

StubeRenA  
Studierstube Render Array  
-  
A Seamless Tiled Display

Gottfried Eibner

June 2003



# Contents

<b>I</b>	<b>7</b>
<b>1 Introduction</b>	<b>9</b>
1.1 Motivation . . . . .	9
1.2 Overview . . . . .	11
<b>2 Commodity hardware today</b>	<b>13</b>
2.1 Projectors . . . . .	13
2.2 Computer systems . . . . .	15
2.3 Screen material . . . . .	15
2.4 Our choices . . . . .	16
<b>3 Related work</b>	<b>17</b>
3.1 Calibration approaches . . . . .	17
3.1.1 Abutted displays . . . . .	18
3.1.2 Regular overlap displays . . . . .	19
3.1.3 Rough overlap displays . . . . .	20
3.2 Computer systems . . . . .	28
3.2.1 Parallel computer architecture . . . . .	29
3.2.2 Networked computer architecture . . . . .	30
3.3 Network hardware . . . . .	30
3.3.1 Ethernet . . . . .	31
3.3.2 Myrinet . . . . .	32
3.4 Data distribution . . . . .	33
3.4.1 Synchronized execution . . . . .	37
3.4.2 Primitives distribution . . . . .	40
3.4.3 Pixel distribution . . . . .	43
3.5 Display synchronization . . . . .	43
3.5.1 Frame buffer switch synchronization . . . . .	44
3.5.2 Refresh rate synchronization . . . . .	44
3.6 Open Inventor . . . . .	46
3.7 Distributed Open Inventor and Studierstube . . . . .	49

<b>4</b>	<b>Design issues</b>	<b>51</b>
4.1	Calibration . . . . .	51
4.2	Cluster architecture . . . . .	52
4.3	Data distribution . . . . .	53
4.4	Display synchronization . . . . .	53
<b>5</b>	<b>Implementation issues</b>	<b>57</b>
5.1	Calibration . . . . .	57
5.2	Studierstube . . . . .	58
5.2.1	Data distribution . . . . .	58
5.2.2	Display synchronization . . . . .	59
5.2.3	Tiled Display . . . . .	63
5.2.4	Summary . . . . .	64
<b>6</b>	<b>Results</b>	<b>69</b>
6.1	Performance . . . . .	69
6.2	Visual accuracy . . . . .	73
6.3	Scalability . . . . .	75
6.4	Maintenance . . . . .	79
<b>7</b>	<b>Future work</b>	<b>81</b>
7.1	Display and event synchronization . . . . .	81
7.2	Synchronized execution . . . . .	81
7.3	Very large displays . . . . .	82
7.4	Detecting the screen wall . . . . .	82
7.5	Other synchronization models . . . . .	82
<b>8</b>	<b>Acknowledgement</b>	<b>83</b>
<b>II</b>		<b>85</b>
<b>9</b>	<b>User Manual</b>	<b>87</b>
9.1	Introduction . . . . .	87
9.2	Application usage . . . . .	89
9.2.1	Server application . . . . .	89
9.2.2	Camera client application . . . . .	89
9.2.3	Projector client application . . . . .	91
9.2.4	User client application . . . . .	93
9.3	Installation and Compilation . . . . .	95
9.3.1	Before compiling the Overlap package . . . . .	96
9.3.2	Before running any part of the Overlap software . . . . .	96

9.4	Running the applications	96
9.5	Known problems	97
<b>10</b>	<b>Developer Manual</b>	<b>99</b>
10.1	Introduction	99
10.2	Class hierarchy	99
10.3	Overlap server code	100
10.3.1	RenaServer	104
10.3.2	RenaReactor	104
10.3.3	RenaLogin	104
10.3.4	RenaLine	104
10.3.5	RenaObject	105
10.3.6	RenaCamera	105
10.3.7	RenaProjector	105
10.3.8	RenaUser	106
10.3.9	main	106
10.3.10	Summary	106
10.4	Camera client code	106
10.4.1	RenaCameraWindow	106
10.4.2	RenaCameraClient	107
10.4.3	RenaCalibration	107
10.4.4	RenaProjectorData	108
10.4.5	RenaMath	108
10.4.6	main	109
10.5	Projector client code	109
10.5.1	RenaProjectorWindow	109
10.5.2	RenaProjectorClient	109
10.5.3	main	109
10.6	User client code	109
10.6.1	RenaUserClient	110
10.6.2	main	110
10.7	Additional code	110
10.7.1	RenaClient	110
10.7.2	RACOM	111
10.7.3	Defined macros	111
10.8	Remarks for future extensions	112
10.8.1	Extension of Overlap	112
10.8.2	Known bugs	113

## Abstract

High-end rendering systems have provided the computer graphics community over a long period with high resolution graphics at high frame rates, allowing to build up large tiled displays. But with the increasing performance of commodity computer and graphics hardware we are able to build such large displays without the costs of high-end rendering systems. The key concept is to build an inexpensive large tiled display for interactive 3D graphics, composed of multiple overlapping projections, driven by a PC based rendering cluster.

In this thesis we will show how to build up a tiled display from scratch using oblique projectors to display high-resolution scenery. Each projector is driven by a PC. The PCs are connected to form a network that is capable to share and distribute data and scenery content. All projected images together form a large high-resolution tiled display. To run the tiled display and compensate distortion coming from oblique projection, we have implemented a software to calibrate the projectors and find their relative poses. The software package undistorts oblique projectors used for the tiled display and generates blending masks for overlapping projection areas. Both undistortion and blending masks are used to run a seamless tiled display. While undistortion guarantees the continuity of geometry displayed by different projectors, blending masks make overlapping regions appear seamless and equally bright. Without blending masks overlapping regions would appear brighter than the other regions. This would lead to regions of different brightness.

**Keywords:** tiled display, render cluster, synchronization, calibration, distributed graphics, virtual reality.

# Part I





# Chapter 1

## Introduction

*“Tiled displays are an emerging technology for constructing semi-immersive visualization environments capable of presenting high-resolution images from scientific simulation. [...] However, the largest impact may well be in using large-format tiled displays as one of possibly multiple displays in building “information” or “active” spaces that surround the user with diverse ways of interacting with data and multimedia information flows. These environments may prove to be the ultimate successor of the desktop metaphor for information technology work.”* [Humphreys et al.00]

### 1.1 Motivation

Large seamless tiled displays are an approach to show information, explore and visualize scientific data from complex simulations, or render 3D scenery at high resolution with high refresh rates and with high details. Tiled displays were first based on high-end rendering system with parallel computing abilities. These high-end systems supported the computer graphics community for a long time with high resolution graphics scenery featuring the possibility to drive several displays concurrently needed for tiled displays at high or constant frame rates.

The drawbacks of high-end systems today are on the one hand their price and on the other hand the fast performance growth of low-cost graphics accelerators in the last years reaching or even putting high-end machines capabilities behind them. Nowadays it is no problem to build a rendering system with a cluster of computers based on cheap PC hardware each equipped with low-cost graphics accelerators connected to one or more displays to produce one large tiled display with an immersive panoramic view and high-resolution

at the same time.

When designing such an equivalent but cheap system we have to keep in mind that we want to design a rendering system that allows an application to render to a large tiled display with the ability to distribute scenery content, user input, and other data like synchronization points relevant for rendering. We have to switch from a shared memory approach to a distributed one. With the reduction of prices for displays, computer hardware, and graphics accelerators we can build such a rendering system multiple times cheaper than high-end rendering systems. While the costs for high-end systems are above a million dollars, the costs of a PC hardware based render cluster should not exceed an order of magnitude less than the high-end systems' costs. With costs in mind and the rapidly increasing performance of present-day graphics hardware, the inexpensive solution also is the better one making it possible to track modern computer and graphics hardware and architecture. So we are able to extend and maintain our system with minimal financial effort and keep pace with newer and faster hardware.

Taking a look at the software side, the advantage of a tiled display render system is its high scalability and flexible extension without the need to change the application's code. We are able to run any application we want on the large display without or with minor changes. This leads us to another goal of large displays. With a tiled display we are able to render large and complex 3D scenery at high detail giving us the chance to explore virtual environments with a new sight. Running already existing applications on the tiled display, we can explore and visualize scientific data from complex simulations or render virtual environments with higher details and at higher resolution than using a single projector or monitor (see figure 1.1). This impressive experience in a virtual environment is the final goal of any virtual or augmented reality system leaving the user without any doubt, that his experience is real.

In this thesis we examine the different approaches taken by other research groups to build a tiled display. The goal is to build a seamless tiled display from scratch using mid-range projectors and a computer cluster to render 3D scenery. All computers within the cluster are autonomous and connected via a local area network to form an integrated facility. The cluster renders a common shared scenery. The rendered images are projected on a display surface and form a large high-resolution tiled display. The aim of this thesis is to show the advantages to render complex scenery with a render cluster and a tiled display. These advantages are high detailed scenery, high frame rates, and multiple setups including front projection, stereo projection, and auto-calibration of arbitrarily positioned projectors.

To be able to render to the tiled display, we used and extended our

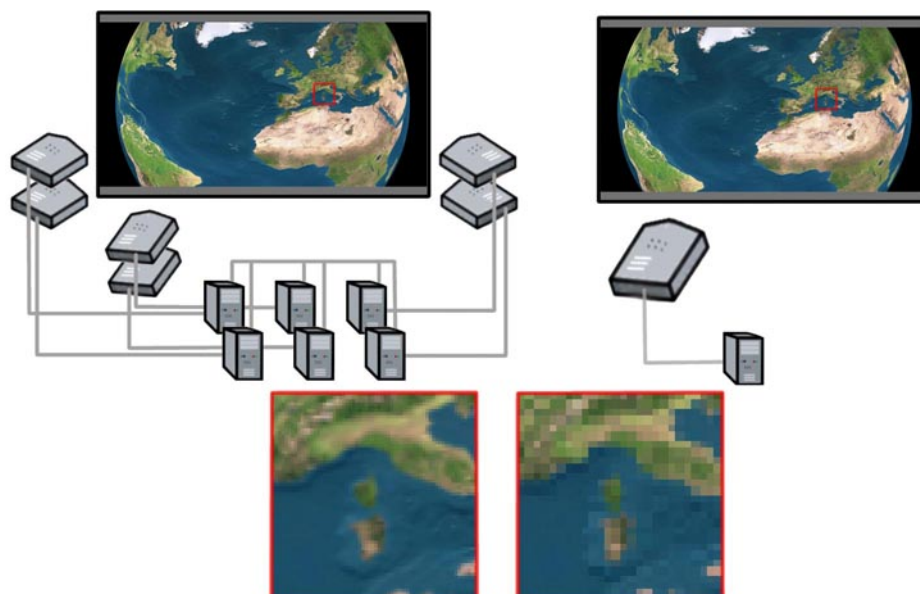


Figure 1.1: A 4.7 Megapixels tiled display, using six mid-range projectors with a resolution of  $1024 \times 768$  pixels each versus a  $1600 \times 1200$  pixels high-end projector reaching a resolution of about 1.9 Megapixels. As shown in the magnified sections of both projections, on the tiled display finer details are visible. If both projections should be equally bright, the high-end projector must be 6 times brighter than the mid-range projectors.

Studierstube framework [Studierstube]. The Studierstube system is a collaborative augmented reality system. Its primary use is to provide a 3D render system which augments real scenes with virtual 3D graphics. It is a multi-user and multi-application system allowing multiple users to collaborate in a virtual world on different tasks.

We create the Studierstube Render Array - StubeRenA - which is a render cluster that drives a seamless tiled display. Each node within the render cluster runs the Studierstube system and represents a “user” in a virtual world. Each “user” renders his part of the scenery on his tile of the tiled display to form the seamless tiled display in the end.

## 1.2 Overview

Chapter 2 investigates the differences of hardware components we need and states our way of choice. We depict the differences of computer architectures

and the choice of projectors and screen materials leading to different tiled display configurations.

Chapter 3 gives an overview of other research projects. We will investigate their approaches and ask for advantages and disadvantages of their systems. We give a classification of the different tiled display setups and the different software solutions to run a tiled display.

Chapter 4 shows our way of choosing between different approaches from other research projects keeping in mind a low budget solution with high visual accuracy and high scalability.

Chapter 5 gives an overview of our implementation to calibrate a tiled display with arbitrary size and resolution, and to enhance the Studierstube framework to render Studierstube applications on the tiled display.

Chapter 6 & 7 give a summary of our approach and serves as a guideline for developers to extend the StubeRenA project in the future. It points out directions that have been left out in this work and describes ideas that came to our mind in the course of research.

