an increase of shared pixel data. Furthermore, for multiple client views to be effective, intelligent camera-placement strategies are required that come with different performance characteristics.

When two pixels belonging to different views warped to the same location the estimated best-quality pixel was used. While this approach works, it is often quite wasteful in the amount of pixels that are transferred and warped only to be discarded in favour of a better quality pixel. The first view that is warped generally produces the best quality pixels, while the second view is mostly used to resolve occlusion artefacts; yet, all of its pixels are transferred and warped. A better approach would be to only render the subset of pixels of the second view that are useful to resolve occlusion artefacts. In particular in cases where the optic flow is very different across sections of the view, it may be beneficial to render these different subsections with different camera placements according to the optic flow of that subset specifically.

Chapter 5

Crosstalk Reduction

Previous work on software crosstalk reduction mostly focussed on the preprocessing of still images with a fixed viewpoint [KLD00, LW94]. However, what is needed is crosstalk reduction in virtual environments; that is, for dynamic scenes, various viewpoints and in real-time. Also, crosstalk should not be assumed to be uniform over the entire display area as some previous methods did [KLD00]. Furthermore, since virtual environments operate in real-time, crosstalk reduction algorithms must be fast enough to run in real-time.

In this chapter, a non-uniform subtractive crosstalk model and reduction implementation are proposed. In addition, a procedure is described for the interactive user calibration of the model parameters. The non-uniformity indicates that crosstalk is not assumed to be constant over the display area. The method is based on a combination of the non-uniform crosstalk characteristics described by Woods and Tan [WT02] and the calibration procedure proposed by Konrad et al. [KLD00]. However, due to the subtractive nature of the reduction algorithm, the basic algorithm suffers from the same disadvantages for uncorrectable regions as previous subtractive methods. In an attempt to reduce the perceived effect of crosstalk in uncorrectable regions, an extensions to the basic non-uniform reduction algorithm is described as well. The efficacy of these algorithms is determined using a quantitative evaluation methodology for the perceptual quality of crosstalk reduction. Details of this evaluation approach are given in Appendix A.

There are a number of general benefits to this method of crosstalk reduction:

- Image quality is enhanced by reducing the perceptually disturbing ghost images due to crosstalk, resulting in images that are sharper and more clear.
- Depth perception is improved by reducing ghost images that have a negative influence on the fusion limits for stereoscopic images.
- The proposed evaluation method measures the amount of perceptually disturbing crosstalk, allowing quantitative assessments to be made about the quality of various crosstalk reduction algorithms.

Furthermore, a number of aspects of the implementation result in increased flexibility for the types of virtual scenes and hardware that crosstalk reduction can be beneficial for:

- A non-uniform reduction method allows for crosstalk to be reduced over the entire area

of the display. Furthermore, since the implementation runs in real-time and on the PDL architecture, dynamic scenes are fully supported.

- An interactive user calibration procedure allows the algorithms to be fine tuned for different kinds of displays, LCS glasses and even users.
- The proposed extension increases the amount of crosstalk reduction that is possible in otherwise uncorrectable regions.

The combined effect of all these contributions is an improved real-time software crosstalk reduction framework that is applicable to a wider range of scenes and results in better image quality and depth perception.

5.1 Crosstalk model and calibration

The amount of perceived crosstalk can vary depending on a number of factors, such as lighting, monitor brightness and the type of hardware. Most crosstalk reduction methods, for example the method described by Lipscomb and Wooten [LW94], need to be adjusted for varying conditions. However, it might not be immediately clear to the users exactly how to do this. Therefore, a simple way of calibrating the crosstalk reduction algorithm is desirable.

To estimate the amount of crosstalk correction required in a given situation, a calibration method is used that is based on a similar psychovisual experiment by Konrad et al. [KLD00]. The latter experiment assumes crosstalk to be uniform across the screen. However, as was shown by Woods and Tan [WT02], this is not the case and results in poor crosstalk reduction towards the bottom of the screen. What is needed is a crosstalk model, combined with an easy-to-use calibration method, that can handle the non-uniform nature of crosstalk.

5.1.1 Non-uniform model

The non-uniform nature of crosstalk is modelled using a simple function of screen height y. This function need not be exact; it only serves as an indication of the shape of the non-uniformity for crosstalk reduction purposes. Crosstalk is classified by two distinct causes: leakage of the LCS glasses and phosphor afterglow [WT02]. The former can be approximated as being almost uniform over screen height, while the latter exhibits a strong non-linear shape. To approximate this shape, the typical decay of CRT phosphors is modelled using a power-law decay function [EIATEPACET93]

$$\phi(t,i,y) = i * (t - y + 1)^{-(\gamma+1)}$$

where $t \in [0,\infty)$ stands for the time after the first vertical blank (VBL) signal, i is the intensity of the phosphor in the next video frame (at $t = 1$), $y \in [0,1]$ is the screen height, and γ is a constant indicating the speed of decay (a constant value of one has been added to γ to simplify integration). As the amount of crosstalk is constant over a specific scan line, $\phi(t,i,y)$ disregards the pixel column x and only depends on y. The purpose of this function is to model the amount of residual intensity in the current display frame for a pixel that was shown in the previous display frame. This first display frame is assumed to be visual for $t \in [0,1)$. At the time $t = 1$ a buffer swap occurs and the next display frames becomes visible for $t \in [1,2)$. This residual intensity in the next display frame gives an indication of the amount of crosstalk due to phosphor afterglow. This time variable t is shifted by the pixel's height y because

	Left screen half	Right screen half
Left-eye frame	I_{intended}	I_{adjust}
Right-eye frame	$I_{\text{unintended}}$	I_0

Table 5.1: Calibration setup for a uniform crosstalk model. The columns show the two halves of the screen, while the rows show the sequentially displayed left- and right-eye frames. The user's right-eye view has been blocked, so the display is only seen through the left eye. The image displayed for the right-eye frame causes crosstalk for the left-eye frame. By altering I_{adjust} the amount of crosstalk between I_{intended} and $I_{\text{unintended}}$ can be determined.

phosphors are excited top-to-bottom. The phosphors at the bottom of the screen have just been excited when the buffer swap occurs at $t = 1$ and cause significant crosstalk. On the other hand, the pixels at the top of the screen have been decaying for a long time when the buffer swap occurs.

To estimate the intensity of the phosphor in the second display frame, integrate $\phi(t,i,y)$ over t:

$$c(i,y) = \int_{1.05}^{2.0} \phi(t,i,y)\,dt = i \cdot \left(\frac{1}{\gamma(2.05-y)^{\gamma}} - \frac{1}{\gamma(3-y)^{\gamma}} \right)$$

The integration interval of $t \in [1.05, 2]$ is chosen because the electron beam spends a small amount of time in the vertical retrace (VRT) state ($t \in [1, 1.05)$), at which time the LCS glasses are still opaque. A power-law decay function was found to better approximate reality than exponential decay functions. Still, $c(i,y)$ is only an approximation of shape, and does not model the physical reality exactly. In reality, the phosphors will also be excited again in the next display frame; however, the goal here is only to approximate the non-linear shape of the intensity of crosstalk depending on the screen height of a pixel, not to determine exact pixel intensities.

Suppose that y is fixed to the screen height of a given pixel that needs to be corrected. Two parameters now remain that can be varied to estimate the amount of crosstalk: γ and i. Also, a constant term needs to be added, which is a combination of a constant amount of crosstalk for the phosphor decay and for the LCS glasses. This leads to the following calibration function:

$$ct(\sigma,\varepsilon,\gamma) = \sigma + \varepsilon \cdot \left(\frac{1}{\gamma(2.05-y)^{\gamma}} - \frac{1}{\gamma(3-y)^{\gamma}} \right)$$

Note, when γ is fixed, the rightmost term can be pre-calculated as it only depends on y. The value for σ describes the amount of crosstalk at the top of the screen, ε at the bottom, and γ is a parameter indicating the curvature of the function. The three parameters can be set to closely match the shape of the crosstalk curves found by Woods and Tan [WT02].

5.1.2 Uniform calibration

To calibrate the display, a procedure similar to the one used by Konrad et al. [KLD00] is used. Konrad's method is first described in more detail, after which the changes required for the non-uniform model are described. Note, whenever it is mentioned that a pixel is set to a certain intensity, what is actually meant is that the linear-light value in the frame buffer is set to a certain value between I_0 and I_{255}. Due to the implicit gamma correction on the graphics card and the display, these linear frame-buffer values correspond to a near-linear actual intensity from the display.

A separate calibration procedure is performed for each of the three red, green and blue color channels due to the different types of phosphors that are used for the three colours, each of which shows different decay characteristics. The display is divided into a left and a right screen half. Because crosstalk is assumed to be uniform for now, but in reality is not, only the center of the display is used to minimize the effect of the non-uniformity of crosstalk. The user looks at the screen through activated LCS glasses where the right-eye is kept completely opaque; hence, the user can only see the display through his left eye. For each screen half, all the pixels are set to a certain constant intensity, which is different for the right- and left-eye frames (see Table 5.1). In this way, the pixels in the invisible right-eye frame cause crosstalk on the visible left-eye frame. Due to crosstalk, the user now observes the following intensities (through his left eye) for each of the screen halves:

$$\begin{aligned} \text{Left screen half:} \quad & I_{\text{intended}} + \theta(I_{\text{unintended}}, I_{\text{intended}}) \\ \text{Right screen half:} \quad & I_{\text{adjust}} + \theta(I_0, I_{\text{adjust}}) = I_{\text{adjust}} \end{aligned}$$

where $\theta(u,i)$ is a function describing the amount of crosstalk between an intended intensity i in the visible (left-eye) frame and an unintended intensity u in the invisible (right-eye) frame. Hence, the unintended intensity u causes crosstalk on the intended intensity i. When $u = I_0$ it is assumed there is no crosstalk, so $\theta(I_0, i) = i$. Therefore, on the right half of the screen, the users perceives I_{adjust} unmodified. The user's task is to adjust the value of I_{adjust} in such a way that the two screen halves match in intensity, resulting in the equality: $I_{\text{intended}} + \theta(I_{\text{unintended}}, I_{\text{intended}}) = I_{\text{adjust}}$. Hence, the amount of crosstalk between an intended and unintended intensity can be specified as:

$$\theta(I_{\text{unintended}}, I_{\text{intended}}) = I_{\text{adjust}} - I_{\text{intended}}$$

The entire procedure is repeated for several values of I_{intended} and $I_{\text{unintended}}$, and for the red, green and blue color channels. This results in three look-up tables, one for each channel, that map from $\mathbb{R}^2 \rightarrow \mathbb{R}$. Next, these tables are linearly interpolated and inverted by an iterative procedure to be able to correct for crosstalk. This results in a function that maps a given unintended and desired perceived intensity to the intensity to be displayed:

$$\theta^{-1}(I_{\text{unintended}}, I_{\text{desired}}) = I_{\text{displayed}}$$

5.1.3 Non-uniform calibration

In this section, the changes required to calibrate the non-uniform crosstalk model are described. The parameter space of the non-uniform model is multi-dimensional, as opposed to one dimensional. The calibration procedure will return function parameters $\sigma, \varepsilon, \gamma$ for a function depending on the screen height y of the pixel. Therefore, interpolating and inverting the resulting calibration tables is not a straightforward task. To avoid this problem, the calibration setup is changed as shown in Table 5.2. Instead of determining the amount of crosstalk an unintended intensity produces on an intended intensity, the intensity to be displayed is directly estimated, given a desired intensity and an unintended intensity. This is equivalent to calibrating for θ^{-1}.

To match both screen halves in intensity, the user first adjusts σ to match the top part of the screen. Then, ε is adjusted to match the bottom part of the screen. Finally, γ is adjusted for the proper curvature. The user adjustable intensity is set to:

$$I_{\text{adjust}} = 1 - \min(1, ct(\sigma, \varepsilon, \gamma))$$

	Left screen half	Right screen half
Left-eye frame	I_{desired}	I_{adjust}
Right-eye frame	I_0	$I_{\text{unintended}}$

Table 5.2: An alternate calibration setup for the non-uniform model, corresponding to Table 5.1. The left half of the screen is crosstalk free. Given $I_{\text{unintended}}$, the intensity to display such that the user perceives I_{desired} can be found by altering I_{adjust}.

This is an approximation of the inverse of $ct(\sigma,\varepsilon,\gamma)$ up to a constant. Hence, when displaying ct^{-1}, the crosstalk is cancelled out. The approximation results in slightly too little crosstalk correction towards the bottom of the screen; for values of y close to 1. However, as was shown by Woods and Tan [WT02], the LCS glasses produce slightly less crosstalk at the bottom of the screen. This LCS leakage was previously estimated to be constant, so the small error made by approximating the inversion compensates for this assumption.

When the two screen halves match, the following can be seen:

$$\begin{aligned} \text{Left screen half:} \quad & I_{\text{desired}} + \theta(I_0, I_{\text{desired}}) = I_{\text{desired}} \\ \text{Right screen half:} \quad & I_{\text{adjust}} + \theta(I_{\text{unintended}}, I_{\text{adjust}}) = \\ & 1 - \min(1, ct(\sigma,\varepsilon,\gamma)) + \theta(I_{\text{unintended}}, I_{\text{adjust}}) \end{aligned}$$

If the amount of crosstalk has been modelled correctly, that is $\theta \approx ct$, the values of I_{adjust} and $\theta(I_{\text{unintended}}, I_{\text{adjust}})$ will compensate each other up to the constant I_{desired}. In other words, given an undesired intensity, the intensity to display such that the user perceives the desired intensity is known. Note, I_{adjust} occurs both by itself and in $\theta(I_{\text{unintended}}, I_{\text{adjust}})$, therefore a complex equality is solved interactively. Also, all equations depend on the height y of the current pixel.

User guided calibration of three correction parameters, on a two-dimensional grid of intended and unintended parameters, for each color channel, is a tedious and time-consuming process. After some initial experimentation, it was found that the value for γ could be fixed at 6.5 in all cases on the used hardware. This is expected to be true for most hardware because γ depends on the type of phosphors used, and CRT monitors are known to use very similar phosphors [WT02]. This reduces the parameter space to only σ and ε. To interpolate the calibration grid for uncalibrated values of I_{desired} and $I_{\text{unintended}}$ linear interpolation is used on the remaining function parameters σ and ε. For each of the three color channels, a 6×6 grid of I_{desired} and $I_{\text{unintended}}$ values was calibrated. This means that the user had to make a total of 108 matches between screen halves.

5.2 Crosstalk reduction implementation

In this section, the GPU implementation of the subtractive crosstalk reduction algorithm is described. First, a crosstalk reduction algorithm that functions under the assumption of still images is examined — that is, scenes that never change and always display the same image — followed by the required changes to implement the algorithm on the PDL architecture for dynamic scenes.

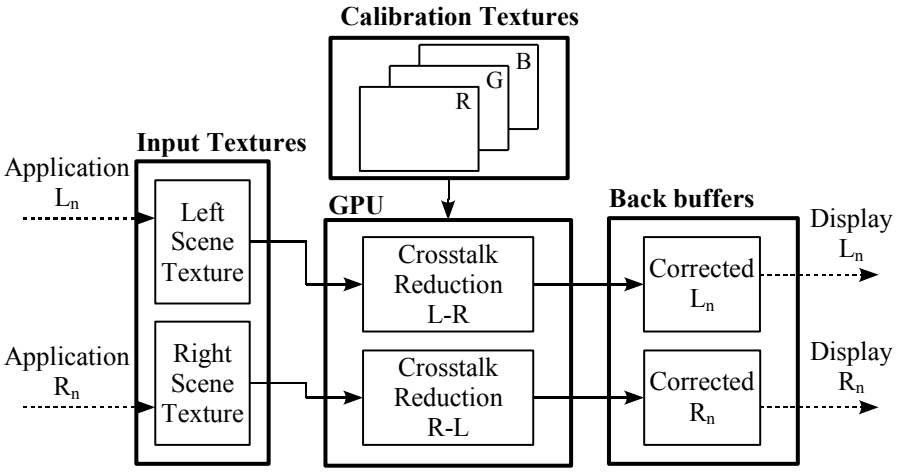

Figure 5.1: A simple crosstalk reduction pipeline for still images. Left and right scene textures are taken as input and are processed on the GPU using the calibration table textures. The output is written directly to the left and right back buffers using MRT-2.

5.2.1 Still images

The algorithm starts out by rendering the stereo scene for the left and right eye to two separate textures. Next, crosstalk reduction is performed, and finally the resulting left and right textures are displayed as screen sized quads. The only requirements are modern video hardware, capable of running fragment programs, and an existing pipeline which is able to render to textures using frame buffer objects (FBOs). An overview of the static crosstalk reduction pipeline is given in Figure 5.1.

Once the scene has been rendered to the left and right floating point textures, they are bound as input to the reduction algorithm. Note that texture targets are bound to video hardware buffers; therefore, everything runs entirely on the GPU — no texture downloads by the CPU are required. The calibration grid is interpolated and stored in three separate 256×256 input textures for each of the red, green and blue calibration colours. This results in three calibration textures, T_{Red}, T_{Green} and T_{Blue}. The textures have a 32 bit floating point, 4-channel RGBA format.

The calibration textures map the pair of pixel intensities $(I_{\text{unintended}}, I_{\text{desired}})$ to the two parameters σ and ε of the calibration function. As this only requires two of the four available texture channels, the inverted pair of intensities is also stored as an optimization. For $c \in \{Red, Green, Blue\}$, the calibration textures have the following layout:

$$T_c(u,v) \rightarrow (\sigma^c_{u,v}, \varepsilon^c_{u,v}, \sigma^c_{v,u}, \varepsilon^c_{v,u})$$

The left and right scene textures, as well as the three calibration textures, are then passed to a hardware fragment program by drawing a screen sized quad. In this way, screen pixels can be processed in parallel on the GPU. When drawing a pixel intended for the left application frame, I_{desired} is fetched from the left scene texture and $I_{\text{unintended}}$ from the right scene texture, and vice versa. Next, the corresponding values of σ^c and ε^c are fetched from the calibration textures for all three color channels.

From the fetched calibration parameters and the implicit constant $\gamma = 6.5$, a correction function depending on the screen height y of the pixel can be constructed. Crosstalk reduction

is performed by setting:

$$I^c_{\text{displayed}} = 1 - \min(1, \sigma^c_{u,v} + \varepsilon^c_{u,v} \cdot f(y))$$

where $c \in \{Red, Green, Blue\}$ as before, $u = I^c_{\text{unintended}}$, $v = I^c_{\text{desired}}$ and

$$f(y) = \left(\frac{1}{\gamma(2.05 - y)^\gamma} - \frac{1}{\gamma(3 - y)^\gamma} \right)$$

Note, as $f(y)$ does not depend on either σ or ε, it needs to be calculated only once for a fixed value of y. Also, all of the above calculations can be done in the fragment program.

Instead of rendering the crosstalk corrected left and right application frames separately in two passes, they can be rendered in a single pass using hardware Multiple Render Targets (MRT). This way, the left and right back buffers are bound simultaneously for writing.

In the dual pass case, correcting a single pixel for either the left or right application frame takes five texture lookups: two to fetch the desired and unintended pixel intensities from the left and right scene textures, and three to fetch the corresponding calibration parameters. This results in a total of ten texture lookups when rendering the left and right application frames.

However, when using MRT both sets of calibration parameters can be fetched simultaneously due to the structure of the calibration textures. Only five texture lookups are required in this case: two to fetch the pixel intensities from the left and right scene textures, and three to fetch the calibration parameters for $(I^c_{left}, I^c_{right})$ and $(I^c_{right}, I^c_{left})$ simultaneously from T_c. Also, both corrected pixels can be written to their corresponding left or right back buffer immediately.

An inherent problem with subtractive crosstalk correction methods is that one cannot compensate the crosstalk when the desired intensity is too low compared to the unintended intensity. When the crosstalk estimation for a color channel results in a value higher than the desired intensity for that channel, the best one can do is to set the channel to zero. To avoid this problem, the desired image intensity can be artificially increased. One way of doing this is by the following linear mapping for $\alpha \in [0,1]$: $I_{\text{desired}} = \alpha + I_{\text{desired}}(1 - \alpha)$. This kind of intensity increase is optional in the current implementation; however, a big drawback of increasing intensity this way is that it significantly reduces contrast.

5.2.2 Dynamic scenes

So far, still images were assumed for the application of crosstalk reduction. However, in virtual environments this is not the case. In an interactive application, subsequent application frames are usually slightly different. For frame sequential, active stereo the application frames are displayed sequentially as $L^a_n, R^a_n, L^a_{n+1}, R^a_{n+1}, ...$, where L^a and R^a stand for the left and right application frames respectively. The problem lies in the fact that R^a_n and R^a_{n+1} are slightly different due to animation. The crosstalk reduction algorithm for still images combines L^a_{n+1} with R^a_{n+1} to produce a corrected left frame, while it should combine L^a_{n+1} with R^a_n. This results in incorrect crosstalk reduction at the differing regions of R^a_n and R^a_{n+1}.

Moreover, if the rendering is slower than the display refresh rate, consecutive display frames are shown that do not directly correspond to application frames. The crosstalk now occurs between these display frames due to the leakage of light from one display frame to the other. In that case, the described crosstalk reduction needs to be performed in a similar fashion between these display frames, not application frames. However, with a classic architecture it is not possible to post-process individual display frames while generating application frames. Since the PDL architecture does guarantee updates of individual display frames

and allows the post-processing of display frames, it can be applied for crosstalk reduction on display frames.

Using the PDL architecture in this way, the crosstalk reduction algorithm can be easily modified to take dynamic scenes and display frames into account. Instead of using the previous and the current application frame as input, the output of the previous reduction step is used as input, in combination with the current display frame, resulting in a new display frame that can again be used as input for the next reduction.

Unfortunately, the difference between reducing crosstalk for display frames on the PDL architecture and for application frames on a classic architecture could not be quantitatively examined; however, one can imagine the effect. When crosstalk is reduced for application frames and not display frames, there are left-eye display frames immediately after a buffer swap for which the crosstalk reduction is incorrect. In fact, two types of errors are introduced:

- The left-eye display frames are not corrected for the preceding right-eye application frames. This leaves some amount of uncorrected crosstalk, resulting in areas that are too bright.
- Instead, these left-eye display frames are corrected for the upcoming right-eye application frame. This crosstalk reduction on areas where it is not required introduces incorrectly darkened areas.

Manual inspection showed that this is particularly relevant for moving objects, where the edges are slightly different from frame to frame. The main effect is not noticeable every frame, as application frames are drawn at a slower rate than display frames. Thus, the effect is most noticeable when application frames are updated. This typically happens at a rate of approximately 15-20 Hz and would explain some perceivable visual jitter, which is completely removed using the PDL architecture.

5.2.3 Quality evaluation

To evaluate the quality of crosstalk reduction, a method described in Appendix A was used. The monitor displayed a frame by frame animation of the optimization procedure for globally optimal Fekete point configurations [vLMFdS00]. Each application frame was displayed in reference mode (P_{ref}), with normal crosstalk (P_{ct}), with non-uniform crosstalk reduction enabled (P_{cor}), and with Konrad's uniform crosstalk reduction (P_{uni}) in sequence. In this manner, photographs were taken of 800 animation frames through the left eye of activated LCS glasses, capturing the images a user would see through his left eye. The results for the right-eye view are assumed to be similar. This resulted in a total of 3200 digital color photos of 2048×1536 resolution. The reference photo P_{ref} for frame 570 is shown in Figure 5.2.

For all animation frames, each of the photos P_{cor}, P_{ct} and P_{uni} were compared to the corresponding reference frame P_{ref} using a VDP-based evaluation method. For a discussion on VDP and the motivation for its use to detect disturbing crosstalk errors see Appendix A.2.2. The VDP evaluation method is used in an attempt to detect only those ghost images that are disturbing to the user. Every pixel value that is not set to zero (black) in the previous display frame causes crosstalk for the current display frame. However, quite often this only results in the same color being displayed with increased intensity, which does not cause a disturbing ghost image. Disturbing ghost images, sometimes called shadows, occur when there is a difference a large difference in pixel values, most notably hue, between the currently displayed pixel and the previous one it receives crosstalk from. If a single patch of the same color is displayed for both the left- and right-eye frames, one can observe a non-linear increase in

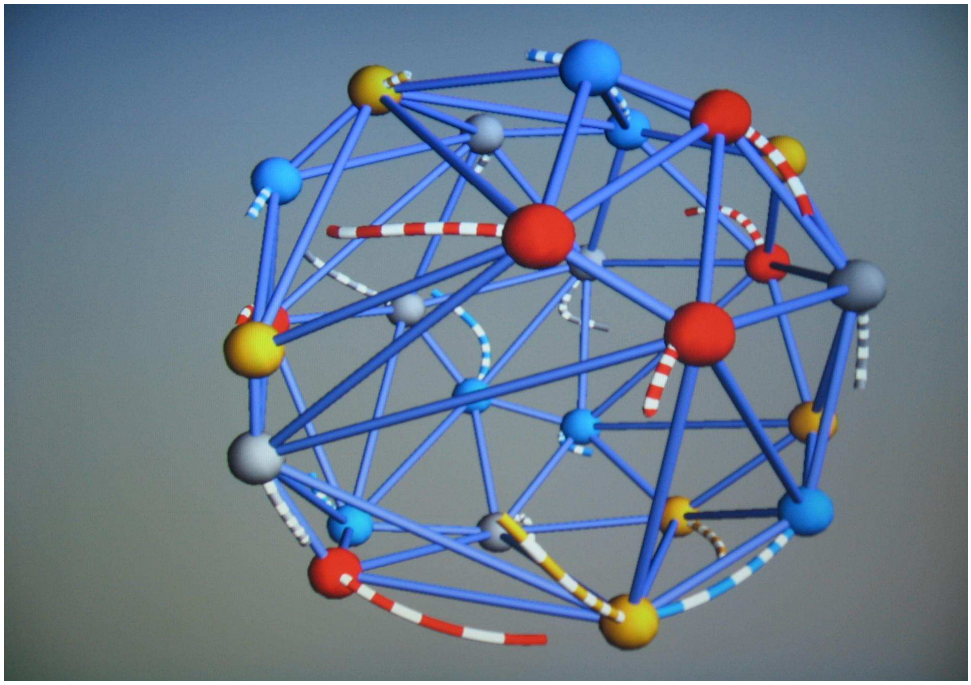

Figure 5.2: A photograph of the reference frame where no crosstalk is present. The photo was taken through the left eye of activated LCS glasses.

intensity from the top of the screen towards the bottom due to crosstalk, but no disturbing ghost images are visible. This gradual increase of intensity from top to bottom is generally not disturbing and including it in the evaluation of errors should be avoided.

Figure 5.3 shows the three photos and the output of the VDP comparison with the reference photo for animation frame 570. The percentages of pixels reported to be perceptually different by the VDP for each frame and correction method have been plotted in Figure 5.4. It can be seen that both uniform and non-uniform crosstalk reduction provide significantly better results than no crosstalk reduction. Also, non-uniform reduction is significantly better than uniform reduction; especially when the animation progresses. This can be explained by the fact that when the animation has been running for some time, more geometry becomes visible at the bottom of the screen. The parameters of the uniform reduction model have been calibrated only for a specific location on the display, in this case the center. The amount of crosstalk becomes larger at the bottom of the screen and the uniform model does not correct this. However, the non-uniform model takes this into account and corrects for crosstalk everywhere. This can also be seen in the photos of frame 570 in Figure 5.3.

When the desired pixel intensity to be displayed is very low, and the corresponding unintended intensity due to crosstalk is fairly high, no adequate crosstalk reduction can be performed. Two types of crosstalk are distinguished: *object-to-background* and *object-to-object*. Object-to-background crosstalk is caused by a geometric object and is visible on the background. This is generally the most perceptually disturbing kind of crosstalk. Object-to-object crosstalk is the crosstalk that is visible on the surface of an object, often caused by the object itself.

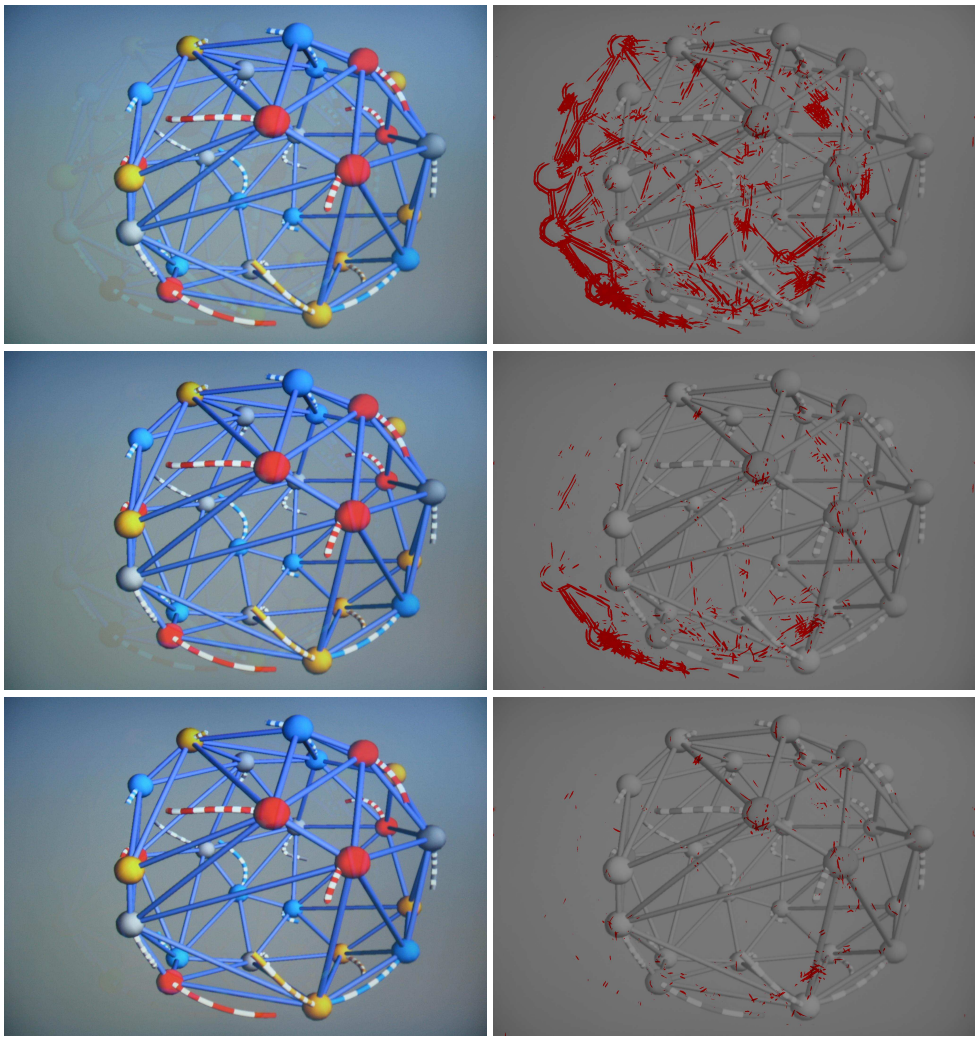

Figure 5.3: Photographs and evaluation for frame 570 of the animation. The left column shows the photos as captured by the camera. The right column shows the corresponding perceptual differences with the reference photo as reported by VDP in red. From top to bottom the images represent: no crosstalk reduction, uniform crosstalk reduction, and non-uniform crosstalk reduction.