Chapter 9 documents the usage of our calibration software to calibrate oblique projectors to build a seamless tiled display.

Chapter 10 documents the code and classes to run our calibration software.

# Chapter 2

## Commodity hardware today

To build large display screens with today's commodity hardware, we have to take a look at the different hardware components we need. They can be divided into three parts, projectors, computer systems, and screen material. We have to investigate all three kinds of hardware to pick out the most suitable for our solution concerning price, durability, and maintenance.

### 2.1 Projectors

Projectors are the most important hardware to investigate, because they create the tiled display in the end. The choice of projector is perhaps the most difficult decision, because there exists a variety of models differing in price, resolution, and brightness. These factors determine the appearance of the whole tiled display. There are also some other factors that account for appearance, but the mentioned one are the most critical one. So the projectors have to fulfill a variety of parameters to be selected as the hardware to use.

- *Price* is a main factor when designing a tiled display. Depending on the resolution of the display we need half a dozen or a dozen of projectors to satisfy our needs. So keeping the price low will save a lot of money at the end. Our restriction was a range between \$3000 to \$5000 dollars. Although there are projectors beyond the \$10K dollar mark which are mainly used for high end multimedia purpose (like in seminar rooms, or theaters) supporting different video formats, automatic correction, and distortion, automatic brightness adaption, and many more. But with the drop-off in prices of the low and middle class projectors it gets possible to fulfill the other factors without exceeding a limited budget.

- The *resolution* of each projector determines the resolution of the tiled display as a whole. Using low resolution projectors increases the number of projectors we need to build a high resolution tiled display. This in turn raises the number of display hardware and finally the number of computer hardware units. But with the ongoing improvement in LCD and DLP technology, the resolution of low-cost projectors will move beyond the one Mega pixel mark in the near future supporting resolutions above  $1280 \times 1024$  pixels.
- *Brightness* comes into concern when using the tiled display in normal environments unlike seminar rooms or theaters which are used with dimmed daylight or artificial lighting conditions. Brightness should be as high as possible to meet our need to set up a tiled display even in well lit rooms. We choose a minimum brightness of about 1500 ANSI-Lumen. This is enough because the use of several projectors on a screen surface increases the luminance perceived by the viewer compared to only one very bright projector (i.e. four 1500 Lumen projectors will be as bright as one 6000 Lumen projector when lighting an area on the screen with fixed size).
- *Weight* as a factor of concern was addressed to keep our tiled display transportable, using it as a portable high resolution display easy to set up and install anywhere we need. Weight also affects scalability. If heavy projectors are used, special mounting racks have to be used, but with light ones we can use off-the-shelf mounting systems.
- *Other factors* also delimits the appearance, clarity, and image quality of the tiled display. These factors include

*color gamut:* using different projector models lowers the impression of one large display since seams between adjacent projectors become noticeable

*data interfaces (analog or digital):* while analog signals can be transmitted over long distances, a digital signal will save the clarity and sharpness of an image since D/A-A/D conversions are unnecessary

*optics:* optical properties affect image quality (flatness of focus, sharpness, image distortion, etc.)

*refresh rates:* high refresh rates are vital for active stereo, but also avoid eyestrain of the viewer

## 2.2 Computer systems

There are two types of computer systems currently used to drive a tiled display. *Shared memory machines* deliver the possibility to share scenery content and data, while *networked PC clusters* keep scalable under some aspects. Using high-end computer systems with shared memory and the ability to drive more than one graphics accelerator shows up to be very expensive. There also exists a PC based shared memory solution, i.e. a PC equipped with several graphics cards to drive several displays. The limiting factor of the PC approach is its poor bandwidth when complex scenes should be rendered. The drawback of both approaches is that only a limited number of displays can be driven, which restricts the resolution of the tiled display. For the PC render cluster, we can use off-the-shelf computer hardware equipped with a commodity graphics accelerator and a network card to be able to distribute, or replicate scenery and application data among the nodes in the cluster.

In each case different problems exist that need to be addressed. Shared data management, computation speed, bandwidth to the rendering pipelines, and display synchronization, to list a few of them. Each system keeps its own solution for these tasks, including software reengineering or adaptation to run common applications on the tiled display. In the end, the image generators must have sufficient performance to deliver real-time performance. See section 3.2 for a brief description of advantages and disadvantages of the two architectural approaches. It should be pointed out that communication and synchronization is much easier in the shared memory architecture, but as mentioned in section 1.2 we want to use off-the-shelf computer hardware charging the budget as little as possible. This approach keeps our system scalable to increase the resolution of the tiled display in the future by adding some computers and projectors to the render cluster.

## 2.3 Screen material

When designing a tiled display an important step is to choose the right screen material. There are two possibilities to drive the display. On the one hand rear or back projection which will be more elegant and comfortable for a user, because he is not limited in movement in front of the screen to watch the scenery. So the user can pick out details shown on the display while nearly touching the screen with his nose, or point to the screen without interfering with the displayed content. On the other hand during front projection the user will sometimes partially occlude the scenery when standing in the beam

of light of any projector. It should be pointed out that [Sukthankar et al.01] and [Jaynes et al.01] show a way to remove shadows on the display surface. But these approaches are difficult. The same display area must be illuminated redundantly by more than one projector under different projection angles to mute shadows. This increases the number of projectors and computers used within the render cluster. During rendering an additional task must be running to detect shadows with cameras. Despite its disadvantages, front projection is the only option when space is limited to setup the tiled display, like in office rooms.

The primary criteria for choosing screens include image performance (e.g. brightness, resolution, angle of view, contrast ratio, conserving polarization), availability of large seamless sheets, type of mounting method, rigidity (or degree of self-support in large-span applications), weight, fragility, portability, and cost. In addition we need to consider physical constraints. For example, if a system must be moved, then we need to use a lightweight screen and lightweight mounting system. This would probably indicate using a flexible fabric screen material and a tension-based mounting system.

## 2.4 Our choices

As we see the choice of hardware directly affects the budget. Since we want an easy to install and transportable tiled display, we take lightweight projectors below 2 kg. We use mid-range projectors with a resolution of  $1024 \times 768$  pixels and a brightness of 1900 ANSI-Lumen. This makes our tiled display even useable in daylight environments. To be prepared for the future and be able to increase the resolution of our tiled display, we take the render cluster approach. We use commodity PC hardware equipped with one graphics accelerator to render 3D sceneries. We also put in network cards to be able to distribute data among a common local area network. The screen material of choice was a polarization conserving front projection sheet. This gives us the freedom to use the screen for passive stereo projection.



# Chapter 3

## Related work

We have investigated related approaches to build up large screen displays with high resolution from the following point of view:

- *Scalability*: The ability to add new hardware components, i.e. adding more displays and/or computers to the render cluster, without changing system, or application code, or without any needs to change the used render architecture, and without lowering the overall performance.
- *Maintenance*: Maintenance comes into concern when thinking about a large display area driven by a dozen or more displays. You can imagine how time consuming manual adjustments of each projector will be. So we need to keep maintenance effort as low as possible. All maintenance steps if any should be done by the tiled display system automatically.
- *System architecture*: This is a major point concerning costs and maintenance. Even with a low budget a suitable rendering cluster can be built showing results comparable to high-end system solutions.
- *Performance*: How a system performs directly influences its usability. The better the performance, the more realistic can the scenery be displayed, dissolving the user's doubt or disbelief in acting in a non-real environment. Our goal is to fade the border between reality and virtuality.

### 3.1 Calibration approaches

A number of research groups have investigated tiled displays and used them to create a single high-resolution seamless display by combining a collection of lower resolution projectors. Research approaches have shown that these

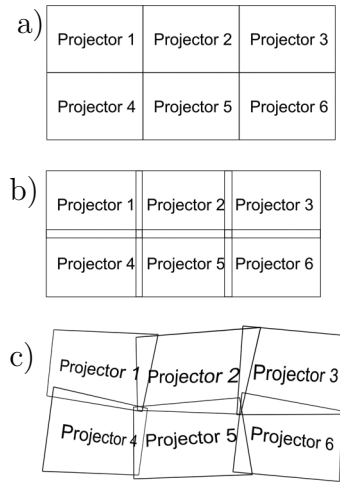


Figure 3.1: The three display groups: a) abutted, b) regular overlap, c) rough overlap.

displays can be divided into three groups: *abutted*, *regular overlap*, and *rough overlap* displays [Gotz01].

### 3.1.1 Abutted displays

Abutted displays are the first multi-projector displays that come in mind when building a tiled display. They are easy to use in a render system, because each pixel on the display screen is rectangular and next to each other and furthermore every pixel in the frame buffer is directly represented by one pixel on the screen. This allows applications to span their display content over many millions of pixels with minor changes in rendering the display content and without any need to undistort oblique projected pixels. But this pixel to pixel relation leads to the need that all projectors in the display are carefully aligned side by side in such a way that neither do pixels overlap nor does any gap between pixels of adjacent projection areas exist. Abutted systems are fairly common and are used in everything from sports stadium scoreboards to trade show exhibits. Some examples of abutted displays are the CAVE [Cruz-Neira et al.93] (while not high-resolution), “Office of Real Soon Now” [Bishop00], and the display wall system at Lawrence Livermore National Laboratory [Schikore et al.00]. This display system scales well, but the maintenance increases with every projector device added to the system. Hence the system must deliver the possibility to adjust each projector, an

high-tech rack systems must be used. This leads to high costs for the rack and for calibration. Since every projector is calibrated by hand, calibration is a time-consuming task. Another drawback of this system is that each projector has to be orthogonal to the display screen to create rectangular images, thus only back-projection is possible which will need additional space behind the display screen. Therefore if a projector is misaligned, pixel errors will be markable on the tiled display disturbing the illusion of a large high-resolution display. See figure 3.1(a) for an abutted display.

Hereld et al. developed sophisticated projector positioner to calibrate projectors by hand. They published an image of this positioner in their paper [Hereld et al.00b]. Also hardware blending masks are used to avoid overlapping regions on the display [Hereld et al.00a]. Their paper allows to get a hint how delicate mechanical fine-adjustment can be.

### 3.1.2 Regular overlap displays

Another approach to build multi-projector displays requires projectors to be carefully aligned so that there is some controlled overlap between projectors. The projectors are required to have a precise geometric relationship that ensure regularity between overlap regions. The overlap regions are used to blend imagery across projector boundaries. Alpha blending techniques are used to fade pixels of overlapping projectors. This is done to help hide both photometric and geometric discontinuities at the boundaries. Due to the overlapping regions the application's code must be changed to handle blended pixels in these regions. Princeton's Scalable DisplayWall [Li et al.00] and Stanford's Interactive Mural [Humphreys,Hanrahan99] are both examples of regular overlap displays. Like the abutted display approach the scalability of the system is good, but maintenance, costs, and space is as bad as with abutted displays, because of precise geometric calibration between the used projectors. Even though hard borders of the former display type are reduced due to controlled overlaps, pixel misalignment is still as noticeable as in the former approach. Figure 3.1(b) depicts a regular overlap display.

In this tiled display approach overlapping projection areas and tapering the brightness of the image from each projector results in a smooth intensity transition from tile to tile. This effect can be achieved in signal electronics [Panoram, Trimension] or in software [Raskar et al.99]. Blending techniques are vital to form a seamless tiled display. Blending tiles in overlapping regions in software is no problem with current graphics hardware. Raskar et al.'s approach is to weight all projected pixels by using alpha-masks for each projector. Each alpha-mask assigns an intensity weight  $[0 - 1]$  for every pixel in the projector. Weights of all projected pixels illuminating the same display

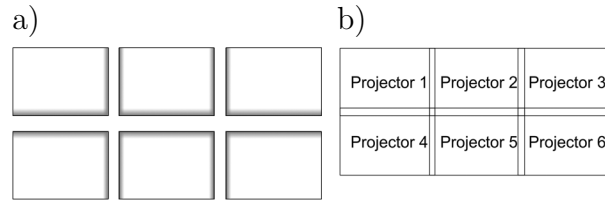


Figure 3.2: Alpha masks (a) for regular overlapping projectors (b) to blend pixels in overlapping regions.

surface point should add up to unity. When using regular overlaps the alpha masks are easy to build, because we know the exactly shape of all overlapping regions. Figure 3.2 shows the alpha masks for the regular overlap display. We will discuss the technique to compute and use alpha-mask to blend pixels in the next section in more detail.