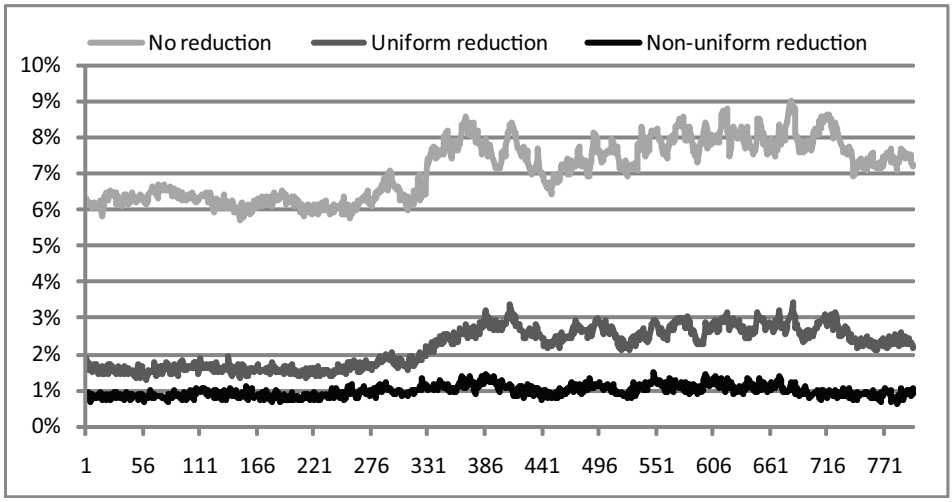

Figure 5.4: On the Y-axis the plot shows the percentage of perceptually different pixels compared to the corresponding reference photo. The animation frame number is shown on the X-axis.

In most cases object-to-background crosstalk can be corrected, as the animation background intensity of 0.6 for all three color channels is fairly high. However, object-to-object crosstalk could not be corrected in all cases. This is due to the fact that the object often does not have a high enough desired intensity in a specific color channel to reduce the crosstalk caused in that channel, for example when a blue object region causes blue crosstalk on a red object region. In this case the desired intensity of the blue channel is zero, while the unintended intensity is not.

Therefore, a second experiment consisted of determining the influence of the background intensity on the ability to perform crosstalk reduction. A sequence of photographs were taken for frame 570 with varying background intensity. For each background intensity, three photos were taken: a reference photo, a photo with normal crosstalk and a photo with non-uniform crosstalk reduction enabled. The perceptual difference evaluation was done as before and is shown in Figure 5.5.

For very low background intensities, almost no correction is possible and crosstalk reduction does not provide a much better result. However, when the background intensity reaches 0.5, almost all object-to-background crosstalk can be corrected. It can also be seen that higher background intensities reduce crosstalk in general. This is due to the fact that the human visual system is less sensitive to intensity differences when intensities are large, and the phosphors reach a peak intensity value.

Finally, the effect of different background/foreground ratios was experimentally evaluated, which is defined as the number of background pixels divided by the number of foreground pixels. Using a constant background intensity of 0.6, the size of the geometry in frame 570 was varied. For larger geometry sizes the background/foreground ratio becomes smaller, resulting in more object-to-object and less object-to-background crosstalk. Again, reference, crosstalk, and non-uniform crosstalk reduction photos were taken. The results are shown in Figure 5.6.

When the background/foreground ratio becomes smaller, more cases of impossible to cor-

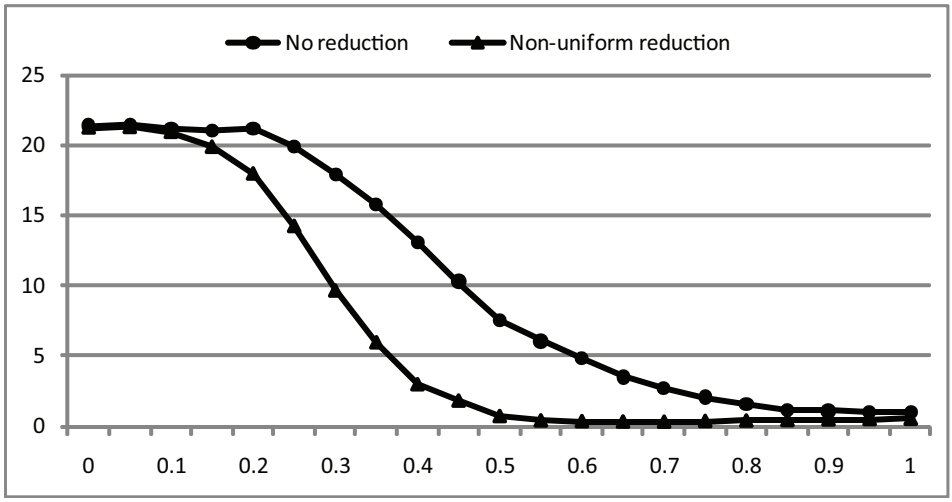

Figure 5.5: Percentage of perceptually disturbing crosstalk with varying background intensity, with and without crosstalk reduction.

rect object-to-object crosstalk occur. This explains why the non-uniform reduction algorithm performs worse with increasing geometry size. However, as it performs the best possible approximate correction, it still produces better quality than no correction. In the case where no correction is performed, the background/foreground ratio has less effect. Increasing the geometry size simply has the effect of replacing object-to-background crosstalk with object-to-object crosstalk, both of which are not corrected anyway. Finally, note that when the geometry becomes very small it becomes more difficult to perceive the crosstalk regions, as can be seen for the non-corrected curve. The reduction algorithm is not affected by this behaviour, as it is capable of correcting the crosstalk regardless.

5.2.4 On-on versus on-off reduction

Current crosstalk reduction algorithms must be performed for each pixel in the image and can be complex and time consuming. However, for a large number of pixels there is no crosstalk present in the form of disturbing ghost images. Therefore, a way to quickly determine whether a pixel shows disturbing crosstalk or not is desirable. In this way, processing can be skipped for all pixels that do not cause disturbing ghost images, thereby increasing the efficiency of the algorithm.

An image is assumed to be crosstalk-free when the image is displayed as normal for one eye and completely black for the other eye. In this way, no leakage of light from one eye to the other is possible since nothing is displayed for the other eye. The crosstalk reduction calibration tables are constructed according to this assumption. As explained in the previous section, if only the same pixel values are displayed in both the left- and right-eye frames, this causes a non-linear increase in brightness from top to bottom over the display area. All pixels receive crosstalk, hence the intensity of every pixel is increased depending on its height on the display. From the calibration data, it can be determined at runtime which pixels cause disturbing ghost images by checking for a difference between the current and previous pixel values and the amount of crosstalk; however, a problem occurs when only those pixels are

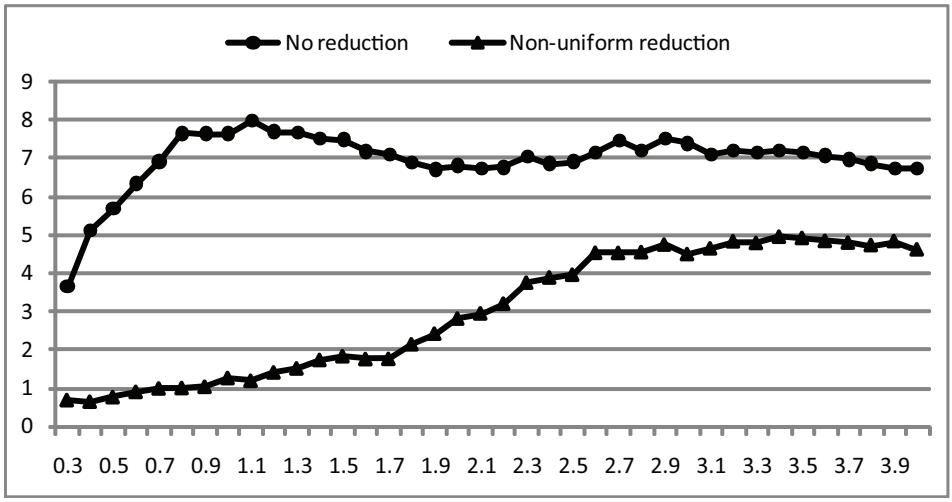

Figure 5.6: Percentage of perceptually disturbing crosstalk with varying background/foreground ratio, with and without crosstalk reduction. The x-axis shows an increasing geometry scale factor, which causes a decreasing background/foreground ratio.

	Left screen half	Right screen half
Left-eye frame	I_{desired}	I_{adjust}
Right-eye frame	I_{desired}	$I_{\text{unintended}}$

Table 5.3: The calibration setup for the (non-uniform) on-on reduction method, corresponding to Table 5.2. The left half of the screen is assumed to be crosstalk free, displaying the same pixel values in both frames. As before, I_{adjust} is user adjusted to determine what intensity to display, given $I_{\text{unintended}}$, such that a crosstalk-free pixel is perceived as assumed by the on-on reduction method.

processed. The previously discussed crosstalk reduction algorithm correctly tries to reduce the increase in intensity for every pixel, in effect darkening the entire image to compensate for global crosstalk. When only the pixels that cause disturbing ghost images are processed, those will appear darker than the unprocessed pixels due to the overall increase in intensity. This results in a ghost image with decreased intensity instead of one with increased intensity, especially towards the bottom of the screen. Therefore, this type of crosstalk reduction must be performed for each pixel, even when the pixel values in both eyes are equal and known not to cause a disturbing ghost image.

One way to avoid this is to change the assumption of what constitutes a crosstalk-free image. The idea is to still eliminate the disturbing ghost images, but to allow the global increase in intensity due to crosstalk. The previous assumption was that the left-eye image is crosstalk-free when the right-eye image is black. This is called *on-off reduction*. However, a different assumption is that the left-eye image is crosstalk-free when the exact same image, with the same value for each pixel, is displayed for the right-eye image. This is called *on-on reduction*. Under this assumption, pixels that have the same value in the current and previous display frame do not require processing because there is assumed not to be any (disturbing)

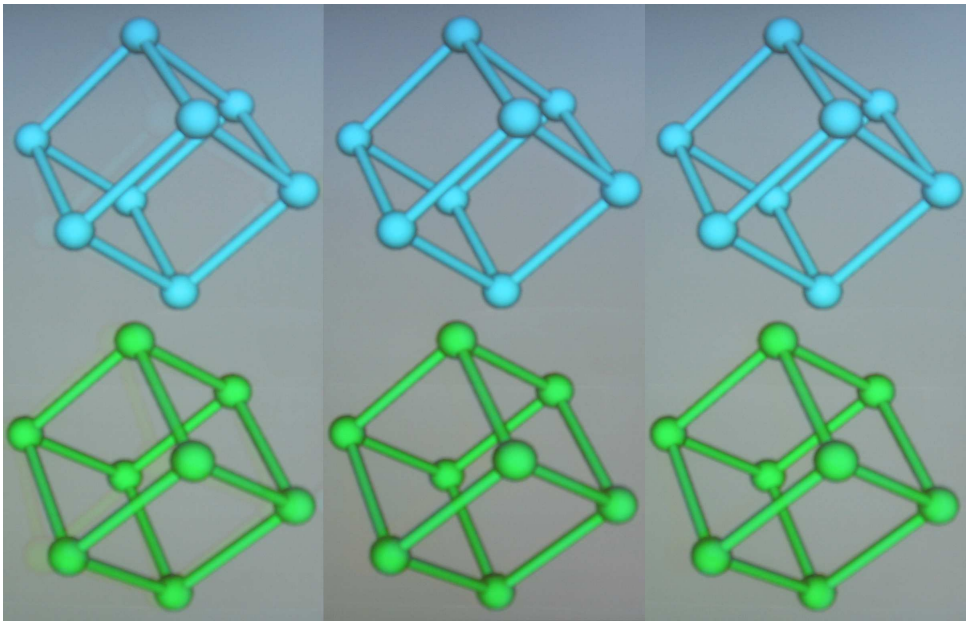

Figure 5.7: From left to right photographs with no crosstalk reduction, on-off reduction and on-on reduction are shown. There is a noticeable global increase in intensity for on-on reduction compared to on-off. The intensity for on-on is similar to the crosstalk image.

crosstalk; even though there is an increase in global intensity. Furthermore, pixels that are processed are reduced towards this same global intensity, depending on their height on the display. Those pixels will no longer be darker than unprocessed pixels, allowing only those pixels to be processed.

To implement on-on reduction, the only required change compared to on-off reduction is in the calibration procedure. All other aspects remain the same. For on-off reduction the calibration procedure displays a left screen half where the left-eye frame is displayed as normal and the right-eye frame is kept blank to model the desired intensity I_{desired} (see table 5.2 (bottom)). However, when calibrating an on-on reduction method, both the left-eye frame and the right-eye frame display I_{desired}. This is shown in Table 5.3.

To evaluate the quality of on-on reduction compared to on-off reduction, a test scene with a uniform grey background was used. Three photographs with normal crosstalk, on-off reduction and on-on reduction were taken and compared to a fourth crosstalk free reference photo, as shown in Figure 5.7 and 5.8. As can be seen from Figure 5.7, the overall intensity for the on-on reduction is higher than for on-off reduction, increasing towards the bottom of the screen (this may be difficult to see accurately in print). The brightness of the on-on reduction is similar to the crosstalk photo; however, no disturbing ghosting images are detected.

Figure 5.8 shows the VDP output of the comparisons to the reference photo as before. For the unreduced scene on the left, the disturbing ghost images are clearly visible. Both the on-off and the on-on reduction algorithms manage to almost completely eliminate all of them. This shows that the on-off and on-on reduction methods perform equally well in this respect. While the on-on reduction method causes a global increase in intensity, it still removes all disturbing ghost images that affect depth perception. The benefit of on-on reduction is that

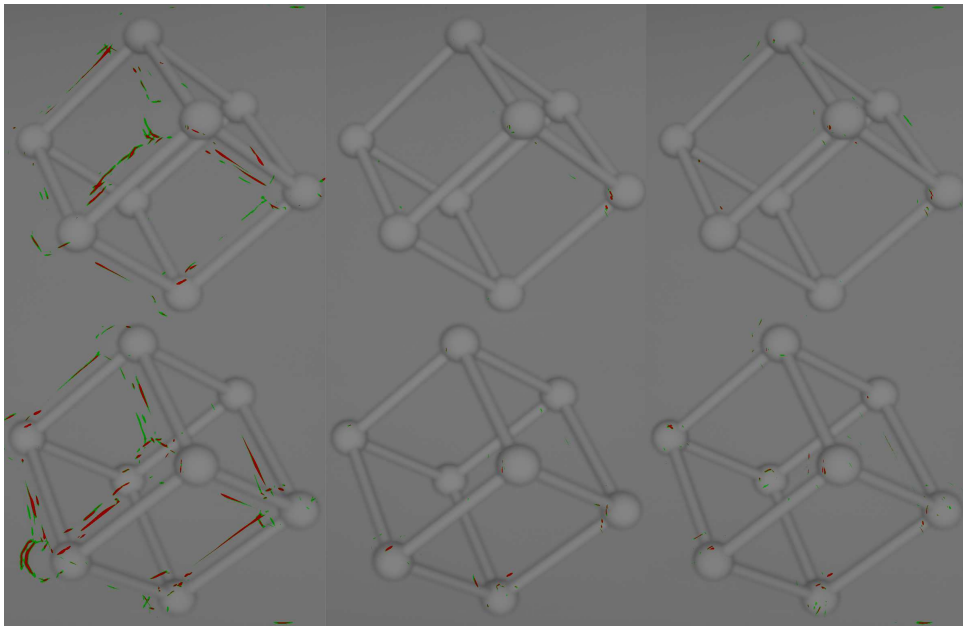

Figure 5.8: Shown from left to right are the VDP outputs between the reference and no crosstalk reduction, on-off reduction and on-on reduction. It can be seen that both on-off and on-on remove the disturbing ghost images which are present in the crosstalk image. Also, the increased intensity for on-on reduction is not perceptually disturbing according to the VDP output.

many pixels do not need to be processed, while on-off reduction had to be performed for each pixel. In this way, 77.8% of the pixels can be skipped on average in the used test scene, as is shown in Figure 5.9.

5.3 Handling uncorrectable regions

Subtractive reduction methods try to eliminate the visible crosstalk by estimating the amount of intensity leakage between left and right frames and subtracting this amount from the displayed intensities. In this way, the crosstalk will cancel out against the darkened regions rendering it invisible. The estimation and subtraction procedure is performed in the RGB color space, for each of the red, green and blue color channels independently. Whenever the estimated crosstalk for any of these channels is larger than the desired display intensity for that channel, the method is unable to eliminate all of the crosstalk completely due to the otherwise resulting negative pixels values. For these uncorrectable regions, all previous reduction methods simply clamp the respective color channels to zero.

The method described here is focused on providing better crosstalk reduction in uncorrectable regions. First, normal crosstalk reduction is performed, and uncorrectable regions where negative pixel values would result are isolated. It is known from the calibration tables how much crosstalk can be eliminated given the desired display intensity. Whenever the display intensity of a pixel is not sufficient for any color channel to perform complete reduction, that pixel is marked as uncorrectable and an extended algorithm is applied. An approach is

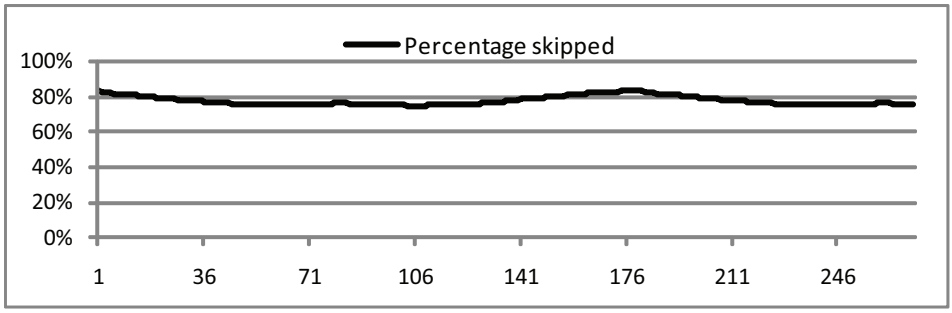

Figure 5.9: For each frame of an animation sequence, this plot shows the percentage of pixels that do not require processing when using on-on reduction. The scene consisted of six wire cubes on a grey background, similar to Figure 5.7.

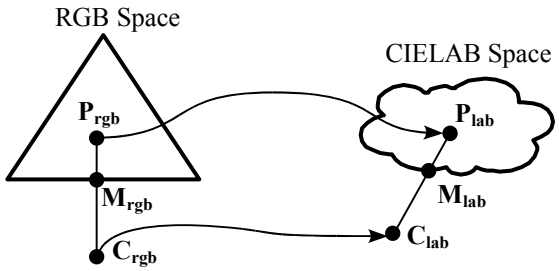

Figure 5.10: P_{rgb} represents an RGB pixel to be displayed. After RGB crosstalk reduction has been applied the desired display pixel C_{rgb} is found to be outside the displayable RGB space. RGB reduction methods now display the closest match M_{rgb} in RGB space. However, with CIELAB reduction, P_{rgb} and C_{rgb} are first converted to CIELAB space, giving P_{lab} and C_{lab}. Now it is often possible to find a better perceptual match M_{lab} than M_{rgb} is to C_{rgb}. Finally, M_{lab} is converted back to RGB space for display.

considered here that performs reduction in the 1976 CIE L*a*b* color space (from now on referred to as simply CIELAB). This approach functions by converting uncorrectable pixels to the CIELAB color space and trying to find a perceptually closer match to the desired color. In this way, some amount of perceivable crosstalk in uncorrectable regions can be reduced where previous subtractive methods could not.

5.3.1 CIELAB color space reduction

Instead of setting the uncorrectable color channel to zero and leaving the others unaffected, an attempt is made to find a perceptually closer match to the desired intensity after crosstalk is added implicitly. The RGB values of pixel P_{rgb} are converted to the perceptually uniform CIELAB color space, resulting in P_{lab}. The numerical conversion is performed under the standard assumption of sRGB phosphors and a D65 white point. Also, the amount of uncorrected crosstalk is estimated and added to the pixel's RGB values giving C_{rgb}, after which these resulting values are also converted to CIELAB color space. This results in two pixels in CIELAB space: P_{lab} corresponding to the desired pixel color, and C_{lab} corresponding to the pixel after crosstalk is added. This idea is illustrated in Figure 5.10.

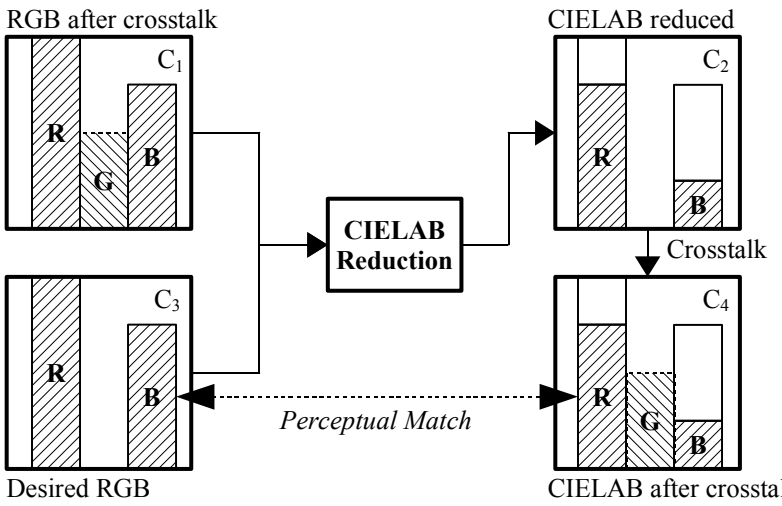

Figure 5.11: The desired display pixel is shown as C_3, while the perceived pixel after green crosstalk is added is shown as C_1. Previous subtractive reduction methods would simply display C_3 as the green channel is zero and can not be subtracted from. This results in a perceived pixel C_1. However, CIELAB reduction tries to find display values (C_2) that, after crosstalk is added (C_4), are perceptually closer to the desired pixel C_3 than C_1 would be.

Next, the uncorrectable increase in CIELAB lightness is estimated by looking at the L-channels of P_{lab} and C_{lab}. Once this is known, a similar amount of can be subtracted from the L-channel of C_{lab} in such a way that after the implicit addition of crosstalk, the pixel will be perceptually closer to the desired lightness in CIELAB space than if only one color channel had been set to zero. The a and b channels could be modified as well, but it was found that this often introduces an unintentional shift in hue that is very noticeable. By only modifying the L-channel this was avoided completely. Thus, by altering the color channels via CIELAB space, the method is able to find a perceptually closer match M_{lab} when one or more RGB channels are found to be uncorrectable. This procedure is shown schematically in Figure 5.11. After the correct CIELAB pixel values are found, they are converted back to normal RGB space and replace the original pixel values. When the resulting pixel is darker than before it will also cause less crosstalk, and this has to be compensated for accordingly. Therefore, the normal crosstalk reduction algorithm is performed again to eliminate any artefacts that might otherwise result from changed color channels in the pixel.

All of the above methods can be implemented entirely on GPU hardware in the pixel shader. The procedure to map pixels between RGB and CIELAB space is costly; therefore, this is only done for pixels that are found to be uncorrectable using the regular subtractive approach. The complete, extended algorithm still runs in real-time on modern graphics hardware.

For the evaluation of the CIELAB reduction compared to subtractive RGB reduction, a scene that shows a very challenging case of color combinations is used. Six shaded wire cubes with combinations of primary colours are positioned in front of the focal plane. A checker board of primary colours is located behind the focal plane. Again, three photographs are taken of this scene: one without reduction, one with RGB subtractive reduction and one with CIELAB reduction, and compared those to a fourth crosstalk-free reference photo. This

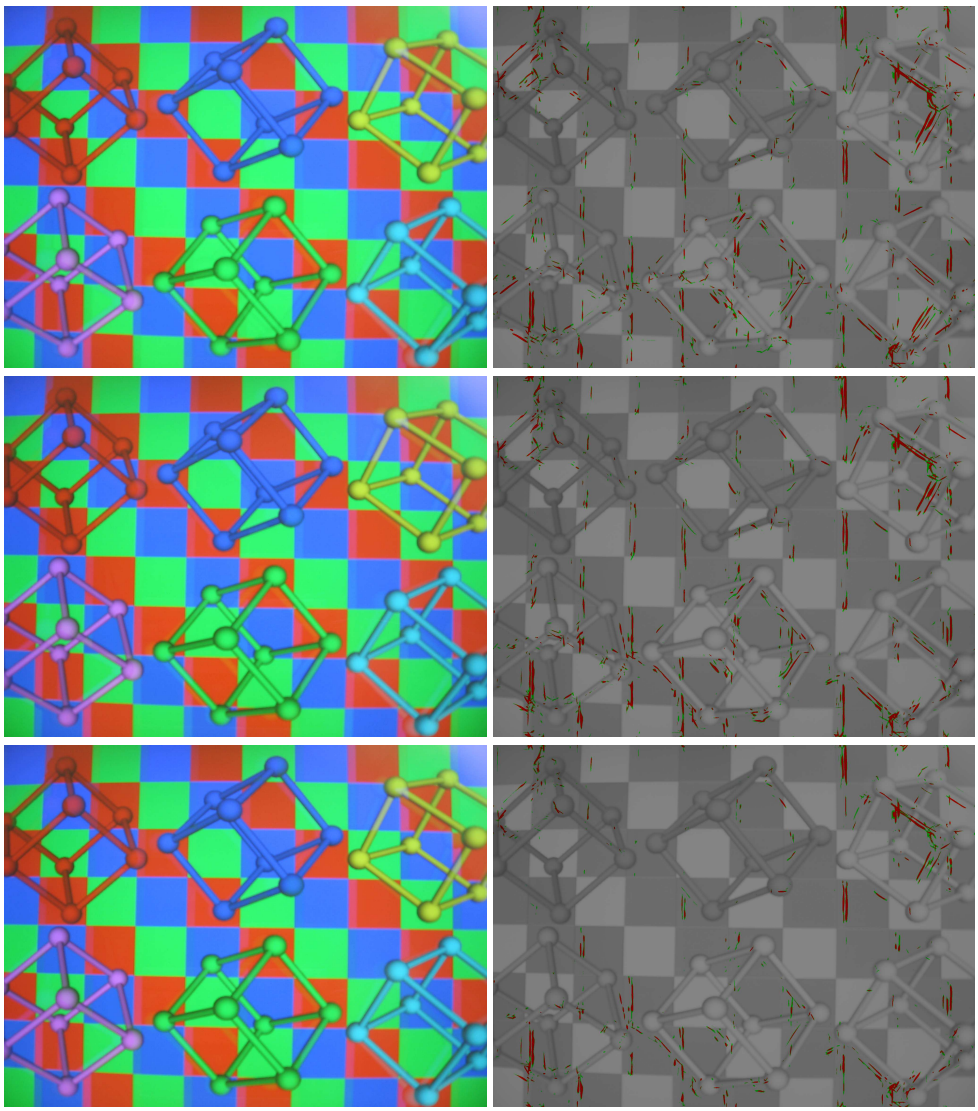

Figure 5.12: (Left) From top to bottom photographs with no crosstalk reduction, normal subtractive crosstalk reduction and CIELAB reduction are shown. (Right) The VDP difference outputs between the reference and corresponding images on the left.

	VDP % Different	Average ΔE
Unreduced	3.5	274.4
RGB reduced	3.35	216.7
CIELAB reduced	2.12	178.5
CIELAB vs RGB	36.7%	17.6%

Table 5.4: The first column shows the percentage of perceptually different pixels according to the VDP. The second column shows the average ΔE for the pixels found perceptually different with CIELAB reduction. Finally, the bottom row shows the percentage of improvement between CIELAB and RGB reduction.

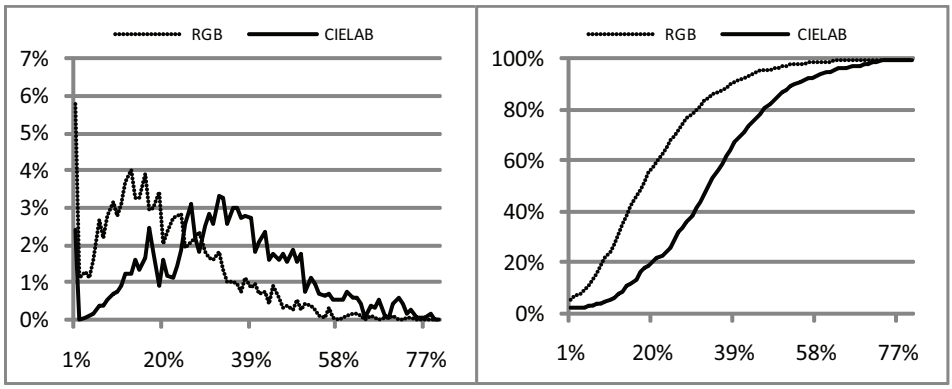

Figure 5.13: The percentage of improvement for RGB and CIELAB reduction over no reduction, for the average ΔE values per scan line for pixels that are found to be perceptually different after CIELAB reduction. On the left a normalized histogram is shown, while on the right a normalized cumulative histogram is shown. It can be seen that CIELAB reduction generally results in a larger improvement than RGB reduction when compared to no reduction.

is shown in the top row of Figure 5.12.

The bottom row of Figure 5.12 shows the output of the VDP comparisons that were performed to detect the disturbing ghost images. The green pixels indicate a perceptual difference with a probability over 75%, while the red pixels are perceptually different with a probability over 95%. It can be seen that RGB reduction performs only slightly better than no reduction at all due to the many uncorrectable regions. However, CIELAB reduction performs much better, reducing the amount of perceptually different pixels according to the VDP by as much as 36.7% compared to RGB reduction. This data is shown in Table 5.4.

Even when the VDP marks a pixel as perceptually different, there is still the question of how much difference is perceived. Therefore, for every pixel that was marked perceptually different by the VDP, the difference between this pixel and the reference pixel is measured in terms of CIELAB ΔE units. The ΔE is a standard measure of perceptual color difference. It is based on the distance between two points in the uniform CIELAB space, thereby providing a quantitative measure of the difference between two colours. The percentage reduction of the average ΔE per scan line compared to normal crosstalk has been plotted in Figure 5.13. Only pixels that were found perceptually different by the VDP are included in these calculations. Figure 5.13 and Table 5.4 show that even though perceptually different pixels remain after

crosstalk reduction, the ghost images are less noticeable in terms of ΔE differences. It can also be seen that CIELAB reduction offers a 17.6% improvement over standard RGB reduction. Also, while the global improvement is 17.6%, for some local cases CIELAB reduction is shown to perform much better. These results show that CIELAB reduction is able to reduce crosstalk in much more general cases than RGB reduction can. Also, even in cases where the crosstalk is still noticeable, it is much less noticeable than with RGB reduction.

Chapter 6

Optical-tracker Simulation Framework

Most previous tracking evaluation methods share the property that they take into account only the camera placements and the virtual space, but do not provide any performance measures for specific optical trackers or tasks. A typical strategy is to maximize the amount of space coverage while maintaining a pre-specified minimum resolution. The framework presented in this chapter is focused on model-based device tracking in virtual environments, as opposed to the larger problem of general optical tracking and space coverage. The aim is to provide quantitative data for real tasks so that the actual, measured performance of different optical trackers can be compared under varying conditions, which are not limited to camera placements alone.

A framework is presented to evaluate the performance of optical trackers in a systematic way. This framework is used to implement a number of quantitative methods to:

- Evaluate and compare the performance of different optical trackers under various conditions. This is useful for deciding which optical tracker implementation to use for a specific virtual environment, under different constraints.
- Study camera properties for various virtual environments. In this way, it can be evaluated how many cameras are required to perform a specific task; what the minimum required quality of these cameras should be in terms of resolution, distortion and focal length; and where they should be placed.
- Study environment conditions for various virtual environments. This facilitates the study of the effects of device occlusion, which is an important aspect for optical tracking. Also, different lighting conditions can be studied, such as infrared, office or day light.

6.1 Implementation

In this section, a detailed description of the proposed framework for the performance evaluation of model-based optical trackers is given. The various components of the framework

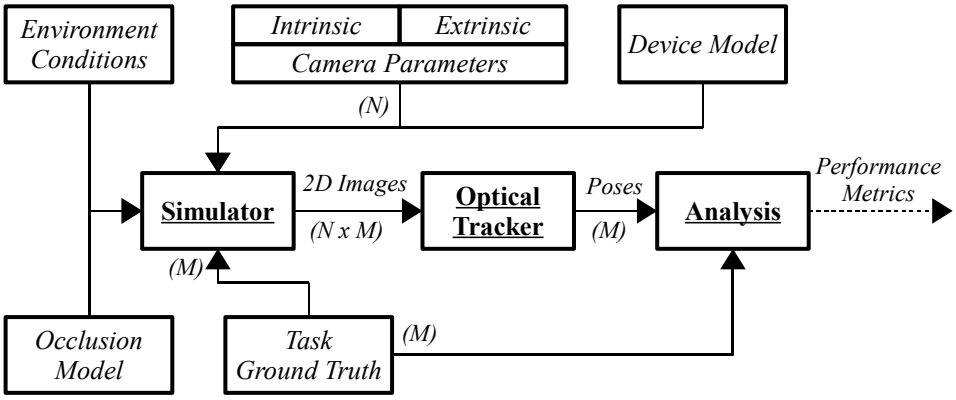

Figure 6.1: An overview of the presented framework for the performance evaluation of model-based optical trackers. The three main components are the simulator, optical tracker and analysis components. Each of these components receive one or more input data streams from supporting components. The input data provided by these supporting components can be varied to evaluate tracker performance under a wide range of conditions. Data flows are indicated by solid arrows between components.

are discussed, along with some examples of typical usage scenarios. Furthermore, a brief description is given of the implementations of two optical trackers. These optical trackers are used in Section 6.2 as examples to illustrate how the presented framework can be used in practice.

An overview of the proposed framework is given in Figure 6.1. The framework consists of three major components: the simulator, the optical tracker and the analysis component. Solid arrows between varying components represent the flow of data. The simulator is responsible for generating two-dimensional image files, which are then used as input for the optical tracker. Next, the optical tracker calculates a pose based on these input images. Finally, the calculated pose is fed to the analysis component and compared to a ground truth resulting in various kinds of performance metrics.

Each major component accepts one or more input data streams generated by supporting components, which are shown in Figure 6.1 to be environment conditions; an occlusion model; camera parameters, both intrinsic and extrinsic; a ground truth in the form of a task; and a device model. By varying the output of the supporting components, the performance of optical trackers can be evaluated under a wide range of different conditions. The three main components and their inputs are described in more detail below.

6.1.1 Simulator component

The purpose of the simulator component is to output a series of two-dimensional images representing captured camera frames. In order to achieve this, several input data are required:

- A device model that is recognized by the optical tracker. This enables the rendering of a simulated, virtual input device, which can then subsequently be tracked by the optical tracker.
- Intrinsic and extrinsic parameters for N virtual cameras. For each of the input cameras, an image is rendered as if the device model was captured with such a camera. In this

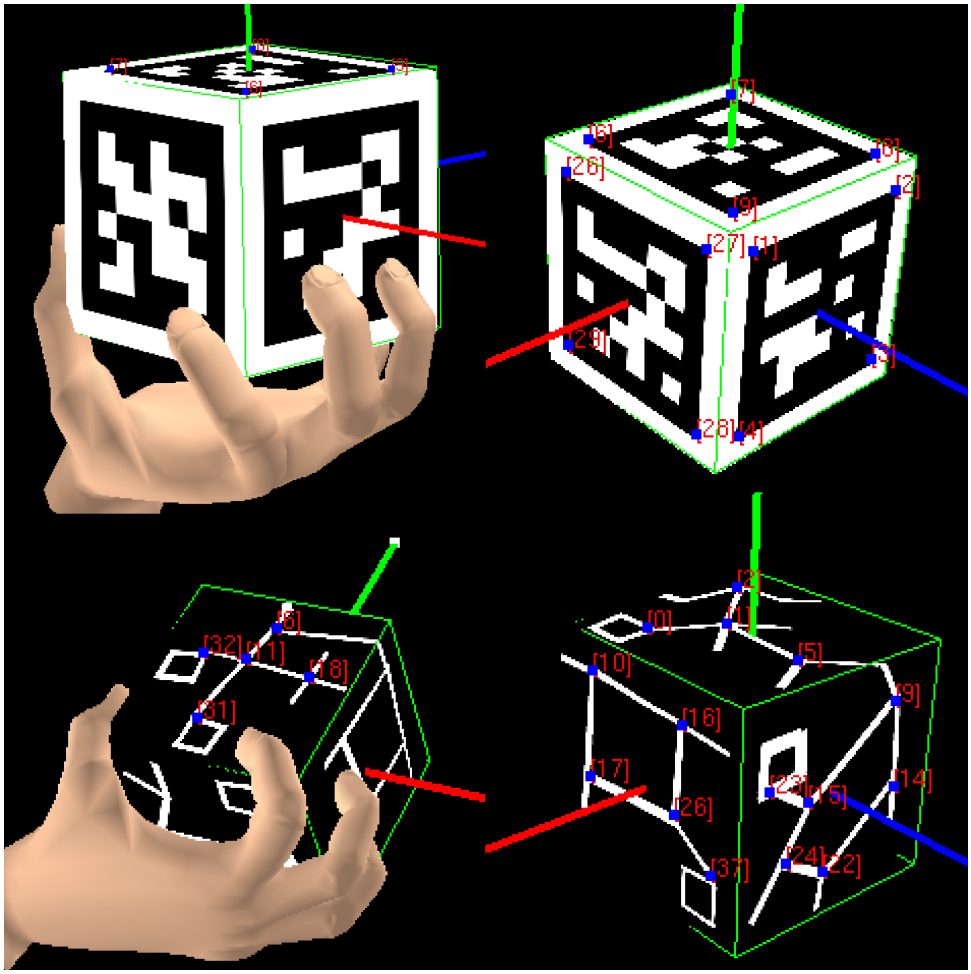

Figure 6.2: The simulated cubical input devices for the ARToolkitPlus-based (top) and GraphTracker (bottom) optical trackers. The 3D hand occlusion model is also shown. The green wire frame box represents the detected pose, with the pose x, y and z axes in red, green and blue. Detected 2D image-features are shown as blue squares with their corresponding ID-number in red.

way, camera placements and the number of cameras can be varied, along with camera properties such as focal length and resolution.

- Simulated environment conditions used for rendering. By altering environment conditions, the rendered images can be modified in various ways. Some examples of environment conditions are lighting models, such as infra-red, office or day light; background images, which can complicate the required image processing; and different types of generated image noise.

- An occlusion model where the input device is partially occluded for different cameras. Occlusion is an important aspect for optical trackers, as line-of-sight is always required

to determine a pose. Examples of occlusion models are the user's hands on the input device, or even people's bodies in larger, CAVE-like virtual environments. Simulated occluders can be rendered in addition to the input device, making it more difficult to detect.

- Ground truth data of where exactly the input device is located. A ground truth position and orientation allows the device to be rendered in that exact location, resulting in an animation sequence of M ground truth device poses. A number of different ways are possible to generate this ground truth location data. A synthetic signal can be used to evaluate different positions and orientations in the environment; however, it is also possible to record and replay a real-life interaction task. Several different types of interaction tasks can be recorded in this manner, such as positioning or pointing. Finally, different sampling schemes can be used to simulate varying camera update rates.

The output of the simulator component is a series of $N \times M$ images: for each of the M ground truth poses, N camera images are rendered. Rendering is performed using standard OpenGL functionality. A 3D polygon mesh of the occlusion model is rendered, along with a texture mapped cube representing the input device. The six textures for each face of the rendered input device are obtained through digital photographs of a real input device, which are often easier to obtain from real props than modelling their marker geometry exactly. Additionally, photographs provide a more realistic representation of the device. The required device models can either be obtained from direct measurements on the textures given the dimensions of the device, or by running an automated model-training procedure inside the simulator. The former approach is followed in the case of the ARToolkitPlus-based [KB99, WS07] optical tracker, while the latter is used for the GraphTracker [SvRvL07] optical tracker. More details about these two optical trackers are given in Section 6.1.2. The virtual input devices for the two trackers are shown in Figure 6.2. After rendering, the OpenGL frame buffer is read back to host memory using the *glReadPixels* function. The raw image data is subsequently passed to the optical tracker component, as if it was captured by a real camera. An overview of the simulator running in interactive mode is given in Figure 6.3.

An important aspect of the simulator is to correctly setup the various camera calibration matrices. The intrinsic parameters of the virtual cameras are given in terms of field-of-view angle and the width and height in pixels of the image plane. The focal-length is given by $f = 1/\tan(fov)$, where fov is the given field-of-view angle. Now let w and h be the width and height of the image plane, then the camera calibration matrix is given by:

$$u_0 = (w+1)/2 \tag{6.1}$$

$$v_0 = (h+1)/2 \tag{6.2}$$

$$a_u = hf/2 \tag{6.3}$$

$$a_v = -a_u \tag{6.4}$$

$$P_{cam2img} = \begin{pmatrix} a_u & 0 & u0 \\ 0 & a_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{6.5}$$

$$P_{img2cam} = \begin{pmatrix} 1/a_u & 0 & -u_0/a_u \\ 0 & 1/a_v & -v_0/a_v \\ 0 & 0 & 1 \end{pmatrix} \tag{6.6}$$

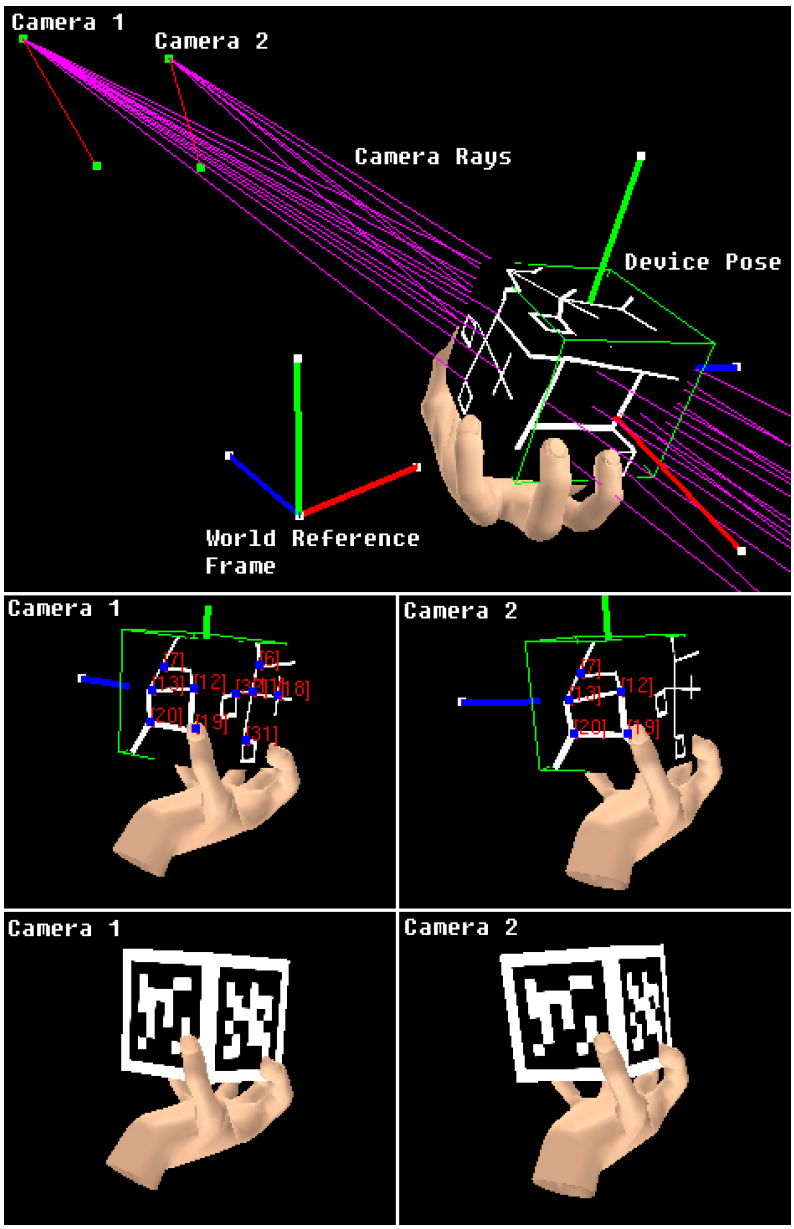

Figure 6.3: The simulator running in interactive mode. The viewpoint and input device pose can be changed using the mouse. The optical tracker component is providing visual feedback about the detected device pose in real-time. The tracker implementation can be switched at run-time. (top) A global overview of where the cameras are, including the 3D camera rays through the detected 2D feature points. (middle) The scene from the viewpoint of the individual cameras. Several GraphTracker feature points are detected, and a device pose can be established. (bottom) Due to occlusion the ARToolkitPlus-based tracker is unable to detect a pose in this identical setting.

$P_{cam2img}$ projects 3-vectors from the camera's reference frame onto homogeneous 3-vectors on the image plane. For optical tracking purposes, the inverse transform $P_{img2cam}$ is required; that is, the transform that maps points on the image plane into three-dimensional lines in the camera's reference frame.

The extrinsic parameters of the virtual cameras are defined by three 3-vectors: C_{pos}, C_{lookat} and C_{up}, representing the camera's position, the point the camera looks at and the up-direction, respectively. From these three vectors, the base vectors corresponding to the camera's reference frame can be calculated. The affine 3x4 transformation matrix $M_{cam2wld}$, formed by the concatenation of four column vectors, transforms homogeneous 4-vectors from the camera reference frame into the world reference frame, and is given by:

$$\vec{z} = (C_{\text{lookat}} - C_{\text{pos}})/\|C_{\text{lookat}} - C_{\text{pos}}\| \tag{6.7}$$

$$\vec{x} = (-z \times C_{\text{up}})/\| - z \times C_{\text{up}}\| \tag{6.8}$$

$$\vec{y} = -x \times z \tag{6.9}$$

$$M_{cam2wld} = \begin{pmatrix} \vec{x} & \vec{y} & \vec{z} & C_{\text{pos}} \end{pmatrix} \tag{6.10}$$

Given this virtual camera setup and a rendered simulation image, the tracker component can subsequently reconstruct a 3D line in the simulated interaction space from the virtual camera through a given a 2D image point. For any given two-dimensional pixel (I_x, I_y) on the image plane, the direction of the three-dimensional line originating at the camera's position and going through the pixel is given by the following expression:

$$M^{33}_{cam2wld} \cdot P_{img2cam} \cdot (I_x, I_y, 1)^T \tag{6.11}$$

where $M^{33}_{cam2wld}$ is the 3×3 submatrix obtained from $M_{cam2wld}$ by deleting the last column. This line direction is required by most, if not all, optical tracking algorithms. The origin of the line in the world reference frame is equal to $C_{\text{pos}} = M_{cam2wld} \cdot (0,0,0,1)^T$. In order to simulate an existing real-world camera, one can either construct the camera calibration matrices manually by performing any one of the standard camera calibration procedures using the real camera, or by calculating the field-of-view given the focal length of the lens and the camera's sensor size, both of which are usually given in the camera specifications. The latter approach is used in this chapter.

For rendering purposes, these matrices need to be loaded into OpenGL. In order to achieve this, standard OpenGL functionality is employed. The 4×4 OpenGL modelview matrix can be loaded from the matrix $M_{wld2cam} = M^{-1}_{cam2wld}$ after conversion for homogeneous coordinates by appending a fourth identity row. Furthermore, a scaling by $(1,1,-1)$ is performed to invert the default OpenGL depth-axis. The OpenGL projection matrix is constructed using the *gluPerspective* function, with as input the earlier specified field-of-view, width and height of the image plane. Since OpenGL internally uses the same principal point (u_0, v_0) as used for $P_{cam2img}$, the resulting projection matrix will be an exact 4x4 homogeneous version of $P_{cam2img}$. In the event that the principal point of the simulated camera is not equal to (u_0, v_0), as is the case for less-common off-axis projections, the OpenGL projection matrix must be constructed manually by a call to *glFrustum*.

6.1.2 Optical-tracker component

The optical tracker component is responsible for calculating a device pose based on the images rendered by the simulator component. In order to achieve this, the tracker receives additional input consisting of the same device model description that was provided to the

simulator, and also with the same intrinsic and extrinsic camera parameters. This is equivalent to the situation where calibrated, real cameras provide captured images. Alternative approaches may restrict this input, for example, in the case of evaluating markerless tracking, structure from motion approaches or camera calibration. The optical tracker component itself can be implemented by many different types of optical trackers that are to be evaluated under certain conditions. The output of the optical tracker component is a device pose, which can be compared to the ground truth pose.

To test the presented framework, two different optical trackers have been used. In this section, the technical details of these two optical trackers are described briefly; the reader is assumed to be familiar with optical tracking techniques. Both trackers are structured in such a way that they consist of two components: a 2D-to-3D point correspondence component and a 3D pose reconstruction component. The correspondence component is responsible for detecting 2D image features and mapping them to 3D device model features. The pose reconstruction component uses this correspondence information to calculate a device pose.

The first optical tracker is an implementation of GraphTracker, which is described in detail in Appendix C. The correspondence component operates by detecting projection invariant graph structures in a 2D image. A sample input device is shown in Figure 6.2. The vertices of these detected graphs are considered to be image features and are mapped to 3D model points. Since feature correspondence is established in a projection invariant manner, only a single camera is required, but more cameras are allowed. In fact, correspondence is established for every input camera image, and the combination of these correspondences is used as input to the pose reconstruction component.

The pose reconstruction component first determines a device pose for each camera individually using the efficient perspective-n-point algorithm [MNLF07], modified to use Horn's absolute orientation method [Hor87]. For each pose found in this way, the reprojection error is determined and the pose with the smallest error is used as a starting pose to an iterative procedure using all cameras. This iterative procedure is an extension for multiple cameras by Chang et al. [CC04] to the iterative pose reconstruction by Lu et al. [LHM00]. Further enhancements would be the use of Micheals-Boult absolute orientation determination [Mic99] and a modification of the iterative procedure to support the robust planar pose algorithm [SP06]; however, these enhancements are currently not implemented.

The second optical tracker is a modified version of ARToolkitPlus by Wagner and Schmalstieg [WS07], which in turn is based on ARToolkit by Kato and Billinghurst [KB99]. ARToolkitPlus is capable of detecting different, square markers in a single camera image. A cubical input device with different markers on each of the six sides of the cube has been constructed as shown in Figure 6.2. Next, ARToolkitPlus is used to establish feature correspondence between the four 2D corners of each square marker and the 3D positions of these corners as given by the device model. For each individual camera image, the set of 2D to 3D feature correspondences is provided to the same pose reconstruction component that was described previously in the case of GraphTracker.

6.1.3 Analysis component

The purpose of the analysis component is to provide various performance metrics based on the pose reported by the optical tracker component and the known ground truth. Therefore, the input to the analysis component consists of the same ground truth animation sequence that was provided to the simulator component and the sequence of poses generated by the optical tracker. The output of the analysis component can be a wide range of performance metrics,

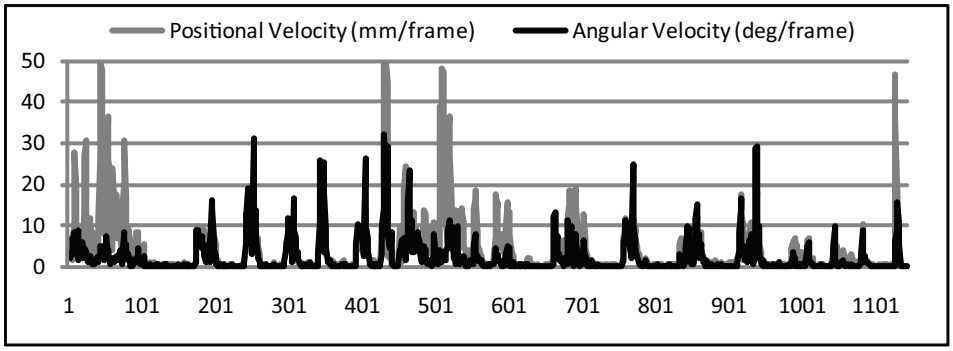

Figure 6.4: Positional and angular velocity per frame of the interaction task sequence. This information can be used to compute a weighted average error depending on the velocity of the input device.

such as position and orientation accuracy, for different animation frames or cameras, either per frame or averaged over the entire sequence, hit:miss ratios that indicate the percentage of frames for which a valid pose can be found, or processing time. Additionally, the animation sequence may contain weights to describe specific frames where accuracy is deemed to be much more important than at other frames, for example at interaction locations near a target. The output of this analysis can be compared for different settings to determine which setting is best suited for a specific task.

Spatial accuracy can be determined by comparing the pose reported by the tracker to the known ground-truth pose. This comparison can be performed for the position as well as the orientation component of the pose. The position component is given by the last column of the 4x4 pose matrix, while the orientation component is given by the upper-left 3×3 submatrix. If the position of the ground truth is given by T_{truth} and the position of the reported pose by $T_{tracked}$, then the position error is given by the distance between these two positions:

$$\varepsilon_{position} = \|T_{truth} - T_{tracked}\| \tag{6.12}$$

The orientation error requires slightly more effort to be expressed as a single number. First, the 3×3 orientation matrices are converted to the quaternions Q_{truth} and $Q_{tracked}$, for the ground-truth and reported pose respectively. Let *Re(q)* be a function that returns the real part of a quaternion, then the orientation error in radians is given by:

$$\varepsilon_{orientation} = |2\arccos(Re(Q_{tracked} \cdot Q_{truth}^{-1}))| \tag{6.13}$$

The orientation error represents the amount of rotation required, about an unspecified axis, in order to transform from the ground truth pose to the tracked pose. The same can be achieved by finding the eigenvalues of the rotation matrix $R_{tracked} \cdot R_{truth}^{-1}$; however, quaternions provide a more intuitive solution. Finally, the orientation error is translated to the $[0, \pi)$ interval and converted to degrees.

The earlier described method of determining spatial accuracy provides an error measure for each individual frame of the task animation; however, what is often required is a method that results in a single measure of accuracy for the entire animation sequence. This can be achieved through standard statistical means, such as the average or median error. To

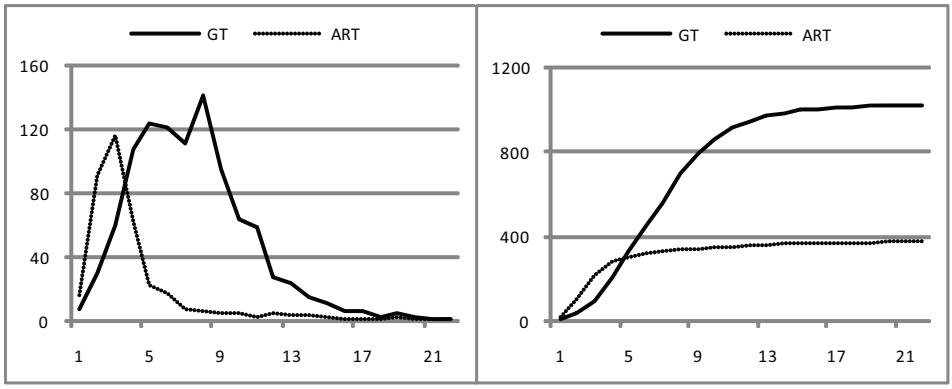

Figure 6.5: (left) Histogram for the positional error for each frame of the animation in millimeters for both GraphTracker (GT) and the ARToolkit-based tracker (ART). When a pose could not be found, the error was assumed to be infinite and, therefore, does not show in the histogram. It can be seen that ART is generally more accurate when a pose is found. (right) Corresponding (non-normalized) cumulative histogram. Here it can be seen that GT finds a pose much more often.

reduce the influence of outliers in the data, the latter method is primarily used, with the exception of the experiment in Section 6.2.4. There, a more complicated way of compacting the error data is used, calculating the weighted average error with respect to the velocity of the input device. The rationale behind this is that users often require more accuracy when they are moving the input device slowly than when they are moving it very fast. In order to achieve this, the positional and angular velocities are calculated for each frame of the task animation sequence, as is shown in Figure 6.4. Observing this figure, an arbitrary threshold velocity of 10 mm/frame is established, as well as 10 deg/frame, to constitute slow movement. Any frame with a higher velocity was not included in the average (a weight of zero), while velocities lower than the threshold were weighed linearly. That is, given velocities v_i, the weighted average error $\bar{\varepsilon}$ with weights ω_i is calculated according to:

$$\omega_i = 1 - \min(1, v_i/10) \tag{6.14}$$

$$\bar{\varepsilon} = \left(\sum_i \omega_i \varepsilon_i \right) / \sum_i \omega_i \tag{6.15}$$

Experimental results employing this scheme are given in Section 6.2.4. Instead of weighing the error by the velocity of the input device, one can also use the position of the input device as a weight. This is useful when there are critical regions in the input space where accuracy is deemed to be more important than at other regions, such as regions near target positions in a pointing task; however, this method has not been implemented.

6.2 Simulated experiments

Four sample uses of the simulation framework are shown to evaluate different aspects of optical tracker performance. In the first experiment, the accuracy of two different optical trackers

	Hit%	Position Err.	Orientation Err.
ARToolkitPlus	45%	3.4 mm	3.2 deg
GraphTracker	97%	7.0 mm	10.8 deg

Table 6.1: Summarized results for the first experiment. The table shows the percentage of hits and the median position and orientation errors for the entire task. GraphTracker detects a pose more often; however, ARToolkitPlus is often more accurate when it detects a pose.

is compared, given a fixed environment. In the second experiment, an attempt is made to determine high-quality camera placements by varying the extrinsic camera parameters. Third, the effect of camera resolution and distance is evaluated by varying the intrinsic camera parameters as well. Finally, the influence of Gaussian distributed random noise on tracker accuracy is studied. These experiments provide answers to interesting sample problems for optical trackers; however, the framework presented is not limited to these experiments alone.

The presented results are not intended to provide an answer to the question of which optical tracker performs strictly better than the other. Instead, these results should be looked upon as examples of the usage of the presented framework to determine and compare the performance of optical trackers over a broad spectrum of conditions. Each optical tracker has its own pros and cons, and the decision of which optical tracker is better suited for a particular task at hand should be based on the combination of relevant metrics and properties, rather than on a single statistic.

6.2.1 Tracker comparison in a fixed environment

The first experiment is a comparison between the two optical trackers, GraphTracker and ARToolkitPlus, described in Section 6.1.2. For this experiment, the implementation of the optical tracker components is varied, along with the corresponding device models. All other components are kept constant. In both cases the virtual input device consisted of a cube with edges of 7cm. The question asked for this experiment is, "Which optical tracker should be used for this particular environment and task?"

For the camera parameters, a setup is used that is typical for the PSS [MvL02], a near-field virtual environment. A dual-camera setup is used, where the cameras are placed above the virtual working space, pointing at the origin at a distance of approximately 55cm. The camera resolution was 640×480 pixels, and the focal length of the cameras is set to 28mm wide angle in terms of 35mm full-frame photographic equipment. Environment conditions are set for infra-red lighting. To simulate this, the device markers are rendered in black and white. To simplify processing, the background is kept strictly black and no image noise is added. The occlusion model consists of a fixed 3D model of a human hand that is attached to the cubical input device, simulating the user holding the input device. This was shown earlier in Figure 6.2 and 6.3. The hand model remains static with respect to the cube's orientation. The task, or ground truth, is a pre-recorded animation sequence of a user executing a real-life task in a near-field virtual environment with a similar cubical input device. This task was recorded using a magnetic tracker so occlusion has no effect in establishing the ground truth. The animation sequence consists of 1146 frames sampled at regular intervals. The combination of all these conditions provides a reasonable simulation of a real near-field virtual environment.