### 3.1.3 Rough overlap displays

The last group of these display systems is the most easiest to set up and maintain because it allows rough overlap regions between projectors. The only requirement is that projectors actually overlap. This means that overlapping regions can be of arbitrary shape and size, and projectors can be installed anywhere near the display screen to project their scenery. This includes also front projection which will not be possible with the former two display types. Figure 3.1(c) shows a display with rough overlap regions.

A drawback of this approach is that application code has to be rewritten in order to counteract the oblique projections and to blend the imagery within the overlapping regions. So as an additional render step the oblique display content of each projector has to be undistorted and blended.

As a set up step, the configuration of projectors and their relative pose has to be detected. This calibration is done automatically by camera and image detection techniques. The projection area of each projector is detected by cameras to solve for projection relationships between all projectors. Each camera used introduces a possible error in misalignment of projectors due to local coordinate system conversion between camera space and projector space. The scalability of this system is limited by the errors done during calibration, thus these errors must be minimized. We will discuss this step later in more detail.

Maintenance is very low, because projectors need not to be aligned anyway and the calibration step is only made once after the system is installed or reconfigured. Even more Raskar et al. present a method to make 'intelli-

gent' projectors be aware of other projectors in their vicinity. The setup is a group of 'intelligent' projectors forming a large display area. If an unit is added to the group it sends a 'request to join' message via a proximity network (like wireless Ethernet, RF or infrared). All surrounding units receive this message and scan for a projection pattern of the new unit with their cameras. If anyone camera of the group sees the pattern, the group's units perform a calibration step and integrate the new unit to the group. Otherwise the new unit is in the vicinity of the group, but does not overlap with its own extent of the display [Raskar et al.03]. These 'intelligent' projectors are aware of one another and can be setup anywhere without any restriction. So this approach of a tiled display is self-configuring when multiple 'intelligent' projectors are put together and has no maintenance at all.

Chen et al. describe a method to minimize the calibration errors when using more than one camera (i.e. more than one local coordinate system to solve for) [Chen et al.01a]. Raskar et al. solve the problem of calibration with more than one camera even for non-planar surfaces and describe in their paper how to compensate for misalignment. In this approach, a series of calibrated stereo cameras are used to determine the display surface and individual projector's intrinsic and extrinsic parameters in a common coordinate frame. The result is an exhaustive description of the entire display environment. Although this approach allowed for a general solution, the computational effort and resources needed to implement this approach introduce their own level of complexity. They developed a two-pass rendering algorithm for edge blending and pixel morphing necessary for overlapping regions and non-planar surfaces [Raskar et al.99]. In the first pass, the desired image for the user is computed and stored as a texture map. In the second pass, the texture is effectively projected from the user's viewpoint onto the polygonal model of the display surface. The display surface model, with the desired image texture mapped onto it, is then rendered from the projector's viewpoint. This is achieved in real-time using projective textures. The rendering cost of this two-pass method is independent of complexity of the virtual model.

## Blending

As mentioned in the last section Raskar et al. developed a blending technique to blend overlapping pixels to form a seamless display. With their method each alpha-mask assigns an intensity weight  $[0 - 1]$  for every pixel in the projector. Weights of all projected pixels illuminating the same display surface point should add up to unity. The weight is additionally modified through a gamma lookup table to correct for projector non-linearities. To

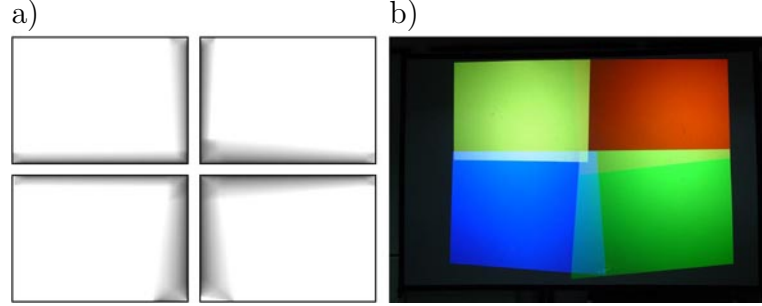


Figure 3.3: Alpha masks for four rough overlapping projectors (a) to blend pixels in the noticeable overlapping regions (b).

find the alpha-mask, they use a camera to view the overlapped region of several projectors. They form a convex hull  $H_i$  in the camera's image plane of observed projector  $P_i$ 's pixels. The alpha-weight  $A_m(u, v)$  associated with projector  $P_m$ 's pixel  $(u, v)$  is evaluated as follows:

$$A_m(u, v) = \frac{\alpha_m(m, u, v)}{\sum_{i=1}^N \alpha_i(m, u, v)} \quad (3.1)$$

where  $\alpha_i(m, u, v) = w_i(m, u, v)d_i(m, u, v)$  and  $i$  is the index of the projectors observed by the camera (including projector  $m$ ). In the above equation,  $w_i(m, u, v) = 1$  if the camera's observed pixel of projector  $P_m$ 's pixel  $(u, v)$  is inside the convex hull  $H_i$ ; otherwise  $w_i(m, u, v) = 0$ . The term  $d_i(m, u, v)$  is the distance of the camera's observed pixel of projector  $P_m$ 's pixel  $(u, v)$  to the nearest edge of  $H_i$ . Figure 3.3 shows the alpha masks for a rough overlap display.

### Calibration

In [Raskar et al.02] he and other researchers show a way to compute the relative pose among oblique projectors utilizing an off-the-self camera. They use *homographies* between the camera image and each projector image to calibrate the projectors and undistort their images. Because a homography is a planar projective transform (a *collineation* in  $\mathbb{R}^2$ ) it is defined up to an unknown scale factor by four pairs of matching points. By displaying and detecting a calibration pattern on each projector, they gather matching points and compute a homography.

A homography can be used to map two different projections of one fixed point. Think of two cameras  $C_{1,2}$ , viewing a fixed single point  $\mathbf{p}$  on a 3D

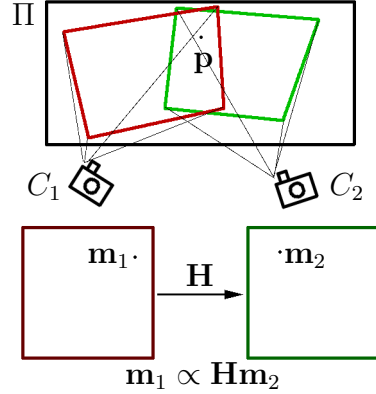


Figure 3.4: Two cameras  $C_1, C_2$  see one single point  $\mathbf{p}$  from different views  $(\mathbf{m}_1, \mathbf{m}_2)$ . The homography  $\mathbf{H}$  defines the relation between the cameras' coordinate frames.

plane  $\Pi$ , the point positions  $\mathbf{m}_{1,2}$  in the two images are related by a  $3 \times 3$  homography matrix  $\mathbf{H}$ . If  $\mathbf{m}_1$  and  $\mathbf{m}_2$  are projections of  $\mathbf{p}$ , then

$$\mathbf{m}_2 \propto \mathbf{H}\mathbf{m}_1 \quad (3.2)$$

where  $\mathbf{m}_1$  and  $\mathbf{m}_2$  are homogeneous coordinates and  $\propto$  means equality up to scale (see figure 3.4). In the same way a pair of homographies can be used to define the relationship between projector to projector coordinates, as we will see in the following section.

Raskar et al. use one single camera  $C$  to record all the projector images. The projector to camera mapping as well as relative projector to projector mapping are then described by the above plane homographies due to the planar display surface used. For notation, they choose homogeneous coordinates for both 2D camera coordinates  $\mathbf{x}_c = (x, y, 1)^T$  and for 2D projector coordinates  $\mathbf{u}_i = (u, v, 1)^T, i = 1, \dots, N$ , from multiple source projectors. In this context, the input consists of  $N$  projector images captured by the single camera  $C$  with the known homography matrix,  $\mathbf{H}_{c1}, \mathbf{H}_{c2}, \dots, \mathbf{H}_{cN}$ , satisfying

$$\mathbf{u}_i \propto \mathbf{H}_{ci}\mathbf{x}_c \quad , \quad i = 1, \dots, N \quad (3.3)$$

The display coordinates on the display surface are denoted as  $\mathbf{x}_r = (x, y, 1)^T$ . The relationship between the display coordinates and camera coordinates can be described by another 2D projective matrix  $\mathbf{H}_{rc}$ . Obviously, we have

$$\mathbf{u}_i \propto \mathbf{H}_{ci}\mathbf{x}_c \propto (\mathbf{H}_{ci}\mathbf{H}_{rc})\mathbf{x}_r \quad , \quad i = 1, \dots, N. \quad (3.4)$$

A new set of  $3 \times 3$  matrices

$$\mathbf{H}_{ri} = \mathbf{H}_{ci}\mathbf{H}_{rc} \quad , \quad i = 1, \dots, N \quad (3.5)$$

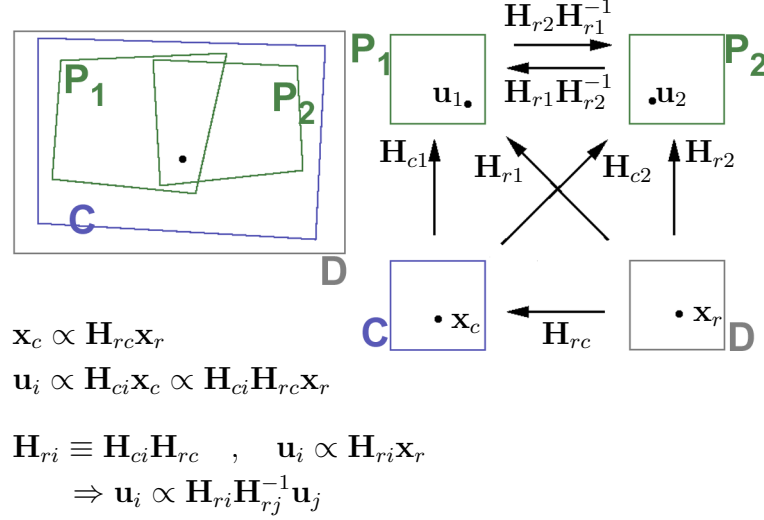


Figure 3.5: Pairs of homographies can be used to define relations between different coordinate frames.

specifies the geometrical relationship between the individual projector and final display coordinate directly. Equation 3.4 and 3.5 determines the pixel mapping between two arbitrary projectors as follows

$$\mathbf{u}_j \propto \mathbf{H}_{rj} \mathbf{x}_r \quad , \quad \mathbf{x}_r \propto \mathbf{H}_{ri}^{-1} \mathbf{u}_i \quad \Rightarrow \quad \mathbf{u}_j \propto \mathbf{H}_{rj} \mathbf{H}_{ri}^{-1} \mathbf{u}_i \quad (3.6)$$

where  $\mathbf{u}_i$  and  $\mathbf{u}_j$  denote the corresponding pixels in projector  $P_i$  and  $P_j$ , respectively (see figure 3.5).