Figure 6.5 shows a histogram for the positional accuracy in millimeters for each of the 1146 animation frames and for both optical trackers. Summarized results are given in Table 6.1. For this fixed camera setup, the percentage of hits for GraphTracker is relatively

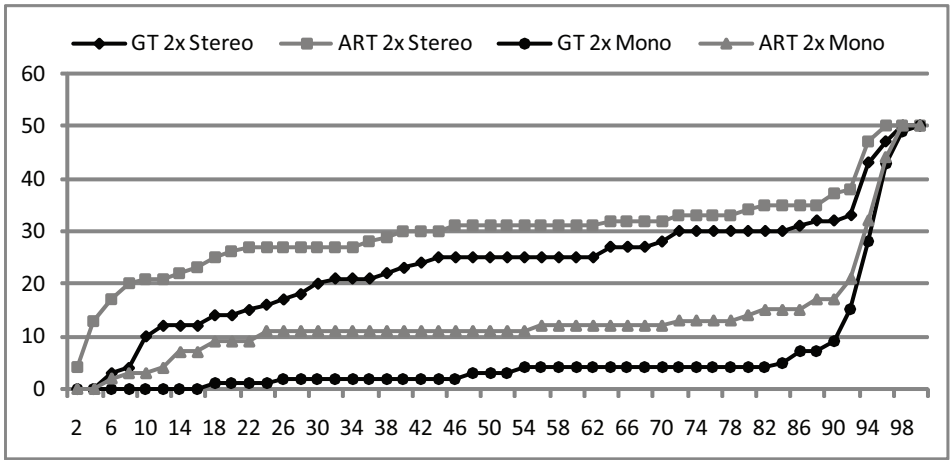

Figure 6.6: Cumulative histogram showing the percentage of hits for 50 setups consisting of two cameras for both trackers. Two different placement strategies were used for both of the two trackers: mono camera placement positioned the two cameras independently and not too close to each other, while stereo camera placement considered the two cameras as a stereo pair and placed them in close vicinity to each other. It can be seen that the mono placement strategy results in the best hit:miss ratios for both trackers. Using the same camera-placement strategy, GraphTracker shows a better hit:miss ratio than the ARToolkit-based tracker.

high at 97% despite the occlusion, whereas the hit percentage for ARToolkitPlus is only 45% due to occlusion. However, whenever a pose is detected by ARToolkitPlus, the reported pose is generally more accurate than the pose reported by GraphTracker. These results suggest that GraphTracker is the better choice for this particular task and environment; however, this may no longer be the case when the cameras can be re-positioned freely. Furthermore, if hit:miss ratio can be sacrificed in favour of enhanced accuracy, then ARToolkitPlus would be the tracker of choice.

6.2.2 Assessing camera placement quality

The second experiment uses a similar setup as the first experiment, only this time the extrinsic camera parameters and the number of cameras will be varied. All other conditions, such as the occlusion model, camera resolution and task, are kept constant as before. By varying the extrinsic camera parameters, high-quality camera placements can be empirically determined for this particular task and setup. The question here is, "Which camera placement gives good results for the execution of this particular task?"

To answer this question, 200 sets of camera placements have been randomly generated. This was done by taking the camera distance to the origin of the near-field camera setup of experiment 1, and randomly placing cameras on the quarter sphere on top and in front of the workspace defined by this distance. In this way, four different batches of 50 camera sets are generated. The first batch consists of stereo pairs of two cameras. For every set the two cameras were constrained to have a distance between 5 and 15cm. The second batch consists of two cameras per set as well; however, the distance constraint was changed so the cameras must be further than 30cm apart. The third batch consists of sets of two stereo-pairs

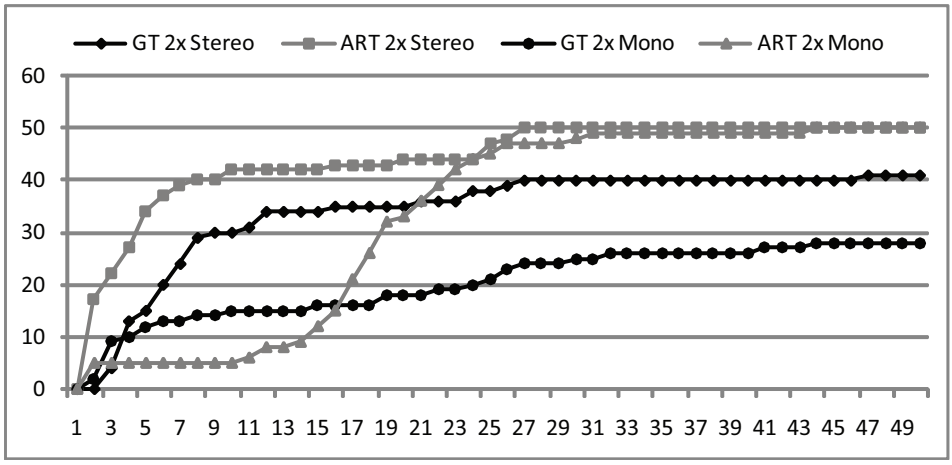

Figure 6.7: Cumulative histogram for the position error in millimeters, using the same camera setup as described in Figure 6.6. In general, a stereo placement strategy results in the best accuracy. It is interesting to note that while the mono camera-placement strategy resulted in the best hit:miss ratios (see Figure 6.6), the same strategy results in the worst accuracy. Similar results can be obtained for orientation errors (also see Table 6.2).

at a distance greater than 30cm. Finally, the fourth batch consists of four cameras, all at a distance larger than 30cm from each other.

Next, for all camera placements, the 1146 frame task was executed. For every camera placement, the average position and orientation accuracy for the task were registered, as well as the hit:miss ratio. These results are summarized in Figures 6.6 and 6.7 for the batches of two cameras. In Table 6.2 all data is summarized by calculating a subsequent average over specific batches of camera sets for the position and orientation accuracy. A number of observations can be made from this data. For two cameras in stereo-setup, ARToolkitPlus achieves poor hit:miss ratios due to occlusion. GraphTracker achieves somewhat better hit:miss ratios; however, the pose accuracy is generally worse than that of ARToolkit in the case it does detect a pose. For two independent, mono cameras the hit:miss ratios improve for both ARToolkitPlus and GraphTracker, but the pose accuracy is often reduced. This is especially so for GraphTracker in general and the position accuracy of ARToolkitPlus. This indicates that stereo camera setups are generally more accurate due to the difficulty of robust pose reconstruction using only a single camera, but result in worse hit:miss ratios due to the smaller coverage of space. In the case of four cameras in the form of two stereo-pairs (see Table 6.2), ARToolkitPlus still shows some poor camera placements with respect to hit:miss ratio. However, for four independent cameras the occlusion problem is reduced significantly due to the large coverage of space. In this case both ARToolkit and GraphTracker are capable of finding a pose in most circumstances, and ARToolkitPlus achieves a much higher accuracy in doing so. It can also be seen that four cameras are generally more accurate than two cameras. These results suggest that if one is free to choose a specific camera placement, then ARToolkit would be tracker of choice for this particular task and environment.

	ARToolkitPlus				GraphTracker			
Placement Strategy	**Stereo**		**Mono**		**Stereo**		**Mono**	
Number of Cameras	**2**	**4**	**2**	**4**	**2**	**4**	**2**	**4**
Hit Percentage (%)	41%	76%	74%	95%	54%	90%	89%	96%
Position Error (mm)	6.7	2.9	17.6	6.8	21.1	4.2	59.8	7.7
Orientation Error (deg)	3.4	1.8	2.1	1.4	7.6	5.1	18.7	6.1

Table 6.2: Summarized results for the 2nd experiment on camera placements. The rows show, respectively, the average percentage of hits, position accuracy in mm and orientation accuracy in degrees over all camera sets. It can be seen that four cameras always perform better than two, but the effect is less pronounced for mono cameras. This is due to the fact that the space coverage of a single stereo pair is low. Furthermore, stereo pairs show less error than mono setups with the same amount of cameras at the expense of a worse hit:miss ratio. This is a consequence of the difficulty to detect an accurate pose using a single camera and the lower space coverage of stereo pairs.

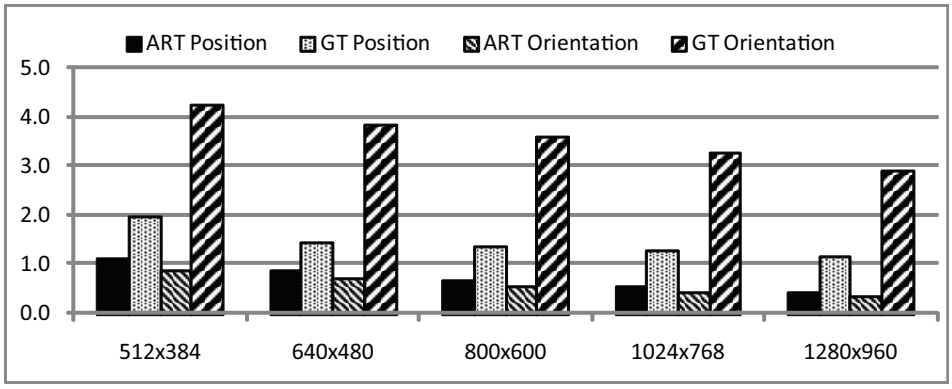

Figure 6.8: Position and orientation errors for both GraphTracker and the ARToolkitPlus-based tracker, expressed in millimeters and degrees, for various resolutions at 0.75m camera distance. A higher resolution results in lower error in all cases.

6.2.3 Determining minimum camera resolution requirements

In the third experiment, the intrinsic camera parameters are varied by changing the resolution; furthermore, the extrinsic parameters will be modified by placing the cameras at different distances from the origin. These camera distances are representative for different types of virtual environments. The other conditions remain constant. This experiment may help in answering the question, "What is the minimum required camera resolution for optical tracking in this virtual environment?"

To determine camera positions for this experiment, the results of the second experiment for the placement of four cameras were used to find the set of camera positions where the positional error was smallest. It turned out that these positions also provided good orientation accuracy, as well as a good hit:miss ratio. Next, these positions were scaled in such a way that three sets of cameras were obtained, for distances of 0.75m, 1.5m and 3.0m from the origin. These distances are representative for near-field, Workbench- and CAVE-like virtual environments. Due to increased camera distance, the focal length of the cameras was increased to a

	ARToolkitPlus			GraphTracker		
	0.75m	**1.50m**	**3.00m**	**0.75m**	**1.50m**	**3.00m**
512×384	96 / 1.11 / 0.88	93 / 3.29 / 3.76	0 / - / -	96 / 1.97 / 4.24	43 / 1220 / 103	0 / - / -
640×480	96 / 0.86 / 0.69	95 / 1.94 / 1.61	58 / 22.28 / 25.0	97 / 1.43 / 3.83	66 / 23.0 / 15.5	0 / - / -
800×600	97 / 0.68 / 0.54	95 / 1.52 / 1.30	86 / 9.20 / 12.95	97 / 1.34 / 3.60	94 / 5.97 / 9.09	16 / 2725 / 94.8
1024×768	98 / 0.52 / 0.42	96 / 1.15 / 0.97	93 / 3.51 / 4.0	98 / 1.26 / 3.25	97 / 2.86 / 4.84	38 / 2531 / 97.9
1280×960	98 / 0.41 / 0.33	96 / 0.89 / 0.76	95 / 1.98 / 1.71	98 / 1.14 / 2.90	98 / 1.53 / 3.91	67 / 23.75 / 16.8

Table 6.3: Results for the 3rd experiment on camera resolution and distance. The rows represent the different resolutions, while for each tracker the columns represent the different camera distances to the origin. Every table cell shows the percentage of hits, the average position error in mm and the average orientation error in degrees, respectively. When the percentage of hits equals zero, accuracy information is not available.

standard 50mm in terms of 35mm full-frame photography equipment. The 1146 frame task was then executed for each of these camera sets at camera resolutions of 512×384, 640×480, 800×600, 1024×768 and 1280×960.

The results of these measurements are given in Table 6.3. This table shows the hit percentage, average task accuracy in millimeters and the average orientation accuracy in degrees. Figure 6.8 gives a graphical overview of the errors at a distance of 0.75m. It can be seen that the performance of GraphTracker is reduced significantly for larger viewing distances. The performance of ARToolkitPlus decreases with distance as well, but not as quickly as for GraphTracker. Furthermore, an increase of resolution results in increased accuracy and better hit:miss ratios. In order to make effective use of an optical tracker, the hit:miss ratio should preferably be at least about 90% and definitely not any lower. Therefore, in order to use these trackers in a CAVE-like environment with cameras placed at 3m distance, a minimum camera resolution of 800x600 is required for ARToolkitPlus (preferably 1024x768) and for GraphTracker a resolution of 1280x960 is still not sufficient. This shows that when designing optical trackers for such environments special considerations have to be taken into account. For environments like the Workbench, with camera distances of 1.5m, both trackers can be used; however, GraphTracker requires higher resolutions. Both trackers show comparable behaviour in near-field environments, where higher camera resolution is apparently an inefficient use of resources. Lower resolution cameras allow for faster image processing and often achieve higher capture rates than high resolution cameras and, thus, are capable of achieving lower tracking latencies.

6.2.4 Evaluating the effect of noise

The fourth experiment is concerned with evaluating the effect of random noise on the accuracy of the optical trackers. There are several sources of random noise in an optical tracking system, of which a common few are described here. First, there is (CCD) sensor noise present in the camera, usually resulting in independent Gaussian distributed additive noise in the pixels of the acquired images; however, other forms of noise, such as speckle noise, are certainly possible. Second, for moving input devices, there's the effect of motion blur on the acquired images due to relatively slow camera shutter speeds. Third, because image processing operates on discrete pixels, the detected position of a blob may shift slightly from frame to frame due to pixels appearing just above or below the detection threshold. Other sources of noise that are usually more systematic in nature are errors in the device model description and erratically changing lighting conditions.

Since it is difficult, if not impossible, to simulate all the sources of noise exactly, a choice

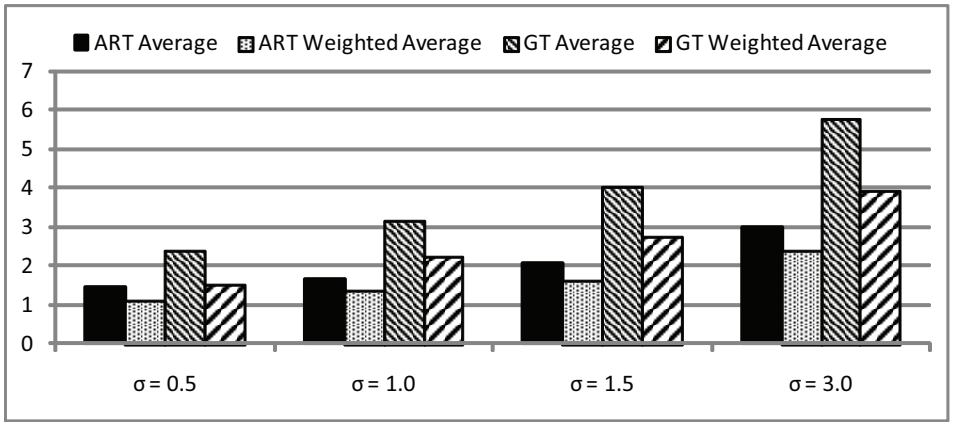

Figure 6.9: Position error for various degrees of simulated Gaussian noise in millimeters. The error increases as the standard deviation of the noise is increased. The error is measured both by taking the average over all frames and the weighted average error with respect to input device velocity. The weighted average error is lower than the average error, indicating better precision at low velocity.

was made to approximate the combined effect of random noise. This is achieved by modifying the image processing step of the optical tracker component in such a way that independent Gaussian distributed noise is added to the detected 2D image coordinates of all blobs. That is, if $I_{x,y}$ is the coordinate of the detected blob's center of mass, then the pose reconstruction algorithm will take as input $\tilde{I}_{x,y} = I_{x,y} + (N(0,\sigma^2), N(0,\sigma^2))$, simulating zero-mean ($\mu = 0$) Gaussian distributed random noise with variance σ^2. This is a reasonable approximation, since the summation of different sources of Gaussian noise is again Gaussian distributed. One aspect that this approach fails to simulate is when the image noise is so extreme that the image processing fails completely; however, this can be considered a rare case in most practical systems.

For this experiment the same camera setup as for the 640×480 resolution experiment at 0.75m was used, which provided good results for both trackers. Gaussian distributed noise with $\mu = 0$ and $\sigma \in \{0.5, 1.0, 1.5, 3.0\}$ was added to the detected 2D image features as described previously. To reduce the per-frame errors into a single number for the entire animation, both the standard average error and the average error weighted by velocity were calculated, as described in Section 6.1.3. The results of this measurement are tabulated in Table 6.4 and partially plotted in Figure 6.9. The relative increase of error with respect to increased levels of noise is shown in Figure 6.10.

A number of observation can be made from this data. First, the weighted average error is always lower than the standard average error, indicating that both trackers are more accurate when the input device is moving slowly. If one is indeed only interested in tracker accuracy when the device velocity is low, this is a favourable property. Second, as the amount of noise increases, so does the tracking error for all cases. From Figure 6.10 it can be seen that the position error of GraphTracker is relatively more affected by increased noise than that of the ARToolkitPlus-based tracker; however, the opposite is true for the orientation error. A possible explanation for this phenomenon is that GraphTracker, in general, puts more weight at orientation accuracy at the cost of sacrificing some positional accuracy. In turn, this could

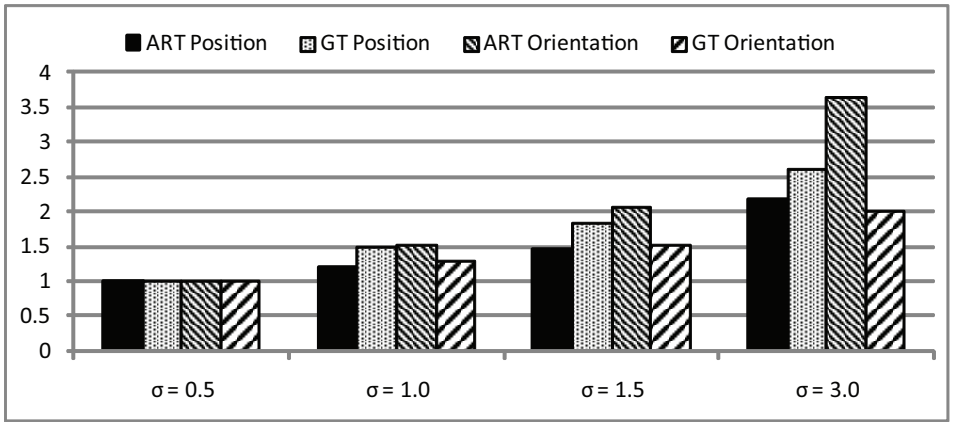

Figure 6.10: Relative error for various degrees of Gaussian noise in millimeters and degrees. The relative error is calculated with respect to $\sigma = 0.5$. This is a measure for the sensitivity to increased noise. GraphTracker appears to be more sensitive to noise for position and less for orientation than ARToolkitPlus.

be due to the detection of more 2D image features than ARToolkitPlus. Alternatively, one can argue that because of the already lower general accuracy of GraphTracker due to model description noise, further reduction in orientation accuracy is not so much influenced by additional random noise.

6.3 Considerations

By using the presented simulation framework to evaluate optical trackers, it was possible to acquire performance metrics for different situations, such as varying camera placements, that would be difficult to acquire quickly in practice. However, care must be taken not to judge tracker performance too rapidly based on a single experiment.

Most optical trackers have their own pros and cons, and the decision of which tracker to use should be based on a number of required factors, some of which may not have been tested. For example, GraphTracker is designed in such a way that it can handle any convex shaped input device, whereas ARToolkitPlus is restricted to planar surfaces. On the other hand, GraphTracker currently requires infra-red lighting, while ARToolkitPlus is capable of operating in normal daylight. Before deciding on a specific tracker, one should carefully consider the requirements and the available environment.

Using the simulator, the accuracy for GraphTracker was shown to be generally worse than that of ARToolkitPlus. This may seem counter-intuitive given the fact that GraphTracker often detects more feature points; however, one of the reasons is that the feature points that are detected are placed closer to each other than the cube corners that the ARToolkit-based tracker detects. Experience shows that when feature points are placed closer to each other, accuracy is decreased because small errors in the positions of feature points result in larger pose fitting errors. Another reasons may be the fact that the tested device model for GraphTracker has been constructed from a real input prop, using digital photographs as textures for the simulated device. Next, an automated model training step [SvRvL07] was performed using the presented framework to acquire the 3D model description. Therefore, the model

	$\sigma = 0.5$	$\sigma = 1.0$	$\sigma = 1.5$	$\sigma = 3.0$
ART Pos Avg	1.44	1.64	2.08	2.98
ART Pos W.Avg	1.10	1.31	1.61	2.38
GT Pos Avg	2.38	3.13	4.02	5.76
GT Pos W.Avg	1.49	2.23	2.74	3.89
ART Angle Avg	1.05	1.55	2.13	3.63
ART Angle W.Avg	0.89	1.35	1.85	3.26
GT Angle Avg	4.75	5.99	7.08	9.83
GT Angle W.Avg	3.18	4.12	4.81	6.42

Table 6.4: Average error (Avg) and weighted average error (W.Avg) in millimeters and degrees under various levels of Gaussian noise, for both position and orientation of the two trackers. Increased noise increases the amount of error in all cases. The weighted average error is always lower than the average error, indicating enhanced accuracy when the input device is moving slowly.

description for GraphTracker is likely to contain small errors, whereas the model description for ARToolkitPlus was constructed in a purely synthetic manner. This illustrates that all factors must be considered when comparing trackers.

In the first experiment, GraphTracker showed a much better hit:miss ratio for the given environment than ARToolkitPlus. If one is free to change the number of cameras and their placements, it can be seen from the second experiment that ARToolkit can be made to perform equally well in terms of hit:miss ratio for certain camera placements. However, these specific camera placements may only be valid for the particular task animation that was executed. For example, a right-handed user holding the input device in such a way that the left side is always completely visible probably benefits most from a camera placement with cameras on the left side. Now, if this user is replaced by a left-handed user, the same setup is likely to perform very poorly in terms of hit:miss ratio. The framework can be used to test many different types of tasks and determine a robust camera placement. In the third experiment it could be seen that GraphTracker performed relatively poorly for larger camera distances. This is due to the fact that GraphTracker has been specifically designed for near-field environments, while ARToolkitPlus was designed as a general marker tracking system. It may be possible to improve the image processing of GraphTracker to allow for larger camera distances.

Besides optical tracker performance evaluation, the simulator can also be used as a valuable development tool. The development of optical trackers is difficult due to a number of reasons. First, it can be hard to detect if all the implemented algorithms are functioning correctly under all circumstances, such as occlusion, image noise and different camera placements. It is often infeasible to regularly change camera placements in a practical real-life setting. Furthermore, once errors are detected in the optical tracker, it is usually very difficult to recreate the exact circumstances under which the error occurred. Using a simulator, the input to the optical tracker can be specified exactly. This deterministic, repeatable input greatly simplifies debugging and development. Finally, the simulator provides a way to visualize and compare the output of various algorithms. In this way, one can quickly determine which algorithm is better suited for a specific situation. Poor tracker performance can then be detected for these specific situations and the algorithms updated accordingly.

Chapter 7

Conclusions and Future Extensions

The following contributions to VR have been described in this thesis:

- An architecture using a programmable display layer (PDL) to generate individual display frames. Applications gained the immediate benefit of judder, latency and crosstalk reduction. Low and constant latency was achieved regardless of the number of polygons. It was also shown that the PDL architecture can produce images with competitive, if not superior, quality compared to level-of-detail methods.
- A new non-uniform crosstalk model and corresponding subtractive reduction method. An interactive calibration procedure was used to determine the model parameters. Furthermore, an extensions was introduced that enabled the reduction of additional crosstalk in uncorrectable regions where subtractive RGB methods failed. Compared to previous methods, this crosstalk reduction framework is applicable to a wider range of scenes and provides better quality.
- A virtual simulation framework to evaluate optical tracker performance under varying conditions. The framework provided an efficient method to study conditions affecting optical tracker performance. In this way, the effect of parameters that can not be easily changed in real-life setups can be studied, such as camera placements and resolution. Furthermore, the framework proved to be a valuable development aid for the construction and improvement of optical trackers.

Although formal user studies were not performed, subjective user feedback was acquired to qualitatively judge the benefit of these contributions. Concerning the use of the PDL architecture, users reacted very positively with respect to the decreased latency and consequent improved interactivity. Navigation and object manipulation particularly benefited from the PDL architecture, as object selection and most system control tasks were executed at the display refresh rate due to the direct influence of the latest scene graph state on the warped images. Furthermore, motion appeared more smooth and responsive and the image-doubling effect due to judder was no longer noticeable. Concerning crosstalk reduction, users reported that the stereoscopic images appeared sharper and clearer. Some basic tests also showed that crosstalk reduction increased the limits of fusion, thereby increasing the depth range of the visual space and improving depth perception.

Throughout this thesis, quantitative metrics have been used to objectively evaluate the quality of the algorithms for image warping, crosstalk reduction and optical tracking. These

evaluation methods were used in two distinct ways: to accurately compare the output of algorithms off-line and to dynamically detect an approximation of the errors at run-time. The former approach was used to compare and evaluate the efficacy of the implemented algorithms. For optical tracking in particular, the off-line evaluation aided decisions about camera placements and intrinsic properties such as camera resolution. The latter approach of dynamically detecting errors in the output of algorithms and resolving these errors by means of specialized algorithms proved beneficial for increasing the effect of the algorithms and, thus, the quality of the output. In this way, occlusion artefacts were reduced for image-warping algorithms, and the negative perceptual effect of crosstalk was further reduced in uncorrectable regions. These results show that dynamic, quantitative evaluation methods can be used to improve algorithmic output only where necessary, according to available processing time.

7.1 Future Extensions

In this section, a number of possible extensions are described that may prove beneficial to the proposed techniques. These extensions have not currently been implemented, but future work may focus on providing new algorithms to do so.

7.1.1 Evaluating image quality

Objectively assessing image quality is an inherently difficult problem. In this thesis, the method used to compare image quality operated on individual, consecutive image frames; however, animations are the main concern in virtual environments, not still images. The primary interest lies in errors that draw the attention of the user in a disturbing way. Errors at the far edges of the display or errors far away from the point of attention are often not noticed at all by the user. Also, small errors in shading are usually not perceived as disturbing. The truly perceptually disturbing errors are caused by rapid flicker in the animation caused by occlusion artefacts. Therefore, a temporal perceptual model should be used for the evaluation of the errors in output animations, such as the Sarnoff JND Vision model [Cor02]. However, the Sarnoff model is mostly aimed at determining image quality and the extent of compression artefacts in MPEG video and may not be immediately applicable to image-warping output. Finding a good temporal comparison method for image-warping animation sequences that can pin-point perceptually disturbing errors would be of great help for the further development of real-time warping systems and should be explored in the future; however, implementing methods that produce accurate results that match well with user observations is no trivial matter.

7.1.2 PDL architecture and image warping

The used image-warping algorithms have a number of limitations. First, scene translucency can not be handled correctly. While certain simple cases of translucency can be resolved by the generation of an extra depth layer, this is infeasible for applications such as volume rendering that require many translucent slices. A second class of geometry that image warping methods can not handle easily is that of deformable objects. While it is possible to warp the pixels belonging to deformable objects, it is difficult to predict the structural changes of the deforming surface. Per-pixel motion vectors corresponding to the motion of the deformable surface may help in dealing with this issue; however, this does not immediately solve the

problem of changing surface topology. As of yet, how to best handle volume rendering and deformable objects in an image-warping architecture is still an open problem.

Standard, z-buffer-based rendering systems are not the best match for image-warping architectures; real-time ray tracers are much better suited because they are capable of ray tracing individual pixels efficiently. Ray tracers can be beneficial in two ways: small subviews can be generated on the client to avoid occlusion artefacts and occlusion artefacts can be resolved more efficiently on the server once detected. Rendering only subsets of pixels on the client allows the use of many different client viewpoints with lower resolution. A major obstacle to this approach is that standard rendering systems are ill-suited to render low resolution images from many different viewpoints. This is due to the fact that a renderer basically has to render all the polygons for every such view, regardless the resolution — with the possible exception of some increased culling. Ray tracers traverse pixels and look-up the corresponding polygons. This enables the rendering of individual pixels from widely varying viewpoints. An interesting approach would be to ray trace relatively small blocks of pixels on the client and determine the viewpoint used for ray tracing according to the optic flow of the block. Each block is then warped by the server as before. Such an approach may significantly reduce occlusion artefacts, and possibly even increase performance.

Concerning the usage of the PDL architecture, an important question is for which scene the architecture is useful. For scenes containing many more than ten million triangles and a display resolution of only one million pixels, a sampling problem arises and, thus, aliasing occurs. Each pixel receives contributions from many triangles, which should be integrated above the pixel area. To a certain degree, this can be achieved by over-sampling and anti-aliasing techniques, but for very large scenes this is not possible. What is needed are output-sensitive and display-resolution-sensitive techniques, such as occlusion culling and level-of-detail approaches, which generate triangle sets matching the display resolution. Future graphics hardware is likely to be able to render such intelligently decimated scenes at 60 Hz. However, the ever increasing shading quality, as well as the computationally expensive dynamic level-of-detail and occlusion culling techniques, may limit the frame rate below 60Hz for some time. Thus, the PDL architecture remains beneficial in all cases where a constant high frame rate and low latency cannot be guaranteed otherwise.

7.1.3 Crosstalk reduction

Monitor-based passive stereo systems make use of a frame synchronized polarizing screen in front of the display and glasses containing different polarization filters for each eye. The nature of crosstalk is very similar to active stereo in this case, but the layered construction of the polarizing screen introduces a view dependent amount of crosstalk. The problem is that the crosstalk increases in a non-linear fashion across the individual horizontal bands that make up the polarizing screen. The exact location of these horizontal bands and their seems with respect to the display depends on the user's viewpoint. Therefore, it is impossible to determine the amount of crosstalk reduction required for a display pixel without knowing the user's viewpoint. This kind of crosstalk can, thus, only be compensated for by using accurate head-tracking.

Recently, stereoscopic displays have begun to appear that make use of TFT-LCD panels instead of CRTs. Different technologies are used to produce stereoscopic images on these displays. One methods is based on the physical separation of pixels for a left- and right-eye view, for example in a scan line interlaced or checker board pattern. Since left and right eye images no longer share pixels using this technique, crosstalk is no longer caused directly by

the display; however, the LCS glasses do still suffer from crosstalk due to imperfections in the shutters. Other LCDs run at higher update frequencies of 100 Hz or more, as opposed to the standard 60 Hz, and succeed in generating stereoscopic images using the regular time-sequential display of left and right images. For this method, the pixels are no longer separated physically, introducing crosstalk from the display itself as well. These observations indicate that crosstalk reduction can also be beneficial for LCDs, not only for CRT displays.

Currently the calibration of the non-uniform model is done interactively by user inspection. The procedure is tedious and error prone. However, in combination with the proposed quantitative evaluation method, it would be possible to automate this task. When running the calibration program, a computer controlled photo or video camera could be placed in front of the LCS glasses. Next, model parameters are adjusted automatically, after which images of the calibration regions can be compared to see if they are perceptually equal. As long as this is not the case, the model parameters can be automatically re-adjusted in a feedback loop. Automatic calibration would be especially desirable in virtual environments with large screens. It is a known fact that the amount of crosstalk is also dependent on the angle of view through the LCS glasses [WT02]. For CRT monitors this does not pose a real problem, but for large screens it does. An automated calibration procedure could calibrate for many different angles of view, making use of head-tracking information.Then, depending on the users viewpoint, different calibration tables can be selected for the crosstalk reduction algorithm.

The CIELAB color space used for additional crosstalk reduction in otherwise uncorrectable areas was actually designed for large patches of fabric and small color differences, not for small regions of color on a computer display. Since the effect of crosstalk is relatively small, the color differences are not very large in general, and the method appeared to provide good results. However, it may be beneficial to examine the use of other color spaces in the future.

The CIELAB reduction method is currently based on the calibrated reduction tables for the RGB-color channels. It may also be desirable to implement a user calibration routine that operates directly in CIELAB, or another perceptual color space, combining the effect of all three color channels. However, it is not yet clear how this can be achieved efficiently with only a small number of calibration parameters and points; calibrating all possible combinations of pixel values in the current and previous display frame is infeasible, so a good, representative subset that can be interpolated needs to be found.

Even with CIELAB crosstalk reduction, it is not possible to completely eliminate visible crosstalk ghosting in all cases. For example, when the background is black it is impossible to perform any reduction. However, this is an inherent problem to software crosstalk reduction and not a specific fault of the proposed method. In many cases, for example scenes with coloured textures, various shades in background color and many coloured objects causing crosstalk onto each other, the method provides an improvement over classic subtractive reduction methods. Even in the cases where visible crosstalk can not be eliminated completely, the method still provides a better, less noticeable alternative to previous methods. In practice it is seen that most, if not all, visible crosstalk can be eliminated if there are not too many very dark regions in the scene.

7.1.4 Optical tracking evaluation

The presented optical tracking evaluation framework does not take end-to-end latency into account. While it is possible to measure execution times for the optical tracker component, the framework is unable to measure real, observed latency. In order to do so a number of ad-

ditional factors need to be simulated, such as the cameras' CCD fill rates, data transmission times, application/simulation update rates and scene graph rendering rates. To be accurate, even the display refresh rate needs to be modelled, along with the difference in update rates between various application components. Since this information varies greatly among different applications, and in some cases may not be available, the choice was made not to provide any latency modelling in the presented framework.

Another aspect that was not simulated are the various types of camera distortions found in real cameras, such as pincushion or barrel distortion, motion blur and focus effects; however, it would be fairly straightforward to add these distortion effects to the simulation renderer. Most real-life cameras are calibrated in such a way as to minimize the effect of distortion in a post-processing step. Therefore, the assumption was made that the simulated cameras are perfectly calibrated and can be treated as linear cameras. Motion blur can easily be simulated by blending in images from previous frames with the current frame. However, motion blur is likely to introduce some noise in the detected feature points, which has already been simulated directly. The same argument is valid for factors such as anti-aliasing in the rendering and blur in the camera images due to focussing issues.

The occlusion model that was used for the simulator consisted of a geometric model of a human hand attached to the input device in a rigid manner. It would be interesting to use an animation sequence where the occlusion model is not fixed and changes in the same way as for a real user. One way to achieve this would be to record a real interaction session with multiple cameras and subsequently making use of hand tracking to reconstruct a realistic 3D occlusion model.

Another problem is that the interface to optical trackers needs to be standardized in order to ease the integration of different optical tracker implementations in a common framework. This may prove difficult for commercially available, closed optical tracking systems, which often integrate camera hardware and operate as a black-box. A number of strategies can be employed depending on the specific optical tracker setup and hardware. A custom virtual camera driver can be written that reads image files from disk or shared memory instead of using a real camera. The simulation framework can then write rendered images to disk, or copy them directly to shared memory, where they are interpreted as camera footage by the virtual camera driver. This driver can then be plugged in to the existing optical tracker system. In cases where it is not possible to change the camera driver used by the tracker, it may be possible to intercept and alter the data stream from the cameras to the driver, which is often a USB, firewire or ethernet link. In that case, the simulation has to produce its images in the same data representation as used by real cameras and send the data over the actual link. This solution is very system dependent, and a virtual camera driver would be the preferred method of dealing with closed tracking systems.

Appendix A

Experimentation Platform and Evaluation Methods

A.1 VR environment

For the implementation of the PDL architecture, two different hardware setups have been used. The first consisted of an Nvidia GeForce 8800 GTX for the client GPU and a stereo-enabled Nvidia Quadro FX5600 for the server GPU. The GPUs were connected over a PCIe 1.1 bus. This system contained an Intel Q6600 2.4 Ghz quad-core processor; therefore, the client and server processes could each utilize a separate core, as well as a GPU. For the stereoscopic display, an Iiyama Vision Master Pro 512 22" CRT monitor has been used, which operated at 120 Hz in order to achieve 60 Hz per eye. For head-tracking a Logitech Acoustic headtracker running at 50 Hz was used. Furthermore, a Polhemus 6-DOF Fastrak device that is sampled at 120 Hz was used. On this system crosstalk and latency were evaluated. This system is shown in Figure A.1

The second system consisted of two independent Nvidia GeForce 260 GTX GPUs connected over a PCIe 2.0 16x bus and an Intel Core2 Quad Q9950 2.83Ghz CPU. For the stereoscopic display a Samsung HL67A750 60 Hz stereoscopic DLP TV was used. On this system the performance of the image warping algorithms for the PDL architecture was evaluated. This system was shown in Figure 2.1.

As crosstalk is only visible by observing the display, result data was acquired by taking photographs of the display through the shutters. For this purpose, a Canon A510 digital camera was used that was fixed in front of activated NuVision LCS glasses, taking photographs of the Iiyama Vision Master Pro 512 CRT display. The photos were taken with the following fixed camera settings: shutter speed 0.5s, aperture F/2.8, fixed manual focus, fixed white balance, ISO 50, with the highest quality settings. The camera was operated automatically over USB in a darkened room, ensuring frame synchronization and unchanged conditions.

For latency measurements, the first described system with the IIyama CRT monitor was used. The used input device was the Polhemus Fastrak. For image capturing, a Leutron Vision LV7500RS B&W camera was used that captured frames at 60Hz.

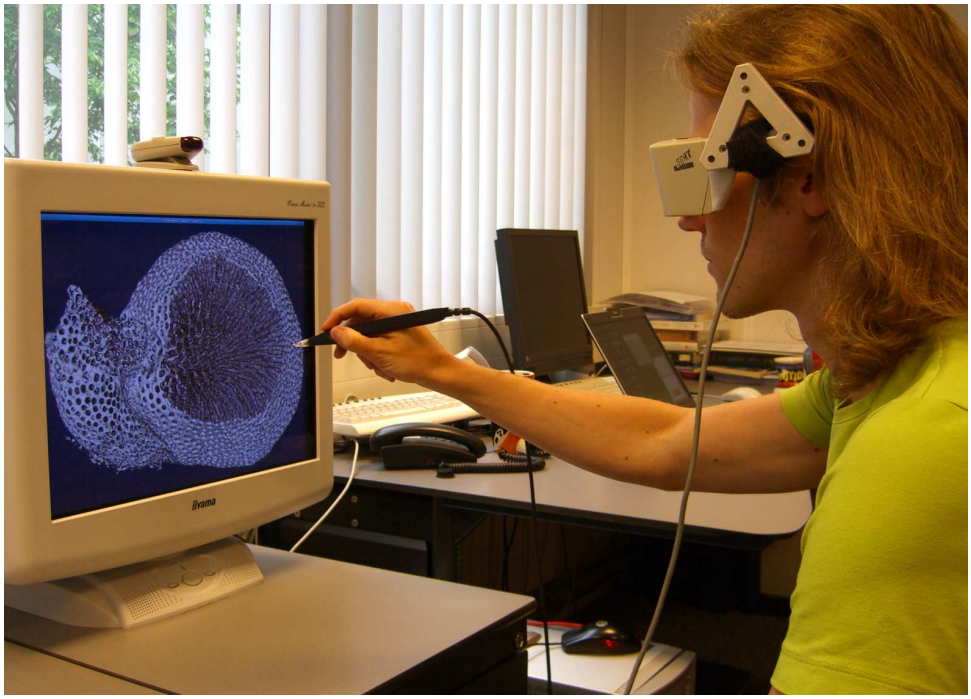

Figure A.1: The system used for evaluating crosstalk reduction and measuring latency on the PDL architecture.

A.2 Evaluating image quality

To evaluate image quality, we have used a method that compares individual output images to reference images (see Figure A.2). The evaluation procedure takes two input images: an image of the actual output and a reference image of what this output should be like. These two images are then compared according to some comparison operator. The result of this comparison is stored in a pixel map representing the detected errors. This evaluation method depends on a number of choices:

- The type of images to use as input for the comparison. For example, the input may consist of color images, depth maps or normal maps, each corresponding to different types of errors.

- How to acquire the images. In some cases digital images in the form of frame buffer dumps may be available; however, in other situations this may not be the case. Other options may be available then, such as photographs of the display.

- How to obtain a reference image to compare to. A reference image is required that reflects the ideal output of the algorithm. This reference needs to be generated in some way, for example by rendering a scene off-line at the highest quality possible.

- What comparison operator to use. The image comparison can, for example, be based on purely statistical methods or on perceptual evaluation methods.

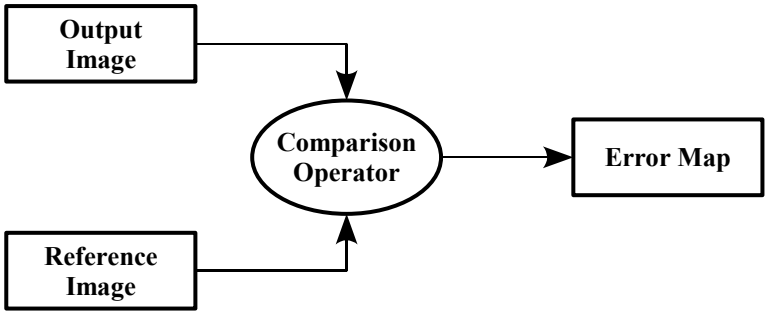

Figure A.2: Evaluating frame-by-frame image quality. An output image produced by an algorithm is compared to a reference image using a comparison operator. The output is a pixel map depicting where errors are detected and, optionally, their magnitude. This pixel map can be further summarized using statistical methods, such as the total sum of errors.

A.2.1 Image warping evaluation

Two types of images are used to evaluate the quality and amount of error in images produced by image warping on the PDL architecture: color images and depth maps. The color images are copies of the digital color buffers, while the depth maps are copies of the z-buffers used for rendering, which contain the depth of each individual pixel. To acquire these images, the hardware frame buffers are written directly to disk in a lossless image format. A reference image is obtained by directly rendering the scene using a conventional rendering system instead of image warping.

Two distinct comparison operators are used for the evaluation, each of which is capable of detecting a different type of error: occlusion errors and sampling errors. The operator detecting occlusion errors achieves this in the following way. For each pixel, the warped depth is compared to the reference depth map. When the depth of a warped pixel indicates it lies further away than a set threshold from the depth of the corresponding reference pixel, it is marked as an occlusion error. Note, if a warped pixel is closer to the viewer than the depth of the reference, then it can not be a hole pixel — but it will marked as a sampling error later.

To detect sampling errors, a two-step method is used. First, both the warped and the reference color images are filtered using a small, three-pixel-wide Gaussian kernel. Next, both images are converted to the Lab perceptual color space. The two images are then compared on a pixel-by-pixel basis, ignoring any pixels that were previously detected to be holes. Pixels are marked as sampling errors if the distance between them is larger than a threshold value of 20 units in the Lab space. This threshold was chosen after some direct experimentation with output images and produced results generally matching expectations about errors. The Gaussian filter and the threshold value are used to avoid marking very small individual pixel differences, which are generally imperceivable, as sampling errors. Additionally, pixels are marked as sampling errors if their depth does not match the reference depth. This kind of sampling error occurs frequently at depth edges, where the splat-size is too large and covers a few pixels behind it.

The summarized error value that is reported for the entire image frame is the percentage of pixels that are marked as either sampling errors or occlusion errors. In some cases, both values are given separately. The output of this approach has been shown previously in Chapter 4.

Figure A.3: Example of an experimental setup to evaluate crosstalk. A camera captures the CRT display through the left eye of activated LCS glasses.

A.2.2 Crosstalk evaluation

Evaluating the quality of crosstalk reduction requires some effort. One of the problems in acquiring the images is the fact that crosstalk is only visible to an external viewer and can not be detected by comparing frame buffer dumps. To externally register the amount of crosstalk in a given scene, digital photographs of the screen are taken through LCS glasses. The digital camera is placed in front of the left eye of the LCS glasses, which are shuttering in the normal frame-synchronized manner. This setup is similar to the one used by Woods and Tan [WT02] and is shown in Figure A.3.

Using this setup, a reference photograph is taken of an application frame where the left scene is displayed in the left eye and the right-eye display is kept blank. In this way, no crosstalk is present in the reference photograph. Next, a photograph is taken of the same application frame where the left and right scenes are displayed in the left and right eyes. This photo includes crosstalk for the left scene, originating from the right scene. Finally, another photograph is taken but this time with the crosstalk reduction algorithm enabled. All photographs are taken with identical camera settings and a relatively slow shutter speed.

An important issue is that the comparison operator should be chosen in such a way that only crosstalk that is perceptually disturbing is detected. The first obvious choice would be to use statistical image comparison metrics. However, there are three conditions for which

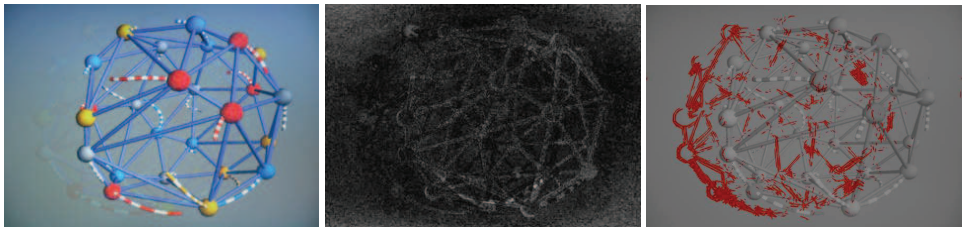

Figure A.4: An attempt to determine the perceptually disturbing crosstalk in an image. (left) An image containing crosstalk. (center) A comparison with a crosstalk-free reference according to the per-pixel RMSE, where pixel intensity represent the magnitude of the RMSE. Not much information can be obtained from this result. (right) A comparison based on the VDP, where detected errors are depicted in red. The errors detected by this method matched well with subjective experience of where the disturbing crosstalk is located in the used test scene.

statistical methods are known not to function optimally: geometric misalignment, global intensity shift and noise. In the case of evaluating crosstalk, geometric misalignment may occur due to small, single-pixel shifts between two photographs. Furthermore, crosstalk causes a global intensity shift, increasing the overall intensity of the display from top to bottom in a non-linear way; however, when the intensity increase is gradual, this is not perceptually disturbing or even discernible. Two images can differ a great deal in their pixel values without showing any perceptually discernible differences. Finally, photographs of the display show relatively high per-pixel noise. Large differences between corresponding pixel values in the two images can be caused due to noise, small image shifts or, especially towards the bottom of the screen, an increase in global intensity, which is of little interest. Therefore, statistical methods of comparison, such as the root mean squared error (RMSE), do not provide any usable information in this case. This discrepancy of statistically based comparison methods was noted before by various authors (an overview is given by McNamara [McN00]). An example of this is given in Figure A.4.

A solution is to use a human-perception-based comparison operator that is capable of isolating the truly disturbing perceptual artefacts due to crosstalk. A survey of many image quality metrics is given by Ahumada [Ahu93]. Concerning perception-based methods, which are usually based on a contrast sensitivity filter (CSF), Zhou et al. [ZCW02] found no significant differences in effect between the most widely used variants. Since an implementation of the Visual Differences Predictor (VDP) [Dal93] is readily-available as a complete software package, the VDP was the first choice for the purpose of evaluating crosstalk. It should be noted that the VDP can in no way guarantee correct results; that is, there may be discrepancies between VDP predictions of differences and actual observed differences. However, after some initial experimentation, it was found that the locations of errors as indicated by the VDP matched well with subjective observations of where errors caused by disturbing crosstalk occurred for the used scenes. Therefore, the VDP was used as the algorithm for detecting disturbing crosstalk; however, other algorithms may have been equally suitable.

To estimate the perceptual differences between two images, the VDP first applies a non-linear response function to each image to estimate the relation between brightness sensation and luminance. Next, the image is converted into the frequency domain and weighted by the human contrast sensitivity function (CSF), resulting in local contrast information. Additional sub-band processing based on the human visual system (HVS), in combination with masking functions, provides scaled contrast differences between the two images. Finally, these con-

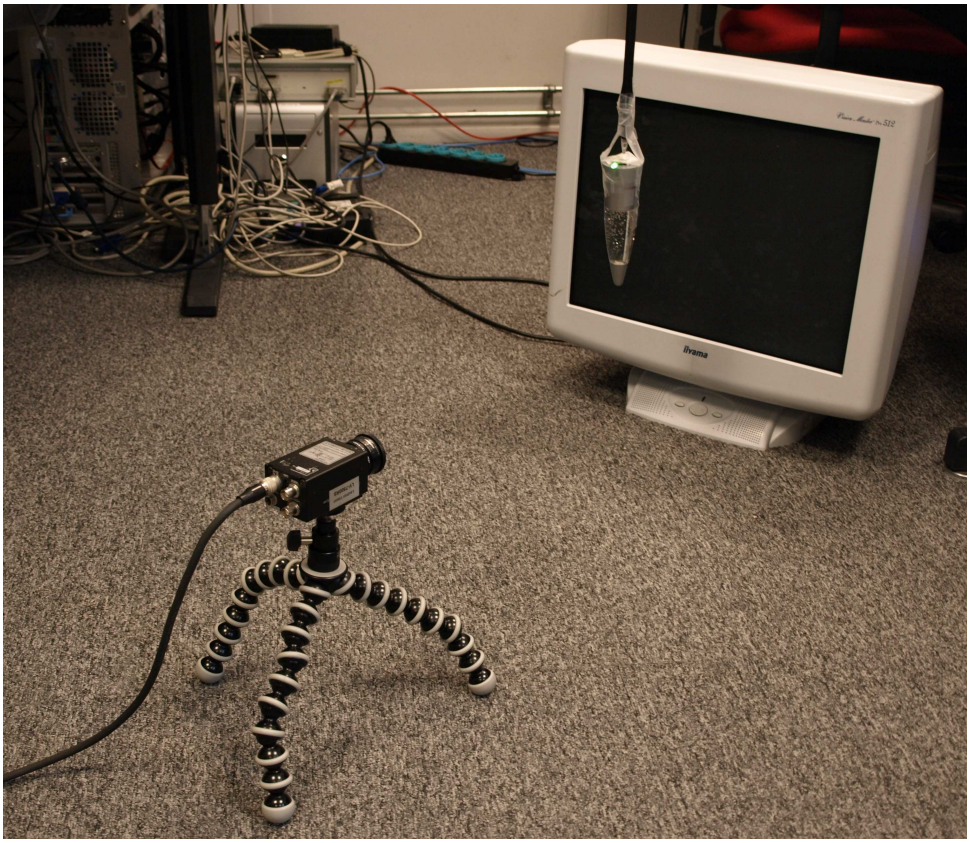

Figure A.5: Example setup to measure latency. A Polhemus Fastrak pen device with an LED attached to it is used as a pendulum. The swinging pendulum is filmed by a Leutron camera at 60 Hz, along with its virtual representation on the Iiyama CRT display.

trast differences are used as input to a psychometric function to determine the probability of perceptual difference. More specific implementation details are given by Daly [Dal93].

The output of the VDP algorithm is a probability map that indicates for each pixel the probability a human viewer will perceive a difference between the corresponding two pixels in the input images. To quantify the results, the percentage of pixels that are different with a probability of over 95% is determined. This is shown in Figure A.4. The threshold was liberally chosen because the typical VDP output per pixel was either very high ($>$95%) or very low ($<$5%), with hardly any values in between. This gives a measure of the amount of perceptually disturbing crosstalk.

A.3 Measuring latency

To measure the latency of the PDL architecture, a real pendulum with an LED attached to it is video-captured, and simultaneously a representation of the tracked pendulum in the form of a sphere is rendered on the display. This is shown in Figure A.5. Using image processing techniques, the (x, y) coordinates of the two blobs can easily be found in the camera images,

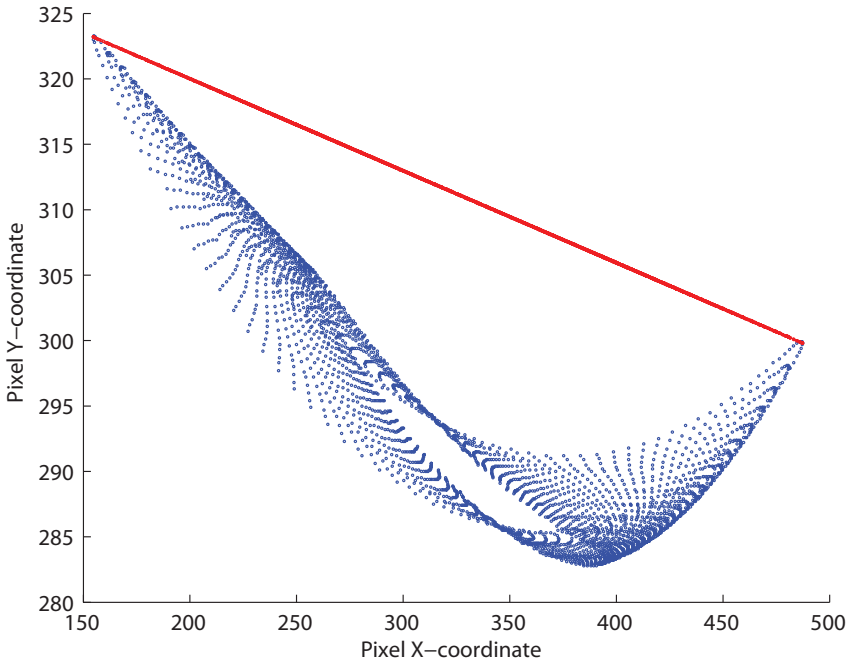

Figure A.6: The blue circles show the pixel coordinates of the detected blob for the real pendulum. The red line represents the corresponding normalized pixel locations.

which represent the LED and the rendered sphere. This results in two sequences of (x,y) image coordinates: one for the real pendulum, and one for the tracked virtual pendulum. For the real pendulum this raw data is shown as blue circles in Figure A.6.

Next, this two-dimensional pixel data needs to be converted into a one-dimensional signal. To do this, the data is normalized by orthogonally projecting all 2D points on the line connecting the two extreme points in the data. This is depicted by the red line in Figure A.6. The distance from the projected point to one of the extreme points is then calculated and divided by the distance between the two extreme points themselves. This procedure maps all 2D points to a 1D range of $[0,1]$. Finally, the $[0,1]$ range is converted to $[-1,1]$, which corresponds more closely to a regular sine wave. The normalized data for both signals is shown in Figure A.7, where the blue line is the real signal, and the red one the tracked signal. The latency of the system is the phase shift between these two signals, which should both be damped sine waves.

To determine the phase shift between the two signals, both signals are converted to the frequency domain. This conversion is accomplished by a fast Fourier transform (FFT). The output of the FFT algorithm is shown in Figure A.8. Next, it is determined at which frequency the complex output of the FFT has the largest (complex) absolute value. This corresponds to finding the peak in Figure A.8. The absolute value of this complex number represents the amplitude of the corresponding sine wave, while its argument represents the phase angle. Since both signals should be similar in shape, up to a phase shift, the FFT peaks should be at the same frequency location for both signals. The complex argument of both peaks represent

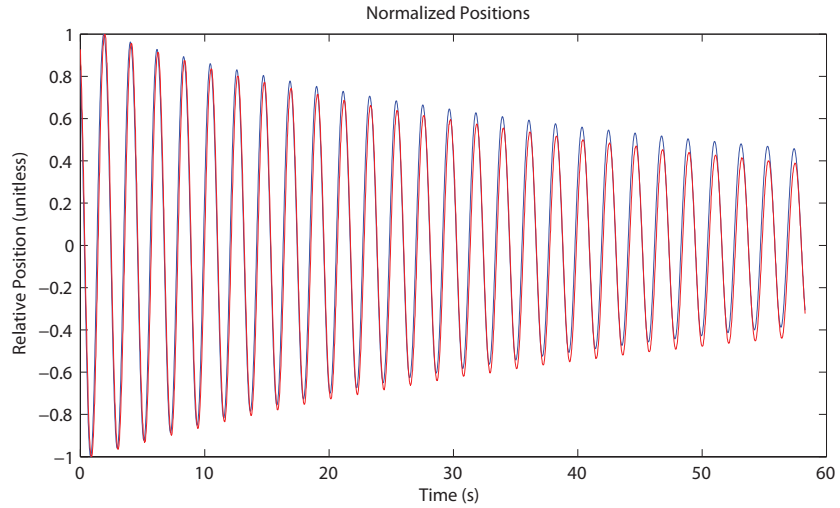

Figure A.7: The two normalized signals set out against time. The red line shows the real pendulum, while the blue line shows the simulated one.

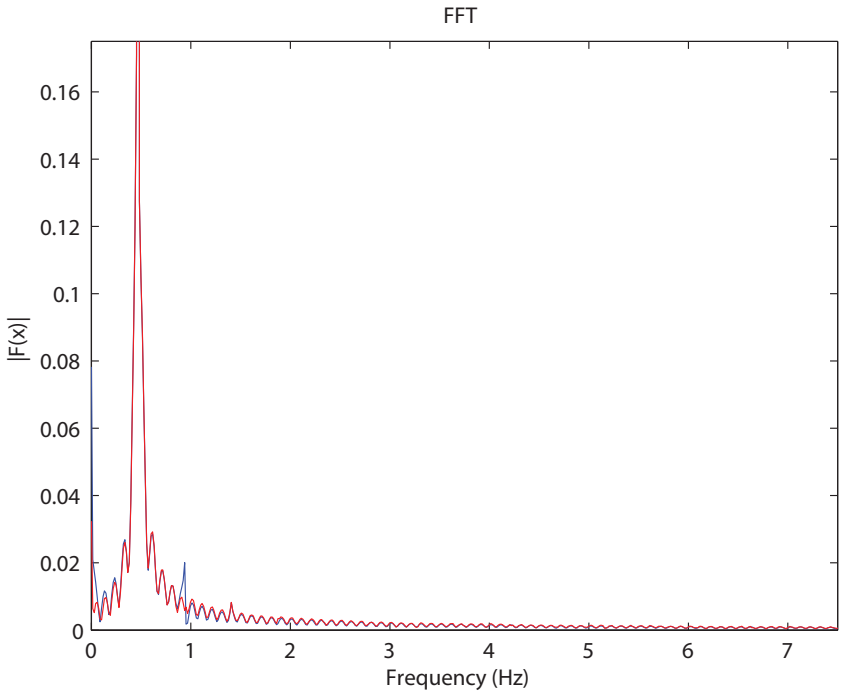

Figure A.8: Fast Fourier transform of both signals. A clearly identifiable peak can be observed around 0.5 Hz, representing the dominant harmonic frequency of the pendulum.

```
--------- Single block of 3498 frames.
   1: Abs = 0.648516 f = 0.46875Hz T = 2.13333 Arg = 1.18135
   2: Abs = 0.653878 f = 0.46875Hz T = 2.13333 Arg = 0.990604
   Phase shift = 0.190746
   Latency = 64.7641ms

--------- Blocks of 256 frames
Frames 0 - 255 :
   1: Abs = 0.993435 f = 0.46875Hz T = 2.13333 Arg = 1.0707
   2: Abs = 0.991735 f = 0.46875Hz T = 2.13333 Arg = 0.879942
   Phase shift = 0.190754
   Latency = 64.7669ms
Frames 256 - 511 :
   1: Abs = 0.913847 f = 0.46875Hz T = 2.13333 Arg = 1.10897
   2: Abs = 0.916834 f = 0.46875Hz T = 2.13333 Arg = 0.918807
   Phase shift = 0.190163
   Latency = 64.5663ms

...

Frames 2816 - 3071 :
   1: Abs = 0.462231 f = 0.46875Hz T = 2.13333 Arg = 1.21722
   2: Abs = 0.4687 f = 0.46875Hz T = 2.13333 Arg = 1.02281
   Phase shift = 0.194412
   Latency = 66.0089ms
Frames 3072 - 3327 :
   1: Abs = 0.435658 f = 0.46875Hz T = 2.13333 Arg = 1.20315
   2: Abs = 0.442459 f = 0.46875Hz T = 2.13333 Arg = 1.00989
   Phase shift = 0.193259
   Latency = 65.6172ms
--- Mean = 65.3352ms Sigma = 0.642597ms 95%-confidence = [64.0757, 66.5947]
```

Figure A.9: Excerpt of the software output for determining latency.

their phase angle.