With the derived equations the weights for the alpha-masks can be easily computed. Remember equation 3.1 that defines the weight for projector  $P_m$ 's pixel  $\mathbf{u}_m = (u, v, 1)^T$ . To find the weight  $A_m(\mathbf{u}_m)$  Raskar et al. use the homographies which are computed with normalized projector coordinates, i.e. the  $u$  and  $v$  coordinates of  $\mathbf{u}_m$  vary between  $[0, 1]$ . Hence, the distance of a pixel to the closest edge in the projector  $P_m$  is described by  $d_m(\mathbf{u}_m) = w(u, v) \min(u, v, 1 - u, 1 - v)$  where  $w(u, v) = 1$  if  $u \in [0, 1]$  and  $v \in [0, 1]$ ,  $= 0$  otherwise. This reduces the weights assignment problem, to a simple minimum function. Inserting the above derived distance function  $d_m$  and equation 3.6 in equation 3.1 leads to

$$A_m(\mathbf{u}_m) = \frac{d_m(\mathbf{u}_m)}{\sum_{i=1}^N d_i(\mathbf{H}_{ri} \mathbf{H}_{rm}^{-1} \mathbf{u}_m)} \quad , \quad m = 1, \dots, N. \quad (3.7)$$

After computing the homographies and blending mask for each projector, they compute a projection matrix and use commodity graphics accelerators to undistort the final output on each projector. The *oblique projection matrix*



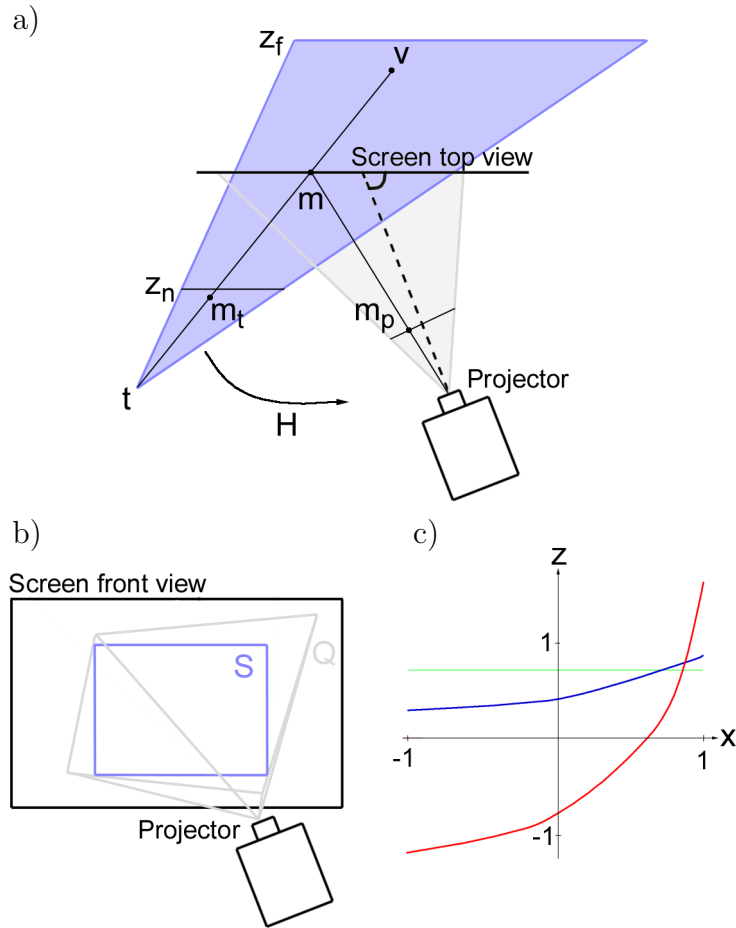


Figure 3.6: A tracked user at position  $t$  viewing a scenery with off-axis projection(a). Oblique projectors create key-stoned imagery (b). The plot (c) shows depth-buffer values along a scan line for points along constant depth (green). Applying uncorrected homography  $\mathbf{H}$  (red), the values range beyond  $[-1, 1]$  and do not change linearly. After using the corrected homography  $\mathbf{H}'$  (blue) traditional graphics pipeline can be used to render correct 3D scenery.

$\mathbf{P}_m$  for projector  $P_m$  is the homography  $\mathbf{H}_{rm}$ , i.e. the homography that maps display surface points to projector points, times a common projection matrix  $\mathbf{P}$  (like one, defined with a view frustum in OpenGL). The homography  $\mathbf{H}_{rm}$  has to be extended to a  $4 \times 4$  matrix  $\mathbf{H}_{rm}^{44}$  to be useable in a common graphics pipeline.

$$\mathbf{H}_{rm} \equiv \begin{pmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{pmatrix} \Rightarrow \mathbf{H}_{rm}^{44} = \begin{pmatrix} h_{11} & h_{12} & 0 & h_{13} \\ h_{21} & h_{22} & 0 & h_{23} \\ 0 & 0 & 1 & 0 \\ h_{31} & h_{32} & 0 & h_{33} \end{pmatrix}$$

In the end the oblique projection matrix for projector  $P_m$  is

$$\mathbf{P}_m = \mathbf{H}_{rm}^{44} \mathbf{P}. \quad (3.8)$$

With the projection matrices and blending masks Raskar and his colleagues built a seamless tiled display with oblique projectors and without any time-consuming mechanical calibration steps.

But this approach has to be extended as we will see. Raskar describes in his papers [Raskar00] and [Raskar99] a single-pass rendering method for planar surfaces with roughly aligned or even oblique projectors and a tracked user to display 3D scenery. He shows in his papers that the naive approach of using a projection matrix  $\mathbf{P}_p = \mathbf{H}\mathbf{P}$ <sup>1</sup> like the one derived above creates correct 3D scenery for an oblique projector  $P$ , but that the depth-buffer values cannot be used for visibility and clipping. In his paper the common projection matrix  $\mathbf{P}$  is used for a tracked user and updated as the user moves. The homography matrix  $\mathbf{H}$  is used to undistort the oblique projection on planar display surfaces. Rendering the virtual point  $\mathbf{v}$  for a tracked user at position  $\mathbf{t}$  using an oblique projector as shown in figure 3.6 with the given projection matrix  $\mathbf{P}$  leads to distorted depth-buffer values.

The oblique projection matrix for a tracked user calculates as follow  $\mathbf{P}_p = \mathbf{H}\mathbf{P}_T$  where  $\mathbf{P}_T = Frustum(\mathbf{t}, \mathbf{S}, z_n, z_f)Translate(-\mathbf{t})$  is the clipping volume from the user's view and is updated as the user moves.  $\mathbf{S}$  is an axis aligned rectangle on the display surface  $\Pi$  bounding the key-stoned quadrilateral  $\mathbf{Q}$  illuminated by the projector.  $Frustum(\mathbf{t}, \mathbf{S}, z_n, z_f)$  defines a view frustum by creating a pyramid with  $\mathbf{t}$  and the four corners of  $\mathbf{S}$  truncated with the near plane,  $z = \mathbf{t}_z - z_n$ , and the far plane,  $z = \mathbf{t}_z - z_f$ .  $\mathbf{H}$  is the known homography or collineation matrix to map screen pixels to projector pixels. The homography is independent of the user location and hence

---

<sup>1</sup> $\mathbf{H}$  is the homography between projector and display surface coordinates. We will use  $\mathbf{H}$  instead of  $\mathbf{H}_{rm}^{44}$  from now on.

remains constant. With the oblique projection matrix  $\mathbf{P}_p$  applied to a 3D scene, the depth values are distorted by the homography  $\mathbf{H}$ .

The depth values of virtual points between near and far plane due to  $\mathbf{P}_T$  are mapped to  $[-1, 1]$ . Let  $(m_{Tx}, m_{Ty}, m_{Tz}, m_{Tw})^T = \mathbf{P}_T(\mathbf{v}, 1)^T$  and  $m_{Tz}/m_{Tw} \in [-1, 1]$ . After collineation, the new depth value is actually  $m_{Tz} = (h_{31}m_{Tx} + h_{32}m_{Ty} + m_{Tw})$  which (i) may not be in  $[-1, 1]$  resulting in undesirable clipping and (ii) is a function of pixel coordinates, changes quadratically and hence cannot be linearly interpolated during scan conversion for visibility computation (see figure 3.6(c)). Using a single  $4 \times 4$  matrix, it is impossible to achieve two hyperbolic interpolations for the depth values. One solution is to render an image using the projection matrix  $\mathbf{P}_T$ , and then warp the resultant image to undistort oblique projection. This would require a two-pass rendering method that first renders the image in texture memory and then achieves warping using texture mapping. Raskar shows that rendering and warping can be achieved in a single pass using an approximation of the depth buffer values. Note that  $m_{Tx}/m_{Tw}$  and  $m_{Ty}/m_{Tw} \in [-1, 1]$  for points rendered inside the rectangle  $\mathbf{S}$ . Hence  $((1 - |h_{31}| - |h_{32}|)m_{Tz}) / (h_{31}m_{Tx} + h_{32}m_{Ty} + m_{Tw})$  is guaranteed to be in  $[-1, 1]$ . Further, by construction of  $\mathbf{P}_T$ , the angle between projector's optical axis and the normal of the planar surface is the same as the angle between the optical axis and retinal plane of frustum for  $\mathbf{P}_T$ . Thus, if this angle is small (i.e.  $|h_{31}|$  and  $|h_{32}| \ll 1$ ), the depth values are modified but the changes are monotonic and almost linear across the depth-buffer as shown in figure 3.6(c).

$$\mathbf{H}' = \begin{pmatrix} h_{11} & h_{12} & 0 & h_{13} \\ h_{21} & h_{22} & 0 & h_{23} \\ 0 & 0 & 1 - h_{31} - h_{32} & 0 \\ h_{31} & h_{32} & 0 & h_{33} \end{pmatrix}$$

With the corrected collineation matrix that undistort oblique projection and corrects depth values during matrix multiplication, we are now able to build the corrected oblique projection matrix

$$\mathbf{P}_p = \mathbf{H}'\mathbf{P}_T.$$

Another approach was taken by Chen et al. [Chen et al.00d]. They provide a mechanism to help reduce roughly aligned projectors by calculating a corrective projection matrix (a collineation matrix from projector space to display space) for each projector. An uncalibrated camera with controllable zoom and focus, mounted on a pan-tilt unit observes geometric relationships - point and line matches - between adjacent projectors to solve the matrix equations. Simulated annealing is used to find a global solution

that minimizes the overall pixel position and line slope error between adjoining projector segments. This approach requires substantial image data and computation. The drawbacks of their solution is its slowness and that data collection and a final solution can take over 30 minutes to compute. Furthermore, this approach corrects the imagery for slightly misaligned projectors, it is not clear if it can handle large misalignment.

UNC's PixelFlex system is an example of a rough overlap display [Yang et al.01].

## 3.2 Computer systems

This section investigates the advantages and disadvantages of different hardware architectures. There exists various kinds of architectures with high-end hardware architecture at the one end of the scale and off-the-shelf PC hardware on the other end. High-end hardware offers the opportunity to share memory among render tasks and rendering pipelines. It also offers the opportunity to use more than one extremely high-end graphics module allowing parallel rendering. Even more, most high-end system supports, when using multiple graphics modules, output synchronization in hardware. Shared memory leaves out the problem of data sharing via networks. Many researches have investigated the high-end system approach in the past, first because commodity hardware of that time did not fulfill their needs, second because of the mentioned capabilities and opportunities. Such a system can handle a tiled display easily, but it is not infinitely scalable, because the number of graphics modules plugged into the system is limited. Another drawback is that these systems are very expensive. Researches quickly found themselves limited in their demands to the hardware in terms of price and scalability. One example is Stanford's *Interactive Mural* [Humphreys,Hanrahan99]. The system was first implemented on a SGI machine. This high-end machine drive up to 8 outputs, so a  $4 \times 2$  display array was achieved. When the group at Stanford wanted a higher resolution, they reimplemented a new version based on a PC cluster, because the high-end machine system cannot scale beyond the eight outputs.

With the needs to build up a scalable tiled display without the costs of a high-end hardware system, we will turn to the commodity hardware approach. Commodity hardware today offers the possibility to render realistic scenery with high resolution and at high frame rates. So researchers came to the point to build a networked cluster with commodity hardware systems to deliver their needs for data sharing and synchronizing. Using off-the-shelf PC hardware and a fast rendering graphics accelerator we are able to drive one or two displays per PC. Even if we want to scale up the system, we just have

to add another PC and another display at no high costs. A drawback of the networked render cluster is its slow data sharing ability when using today's network technologies. So we are forced to think of a good design when it comes to the aspect of parallel rendering. This comes especially true when it comes to the point of sharing the scenery content and distributing it within a render cluster. But also with this drawback, the costs of the cluster and its good scalability makes it worth investigating. The approaches of other research groups have shown that it is possible to render complex scenes on a tiled display in real-time, e.g. Princeton's *Scalable Display Wall* [Li et al.00] and Stanford's new version of *Interactive Mural* [Humphreys et al.00].

### 3.2.1 Parallel computer architecture

The hardware approach in the last ten years to provide high-resolution graphics at high frame rates was tightly coupled with high-end graphics machines. These machines mostly offer the possibility for parallel computing and even for parallel rendering. These systems are equipped with one processor and one graphics module at the lowest level, or at the highest level with several processors and several graphics modules. They give the possibility to share memory among different render tasks needed to render on a tiled display.

The advantages of such a system is that a shared render context where different tasks run concurrently and in parallel to render a high-resolution scene is already supported in hardware. Each task will represent and render the scene of a specific tile on the tiled display. Since a tiled display will show a complex scene, the geometry for the different tasks will be the same, but each task only renders a part of the whole scene. With the opportunity to share data and memory on such systems, very little changes have to be done to transfer visual context and geometry data between the different tasks.

The drawback of these approaches are, while they use very specific and dedicated hardware components, that these rendering systems are very expensive, often costing millions of dollars. Another drawback is the needed scalability and extensibility for tiled displays. These system often allow to extend the number of processor and graphics modules only to a specified limit. When this limit is reached, the maximum resolution of the tiled display will be reached.