The test data consists of 3498 captured images at a 60Hz sampling rate. For this data, the FFT peaks were found to be at 0.47Hz in both signals, with a phase angle difference of 0.19 radians. The frequency is calculated under the assumption of a steady 60Hz sampling rate from the cameras. The absolute value of the peaks is not equal to 1.0 (they are actually approx. 0.65) because of the damping factor present in the signal; however, this does not seem to influence the resulting phase angle. From the 0.19 rad difference in phase angle and the 0.47Hz frequency, it can be calculated that the phase shift in seconds equals $0.19/(2\pi * 0.47) = 0.065$, or approximately 65ms of latency. To estimate the amount of (random) error in this measurement, the same algorithm is repeated for blocks of 256 frames each, representing approximately 4 seconds, or two periods of the pendulum. In this short duration the amplitude of the sine wave should stay roughly constant. Next, the mean and the standard deviation of the measured latencies are calculated for all the 256 frame blocks. An excerpt of the software output is given in Figure A.9. Note, the absolute values of the FFT peaks decrease for the 256 frame blocks, indicating damping; however, the phase angle difference remains roughly constant. Future extensions may be the use of a Z-transform to implicitly capture the damping effect, and estimating the FFT peaks by linear interpolation between discrete frequencies. However, these approaches do not appear to be immediately necessary.

Appendix B

Quality Comparison with Level-of-detail Methods

When a 60 Hz frame rate can not be realized, classic VR-architectures often use static level-of-detail (LOD) approaches to reduce the number of polygons rendered by such an amount that new application frames can be generated at 60 Hz. The geometric models are decimated by successively removing all those polygons that are considered to have the least visual significance, until a target number of polygons is reached for which a 60 Hz frame rate can be guaranteed. However, this comes at the cost of reduced image quality. In this appendix, a comparison is made between the image quality of the proposed image-warping architecture and that of a static level-of-detail method.

Two models are used for this comparison: the 10M polygon Thai Statue model from XYZ RGB Inc. and a 17M polygon coral model. The former is a model of a scanned statue with a relatively smooth, low-frequency surface, while the latter is a model of a CT scan of a coral consisting of high-frequency data and many holes in the surface. Both models are good examples of large real-life polygonal datasets. To estimate image quality, the 60 Hz image-sequence output of the PDL image-warping server was recorded for approximately 1600 animation frames. A pre-recorded animation sequence is used where the models rotate about their Y-axis and the camera hovers around the models according to recorded user inputs. This is called the dynamic animation sequence, since the objects as well as the camera move. In addition, similar output is recorded from a stand-alone reference implementation that renders every animation frame as it should appear without error. Finally, the same reference scene is rendered using corresponding level-of-detail (LOD) models that are decimated to various decreasing amounts of polygons. For the PDL server, rendering and recording is performed off-line by running the client and server in a special synchronized mode. In this way, a client frame rate of 6 Hz is simulated in combination with a 60 Hz server rate; i.e., for every application frame, ten warped display frames need to be generated. In all cases a resolution of 1024x768 pixels is used. The rate of object rotation about the Y-axis for the dynamic scenes is 45 degrees/s. The extent of the axis-aligned bounding boxes in rendering units of the coral and the statue object are $(130, 94, 83)$ and $(235, 396, 203)$ units. The translational and angular velocities for the camera are different for each scene. For the coral scenes, the average translational velocity of the camera is 27.2 and 37.2 units/s for the dynamic and static scenes, while the average angular velocities are 43.6 and 49.3 degrees/s, respectively. For the

	Statue		Coral	
Dynamic Scene	**Error**	**Stdev**	**Error**	**Stdev**
LOD 2	0.03	0.01	10.01	3.84
LOD 4	0.33	0.16	15.85	4.55
LOD 8	1.45	0.54	21.88	4.92
LOD 16	3.58	1.09	27.76	4.93
LOD 32	6.51	1.73	32.61	4.94
Warping / Optic flow	0.72	0.61	2.10	1.31

Table B.1: Overview of the average errors and standard deviations for various level-of-detail (LOD) methods. The PDL architecture was used to perform image warping using an optic-flow-based client-side camera-placement strategy.

statue scenes, the average dynamic and static animations' translational velocities are 48.7 and 66.2 units/s, with average angular velocities of 34.2 and 49.2 degrees/s. A sample frame of this animation for the coral model is shown in Figure B.3.

For the level-of-detail models, the numbers of polygons for each of the two models have been reduced by fractions 2, 4, 8, 16 and 32 of the total amount of polygons. For example, LOD-4 of the coral model consists of 4.25M polygons, while the same level for the statue model consists of 2.5M polygons. Free, out-of-the-box software for mesh decimation that could efficiently handle datasets of 10M polygons or more could not be found; most of the tools found simply hanged or quickly ran out of memory. Therefore, the used models were split in separate chunks of 6M polygons each. LODs were generated for each chunk and then the individual meshes were recombined into a single whole. The MeshLab software (version 1.1.1) was used to achieve this [CCR08]. Mesh decimation was performed using the quadric edge collapse algorithm with a default quality threshold of 0.3 to reduce the mesh to the various target numbers of polygons.

Figure B.1 gives an overview of the errors for the dynamic statue and coral scenes for each animation frame. Average errors and their standard deviation are summarized in Table B.1. The image-warping frames with the highest error for both the coral and statue scenes are shown in Figures B.4 and B.5, respectively. Looking at Figure B.1, it is obvious that the image-warping quality for the coral scene is superior to that of the LOD approach for all levels. For the statue scene, the average warping quality lies somewhere between LOD-4 and LOD-8. There are two reasons for these large differences. First, the original statue model appears to be severely over-sampled, since reducing the amount of polygons by half from 10M to 5M (LOD-2) hardly introduces any error at all. This is an effect of the inherent smoothness of the statue model's surface. The second reason is that the coral model consists of very high-frequency data and almost twice the number of polygons. Since LOD-2 already introduces a significant error for the coral, this model does not share the smooth, over-sampled nature of the statue and is less-suited to LOD methods. Another observation that can be made from Table B.1 is that the standard deviation of the error for the statue scene is generally lower in the LOD case. The standard deviation gives a reasonable estimate of the amount of occlusion errors that appear and disappear from frame to frame. For warping the error is lowest when a new frame is received from the client, and usually highest when extrapolation is at its maximum just before the receipt of a new frame. This fluctuation in error leads to higher standard deviations.

In Figure B.2 both the frame rates of a stereoscopic reference implementation and the

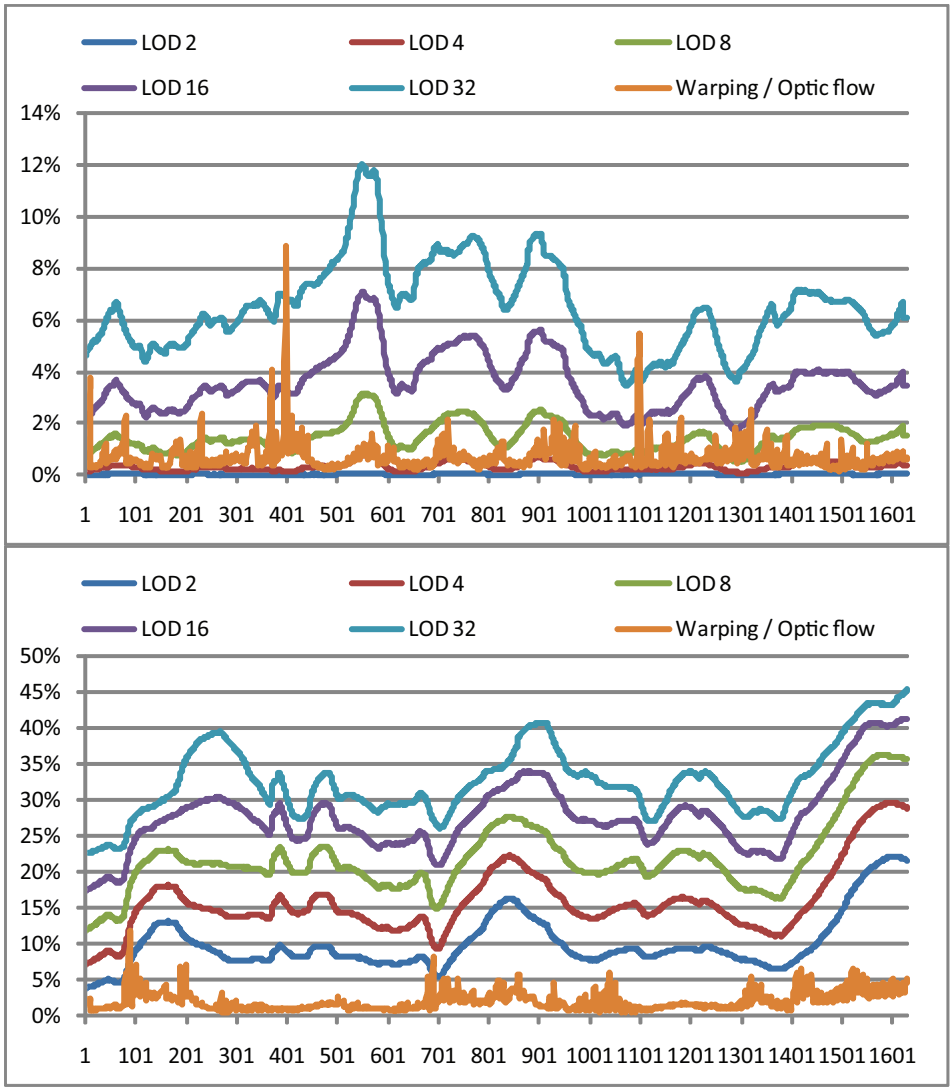

Figure B.1: Percentages of error pixels for each frame of a dynamic animation sequence. The top plot shows the errors for the 10M polygon statue scene; the bottom plot, the 17M polygon coral scene. The camera-placement strategy used for image warping is the optic-flow-based approach. This strategy occasionally results in a poor camera setup, causing the error spikes near frame 400 and 1100.

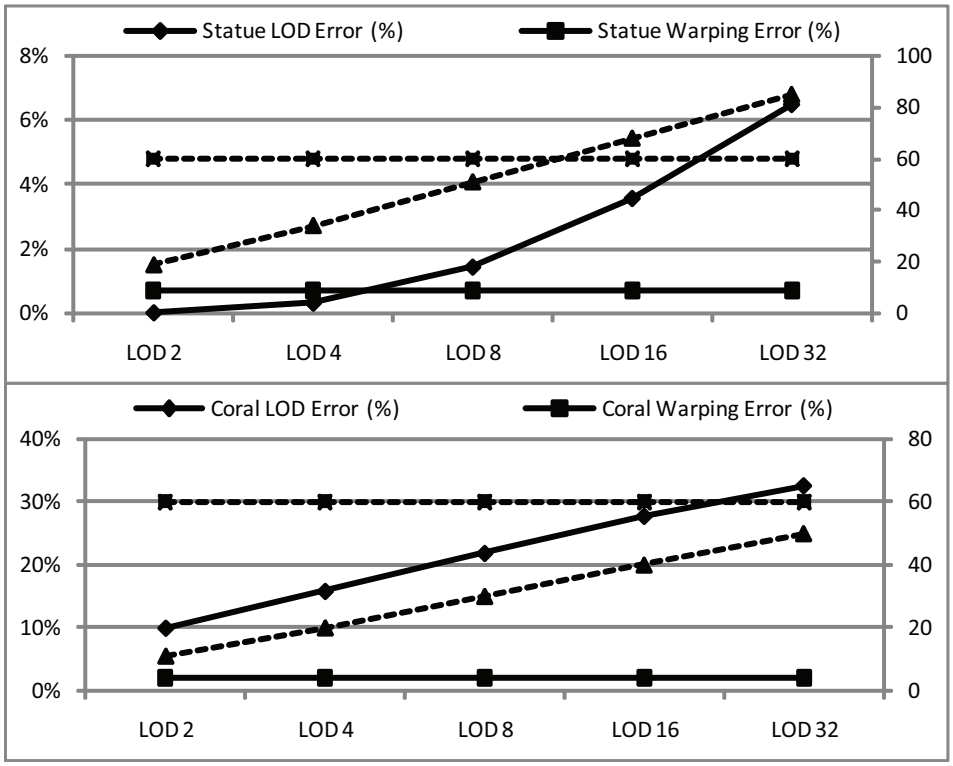

Figure B.2: Average errors and frame rates for level-of-detail methods and image warping, set out against the various detail levels. The left vertical axis indicates an average error percentage, while the right axis is used to indicate frames-per-second. The points of interest are where the error and FPS lines cross for image warping and the LOD approach. This shows that when both approaches run at 60 Hz, image-warping results in the best image quality.

average errors for warping and the various levels-of-detail are given. For image warping, the number of polygons on the client-side has no effect on either the frame-rate or the quality and are, therefore, plotted as straight lines. The points of interest in this figure are the crossing points between the quality and frame rate series. For the statue scene the crossing point for frame rate lies somewhere half-way between LOD-8 and LOD-16. This means that to achieve 60 Hz frame rates with LOD methods, the amount of polygons needs to be reduced by roughly a factor of ten, resulting in 1M polygons. However, the crossing point for the amount of error lies to the left of the FPS crossing, between LOD-4 and LOD-8, where the errors in both methods are considered equal. This means that if both image warping and LOD run at 60 Hz for the statue scene, the image quality for warping is better than that of the LOD approach. For the coral scene the FPS crossing is not contained within the graph, meaning that even at LOD-32 a 60 Hz frame rate is not achieved. There is also no error crossing, which implies that the warping quality is always better in this case.

These results show that the quality of the PDL architecture in combination with image warping is competitive, and in many cases superior, to classic LOD methods for large models; especially when a 60Hz frame rate is required. For LOD methods, the models need to

be decimated to such extent that a large reduction in image-quality occurs for LOD methods. Image warping, on the other hand, can maintain high quality images because the original, high-detail geometry is rendered and subsequently warped to produce 60Hz display updates. One further advantage of image warping is that no extensive pre-processing is required on the polygonal datasets. In contrast, generating the decimated datasets for the various levels-of-detail requires a considerable amount of pre-processing time and resources. Under certain circumstances, for example when large, static datasets are regularly updated with newer versions, pre-computing static LODs may be infeasible.

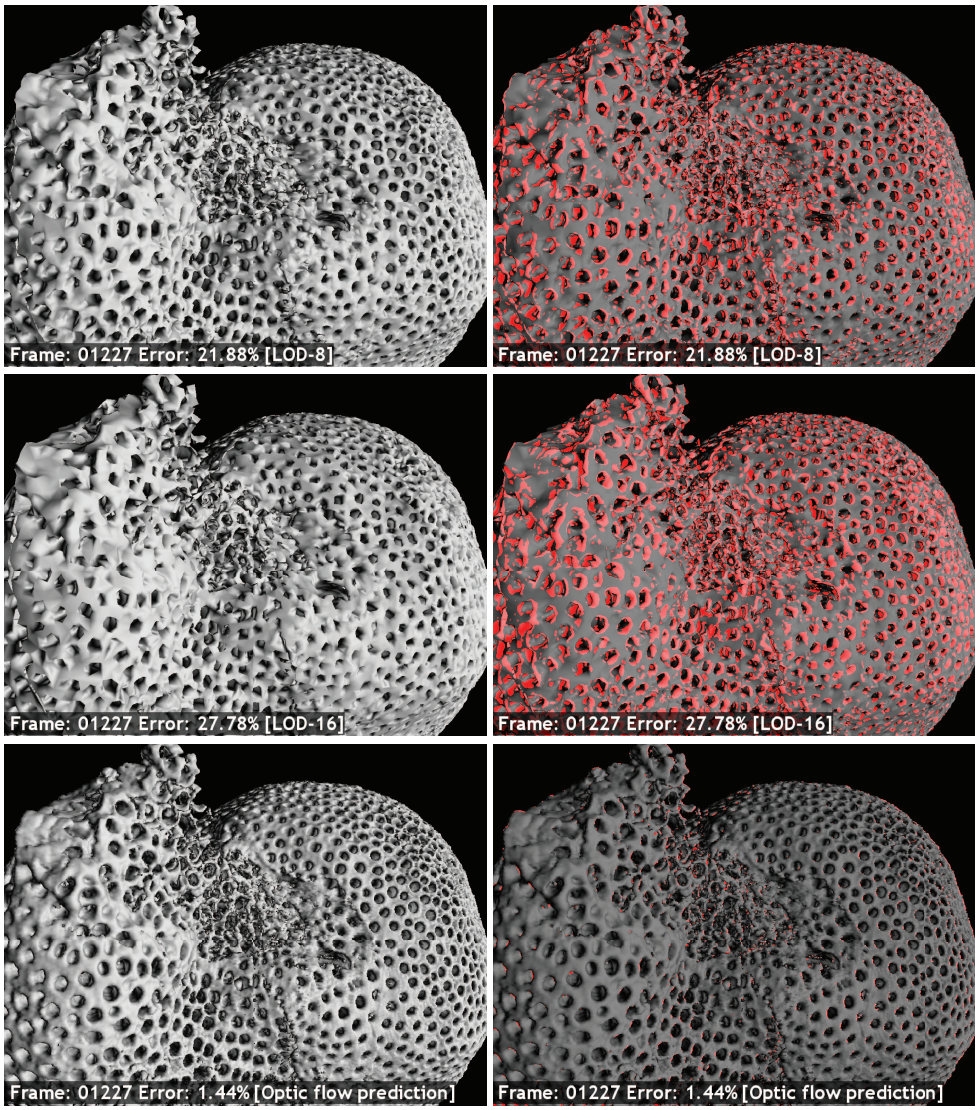

Figure B.3: Comparison between level-of-detail approaches and image warping. The top-most image in the left column shows the 17M polygon coral model where the number of polygons is reduced by a factor of 8, while for the center image the polygon reduction factor is 16. The bottom image shows the same frame produced by image warping. The right column gives a visual overview of the error locations. It can be seen that image warping results in much higher detail at the expense of a different kind of errors in the form of holes. Note, the LOD-16 model is still not sufficiently decimated to allow for 60 Hz rendering.

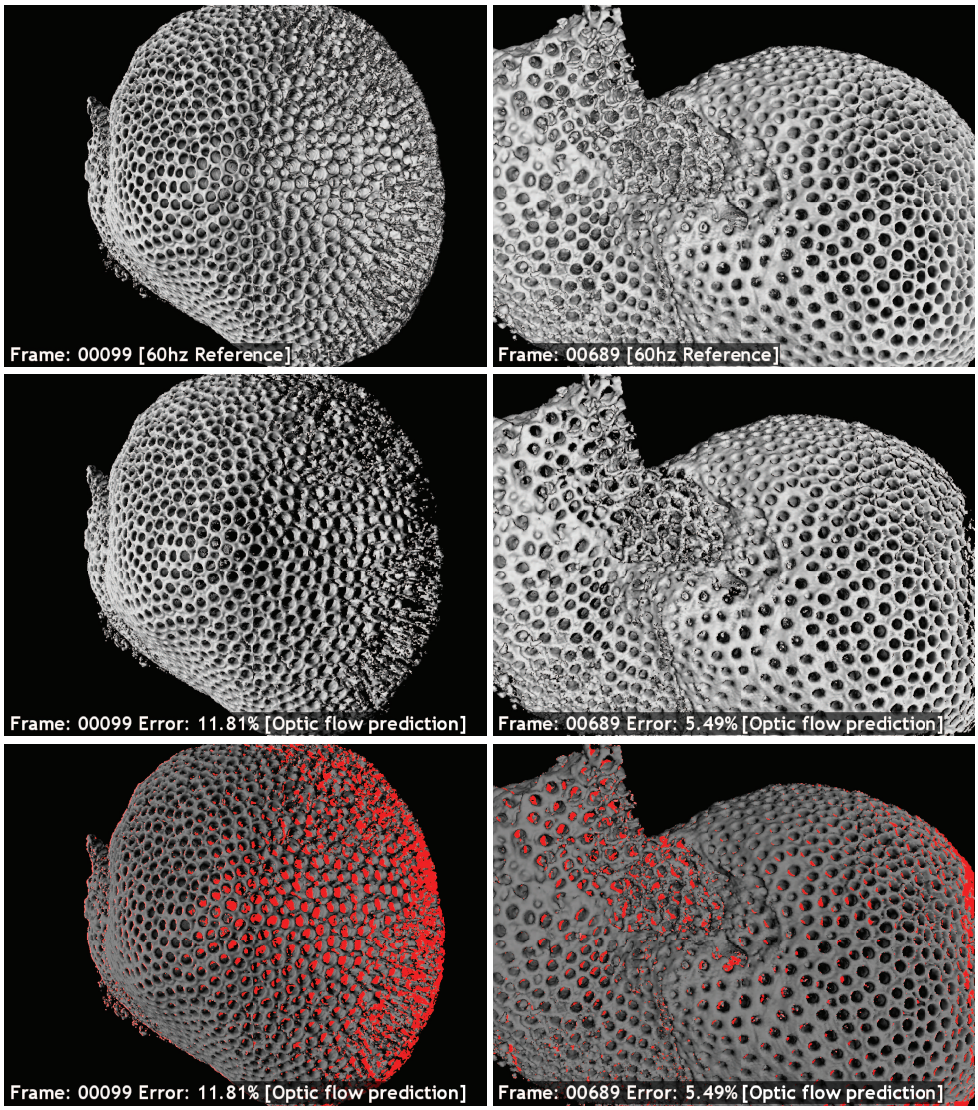

Figure B.4: From top to bottom are shown a reference image of how a rendered scene should look without error, the same scene as rendered by the image-warping process, and the errors made depicted by red pixels. The left column shows the worst frame, with the largest error, for the dynamic coral scene animation sequence. The right column shows another frame with a high amount of error. These large amounts of errors are almost always due to poor camera placements for the two warped views. Note, these frames show a particularly high error because of erroneously predicted camera placements. In practice, the amount of error is much lower for the vast majority of frames (see Figure B.1).

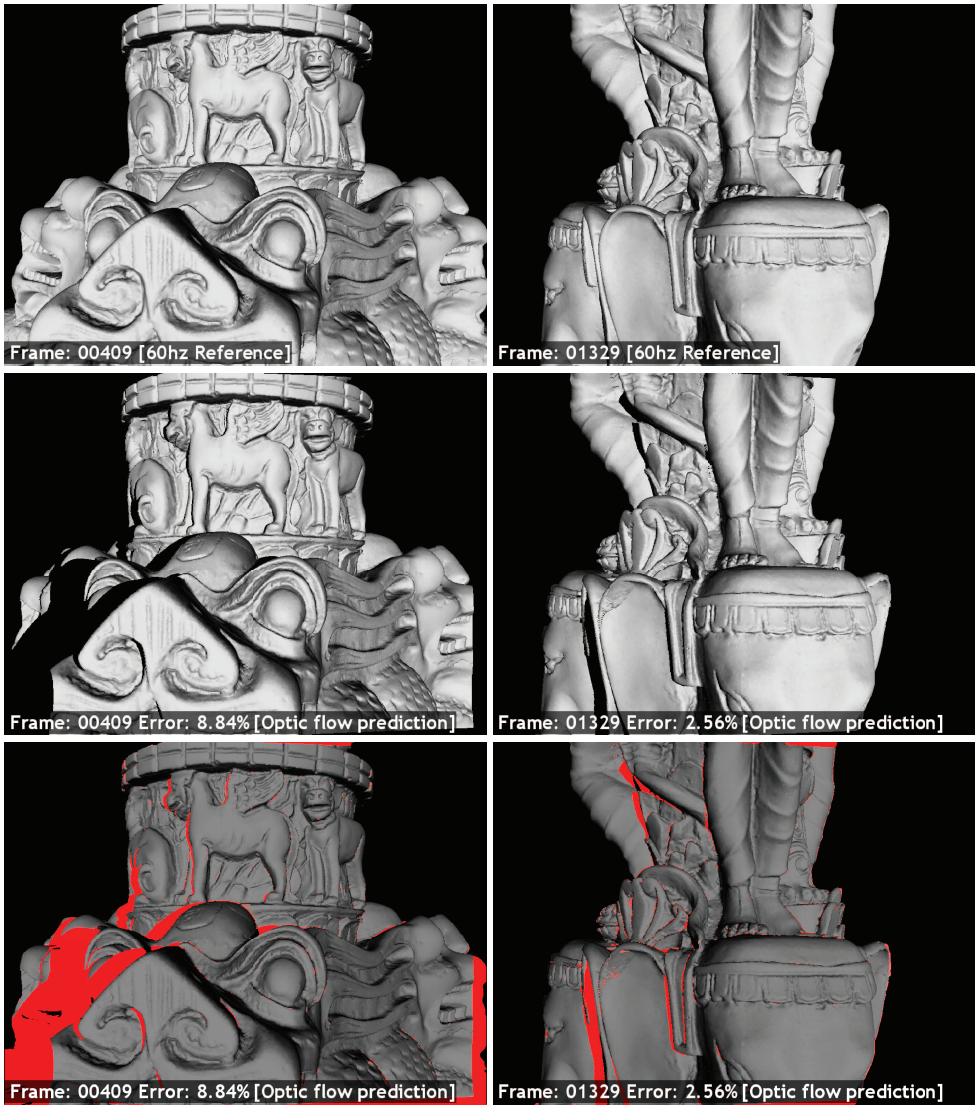

Figure B.5: Images similar to the ones shown in Figure B.4 for the dynamic statue scene. As before, from top to bottom are shown the reference and image-warped frame and the depiction of the error. The left column shows the worst error frame, which could be considered an outlier, while the right column shows a typical frame with high error that occurs occasionally. Again, the error is much lower for the majority of frames.

Appendix C

Projection Invariant Optical Tracking: GraphTracker

An optical tracking method is presented that is based on the projective invariant topology of graph structures. The topology of a graph structure does not change under projection; in this way, the point correspondence problem is solved by a subgraph matching algorithm between the detected 2D image graph and the model graph. The goal is to construct an optical tracker that is less sensitive to occlusion issues. A sample input device is shown in Figure C.1. The method can handle any number of points in many spatial configurations, for example on the surface of a cylinder, sphere or cube.

There are four advantages to this method. First, the correspondence problem is solved entirely in 2D and therefore no stereo correspondence is needed. Consequently, any number of cameras can be used, including a single camera. Second, as opposed to stereo methods, the same model point needs not be detected in two different cameras images; consequently, the method is more robust against occlusion. Third, the subgraph matching algorithm can still detect a match even when parts of the graph are occluded, for example by the users hands. This also provides more robustness against occlusion. Finally, the error made in the pose estimation is significantly reduced as the amount of cameras is increased.

C.1 Implementation

The graph tracking algorithm is based on the detection and matching of graphs to solve the correspondence problem. An overview of the processing pipeline is given in Figure C.2 and C.3. The first step in the pipeline is to perform some basic image processing to acquire a skeleton of the regions in the input image. This skeleton is sufficient to reconstruct the topology of the graph. The clockwise planar ordering of edges is kept track of as well. Next, some graph simplification is performed to eliminate spurious edges, followed by the ordered subgraph isomorphism testing phase to determine correspondence. Image points are vertices of degree three or greater in the detected graph. Once a correspondence has been determined between the image points and the model, a closed form pose estimation algorithm is performed. The pose estimation algorithm first calculates the 3D positions of the image points, followed by an absolute orientation algorithm to fit the point cloud to the model

Figure C.1: A 7x7x7cm cubical input device augmented by a graph pattern of retro-reflective markers. All six faces of the cube are shown. Note that graph edges are allowed to cross over between faces of the cube and do not need to be straight lines.

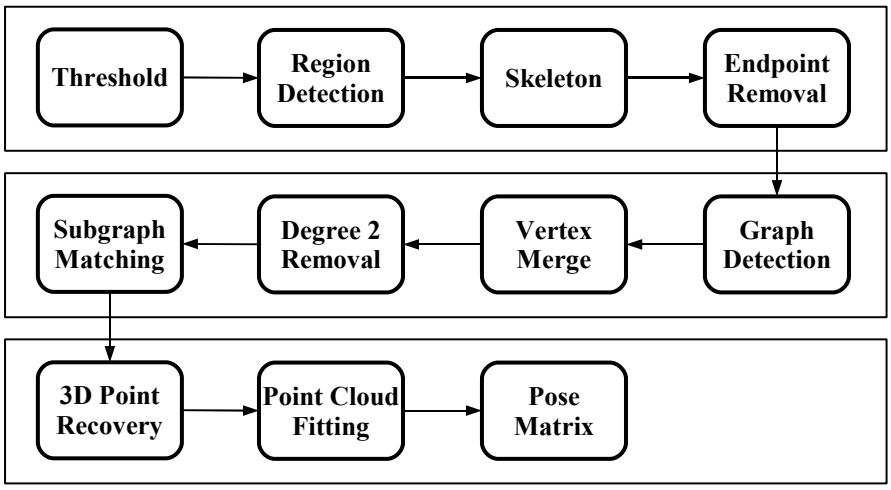

Figure C.2: The sequence of stages in the pipeline to go from a camera image to a device pose.