The PowerWall at the University of Minnesota and the InfiniteWall at the University of Illinois at Chicago are examples, each driven by an SGI Onyx2 with multiple InfiniteReality graphics pipelines.

### 3.2.2 Networked computer architecture

In the last few years the huge barrier between high-end graphics computer systems as mentioned in the section above and commodity graphics accelerators was broken. Nowadays graphics accelerators for commodity computers even top the performance of high-end graphics systems. Thus we have the possibility to build a tiled display with off-the-shelf hardware components with the performance of high-end graphics machines.

To create a tiled display we need a bundle of computers, each equipped in such a way that it can render 3D scenery, display the scenery on the tiled display, and communicate with other computers that contribute to the tiled display. We will name this bundle a *render cluster*. Within the render cluster different machines solve different problems as described in section 3.4. We have to solve the problem to render to the tiled display within an ordered manner and to show a homogeneous content. This is done via data distribution and communication which are the backbone of a render cluster. So each machine is equipped with a network card and connected to a local area network. The next section gives a brief description on different network cards and approaches and their advantages and disadvantages.

Since each machine executes different tasks, we will divide the machines into two parts according to their tasks. One part is formed by *display nodes*, where each display node is responsible to render the correct part of the scene and to display it on the tiled display. The other part consists of *control nodes*, where each control node's task is to handle user input, and run the application code. This is only one possible division of the cluster. In another architecture, an control node handles user input and distributes it to the display nodes, while each display node runs the application code, renders the scene, and displays it on the tiled display. We will investigate for the different task approaches in section 3.4.

## 3.3 Network hardware

This section investigates different network designs available today. It points out their usage in the terms of scalability, bandwidth, transfer rates, and latency. With present-day network technology it is possible to transfer data with more than one Giga-bit per second. Thus current technology gives us the possibility to render even very complex scenes at high frame rates without the need to store the scenery data on the display nodes.

### 3.3.1 Ethernet

In 1972 the first experimental version of Ethernet was developed by Metcalfe and his Xerox PARC colleagues to link Xerox Altos to one another, and to servers and laser printers. This very first version was called the *Alto Aloha Network*, but in 1973 Metcalfe changed the name to *Ethernet*, to make it clear that the network mechanisms had evolved well beyond the Aloha system, and that Ethernet supports any computer. Since 1973 the Ethernet gained a great popularity and is the most commonly chosen network layer for local area networks.

The Ethernet system consists of three basic elements

- the physical medium used to carry Ethernet signals between computers,
- a set of medium access control rules embedded in each Ethernet interface that allow multiple computers to fairly arbitrate access to the shared Ethernet channel, and
- an Ethernet frame that consists of a standardized set of bits used to carry data over the system.

The computers linked to an Ethernet network, also named stations, operate independently of all other stations on the network. There is no central controller and therefore each station can send and receive data to and from the shared medium. Thus a protocol is needed that is aware of collision detection and multiple access to the medium. Ethernet signals are transmitted serially, one bit at a time, over the shared signal channel to every attached station. To send data a station first listens to the channel, and when the channel is idle the station transmits its data in the form of an Ethernet frame, a so called packet. After each frame transmission, all stations on the network must contend equally for the next frame transmission opportunity. This ensures that access to the network channel is fair, and that no single station can lock out the other stations. Access to the shared channel is determined by the *medium access control* (MAC) mechanism embedded in the Ethernet interface located in each station. The medium access control mechanism is based on a system called *Carrier Sense Multiple Access with Collision Detection* (CSMA/CD).

The CSMA/CD protocol functions somewhat like a dinner party in a dark room. Everyone around the table must listen for a period of quiet before speaking (Carrier Sense). Once a space occurs everyone has an equal chance to say something (Multiple Access). If two people start talking at the same instant they detect that fact, and quit speaking (Collision Detection).

These techniques make the Ethernet a powerful tool to build up local area networks (LANs) very quickly with heterogenous hardware. Nowadays LANs are build up with 10 Mbps Ethernet and 100 Mbps Fast Ethernet.

Also the 100 Mbps Fast Ethernet seemed to be the fastest transfer rate possible with the Ethernet technology, recent research have shown that also 1 Gbps is possible with this technology, and, as technology outlays itself, 10 Gbps Ethernet technology is in a ready to be released state.

Thus the ethernet technology offers different bandwidth ranging from 10 Mbps up to 1 Gbps. The latency depends on the system overall accesses. The more stations want to access the network at the same time the worse the latency get due to the collision detection algorithm. The scalability of the system is very good, since there is no restriction in adding stations to the network.

### 3.3.2 Myrinet

[Boden et al.95] developed another technology introduced as *Myrinet* to provide a fast network with 1 Gbps and higher transfer rates. Myrinet is a new type of local-area network (LAN) based on the technology used for packet communication and switching within "massively parallel processors" (MPPs). Think of Myrinet as an *MPP message-passing network* that can span campus dimensions, rather than as a wide-area telecommunications network that is operating in close quarters. The technical steps toward making Myrinet a reality included the development of

- robust communication channels with flow control, packet framing, and error control,
- self-initializing, low-latency, cut-through switches,
- host interfaces that can map the network, select routes, and translate from network addresses to routes, as well as handle packet traffic, and
- streamlined host software that allows direct communication between user processes and the network.

Myrinet was developed from the results of two research projects, the Caltech Mosaic, an experimental, fine-grain multicomputer, and the USC Information Sciences Institute (USC/ISI) ATOMIC LAN, which was built using Mosaic components (see [Boden et al.95] References 1 to 3).

**Multicomputer Message-Passing Networks.** A multicomputer is an MPP architecture consisting of a collection of computing nodes, each



with its own memory, connected by a message-passing network. The Caltech Mosaic was an experiment to “push the envelope” of multicomputer design and programming toward a system with up to tens of thousands of small, single-chip nodes rather than hundreds of circuit-board-size nodes. The fine-grain multicomputer places more extreme demands on the message-passing network due to the larger number of nodes and a greater interdependence between the computing processes on different nodes. The message-passing-network technology developed for the Mosaic achieved its goals so well that it was used in several other MPP systems, including the medium-grain Intel Delta and Paragon multicomputers, the Stanford DASH multiprocessor, and the MIT Alewife multiprocessor.

Myrinet provide a network technology with extremely fast transfer rates with parallel transfer schemes and it is a good candidate to upstage the Ethernet technology. The parallel transfer schemes restricts the scalability of the system. It allows two nodes to communicate without disturbing two other nodes within the same network. But the number of nodes within a network is limited. To keep the system scalable a hierarchial network must be built.

## 3.4 Data distribution

We will define some terms that we use in this thesis to be clear what we are talking about. A *render cluster* is a collection of autonomous computers linked by a network, with software designed to produce an integrated computer facility. This facility is able to render to a tiled display and distribute data within the render cluster. Each computer within the render cluster is referred as *render node*. As other authors in the computer community have shown, render nodes have different *tasks* to process to run a tiled display. We will divide these tasks into four groups, *user interface*, *application*, *render*, and *display* task. Depending on the tasks a render node processes, it belong to one of the two node groups. *Control nodes* process user input (e.g. mouse movement, tracker data, etc.), while *display nodes* show framebuffer content and drive the different displays of the tiled display (e.g. projectors). The application task executes application code, and the render task renders and draws scenery content. Depending on the individual performance of the two node groups, both tasks can be shifted among the two groups, see figure 3.7. So a minimal render cluster may have one control node, for user input, and several display nodes, for displaying.

The aim of data distribution is to distribute rendering content, user input, and shared application data to each render node to show huge high-resolution

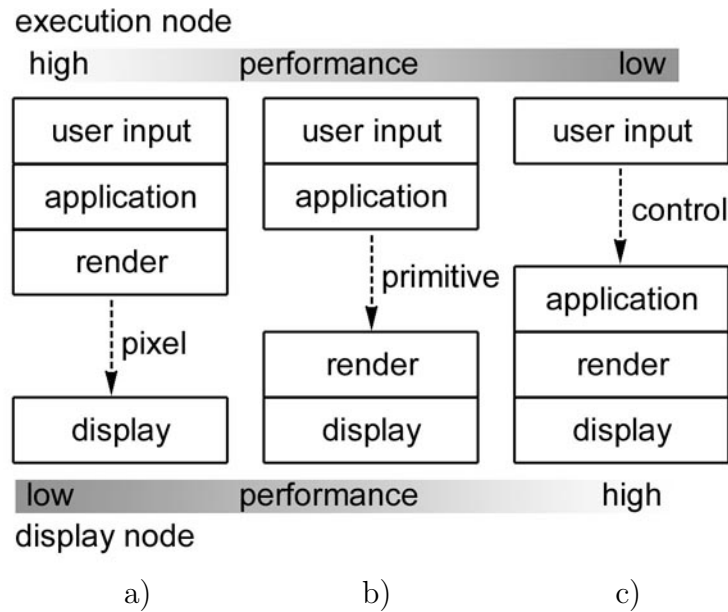


Figure 3.7: The four task groups shifted between a control and a display node, depending on each nodes performance.

scenery on a tiled display. Since we know the different tasks for a tiled display, we also know the two groups within the render cluster that run the tasks to manage the tiled display. It is a design philosophy which group (control, or display) handles which tasks. But at this point we can distinguish between two different distribution techniques. On the one hand the *client-server approach*. Control nodes act as clients, while display nodes run as servers waiting for their clients' data (e.g. user input, scenery, images) and show their part of the scenery on the screen wall. On the other hand the *master-slave approach*. In this architecture control nodes run as masters, processing user input, and distribute their data to display nodes which themselves run as slaves and are triggered by the masters to show the scenery. Both distribution techniques are an issue of implementation design, so we will not go into detail for now. Both techniques have their own advantages and disadvantages. But both provide similar or equivalent concepts for distributing and sharing data among a render cluster. We will discuss our choice of distribution technique in chapter 4.

We will give a short overview at the different approaches for the task shift problem and discuss their advantages and disadvantages in the following sections. In a poor performance control node scenario, control nodes only process user inputs, while display nodes have to process all other tasks

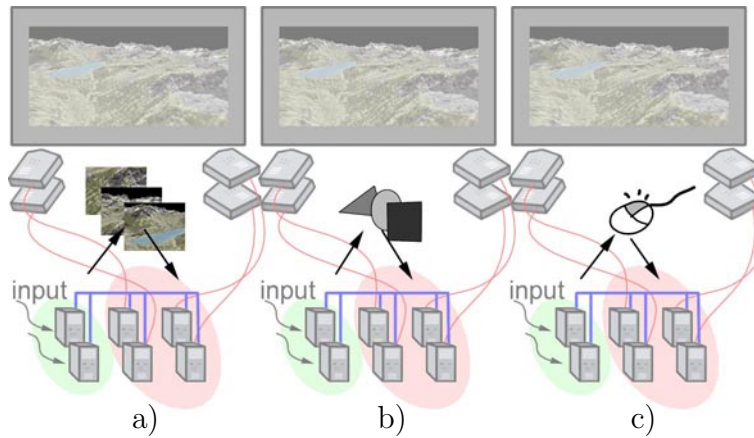


Figure 3.8: A render cluster with the render nodes divided into two groups, control nodes (light-green) and display nodes (light-red). With high-performance control nodes compressed image data is sent to the display nodes (a). With mid-performance control and display nodes primitives are distributed (b). And with high-performance display nodes, only control data must be sent (c).

(application, render, and display). So the display node side keeps the main part of work. When assigning maximum load to control nodes, each control node has to process user input, execute application's code, and render the scenery. The display nodes only display pixels distributed by the control nodes. So the main part of work is on the control node side. In the medium performance case the tasks are evenly share among the node groups. The control nodes perform user input and application tasks, while the display nodes process render and display tasks.

We have to care about synchronization when multiple nodes run the application code – regardless of which group they belong to. We have to synchronize both, the application's execution path on each node processing the application code and the rendering output. Depending on each node's performance several scenarios arise. Figure 3.8 depicts the task load for each node group. The performance of the individual nodes influences the architectural design for the render cluster. Depending on the performance of each node group we shift the task load to the higher performance group. Of course the control and display group can collapse if high performance machines are used. In this case all nodes can handle all four tasks in real-time and no distinction between control and display nodes is possible. We will go into more detail in the following sections to describe the different scenarios and

their feasible solutions.

Scenery data (like geometry, position, orientation, etc.) must be shared and updated among the render cluster. We can choose between three alternatives for distributing scenery data to show the scenery correctly on a tiled display [Chen et al.00b].