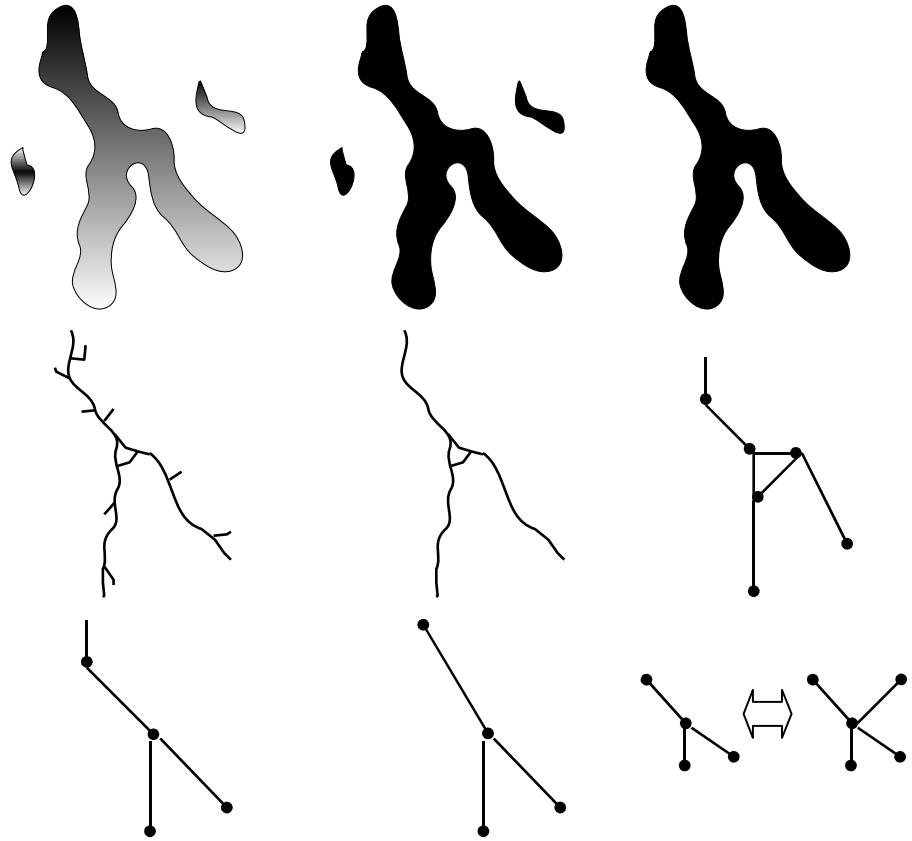

Figure C.3: A schematic visualization of the various stages in converting a camera image to a graph topology that can be matched (also see Figure C.2). From top-left to bottom-right the images show a visualization of the state after: image acquisition, thresholding, region detection, skeletization, end-point removal, graph detection, short edge removal, degree-two removal, and graph matching.

and find a transformation matrix. Finally, an optional iterative solution to the pose estimation problem is proposed. The five major stages are each described in a separate subsection below.

C.1.1 Image processing

Due to the use of retro-reflective markers and infra-red lighting, the preliminary image processing stage is straightforward. It is divided into four stages: thresholding, region detection, skeletization, and end-point removal. All stages use simple algorithms as described by Gonzales and Woods [GW02].

First the input image is processed by an adaptive thresholding algorithm. Next, regions are detected and merged starting at the points found by thresholding. The detected regions are processed by a morphology based skeletization algorithm. The resulting skeleton suffers from small parasitic edges, which are removed in the final phase. The final result is a strictly 4-connected, single pixel width skeleton of the input regions. This is the basic input to the

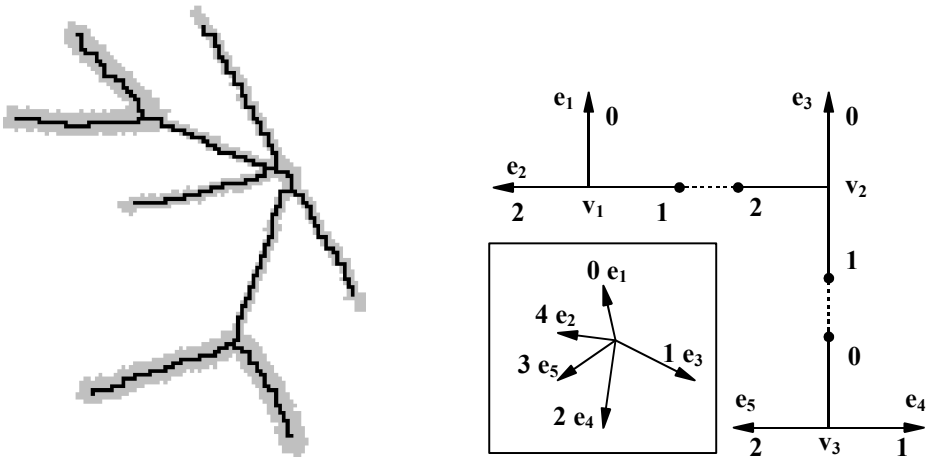

Figure C.4: (left) A detected region in light grey with its skeleton shown in black. The top-right junction of 5 edges is split over 3 vertices. The goal is to combine these vertices into a single vertex while maintaining edge ordering. (right) Merging vertices v_1, v_2, v_3 results in the vertex with incident edge ordering as given in the dashed inset. The numbers indicate the order of each incident edge e_i. The order of merging does not matter, for example merge(merge(v_1, v_2), v_3)) equals merge(merge(v_2, v_3), v_1))

graph topology detector.

C.1.2 Graph detection

The graph detector makes some basic assumptions about the structure of graphs: any pixel that does not have exactly two neighbours in the skeleton is a vertex, and an edge exists between any vertices with a path between them consisting of pixels with exactly two neighbours. The implications are twofold. First, edges do not have to be straight lines; as long as two vertices are connected in some way there is an edge between them. This means the same graph can be on arbitrary convex surface shapes. Secondly, vertices of degree two cannot exist unless there is a self-loop at the vertex; i.e., a degree one vertex with an extra self-loop becomes a degree two vertex.

Next, the algorithm used to detect the graph topology from a given skeleton image is discussed. Let $V = \{v_i\}$ be a set of vertices and $M(x,y) \rightarrow v_i$ a map that maps image coordinates to this set whenever a vertex exists at that coordinate. The vertex set initially contains only the (arbitrary) starting point. As long as the set is not empty, a vertex is taken from it and processed.

Once a starting vertex has been chosen, an arbitrary neighbour of that vertex is chosen and a recursive walk to adjacent pixels is performed. Only the three neighbours of a pixel that are different from the neighbour this pixel was reached from are considered. First, the map M is consulted to see if this pixel is an existing vertex, and if so, an edge is inserted. Also, the pixel is set to zero in the image to indicate it has been searched. Next, the neighbours of this pixel are examined. Whenever there is exactly one neighbour to a pixel, it is sufficient to simply move to this neighbour and continue the recursion. When there are multiple neighbours, the pixel is added to the vertex set V and an edge is inserted. When there are zero neighbours, an

edge is inserted, but the vertex is not added to the vertex set. After inserting an edge, a new starting vertex is chosen from the set V and the procedure is repeated. The process terminates once the set V is exhausted, at which time the entire topology has been constructed.

For reasons explained in Section C.1.3, an ordering is imposed on the incident edges of a vertex. The pixel direction (N,E,S,W) that a vertex is left from is noted, as well as the direction a vertex is reached from in the recursive walk over pixels (also see Figure C.4). Every edge now has an ordering attribute on both ends. The starting points of edges are used for this ordering as the end points might affect the ordering when occluded. Note, this ordering of incident edges is projection invariant up to cyclic permutations.

Even though small edges were removed from the skeleton in the image processing phase, it is still possible for the detected graph to contain very short parasitic edges. These edges have a harmful effect on the matching algorithm, and thus they are removed in a subsequent step by merging their end-points. However, care must be taken not to affect the ordering of incident edges by merging two connected vertices (see Figure C.4).

Finally, all vertices of degree two, except those containing self-loops, are removed in a similar fashion as short edges. These vertices can occur due to the chosen starting point, but cannot exist in theory.

C.1.3 Graph matching

After detecting one or more graphs in the image, the detected graphs are matched as subgraphs in the model graph to solve the point correspondence problem. An error-tolerant subgraph matching algorithm is used to achieve this. A subgraph isomorphism is a mapping of the vertices from one graph to the other that maintains the structure of the graph. All subgraph isomorphisms must be detected to verify a unique match. The problem of finding all subgraph isomorphisms is a notoriously complex one. The decision problem is known to be NPC, and finding all possible subgraphs cannot be done sub-exponentially [Epp99]. A slightly modified version of the VF algorithm by Cordella et al. [CFSV96][CFSV04] is used to perform this matching in worst-case exponential time. However, test cases show that in practice the algorithm is fast enough to be used in real time.

The matching has been simplified further with the following extensions. First, as edges can be occluded, vertices of degree one do not provide a reliable position. Therefore, the matcher ignores all vertices of degree one while matching. They do, however, add to the degree of their adjacent vertex. Second, to reduce the amount of isomorphisms, an ordering is imposed on the incident edges of a vertex. For a match to be valid, this ordering has to be a cyclic permutation in the model, possibly with gaps for missing edges. In this way, a star graph with one center point and five edges only has five automorphisms, as opposed to $5! = 120$.

Whenever more than one subgraph isomorphism is detected, a scan is performed to detect fixed points that have the same mapping in all the isomorphic mappings. In this way, some points can be uniquely identified even when multiple isomorphic mappings exist (see Figure C.7). All fixed points, which have degree greater than one, and their uniquely corresponding model points are provided as input to the pose reconstruction algorithm.

C.1.4 Closed-form pose reconstruction

Once the correspondence between 2D image points and 3D model points is known, the 6-DOF device pose can be reconstructed. The followed approach is closely related to the method suggested by Quan [QL99]. A system of polynomial equations is solved by partial

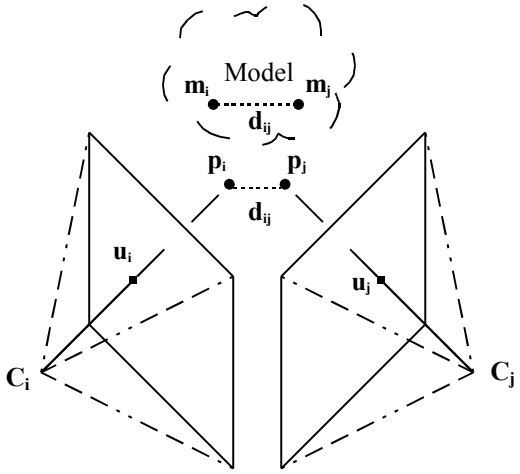

Figure C.5: Schematic view of the perspective n-point problem with two cameras. The goal is to reconstruct the p_i given the camera positions C_i, image points u_i and corresponding model points m_i

algebraic elimination and singular-value decomposition, followed by Horn's [Hor87] absolute orientation determination algorithm using quaternions. Quan's method does not directly support multiple cameras, but the extension is straightforward.

Given camera positions C_i, 3D image points u_i on the focal plane, corresponding 3D model points m_i, and the camera calibration matrices, the task is to calculate the 3D positions p_i and the transformation matrix M that maps p_i to m_i (see Figure C.5). Since all p_i reside in a different frame as the m_i, only inter-point relations in the same frame can be used. Each point p_i lies on the 3D line through its corresponding image point u_i and the camera location C_i. For each pair of such lines, the line equation can be written in parametric form and solved for the parameters (t_i, t_j), where the distance between those points equals $d_{ij} = ||m_i - m_j||^2$:

$$||(C_i + t_i(u_i - C_i)) - (C_j + t_j(u_j - C_j))||^2 = d_{ij} \tag{C.1}$$

Simplifying this equation and setting $D_i = u_i - C_i$ and $C_{ij} = C_i - C_j$ results in a polynomial in two unknowns, (t_i, t_j), with the following coefficient matrix:

$$\begin{pmatrix} -d_{ij} + C_{ij} \cdot C_{ij} & -2(D_j \cdot C_{ij}) & D_j \cdot D_j \\ 2(D_i \cdot C_{ij}) & -2(D_i \cdot D_j) & 0 \\ D_i \cdot D_i & 0 & 0 \end{pmatrix} \tag{C.2}$$

Every pair of points defines an equation of this form, but solving such a system of non-linear equations algebraically proves extremely difficult. A direct solution can be obtained using Gröbner bases; however, the number of terms in this solution is extremely large and it is sensitive to numerical errors.

The coefficient matrix can be simplified by normalizing the direction vectors of the parametric lines ($D_i \cdot D_i = D_j \cdot D_j = 1$) and translating all cameras to the same point at the origin ($C_{ij} = \bar{0}$). Once these simplifications have been made, the polynomial system reduces to the same form as defined by Quan [QL99]:

$$P_{ij} = t_i^2 + t_j^2 - 2(D_i \cdot D_j) - d_{ij} = 0 \tag{C.3}$$

Because of the simplifying camera translations, the new distance calculation becomes:

$$d_{ij} = ||(m_i - C_i) - (m_j - C_j)||^2 = ||m_i - m_j - C_{ij}||^2 \tag{C.4}$$

Given N input points, there are $\binom{N}{2}$ constraining equations. Using three equations P_{ij}, P_{ik}, P_{jk} in t_i, t_j, t_k a 4th-degree polynomial can be constructed in t_i^2 by variable elimination. If the first parameter is fixed to t_i, then $\binom{N-1}{2}$ of such polynomials in t_i^2 can be constructed. For $N > 4$ this system is an over defined linear system in $(1, t_i^2, (t_i^2)^2, (t_i^2)^3, (t_i^2)^4)$, which can be solved in a least-squares fashion by using a singular value decomposition on a $\binom{N-1}{2} \times 5$ matrix. In the case of $N = 4$ a slight modification has to be made, as the linear system is under determined in this case, but a unique solution can still be found (see [QL99] for details). This means that a minimum of four points must be detected over all the cameras to reconstruct the 3D point cloud.

Note that after calculating t_i, this value can not simply be substituted in the original polynomial equations. As the obtained solution is a least squares estimate, there might not exist a solution to the polynomial equation using this variable. Recall that the polynomial equation represents a constraint on the distance between two lines. By fixing a point on one of these lines, there is no guarantee that there even exists a point on the other line for which the distance constraint holds. One could minimize the difference in distance; however, to be precise all points need to be taken into account. Therefore, all of the t_i are solved separately using the method described above.

At this point, the least squares perspective n-point problem for multiple cameras has effectively been solved in closed form. The next step is to determine a transformation matrix between the determined 3D point cloud and the 3D model. To accomplish this, the closed form absolute orientation method of Horn [Hor87] is used. As this algorithm can be left unmodified, it is not described any further here.

C.1.5 Iterative pose reconstruction

Occasionally the closed form perspective n-point algorithm for multiple cameras fails to find a reliable pose. This could be caused by a number of factors, such as slight errors in the device model description, camera noise, numerical instability and near-critical configurations. In an attempt to partially overcome these problems, an iterative pose estimation algorithm as described by Chang and Chen [CC04] has been applied. The iterative algorithm can be seen as an optional post processing step to enhance the reliability of the solution. The accuracy of the solution is not greatly affected, as the closed form algorithm already provides a least-squares solution.

The iterative algorithm relies on an initial pose M_0, which can be provided either as the solution of the perspective n-point problem or as the device pose estimated in the previous frame. Once an initial pose is given, the 3D model points m_i are transformed back to camera space. Now two possibilities occur for these transformed points p_i^t:

- The point p_i^t is visible in multiple cameras. In this case a desired point location p_i^p can be determined by finding the best intersection of all the camera rays through corresponding 3D camera points u_i on the focal plane.
- The point p_i^t is visible in only one camera. Now, the desired point location p_i^p can be found by a perpendicular projection of the transformed point on the corresponding camera ray.

Once the sets $\{p_i^t\}$ and $\{p_i^p\}$ of transformed and desired point locations are known, a least-squares rigid transformation M_ε between the two sets can be determined using Horn's method [Hor87]. This transform M_ε can be seen as an error measure of the initial pose. The procedure can be repeated by setting the new initial pose to $M_n = M_\varepsilon \cdot M_{n-1}$. The algorithm terminates when M_ε is sufficiently close to identity.

C.1.6 Model estimation

The graph-topology-based tracking method requires a model that describes the graph topology of the markers on the device, along with the 3D locations of the graph vertices in a common frame of reference. Creating such a model by hand is a tedious and error prone task, and is only possible for simple shaped objects.

Van Rhijn and Mulder [vRM05] presented a method for automatic model estimation from motion data, using devices equipped with point shaped markers. When a new device is constructed, a user moves it around in front of the cameras, showing all markers. The system incrementally updates a model that describes the 3D markers locations in a common frame of reference. This enables a user to quickly construct new interaction devices and create accurate models. As a result, the flexibility and robustness of the tracking system is greatly increased.

These techniques can be adapted to automatically estimate a model needed for the graph-topology-based optical tracker. Two methods are developed that differ in the amount of information that has to be specified manually. The first method is a fully automatic model estimation method, where no prior information about the interaction device is required. The aim is to obtain a model that describes the graph topology and 3D vertex locations. In the second case, the graph topology has to be pre-specified in an abstract form. The aim is to obtain a model describing the 3D vertex locations, which are needed for pose estimation. This approach greatly simplifies the model estimation procedure and makes it more robust. In the next sections, both methods are described in more detail.

Fully automatic model estimation

When the graph topology is unknown, two cameras are needed to determine the 3D locations of graph vertices. The model estimation procedure is then a straightforward extension of the method presented in [vRM05]. The method proceeds as follows. First, stereo correspondence is used to obtain the 3D locations of the graph vertices. Next, frame-to-frame correspondence is exploited to assign a unique identifier to each vertex during the time it is visible. A graph $G = (V, P, D, E, O)$ is maintained, where

- V is a set of vertices v_i
- $P \subseteq V$ is a set of 3D locations, where p_i assigns a 3D location to vertex v_i in a common frame of reference
- $D \subseteq V \times V$ is a set of distance edges, where d_{ij} represents the average Euclidean distance between vertices v_i and v_j. A distance edge d_{ij} is only present if it remains static during motion.
- $E \subseteq V \times V$ is a set of edges, where an edge e_{ij} is only present if v_i and v_j have a connecting marker path.

- $O \subseteq V \times V$ is an edge ordering, where o_{ij} assigns a planar ordering index to edge e_{ij} with respect to vertex v_i, in clockwise fashion.

The procedure starts by adding all visible vertices to G. The distances d_{ij} are determined, and the camera images are analysed to determine if visible vertex pairs (v_i, v_j) are connected. If so, edges e_{ij} are created. As vertices are moved around, the Euclidean distance between each vertex pair is examined and compared to the distance d_{ij} in G. If the difference in distances exceeds a certain threshold, the distance edge d_{ij} is deleted.

When a vertex appears for which no frame-to-frame correspondence can be determined, the system needs to distinguish between a new vertex appearing and a previously occluded vertex reappearing. This is accomplished by predicting the location of all occluded vertices of the model. The graph G stores a model of all 3D vertex locations in a normalized coordinate system, giving the set P. Vertex locations are averaged over all frames to reduce inaccuracies due to noise and outliers. The locations can be determined by calculating the rigid body transform that maps the identified vertices to the corresponding model vertices in a least-squares manner [Hor87]. This transform is used to predict the locations of occluded vertices. When a vertex is found for which no frame-to-frame correspondence could be established, its location is compared to the predicted occluded vertex locations. If the vertex is classified as new, it is added to the graph, and distance edges d_{ij} are created to connect it to the visible vertices.

Each frame, the camera images are examined to determine if vertices v_i and v_j have a connecting marker path. To handle temporary occlusion of the marker paths between vertices, an edge e_{ij} is only removed from the graph if there is enough evidence to do so. For each edge e_{ij}, an edge ordering o_{ij} is maintained with respect to vertex v_i (see Section C.1.2).

Due to possible inaccuracies in the stereo correspondence method, false 3D vertex locations may appear in the motion data. To reduce the probability that these are included in the model, a pyramid based clustering of the graph G is performed with respect to the distance edges d_{ij}, which finds all rigid subgraphs. Pyramids or 4-cliques are detected in G, and are considered to be connected if they share a triangle. Clusters are defined by the connected components of the pyramid graph. Although connected pyramids do not necessarily form a clique, vertices within a cluster are part of the same rigid structure. The rigid body transform of each cluster is computed using the visible vertices, and used to predict the location of the occluded vertices. Note that this also allows for simultaneous model estimation of multiple objects.

Model estimation with specified graph topology

When the graph topology is given, the model estimation procedure as detailed in the previous section can be greatly simplified. In this case, the only unknown in the model graph G_T is the set of 3D vertex locations P. Since the topology is given, the graph matching techniques described in Section C.1.2 can be applied to find a subgraph in each camera image. Therefore, graph vertices are uniquely identified in 2D. If a vertex is visible in at least two cameras, simple epipolar geometry can be used to obtain its 3D location.

As a consequence, the model estimation procedure is greatly simplified and more robust. Stereo correspondence, which is complicated and can result in invalid 3D vertex locations, is not needed. Since the vertex identification is already known in 2D, frame-to-frame correspondence and occluded vertex location prediction are also not required. The clustering step, which is used to identify false 3D vertex locations and to distinguish between vertices from different objects, can be omitted. The data that is used to update the model graph is reliable,

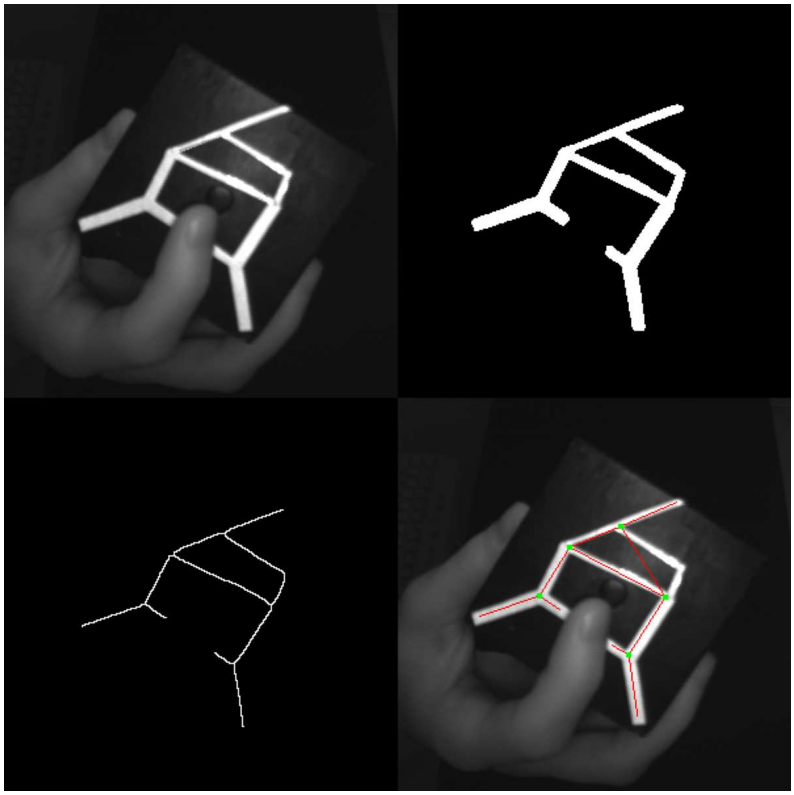

Figure C.6: A number of processing steps performed to match a graph in a camera image. (top left) A raw image as captured by the infra-red cameras. (top right) The resulting image after thresholding and region detection. (bottom left) The result of running a skeletization algorithm. (bottom right) The detected graph after reading and matching the graph from the skeletized image. Five unique points are identified.

and the vertex identification determines which vertices belong to which object. This reduces the model estimation procedure to expressing the identified 3D vertex locations in a common frame of reference throughout the motion sequence.

C.2 Evaluation

The current implementation uses a cubical input device augmented by retro-reflective markers, as shown in Figure C.1. Figure C.6 shows a practical example of the image processing pipeline, which has been shown schematically in Figures C.2 and C.3 earlier. A captured camera image is shown, followed by some steps required to perform graph detection. A number of steps, such as end-point removal and vertex merge, have been omitted.

In the presence of occlusion, several points can still be uniquely identified, allowing the device pose to be reconstructed. A few practical examples of occlusion handling are shown in Figure C.7. Also, using multiple cameras increased the number of detected points as expected.

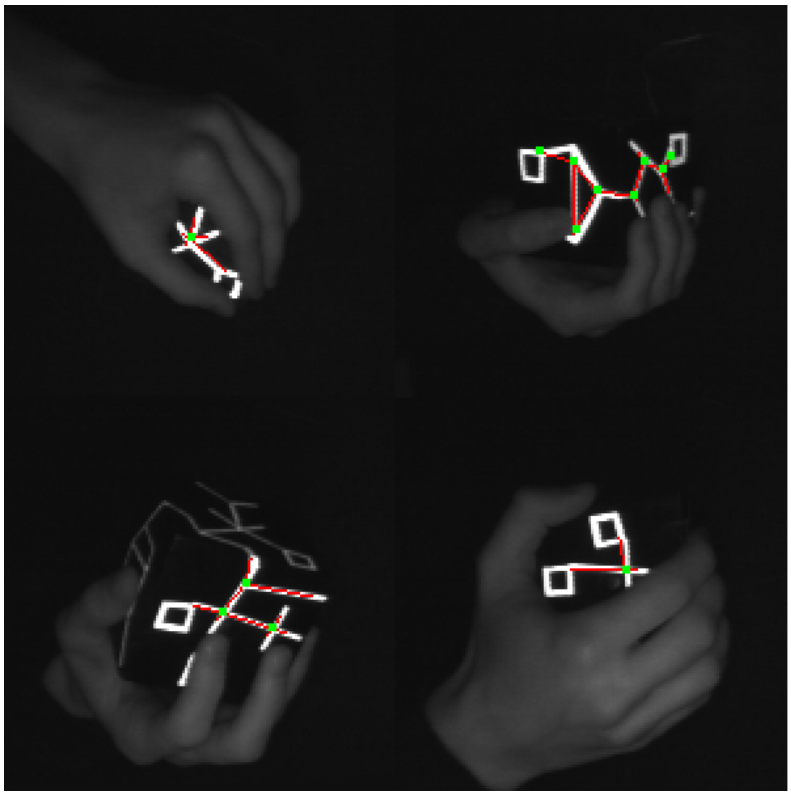

Figure C.7: When parts of the graph are occluded, some fixed points can still be detected. An interesting example is the bottom-right image; the detected subgraph matches the model in two ways. The point connecting the two self-loops can be uniquely identified by noting a fixed point. However, the points representing the loops themselves cannot be uniquely identified as the two points can be interchanged freely.

The implementation uses relatively unoptimized image processing algorithms, and runs entirely on the CPU. Even so, the entire reconstruction, from image processing to pose estimation with multiple cameras, takes between 10 and 20ms per frame. Of this time more than half is spent in the unoptimized image processing phase. These results show that the method is well suited to run in real-time with multiple cameras.

C.2.1 Graph counting

As discussed previously, subgraph matching is used to solve the point correspondence and identification problem. To find a unique match between a smaller subgraph and the model graph, there must not be more than one subgraph in the model with the same topological structure. Also, the (sub)graph itself should not be automorphic, as this prevents the unique identification of vertices. This raises the question of exactly how many topologically different graphs can be constructed, and thus the extensibility of the system.

Some of these graph counting questions are still unanswered in the field of mathematics;

	Number of Vertices									
Degree $\geq$	1	2	3	4	5	6	7	8	9	10
0	1	2	4	11	33	142	822	6966	79853	1140916
2	0	0	1	3	10	50	335	3194	39347	579323
3	0	0	0	1	2	9	46	386	3900	48766
4	0	0	0	0	0	1	1	4	14	69

Table C.1: Number of topologically different planar graphs using a fixed amount of vertices of degree at least n.

therefore, it is hard to give an exact answer to the extensibility question. However, similar, known counting problems can be investigated to give a good indication. One of these is the number of topologically different planar graphs given a fixed number of vertices. The model graph is not necessarily planar, but it will always be locally planar, as it is mapped on a tangible convex input device and edges cannot cross without intersecting. Therefore, the number of planar graphs serves as an indication of the extensibility and is given in Table C.1.

It can be seen that the number of graphs explodes for seven vertices or more. The graph detection algorithm requires the vertices used for pose estimation to be of degree at least three; however, this does not imply that for graph construction all vertices are necessarily of degree three or more. Extra vertices of degree one or more can be used to aid in constructing different topologies. Nonetheless, even for graphs having only vertices of degree three or more, there are many topologically different graphs from 7-8 vertices onward (see Table C.1).

There are two more properties of the algorithm that result in even more graphs to choose from. First, even though vertices with degree zero are not allowed, loops on vertices are allowed. The existence of loops can be seen as somewhat equivalent to removing the constraint that vertices have degree at least one in a counting argument. This means that the number of graphs to choose from is closer to the first row of Table C.1.

Second, by using the imposed planar ordering as described in Section C.1.3, a distinction can be made between graphs that would otherwise be topologically equivalent. The ordering of edges reduces the search space for graph matching by increasing the number of topologically different graphs.

Finally, it should be noted that not all topologically different graphs are equally good choices. The distance between two graphs is often defined as the number of graph edit operations required to transform one graph into the other. To allow for better detection and occlusion handling, it is preferable that graphs exhibit large editing distances to each other. Determining the distance between two graphs is often a complex problem by itself (e.g., [KKV90]). Therefore, counting the number of non-isomorphic graphs with maximum distance is a difficult problem that has not been solved. One useful approach for future work might be to use some graph representation of an error correcting code, as these codes follow strict distance criteria.

C.2.2 Theoretical error analysis

Next, a mathematical model is described to analyse the errors made in pose estimation under the presence of noise, using multiple cameras. Three sources of errors can be distinguished: inaccurate vertex positions in the camera images, errors in the device description, and errors

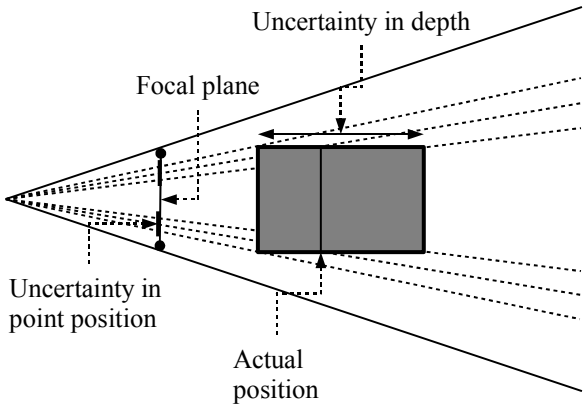

Figure C.8: Expected error perpendicular to the camera plane.

in the camera calibration. Only the first source of errors is considered here, and both the device as well as the camera errors are assumed to be negligible.