We can render the scenery on control nodes for poor performance display nodes and then distribute packed image data to the latter to display the output (see figure 3.7(a) and figure 3.8(a)). Since we only distribute pixels to display nodes we describe this technique as *pixel distribution* in section 3.4.3.

With higher performance display nodes and the ability to render with graphics accelerators, we can distribute the primitives instead of pixels to save network traffic. After distributing the primitives building up the scenery we render them directly on the display nodes. Thus a copy of the application code runs on each control node while the display nodes render the scene (see figure 3.7(b) and figure 3.8(b)). For these midrange performance display nodes display content distribution can be viewed from the used API to render the content. One can use an OpenGL API to render 3D scenery, or OS dependent APIs to render a large desktop environment. Using and replacing API-calls to distribute the render content is known as *primitives distribution*. An obvious solution for primitives distribution is to replace the installed system's driver for the demanded API with an enhanced version of it to be able to distribute the different API calls that are needed to render the scenery on different display nodes. We will have a closer look on related works handling this style of data distribution in section 3.4.2.

With high performance display nodes, we can in turn run a copy of the application code on each display node. The control nodes only have to process user inputs (see figure 3.7(c) and figure 3.8(c)). Thus control is the only data distributed from control nodes to display nodes and is referred as *control distribution*. We will not discuss control distribution because there is no tile specific implementation needed. The control data are sent to all display nodes and processed by the application code. Since control data are normally small data packets the traffic load on the network is the least.

Special care has to be taken if we run multiple copies of the application code on anyone group of nodes. In this scenario we have to take care about a parallel execution of the codes. Section 3.4.1 describes two approaches to synchronize application's execution path on different nodes. It turns out that synchronization of the application's execution path on each node running the application will show identical behavior. This kind of synchronization in respect to execution is known as *synchronized execution*.

### 3.4.1 Synchronized execution

The basic idea in synchronized execution is to run multiple instances of an application on different render nodes. The application's execution path will be synchronized within a synchronization boundary, so that the application instances assume identical behavior within this boundary. As a result of synchronization these instances generate identical scene descriptions, which can simply be identical OpenGL 3D primitives across all render nodes or some higher-level scene description that each node instantiates in a tile-specific fashion. In the first case the graphics accelerator performs tile-specific culling. With the appropriate projection matrix set, as described in section 3.1.3, it processes all primitives, but only renders those that fall within its view frustum. In the second case the higher-level scene description can be used to cull parts of the scene that are totally outside the tile's view frustum. If we use scene graphs that organizes the scene data in a hierarchy of bounding volumes in respect to object vicinity, whole subgraphs can be pruned near the root without visiting and culling each leaf of the subgraph. Figure 3.9 depicts the difference if view frustum culling is done in hardware, or in a higher-level scene description fashion.

We will now investigate two approaches taken by Chen et al. and their usability.

#### System-level synchronization

The basic idea in system-level synchronization (SSE) is to run an application on multiple nodes in a transparent fashion, i.e. without modifying and relinking the application code. Chen and his colleagues showed in [Chen et al.00c, Chen et al.00b] that multiple copies of a single-threaded application can run unchanged while intercepting system specific API-calls to enable execution path synchronization. Synchronizing a single-threaded program at the system call level leads to synchronized program execution. The reason is that code execution is deterministic from the microprocessor architecture's point of view. Only external events can influence a single-threaded application's execution path. The way these external events affect the program behavior is through the system-call interface. It follows that if the interaction between the program and the OS environment is identical on all nodes, the program instances will all follow the same execution path and exhibit the same behavior, and as a result, produce the same graphics primitives.

A few simplifying assumptions must be made for the SSE approach. First, programs that interact with the rest of the system via shared-memory seg-

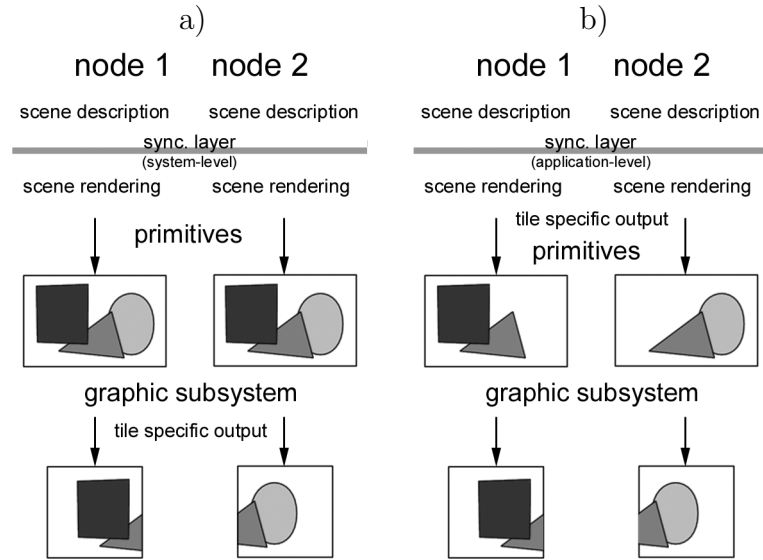


Figure 3.9: Running multiple applications synchronized with system-level synchronization (a) and application-level synchronization (b). While in the first approach the whole scenery must be drawn on each node, in the latter approach only the part of the scenery that belongs to the tile's view volume must be rendered.

ments are ignored. In such a case reads and writes to the shared-memory segments have to be intercepted and distributed in order to achieve identical application states among the nodes. This leads to a permanent interception of the application and increases the overhead to run synchronized. Second the applications must use system-call APIs defined on the OS, like accessing the CPU cycle counter via the `QueryPerformanceCounter()` on the Windows platform, so that these accesses can be intercepted to be used for synchronization.

This mechanism has one limitation. It cannot guarantee to work with arbitrary multi-threaded programs. It is hard to mimic identical interleaving of threads among multiple program instances. In theory this mechanism should also work for those multi-threaded application in which only a single thread interacts with the environment while the other threads simply compute the scene without affecting the internal states of the application.

Chen et al. show that among numerous system calls an application makes, only a handful of them can alter application states. These calls include to query window messages and the system timer. Their approach of a system-level synchronization layer performs synchronization for these specific system calls.

This approach to synchronize multiple instances of an application is the most suitable when program code is not available and cannot be altered. So it represents an easy and fast approach to make synchronization work. But if code is available another approach is more practical. This is even more true when running multi-threaded applications. As a drawback of this mechanism, since the code of the application is left untouched, the whole scenery and all primitives are generated on each node processing an application's instance. If we are rendering hundreds of thousands of polygons the frame rate will drop below an interactive rate. So we have to think of a more elegant way to render only a subset of the polygons that really belongs to the view of a specific tile.

### Application-level synchronization

System-level synchronization can synchronize multiple program instances running on nodes within a render cluster. However, it is not guaranteed to work for multi-threaded programs. Also the former approach cannot separate the program into scene management and scene rendering components, i.e. the parts of a program where the scene is described and managed in some way with respect to tile-specific boundaries and where the scene description is interpreted and primitives are rendered. With system-level synchronization the scene rendering component generates all graphic primitives describing the scene and the graphics accelerator performs tile-specific rendering and culling.

Chen et al. presented a way to solve this problems. The synchronization boundary is moved into the application itself. The same mechanism for system-level program synchronization can be used to synchronize program instances at the application level. Instead of synchronizing system calls, they synchronize function calls within the application. They provide a simple API call `SynchronizeResult()` to let all display nodes get consistent results. This function is used for each function in the application code that must be synchronized and must show identical results. A synchronized function operates on result of the replaced function and stores it temporarily. Then the `SynchronizeResult()` function is called with the temporary result to get consistent results among all nodes. After this function returns the result is identical on each node and can be used by the application. Table 3.1 shows the scheme how a synchronized function is built.

This solution is the best choice in designing a tiled display. Each application instance can take advantage of application-level synchronization by simply performing high-level object culling based on its tile view frustum. Given  $N$  projectors in a tiled display, each screen tile frustum is approxi-

---

```

synchronized syncF(a,b,c,...) ⇒ result:
    temp ← F(a,b,c,...)
    SynchronizeResult(temp)
    result ← temp

```

---

Table 3.1: To run multiple application instances synchronized, function calls are synchronized in the application-level synchronization approach.

mately  $1/N$  of the global frustum. In general this results in a small fraction of objects to render for each tile. Instead of generating all the primitives and letting the graphics accelerator throw out primitives that fall outside the local frustum, as in the former approach, the scene rendering component of each program instance can avoid generating groups of graphics primitives, by comparing their bounding volume with its view frustum. Such high-level culling needs much less computation time and consumes far less local bus bandwidth.

### 3.4.2 Primitives distribution

Leaving application code unchanged is one of the major goals in designing a tiled display in order to run an application on the tile display where source code is not available. The easiest way is to replace the driver used to show the graphics content, mostly an graphics API like OpenGL or the OS widget construction routines, with an enhanced version on one machine to supply rendering on each display node invoking the graphics API routines on them. Every display node renders a part of the tiled display without running the application code itself. The control node delivers the appropriate display nodes with the given geometry and appearance of the scenery. Thus only graphics API commands must be sent from the control side to the display side keeping the traffic load via the network as low as possible. This approach was implemented in a server-client model where the client act as the application executing machine (control node), using the display nodes as servers to show the scenery, and in a master-slave model where the master executes the application intercepting the graphics primitives and send them over the network to the display nodes which are acting as slaves.

#### OpenGL extensions

Researchers at Stanford University [[Humphreys et al.00](#)] investigated the design of a distributed graphics system that allows an application to render to



a large tiled display. Their goal was to run a normal OpenGL application unchanged on a large tiled display. They showed when rendering large scenes with a heavy load of geometry on their distributed OpenGL implementation **WireGL**, the total render time was comparable to rendering on a single machine. The render time fell off to about 87% while using 32 render servers building a  $8 \times 4$  tiled display.

They developed an efficient protocol to send OpenGL commands to the render servers by replacing the OpenGL driver on the client host. Their assumption was the spatial locality of successive primitives. After packing many primitives together (*bucketing*), the packet will be sent to just one server in most cases. Only in overlapping regions a copy must be sent to each server that contributes to this region. The algorithm also keeps track of state changes and updates the render servers' states when needed, saving traffic load on the network, because not every state change belongs to all servers.

They have shown in their results that scalability is possible until the bounding box of the packed geometry data reaches the size of a tile in the display, i.e. when the packing algorithm is forced to send every packet to more than one server, the frame time of their distributed rendering system will slow down.

“Although bucketing is critical to achieving output scalability, it is important to note that it does not allow us to scale the size of the display forever. As the display gets bigger (that is, as we add more projectors), the screen extent of a particular bounding box increases relative to the area managed by each rendering server. As a result, more primitives will be multiply transmitted, limiting the overall scalability of the system.” [Humphreys et al.00]

But they have proven that it is many times faster to explicitly send the packets to the servers they belong to than to simply broadcast all the OpenGL commands to all servers. This comes especially true if the number of displays is increased. They compared the render time using their isosurface set **March** with over 1 million triangles.

“Broadcasting commands to twice the number of servers halves the rendering speed, as expected. For **March**, WireGL's rate for 32 rendering servers only decreases by 13% from the single server configuration, compared to broadcast, which runs 25 times slower. WireGL's performance decrease is due to a small number of geometry primitives crossing multiple server outputs, resulting in additional transmissions of the geometry buffer.” [Humphreys et al.00]

**Chromium** is an extension of the WireGL concepts. It uses WireGL as a codebase for distributed rendering, extending it in the way of having not only one render client running an application, but to permit multiple render clients running the same application in parallel. Thus every render client has only to render a piece of the tiled display submitting render commands to a few render servers instead of shipping the whole scenery to all servers. This extension leads to a performance speed-up when rendering scenery with millions of triangles or primitives [Chromium]. Application like **March** which extracts and renders an isosurface from a volumetric data set – a typical scientific visualization task – should perform at higher frame rates when rendered in parallel. The dataset of **March** is a  $400^3$  volume, and the corresponding isosurface consists of 1,525,008 triangles. So, WireGL renders **March** at 0.25 frames per second. Rendering **March** with Chromium with multiple render client increases the frame rate significantly.

Chromium represents another proof that running multiple application instances in a tiled display improve overall performance.

### Virtual displays

Another design is to replace the normal display with a virtual display as supported by most modern OS. While running an application on the control node and drawing its content on a virtual display, the virtual display driver will send the content to the appropriate display nodes to show the output on the tiled display. The benefit of this approach is to capture display content on its lowest level, so no one has to care about which graphics API is used to draw the content or whether display content is accessed directly drawing pixels into the virtual display frame buffer. The solution's drawback is its bad scalability and the massive impact on the network bandwidth if scenery content is almost drawn pixel by pixel. While using a 3D graphics API to draw the content saves many instructions that have to be shipped in the virtual display approach. Think of drawing a triangle on the screen, with a virtual display every pixel with its color will be sent over the network in this approach. In the former approach only a few primitive attributes will be sent and the pixels will be drawn by the display nodes during rasterization.

At Princeton [Chen et al.00b] enhanced the virtual display driver (VDD) and the OpenGL driver with new versions of the OS bounded DLLs. The new DLLs intercept the API's commands on the application executing machine and send them over the network to the display nodes. By replacing the VDD driver they are able to render the Windows desktop on their tiled display without changing any code of Windows desktop programs. With the replacement of the OpenGL-DLL they rendered different OpenGL applica-

tions without changing application code.

### 3.4.3 Pixel distribution

[Chen et al.00a] developed a method to display ultra-high-resolution videos with resolutions matching the resolution of a tiled display. In their approach they designed a parallel MPEG2-video decoder for a PC cluster based tiled display. Thus the render cluster itself driving the tiled display can be build up of low performance hardware PC, because only pixels transmitted as image parts for each tile in the display have to be unpacked and displayed on the display nodes. So this approach is a good example for pixel distribution leaving the 'render' costs on the MPEG2 decoder nodes.

“In a parallel MPEG-2 video decoder for PC cluster based tiled display wall systems, three major components work together. A splitter divides the input stream into small work units and sends them to the decoders. The decoders might need to communicate with each other to decode a picture. Finally, the decoded pixels might be redistributed before being displayed. There are two challenges in successfully building such a system: high performance and scalability. The system should be able to play ultra-high-resolution videos at an interactive frame rate while keeping the communication requirement at a minimum such that an off-the-shelf network can be used.” [Chen et al.00a]

Another approach for distributing pixels was investigated at Princeton by [Samanta99]. They sought for a balanced render algorithm to balance the render load evenly over all display nodes. Each node renders not only parts that belong to his tile in the tiled display, but also parts for other nodes. After each node has rendered its parts, pixels that do not belong to the node's tile must be redistributed to the right render node to display the scenery correctly. Although this technique is very sophisticated and exploit the cluster's performance the best, the tightrope walk is to keep pixel and render distribution low while frame rates should be high.

## 3.5 Display synchronization

There are many synchronization models that come to mind. We can distinguish mainly between two concepts, hardware synchronization, and software synchronization. Furthermore we divide synchronization models into

four parts, *application-level*, *system-level*, *frame buffer switch* synchronization, and *refresh rate*, or *vertical blank* synchronization known as *genlocking*. All except genlocking can be used to synchronize program execution to provide parallel execution of the same application on the different display nodes to improve overall performance and render times.

The first two synchronization models directly synchronize the applications code execution. They allow to run multiple application instances within the render cluster, as described in section 3.4. Although genlocking can be used for another task important to tiled displays (see description below), it cannot or only in a limited way be used to synchronize program execution. If the render node limits its render time to fit into one frame cycle application execution synchronization can be achieved with genlocking techniques. In any other case a second synchronization model must be used to achieve coherent rendering content.

We will now have a deeper look on the latter two synchronization models and their possible realizations in hardware, or software.

### 3.5.1 Frame buffer switch synchronization

Although this synchronization model is very similar to the system-level synchronization, we mention it here, because it has a special purpose regarding rendering. With parallel rendering on every render node we want to synchronize the rendering result of every node. A common task in rendering is a frame buffer switch, i.e. when rendering is done the current rendered scene has to be displayed. When a frame buffer switch occurs the back buffer with the current rendered content must be displayed becoming the front buffer and vice versa the front buffer must be hidden to render a new frame. But in a synchronized model, before a render node is allowed to switch its buffers, it has to wait until all display nodes within the render cluster have finished their actual frame and are ready to display the new content. Thus, this approach is easy to implement, capturing the OS-`switchBuffer()` call to implement this behavior.

[[ChannelSync](#)] offers a low-cost solution for frame buffer switch synchronization with a low-cost network.

### 3.5.2 Refresh rate synchronization

This is the most advanced model of synchronization, because with refresh rate synchronization we are able to do *active stereo* on the one hand. On the other hand it also provides the most visual accuracy concerning frame buffer switches or refreshes. Think of viewing a high-resolution video with

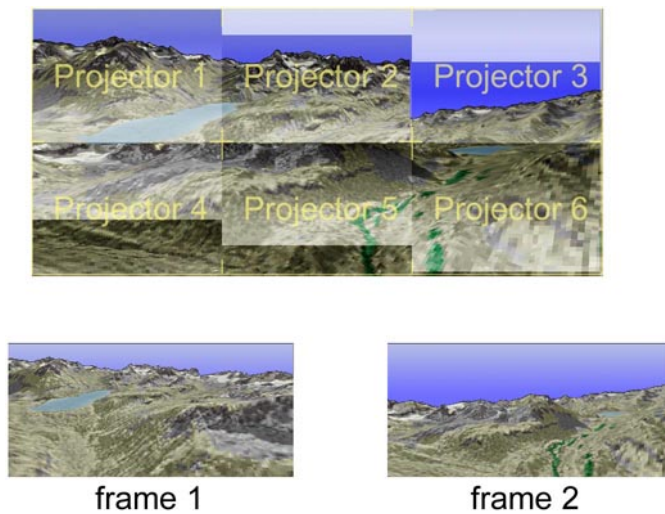


Figure 3.10: Unsynchronized tiles in a tiled display lead to visible differences. Projectors 1&2 show frame 1, while projectors 3-6 show frame 2. Refresh rays are indicated as brighter regions in the images.

many short scenes, so that scene cuts appear every few seconds. With refresh rate synchronization you are not able to distinguish between each individual display, because every display refreshes its content together. While only synchronizing buffer switches, i.e. showing the content of the next video frame, one display could have been half finished while the other is just beginning to refresh its content. So, slight differences can be visible in the content of the whole tiled display and this would be even more true when a scene cut arises. Figure 3.10 depicts this situation. Using refresh rate synchronization visible differences are avoided and all displays starting to draw a new frame at the same time, see figure 3.11.

This model can be solved either in hardware or software. While the software solution is time-critical and needs to be supported by the OS to have exact time slices for synchronizing the refresh rate without being interrupted by other threads even OS ones, hardware supported genlocking is free from OS-dependencies and able to synchronize the refresh rates externally. With genlocking on board, one need not be concerned with the implementation at all.

There exists only one *software* solution and fortunately it is open source [[SoftGenLock](#)]. So it is possible to look behind the scenes and see what the researchers have done to make this possible. They used a real-time OS (real-time Linux) to synchronize the different graphics cards. With a

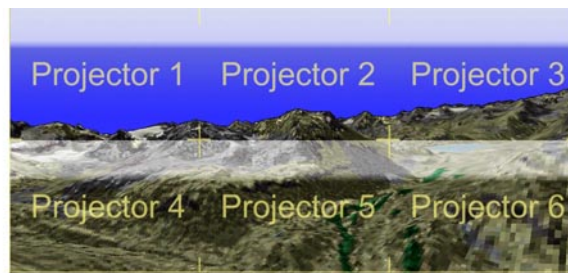


Figure 3.11: Synchronizing the refresh rates of all tiles avoid visible differences since all displays start a new frame at the same time. Refresh rays are indicated as brighter regions in the images.

dedicated network driven via the parallel port they are able to synchronize up to 6 display nodes within  $6 \mu\text{s}$  providing a fast synchronization but with the lack of scalability. Their main idea is to prohibit the vertical blank signal on every graphics card until all cards reaches the end of their frame buffers. Then all cards together are forced to send the vertical blank signal starting at the same time to refresh their displays. A clear and obvious approach, but with the disadvantage to have direct access to the video card registers to be able to delay the refresh signals. Thus, one needs to know the exact addresses of the needed graphics card registers and is not allowed to program in a high-level language due to time-critical computations.

The *hardware* implementation uses a similar ability of some graphics card to suppress the refresh signal. Also a dedicated cable is used on the output of the graphics cards to detect and prune the refresh signal until all graphics card have finished their frame output. With cascading the synchronization hardware this implementation approach is scalable. Researches from Ars Electronica Futurelab have implemented this approach [[ArsBox](#)].

## 3.6 Open Inventor

Open Inventor [[Wernecke94](#)] is an object-oriented toolkit for the development of interactive 3D graphics applications. It is the base of the Studierstube API, because of its profound design and philosophy of extensibility.

### Scene Graph

The database that represents a virtual scene in Open Inventor is called scene graph. The scene graph is a directed acyclic graph and rooted by a single

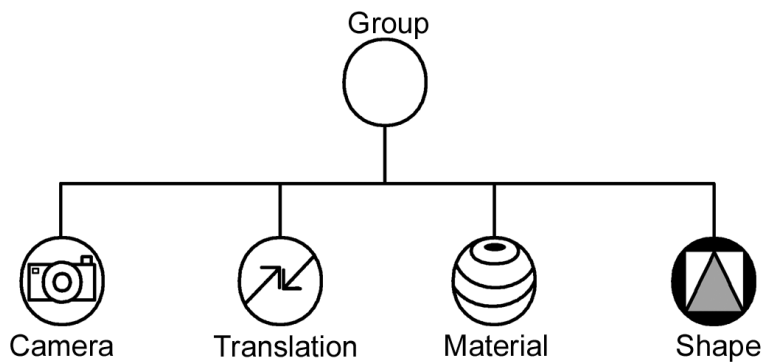


Figure 3.12: A simple scene graph.

node. There exists many different nodes with dedicated behavior to display the scene, handle user input, and change the scene content based on some calculations or simulations. A single node can hold a collection of information. We will list a few built-in nodes:

- *Group nodes* structure the scene graph. They serve as containers holding several nodes or assembling several subgraphs. They are the only nodes within the scene graph, while all other nodes forming the leaves of the graph.
- *Attribute nodes* change the way subsequent shape nodes are rendered. Material nodes for instance change the appearance of a scene object's surface, transformation nodes alter a scene object's scale, its position or its alignment.
- *Camera and light nodes* represents the position and orientation of cameras and lights to view and illuminate the scenery.
- *Shape nodes* as their name suggests form the base for a geometric description of real objects. There are built-in shapes, like cubes, spheres or cylinders, and there are more complex ones like face sets that allow the application developer to define arbitrary shapes described by a set of triangles.

Figure 3.12 shows a scene graph that uses a group node that contains the whole scene and roots the scene graph. A camera node is used to make the scenery viewable. The translation node is used to alter the position of the virtual object described by the two following nodes. The material node sets up surface characteristics. It changes the current render attributes for all



following shape nodes. This affects the last node, a shape node that describes the appearance of the object.

Open Inventor's major strength is its extensibility. The application developer can implement new nodes that can be inserted into the scene graph. Open Inventor's library provides macros and helper functions that aid the developer to create new nodes.

## Actions and events

The database of Open Inventor is traversed by either an action or an event. Both call a distinct function of a node when the node is traversed. Actions and events are a common design pattern, known as "visitor pattern". [Gamma et al.94] describes what a visitor should do: "... represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates."

Actions (and events) can be considered as a design decision by the Open Inventor developers to ensure far-reaching extensibility. Let's assume we want to extend the functionality of a hierarchy of classes, like the nodes of Open Inventor. In terms of extension in an object oriented fashion, like in C++, we have to implement the functionality for each class adding a new method - in the case we can access the source code. If we cannot, subclassing each class and extend it the new with the desired method is the only way in C++. This is an error-prone and unreasonable process if the number of classes increases and the level of complexity is high.

Inventor employs actions to solve this problem. An action maintains a list of static methods, one for each node class. When such an action is applied to the root of the scene graph, this list is used to determine the method to call for the particular node that is just traversed [Wernecke et al.94] Actions practically encapsulate methods as a class of their own. The action of interest for us is the *SoGLRenderAction*. This action traverses the scene graph calling the appropriate function of each node encountered. In order to define the action, Open Inventor must be provided a function for every node type. *Group nodes* successively pass the action to its children starting with the first one. *Attribute and light nodes* change the current render state, i.e. the way all subsequent shape nodes will be rendered. *Camera nodes* set the viewing point of the scene, changing the location of the global coordinate system. *Shape nodes* will render the objects they are describing.