As depth information cannot be obtained by using a single projected point, a camera with two detected points is assumed, as shown in Figure C.8. If the error in the projected image points is further assumed to follow a Gaussian distribution with expectation ε_b, the resulting expected errors ε_{XY} in the XY-, and ε_Z in the Z-direction are given by:

$$\varepsilon_{XY} = \frac{d}{f}\varepsilon_b \qquad \varepsilon_Z = \frac{d^2\varepsilon_b}{lf - d\varepsilon_b} \tag{C.5}$$

where d is the distance from the points to the camera's center of projection, f the focal distance, and l the distance between the points. Note that ε_Z is larger than ε_{XY}, and thus the largest error is made in estimating depth information. Extending equation C.5 to N points results in

$$\varepsilon_{XY}^N = \frac{1}{\sqrt{N}}\varepsilon_{XY} \qquad \varepsilon_Z^N = \frac{1}{\sqrt{N}}\varepsilon_Z \tag{C.6}$$

Equation C.6 shows that the error decreases in all directions when more points are visible to the camera. These equations can be extended to the case of multiple cameras. To do this, the expectations have to be transformed to a common reference frame. Writing N_c for the number of cameras, and ε_i for the transformed expectation of errors in the i-th camera gives

$$\varepsilon = \frac{1}{N_c}\sqrt{\sum_i \varepsilon_i} \tag{C.7}$$

for each of the three directions with respect to the reference frame. Equation C.7 shows that the expected error is reduced by increasing the amount of cameras. The initial claim that the usage of multiple cameras reduces the amount of error in pose estimation is supported by this result.

C.2.3 Tracking accuracy

Determining the absolute accuracy of a tracker over the entire working volume is a tedious and time-consuming process. A grid of sufficient resolution covering the tracking volume

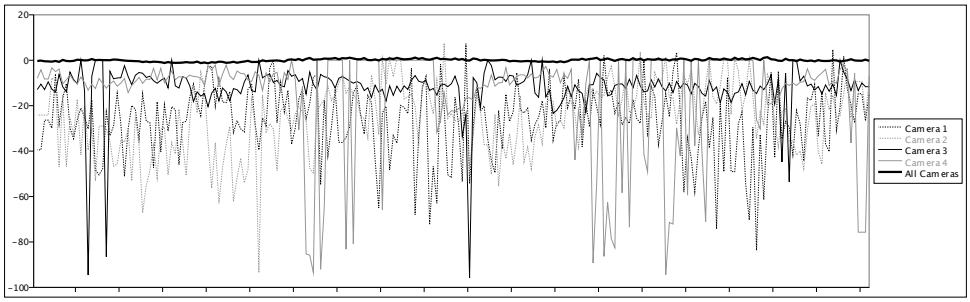

Figure C.9: Positional tracking accuracy with respect to the XZ-plane for single cameras and all four cameras combined. The vertical axis shows the distance to the XZ-plane in millimeters. The horizontal axis represents a sequence of about 200 frames. When a camera could not detect a pose the values are omitted.

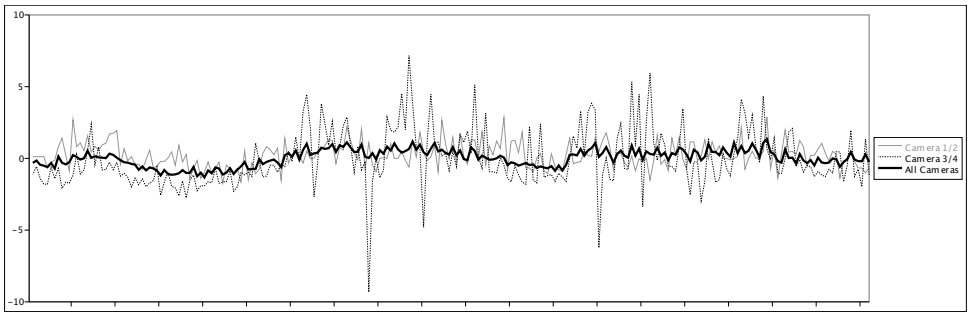

Figure C.10: Positional tracking accuracy with respect to the XZ-plane for two pairs of cameras and all cameras combined.

has to be determined. Next, the device has to be positioned accurately at each grid position, and the resulting tracker measurement has to be determined. A different, simplified approach is used here to measure the accuracy of both the reported position and orientation. Measurements were made in the following ways:

- Position accuracy was measured by sliding the device over a plane of constant height in the tracking volume and logging the positional result. Thus, the measured device positions should all lie on the XZ-plane. The average distance to the XZ-plane is now determined, which gives an indication of the systematic error made. The standard deviation of this set of distances (RMSE) gives a good indication of the random error.
- Orientation accuracy was also measured by sliding the device over a fixed plane; however, this time the angle between a device axis perpendicular to the plane and the plane normal was measured. Thus, ideally the measured angles are equal to zero. Again the average and standard deviation of this set give good indications for the systematic and random error.

Although this procedure does not provide absolute accuracy, it does provide relative accuracy with respect to the plane. The results of the XZ-plane measurements for one, two and four cameras are shown in Figure C.9 and C.10 for position, and Figure C.11 and C.12 for orientation. Table C.2.3 shows summarized results for both measurements.

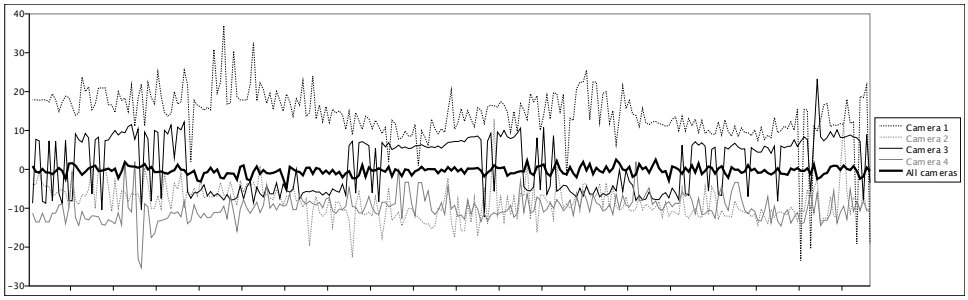

Figure C.11: Angular tracking accuracy with respect to the XZ-plane for single cameras and all four cameras combined. The vertical axis shows the angle with the XZ-plane in degrees. The horizontal axis represents a sequence of frames.

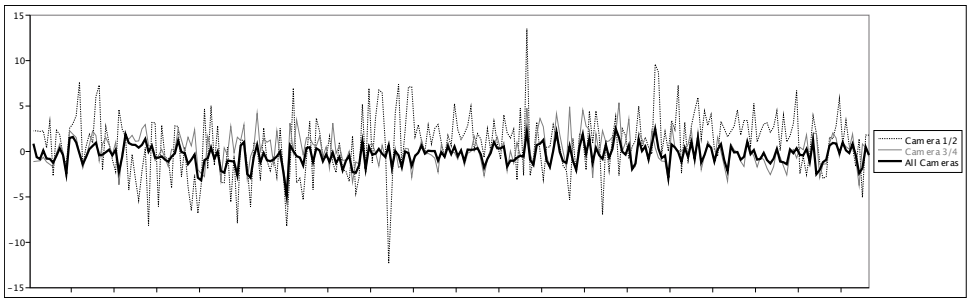

Figure C.12: Angular tracking accuracy with respect to the XZ-plane for two pairs of cameras and all cameras combined.

	Positional Accuracy (in mm)		Angular Accuracy (in degrees)	
Camera	Average	RMSE	Average	RMSE
1	-26.4	15.4	14.3	6.67
2	-25.1	15.3	-8.4	4.36
3	-12.7	10.8	0.99	7.15
4	-20.5	24.3	-9.9	3.27
1/2	0.232	0.90	0.82	3.41
3/4	-0.0754	2.01	0.14	1.69
1/2/3/4	0.0324	0.57	-0.3	1.08

Table C.2: Measurement-to-plane summarized results for 1/2/4 cameras. The average distance to the XZ-plane and the RMSE are given in the first two columns. The average angle with the plane and corresponding RMSE are given in the last two columns.

Camera	1	2	3	4	Stereo	Invariant
Points	4.7	4.9	5.1	4.7	5.7	7.8

Table C.3: Average number of detected unique points for single cameras, stereo and projection-invariant tracking during a random interaction session.

For a different, random interaction session the average number of detected points are listed in Table C.3. The first four columns show the average number of detected points per camera individually. The last two columns show the number of points a stereo correspondence based tracker would detect, as opposed to the tracker based on projection invariants. The projection-invariant tracker clearly detects more points on average.

A number of observations can be made from these results:

- Both the systematic and the random error decrease as the number of cameras increases. A pair of cameras is more accurate than either of the single cameras, and four cameras are yet slightly more accurate than either pair of cameras.
- In the case of cameras 1 and 2, the XZ plane is almost parallel to the camera planes. In this case the error is mostly determined by the depth estimation. Hence, the error is dominated by ε_Z as given in Equation C.5. For cameras 3 and 4 the XZ plane is at a near 45 degree angle, and thus the systematic error is decreased as can be seen in Table C.2.3. However, the random error is increased as the cameras are positioned further away. The combination of all cameras is even more accurate as the device is viewed from more directions now.
- The total number of detected points increases as the number of cameras increases. Hence, a pose can be determined a larger percentage of the time with more cameras. The theoretical accuracy is increased as well, since more points are being used in the calculations. Stereo correspondence can often not be found, while individual cameras do see enough points combined for pose reconstruction.
- Occasionally a single camera reports a better pose than a combination of cameras. This is usually caused by one of the other cameras reporting a very inaccurate pose. Also, camera 3 seemingly reports a very good pose as the average angle is very close to zero. However, the standard deviation is fairly high, indicating an unreliable pose. For accurate, robust tracking the combination of both a low average and standard deviation is important.

Bibliography

[Ahu93] A. J. Ahumada. Computational image quality metrics: A review. *Society for Information Display International Symposium Digest of Technical Papers*, 24:305–308, 1993.

[BES95] P. J. Bex, G. K. Edgar, and A. T. Smith. Multiple images appear when motion energy detection fails. *Journal of Exp. Psychology: Human Perception and Performance*, 21:231–238, 1995.

[BJH+01] Allen Bierbaum, Christopher Just, Patrick Hartling, Kevin Meinert, Albert Baker, and Carolina Cruz-Neira. VR Juggler: A virtual platform for virtual reality application development. In *VR*, pages 89–96, 2001.

[Bro99] F. P. Brooks. What's real about virtual reality? *Computer Graphics and Applications, IEEE*, 19(6):16–27, 1999.

[BW00] W. Bles and A.H. Wertheim. Appropriate use of virtual environments to minimise motion sickness. *RTO MP58*, pages 7.1–7.9, 2000.

[CBPZ04] L. Cheng, A. Bhushan, R. Pajarola, and M. El Zarki. Real-time 3D graphics streaming using MPEG-4. In *Proc. IEEE/ACM Workshop on Broadband Wireless Services and Applications*, 2004.

[CC04] Wen-Yan Chang and Chu-Song Chen. Pose estimation for multiple camera systems. In *ICPR*, pages 262–265, 2004.

[CCR08] Paolo Cignoni, Massimiliano Corsini, and Guido Ranzuglia. Meshlab: an open-source 3D mesh processing system. *ERCIM News*, (73):45–46, April 2008.

[CFSV96] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An efficient algorithm for the inexact matching of ARG graphs using a contextual transformational model. In *ICPR '96: Proceedings of the International Conference on Pattern Recognition, Volume III-Volume 7276*, pages 180–184, 1996.

[CFSV04] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Trans. Pattern Anal. Mach. Intell.*, 26(10):1367–1372, 2004.

[Che00] Xing Chen. Camera placement considering occlusion for robust motion capture. Technical report, CGL Stanford, 2000.

[Cor02] Sarnoff Corporation. Measuring Image Quality: Sarnoff's JNDmetrix Technology. 2002.

[Dal93] Scott Daly. The visible differences predictor: an algorithm for the assessment of image fidelity. In *Digital images and human vision*, pages 179–206. MIT Press, 1993.

[Dor99] K. Dorfmüller. Robust tracking for augmented reality using retroreflective markers. *Computers and Graphics*, 23(6):795–800, 1999.

[EIATEPACET93] Electronic Industries Alliance Tube Electron Panel Advisory Council (EIA-TEPAC). Optical characteristics of cathode-ray tube screens, TEP116-C, 1993.

[Epp99] David Eppstein. Subgraph isomorphism in planar graphs and related problems. *J. Graph Algorithms & Applications*, 3(3):1–27, 1999.

[EYAE99] S. R. Ellis, M. J. Young, B. D. Adelstein, and S. M. Ehrlich. Discrimination of changes in latency during voluntary hand movement of virtual objects. In *Human Factors and Ergonomics Society*, 1999.

[Fia05] Mark Fiala. ARTag, a fiducial marker system using digital techniques. In *CVPR (2)*, pages 590–596, 2005.

[FPS90] J. E. Farrell, M. Pavel, and G. Sperling. The visible persistence of stimuli in stroboscopic motion. *Vision Research*, 30(6):921–936, 1990.

[GW02] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing*. Prentice-Hall, ISBN: 0201180758, second edition, 2002.

[HH02] Eduardo Hidalgo and Roger J. Hubbold. Hybrid geometric - image based rendering. *Comput. Graph. Forum*, 21(3), 2002.

[Hor87] B.K.P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4(4):629–642, 1987.

[HZP06] Thomas Hübner, Yanci Zhang, and Renato Pajarola. Multi-view point splatting. In *Proc. ACM GRAPHITE*, pages 285–294, 2006.

[Kal60] R. E. Kalman. A new approach to linear filtering and prediction problems. *Transactions of the ASME - Journal of Basic Engineering Vol. 82*, pages 35–45, 1960.

[KB99] Hirokazu Kato and Mark Billinghurst. Marker tracking and HMD calibration for a video-based augmented reality conferencing system. In *IWAR '99: Proceedings of the 2nd IEEE and ACM International Workshop on Augmented Reality*, pages 85–94, 1999.

[KFNN03] Stanislav Klimenko, Pavel Frolov, Lialia Nikitina, and Igor Nikitin. Crosstalk reduction in passive stereo-projection systems. *Eurographics*, 2003.

[KKV90] Ewa Kubicka, Grzegorz Kubicki, and Ignatios Vakalis. Using graph distance in object recognition. In *CSC '90: Proceedings of the ACM annual conference on Cooperation*, pages 43–48, 1990.

[KLD00] J. Konrad, B. Lacotte, and E. Dubois. Cancellation of image crosstalk in time-sequential displays of stereoscopic video. In *IEEE Transactions on Image Processing, Vol. 9 No. 5*, pages 897–908, 2000.

[KO02] Ryugo Kijima and Takeo Ojika. Reflex HMD to compensate lag and correction of derivative deformation. In *Proc. IEEE VR*, page 172, 2002.

[LHM00] Chien-Ping Lu, Gregory D. Hager, and Eric Mjolsness. Fast and globally convergent pose estimation from video images. *IEEE Trans. PAMI*, 22(6):610–622, 2000.

[LW94] J. S. Lipscomb and W. L. Wooten. Reducing crosstalk between stereoscopic views. In *Proc. SPIE Vol. 2177*, pages 92–96, 1994.

[Mar99] William R. Mark. *Post-Rendering 3D Image Warping: Visibility, Reconstruction, and Performance for Depth-Image Warping*. PhD thesis, University of North Carolina at Chapel Hill, 1999.

[Mar01] Dave Marsh. Temporal rate conversion. *Microsoft archived paper: http://www.microsoft.com/whdc/archive/TempRate.mspx*, 2001.

[MB95] Leonard McMillan and Gary Bishop. Plenoptic modeling: An image-based rendering system. *Computer Graphics*, 29(Annual Conference Series):39–46, 1995.

[McN00] Ann McNamara. STAR: Visual perception in realistic image synthesis. In *Eurographics 2000*, August 2000.

[Mic99] R. Micheals. A new closed-form approach to the absolute orientation problem. Masters thesis, Lehigh University, 1999.

[Min93] Mark R. Mine. Characterization of end-to-end delays in head-mounted display systems. Technical report, 1993.

[MMB97] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3D warping. In *Symp. Int. 3D Graphics*, pages 7–16, 180, 1997.

[MNLF07] F. Moreno-Noguer, V. Lepetit, and P. Fua. Accurate non-iterative O(n) solution to the PnP problem. In *Proc. of the ICCV*, 2007.

[MRWB03] M. Meehan, S. Razzaque, M. C. Whitton, and F. P. Brooks. Effect of latency on presence in stressful virtual environments. In *IEEE Virtual Reality*, page 141, 2003.

[MvL02] J. D. Mulder and R. van Liere. The Personal Space Station: Bringing interaction within reach. In *VRIC*, pages 73–81, 2002.

[OCMB95] Marc Olano, Jon Cohen, Mark Mine, and Gary Bishop. Combatting rendering latency. In *Proc. ACM SI3D*, pages 19–ff., 1995.

[PLAdON98] Voicu S. Popescu, Anselmo Lastra, Daniel G. Aliaga, and Manuel M. de Oliveira Neto. Efficient warping for architectural walkthroughs using layered depth images. In *IEEE Visualization '98*, pages 211–216, 1998.

[PR94] Ronald Pose and Matthew Regan. Techniques for reducing virtual reality latency with architectural support and consideration on human factors. In *MHVR*, pages 117–129, 1994.

[QL99] Long Quan and Zhongdan Lan. Linear n-point camera pose determination. *IEEE Trans. Pattern Anal. Mach. Intell.*, 21(8):774–780, 1999.

[RP94] Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. In *SIGGRAPH*, pages 155–162, 1994.

[RPF01] M. Ribo, A. Pinz, and A. Fuhrmann. A new optical tracking system for virtual and augmented reality applications. In *Proceedings of the IEEE Instrumentation and Measurement Technical Conference*, pages 1932–1936, 2001.

[SBM04] J. Stewart, E. P. Bennett, and L. McMillan. Pixelview: a view-independent graphics rendering architecture. In *Proc. ACM HWWS*, pages 75–84, 2004.

[SGHS98] Jonathan W. Shade, Steven J. Gortler, Li-Wei He, and Richard Szeliski. Layered depth images. *Computer Graphics*, 32(Annual Conference Series):231–242, 1998.

[SGLS93] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *Information Systems*, 11(3):287–317, 1993.

[SMI05] P. J. H. Seuntiens, L. M. J. Meesters, and W. A. IJsselsteijn. Perceptual attributes of crosstalk in 3D images. *Displays*, 26(4-5):177–183, 2005.

[SP06] Gerald Schweighofer and Axel Pinz. Robust pose estimation from a planar target. *IEEE Trans. PAMI*, 28(12):2024–2030, 2006.

[Sut70] I. E. Sutherland. Computer displays. *Scientific American*, 222(6):57–81, 1970.

[SvL08] F. A. Smit and R. van Liere. A framework for performance evaluation of model-based optical trackers. In *Proc. EGVE*, pages 33–40, 2008.

[SvL09] F. A. Smit and R. van Liere. A simulator-based approach to evaluating optical trackers. *Elsevier Computers & Graphics*, 33(2):120–129, 2009.

[SvLBF09] F. A. Smit, R. van Liere, S. Beck, and B. Fröhlich. An image warping architecture for VR: Low latency versus image quality. In *Proc. IEEE Virtual Reality (VR)*, pages 27–34, 2009.

[SvLBFss] F. A. Smit, R. van Liere, S. Beck, and B. Fröhlich. A shared-scene-graph image-warping architecture for VR: Low latency versus image quality. *Elsevier Computers & Graphics*, 2009 [in press].

[SvLF07a] F. A. Smit, R. van Liere, and B. Fröhlich. The design and implementation of a VR-architecture for smooth motion. In *Proc. ACM VRST*, pages 153–156, 2007.

[SvLF07b] F. A. Smit, R. van Liere, and B. Fröhlich. Non-uniform crosstalk reduction for dynamic scenes. In *Proc. IEEE Virtual Reality, March 2007*, pages 139–146, 2007.

[SvLF07c] F. A. Smit, R. van Liere, and B. Fröhlich. Three extensions to subtractive crosstalk reduction. In *Proc. EGVE*, pages 85–92, 2007.

[SvLF08] F. A. Smit, R. van Liere, and B. Fröhlich. An image warping VR-architecture: Design, implementation and applications. In *Proc. ACM VRST*, pages 115–122, 2008.

[SvLFss] F. A. Smit, R. van Liere, and B. Fröhlich. A programmable display layer for virtual reality system architectures. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 2009 [in press].

[SvRvL06] F. A. Smit, A. van Rhijn, and R. van Liere. Graphtracker: A topology projection invariant optical tracker. In *Proc. EGVE*, pages 63–70, 2006.

[SvRvL07] F. A. Smit, A. van Rhijn, and R. van Liere. Graphtracker: A topology projection invariant optical tracker. *Elsevier Computers & Graphics*, 31(1):26–38, 2007.

[SWI06] A. State, G. Welch, and A. Ilie. An interactive camera placement and visibility simulator for image-based VR applications. In *Proc. of the SPIE*, volume 6055, pages 640–651, February 2006.

[TK96] Jay Torborg and James T. Kajiya. Talisman: commodity realtime 3D graphics for the PC. In *Proc. ACM SIGGRAPH*, pages 353–363, 1996.

[vLMFdS00] Robert van Liere, Jurriaan Mulder, Jason Frank, and Jacques de Swart. Virtual fekete point configurations: A case study in perturbing complex systems. In *VR '00: Proceedings of the IEEE VR 2000*, page 189, 2000.

[vLvR04] R. van Liere and A. van Rhijn. An experimental comparison of three optical trackers for model based pose determination in virtual reality. In *Proc. of EGVE*, 2004.

[VRC07] VRCO. CAVElib, http://www.vrco.com/. 2007.

[vRM05] A. van Rhijn and J. D. Mulder. Optical tracking and calibration of tangible interaction devices. In *Proceedings of the Immersive Projection Technology and Virtual Environments Workshop*, pages 41–50, 2005.

[WAB93] C. Ware, K. Arthur, and K. S. Booth. Fish tank virtual reality. In *Proc. Human factors in computing systems*, pages 37–42, 1993.

[WKC94] D. S. Wallach, S. Kunapalli, and M. F. Cohen. Accelerated MPEG compression of dynamic polygonal scenes. *Computer Graphics*, 28:193–196, 1994.

[WMG+07] Ingo Wald, William R. Mark, Johannes Günther, Solomon Boulos, Thiago Ize, Warren Hunt, Steven G. Parker, and Peter Shirley. State of the art in ray tracing animated scenes. In *STAR Proc. of Eurographics 2007*, pages 89–116, 2007.

[WS07] Daniel Wagner and Dieter Schmalstieg. ARToolkitPlus for pose tracking on mobile devices. In *Proc. of the CVWW*, February 2007.

[WT02] Andrew J Woods and Stanley S Tan. Characteristic sources of ghosting in time-sequential stereoscopic video displays. In *Proc. SPIE Vol. 4660*, pages 66–77, 2002.

[YS90] Yei-Yu Yeh and Louis D. Silverstein. Limits of fusion and depth judgment in stereoscopic color displays. *Hum. Factors*, 32(1):45–60, 1990.

[ZCW02] H. Zhou, M. Chen, and M. F. Webster. Comparative evaluation of visualization and experimental results using image comparison metrics. In *Proc. of IEEE Visualization*, pages 315–322, 2002.

A Programmable Display-Layer Architecture for Virtual-Reality Applications

Summary

Two important technical objectives of virtual-reality systems are to provide compelling visuals and effective 3D user interaction. In this respect, modern virtual reality system architectures suffer from a number of short-comings. The reduction of end-to-end latency, crosstalk and judder are especially difficult challenges, each of which negatively affects visual quality or user interaction.

In order to provide higher quality visuals, complex scenes consisting of large models are often used. Rendering such a complex scene is a time-consuming process resulting in high end-to-end latency, thereby hampering user interaction. Classic virtual-reality architectures can not adequately address these challenges due to their inherent design principles. In particular, the tight coupling between input devices, the rendering loop and the display system inhibits these systems from addressing all the aforementioned challenges simultaneously.

In this thesis, a virtual-reality architecture design is introduced that is based on the addition of a new logical layer: the Programmable Display Layer (PDL). The governing idea is that an extra layer is inserted between the rendering system and the display. In this way, the display can be updated at a fast rate and in a custom manner independent of the other components in the architecture, including the rendering system. To generate intermediate display updates at a fast rate, the PDL performs per-pixel depth-image warping by utilizing the application data. Image warping is the process of computing a new image by transforming individual depth-pixels from a closely matching previous image to their updated locations. The PDL architecture can be used for a range of algorithms and to solve problems that are not easily solved using classic architectures. In particular, techniques to reduce crosstalk, judder and latency are examined using algorithms implemented on top of the PDL.

Concerning user interaction techniques, several six-degrees-of-freedom input methods exists, of which optical tracking is a popular option. However, optical tracking methods also introduce several constraints that depend on the camera setup, such as line-of-sight requirements, the volume of the interaction space and the achieved tracking accuracy. These constraints generally cause a decline in the effectiveness of user interaction. To investigate the effectiveness of optical tracking methods, an optical tracker simulation framework has been developed, including a novel optical tracker to test this framework. In this way, different optical tracking algorithms can be simulated and quantitatively evaluated under a wide range of conditions.

A common approach in virtual reality is to implement an algorithm and then to evaluate the efficacy of that algorithm by either subjective, qualitative metrics or quantitative user experiments, after which an updated version of the algorithm may be implemented and the cycle repeated. A different approach is followed here. Throughout this thesis, an attempt is made to automatically detect and quantify errors using completely objective and automated quantitative methods and to subsequently attempt to resolve these errors dynamically.

A Programmable Display-Layer Architecture for Virtual-Reality Applications

Samenvatting

Twee belangrijke technische doelstellingen van virtual-reality systemen zijn het realiseren van overtuigende beelden en effectieve 3D gebruikersinteractie. In dit opzicht vertonen moderne virtual-reality systeemarchitecturen nog een aantal gebreken. Vooral het reduceren van crosstalk, judder en latentie zijn moeilijke problemen, elk waarvan zowel de visuele kwaliteit als de effectiviteit van de gebruikersinteractie negatief beïnvloedt.

Om de visuele kwaliteit van de beelden te verhogen worden vaak complexe scenes bestaande uit grote modellen gebruikt. Het weergeven van van zo'n complexe scene is een tijdrovend proces dat resulteert in hoge latentie en daardoor effectieve gebruikersinteractie belemmert. Klassieke virtual-reality systeemarchitecturen kunnen deze uitdagingen niet voldoende adresseren vanwege hun inherente ontwerpprincipes. De rigide koppeling tussen inputmechanismen, het rendersysteem en het beeldscherm verhindert het gelijktijdig oplossen van al de voorgenoemde problemen in het bijzonder.

In dit proefschrift wordt een ontwerp voor een virtual-reality systeemarchitectuur geintroduceerd dat is gebasseerd op de toevoeging van een nieuwe logische laag: de *Programmable Display Layer (PDL)*. Het kernidee bestaat uit de toevoeging van een extra laag tussen het rendersysteem en het beeldscherm. Op deze wijze kunnen er snel nieuwe beelden vertoond worden op het beeldscherm onafhankelijk van de andere componenten in de systeemarchitectuur, inclusief het rendersysteem. Om op deze manier snel nieuwe beelden te kunnen genereren maakt de PDL architectuur gebruik van *image warping* voor individuele pixels door gebruik te maken van extra applicatie data. Image warping is een proces dat individuele pixels van een eerder gegenereerd beeld omzet naar hun bijgewerkte locatie in een nieuw beeld. De PDL architectuur kan voor meerdere algoritmen gebruikt worden om zo een aantal problemen op te lossen die niet gemakkelijk oplosbaar zijn voor klassieke architecturen. In het bijzonder worden er algoritmen onderzocht en geimplementeerd op de PDL architectuur om judder, crosstalk en latentie te verminderen.

Wat betreft technieken voor gebruikersinteractie bestaan er verscheidene inputmethodes met zes vrijheidsgraden, waarvan optische input een populaire optie is. Nochtans introduceren de optische inputmethodes ook verscheidene beperkingen afhangende van de gebruikte camera-opstelling, zoals zichtbaarheids vereisten, het volume van de interactieruimte en de bereikte nauwkeurigheid. Deze beperkingen veroorzaken over het algemeen een daling in de doeltreffendheid van gebruikersinteractie. Om de doeltreffendheid van optische inputmethodes te onderzoeken is er een simulatieraamwerk ontwikkeld om deze methodes te evalueren, inclusief een nieuwe optische inputmethode om dit raamwerk te testen. Op deze wijze kunnen verschillende optische inputmethodes gesimuleerd en kwantitatief geëvalueerd worden onder een breed scala van voorwaarden.

Een veelvoorkomende aanpak in virtual-reality is het implementeren van een algoritme en vervolgens het bepalen van de effectiviteit van dit algoritme door middel van subjectieve, kwalitatieve metrieken of kwantitatieve gebruikersexperimenten, waarna een bijgewerkte versie van het algoritme wordt geimplementeerd en de cyclus herhaald wordt. In dit proefschrift wordt een andere aanpak gevolgd. Er wordt steeds een poging gedaan om automatisch fouten

te ontdekken en te kwantificeren door gebruik te maken van volledig objectieve en geautomatiseerde kwantitatieve methoden en om deze fouten vervolgens dynamisch op te lossen.

Curriculum Vitae

Ferdi Smit was born on the 26th of April, 1980 in Amsterdam, The Netherlands. He received his high school Atheneum diploma in 1998 from the Casimir Lyceum, Amstelveen, after which he began a Master's degree track in mathematics as well as computer science at the Vrije Universiteit, Amsterdam. One year later, in 1999, he received a propedeuse degree in mathematics and another in computer science. For this he was awarded with a Civi Propedeuse Award in the field of computer science. During his university studies, he worked part-time at the Nederlandse Omroep Stichting (NOS) as a web developer for two years. In May 2005, he received his M.Sc. degree in computer science from the Vrije Universiteit. His Master's thesis research was concerned with the rendering of large terrain data on a parallel cluster and scalable tiled display, and it was supervised by prof.dr.ir. Henri E. Bal and dr. Tom van der Schaaf. Shortly thereafter, he joined the Visualization and 3D Interfaces theme (INS-3) at the Centrum Wiskunde & Informatica (CWI), Amsterdam, in pursuit of a Ph.D. degree in computer science. There he carried out the research described in this thesis under the supervision of prof.dr.ir. Robert van Liere and as a member of the Quantitative Spatial Interaction Design (QUASID) group.