To render and view a scenery, we must merely apply a *SoGLRenderAction* to the scene graph's root. The action concept will cause the run-time system to traverse the scene graph and to render the database as desired.



## 3.7 Distributed Open Inventor and Studierstube

In this section we investigate the two concepts of virtual reality operating systems and shared data for interactive 3D graphics.

### Distributed Open Inventor

Distributed Open Inventor [Hesina et al.99] is an extension to the popular Open Inventor toolkit for interactive 3D graphics. The toolkit is extended with the concept of a distributed shared scene graph, similar to distributed shared memory. From the application programmer's perspective, multiple workstations share a common scene graph. The proposed system introduces a convenient mechanism for writing distributed graphical applications based on a popular tool in an almost transparent manner. Local variations in the scene graph allow for a wide range of possible applications, and local low latency interaction mechanisms called input streams enable high performance while saving the programmer from network peculiarities. Although Distributed Open Inventor intended use is a collaborative setup for multiple users viewing a shared scene graph, this approach is a good candidate for a tiled display since it provides all needed distribution techniques vital for a tiled display.

### Studierstube

The Studierstube framework is an operating system for applications in virtual reality. It supports multiple users collaborating in a distributed system. While initially developed for scientific visualization, the generic approach supports almost any kind of VR application on a wide range of VR-hardware. It is based on the Distributed Open Inventor which itself is an extension of the Open Inventor in the sense of a distributed shared scene graph. The Studierstube framework consist of inherent concepts, like real-time graphics, high-level interaction methods, multi-user integration, and distributed execution. Thus making this framework a ready-to-use system to build up a tiled display for virtual reality.

With the Studierstube framework we have not only a well working rendering system for VR-applications supporting multiple users and parallel application execution, but also the demanded shared memory, respectively shared scene graph concept mandatory for a tiled display. With the underling Open Inventor API it is easy to extend for rendering on a tiled display.



# Chapter 4

## Design issues

In this chapter we present our solution to build up a scalable tiled display with commodity PC and graphics hardware and low-cost projectors.

### 4.1 Calibration

We have implemented the approach taken by Raskar et al. They calibrate oblique projectors with image detection techniques, and compute collineation matrices between projector space, camera space and display space, also named homographies. With these homographies it is easy to generate a projection matrix for each display that can be used as a pre-transformation matrix in the render step. It undistorts oblique projection from roughly positioned and aligned projectors. The technique describe by [Raskar et al.02] uses structured patterns projected by each projector to extract feature points and finds the corresponding collineation between camera and projector pixels. With this collineation it is easy to compute projector to projector and display to projector homographies. The projector to projector homographies are used to compute the needed blending masks for intensity weighting in overlapping regions.

During rendering the projector to display homographies can be used as pre-transformation matrices to undistort oblique projection. Blending mask using alpha blending techniques are used as a post-rendering step to blend multiple pixels in overlapping regions. So the scenery will be corrected, rendered, and blended in a single-pass saving a lot of computation time (see section 3.1.3 for a brief description).

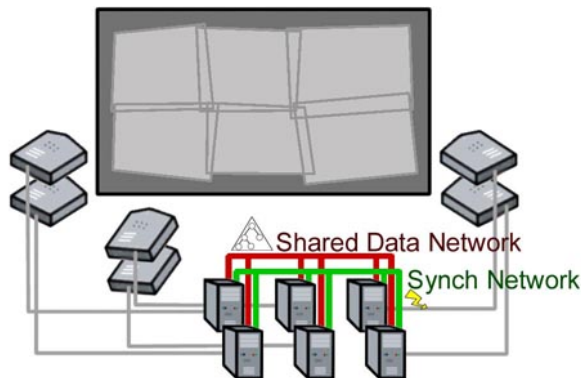


Figure 4.1: Two dedicated networks to separate shared scene data (red) from synchronization events (green).

## 4.2 Cluster architecture

As mentioned in chapter 3, there are many possibilities to build up a render cluster. To keep the costs low we decided for the off-the-shelf PC hardware solution using dual processor PCs with a commodity graphics accelerator to drive the tiled display. Each computer drives two displays from two outputs and is able to do quad buffering for active stereo displays.

To be able to build a render cluster to share data and synchronize displays, we need to connect the PCs within a network. We had to choose between normal 100 Mbps Fast Ethernet and other high-speed networks, like Myrinet. The latter one was the better in the term of high transfer rate which are recommendatory for graphic applications. But we did not take any high-speed network cards which are designed to send a Giga bit per second of data. First, because of high price and second, because our Studierstube framework performs well with normal ethernet cards since data sharing and replication between the display nodes is not done very often. Two 100T-Ethernet network cards were put in each computer creating two dedicated networks. The reason for two network cards per computer is that one network is used for data sharing and the other for synchronization. So interference of both is avoided to guarantee full bandwidth when data is shared and synchronization points are sent (see figure 4.1).

We use high performance computers equipped with dual processors and one graphics accelerator each, as mentioned above. To run the computers as a sharing scene graph cluster we use our Studierstube framework. This allows us to create a render cluster with one control node running as master to collect user data, like tracking data, mouse motion. The other nodes

within the cluster are display nodes running as slaves. They process all other tasks - executing application code, rendering, and displaying the scenery on the tiled display. With this approach a copy of our Studierstube framework runs on each display node concurrently. In terms of tiled displays types we are running multiple copies of application codes on the different display node and since we are also sometimes using both graphics cards output channels, we run two copies of the code on each display node. So we have to take care about a synchronized execution of code on every single display node. We used system-level synchronization with a simple frame buffer switch synchronization. The application code halts when ever a buffer switch should occur. When all display nodes are ready to switch their frame buffer content, the application will be notified to continue code execution. Figure 4.2 shows a time diagram and how every copy of the application is treated when synchronization points are met.

### 4.3 Data distribution

The Studierstube framework already delivers data distribution. Since this framework is build on top of Open Inventor its data distribution strategies are tightly coupled with node creation and destruction, and event generation. The distributed Open Inventor extension provides easily manageable data distribution (see section 3.7). This extension is build in a master-slave scheme, where one master holds the master copy of the scene graph and delivers its slaves with updated copies of the scene graph as soon as any content of the scene graph is changed. So in our solution we choose the synchronized execution model where a master synchronize all its slaves to render the correct view of the scene. We take a lazy synchronization model between the application driven by the slaves using frame buffer switch synchronization. So the application's execution path is stopped by each slave, when its scene part is completely rendered. After all slaves have finished rendering their scenes, the frame buffer switch occurs and implicitly synchronizes all slaves' execution paths.

### 4.4 Display synchronization

Since we want to enhance our existing Studierstube framework, it is the easiest way to enhance this framework with a synchronization scheme. Studierstube is build on top of Open Inventor which is a 3D scene graph API allowing to build complex scenery by describing the scene with a scene graph. The

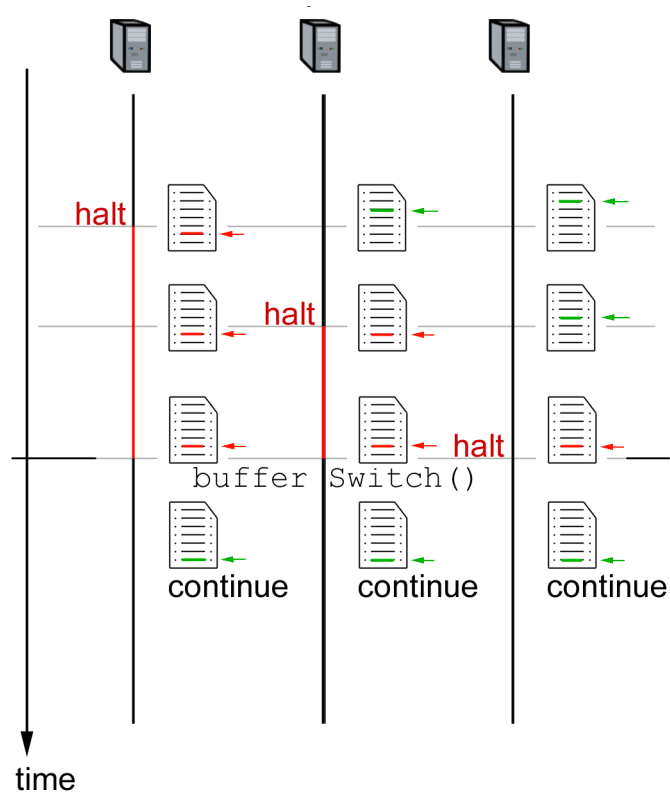


Figure 4.2: A time diagram showing system-level synchronization of multiple copies of a single-threaded application. After each buffer switch the code's execution path is synchronized again.

scene graph contains different nodes, some representing geometry, transformation, or grouping sub graphs, see section 3.6 for a brief description of Open Inventor. To synchronize the result - a rendered frame - we create a new node and add it at the end of each display node's scene graph which synchronize the render result of all display nodes. Since Open Inventor traverses its scene graph to render the scenery describe by the scene graph, it is the easiest way to do synchronization within the Open Inventor traversal philosophy. While appending this new node at the end of the scene graph, it will trigger its synchronization method after all scenery was rendered. When one of these nodes is traversed during rendering, it notifies a render control server that it has finished rendering. After all display nodes have sent their signals to the server, the render control server then notifies each node to swap the buffers allowing them to show up the new content. So we achieve a uniform content switch (new frame content) over the whole tiled display. We described this technique in section 3.5 as the frame buffer switch synchronization.

But we have to mention that when synchronizing via the frame buffer switch, we have to take care of application synchronization as a second step, too. So we have to ensure that user inputs and application data changes, so called application events, are synchronized on each display node when a buffer switch occurs. Since we can not guarantee that events are delivered to the display nodes in the time-order they occurred on the control node, we have to divide these events into two parts. First, all events that occurred before the buffer switch have to be distributed to all display nodes before the buffer switch takes place. Second all events occurred after the buffer switch compared to the control node's execution path at that time must be retarded until the next buffer switch occurs. Retarding the events after finishing rendering can be done on the master side or the slave side, i.e. on the control node or the display nodes. If this is done on the display nodes, each display node has to collect the events occurred the first display node has finished rendering and not to trigger them until all display nodes are allow to switch the frame buffer. Figure 4.3 shows the scenario where events are collected on the slave side by the display nodes and are retarded until the next buffer switch occurs. When the first display node has finished rendering all subsequent events have to be retarded and stored on all display nodes, until every display node continues program execution.

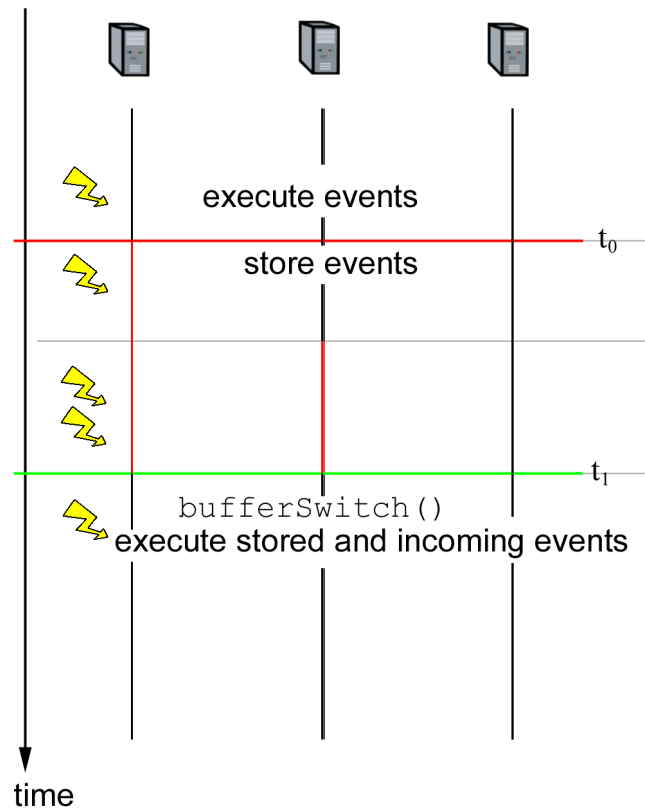


Figure 4.3: A time diagram showing event retarding on each display node. The control node sends events via multicast (flashes), so that every event is distributed to all display nodes. After the first display node has finished rendering (at  $t_0$ ), all subsequent events have to be stored on each display node until a buffer switch occurs (at  $t_1$ ).



# Chapter 5

## Implementation issues

This chapter describes the taken steps to fulfill our needs to drive the tiled display, keeping in mind scalability, maintenance, and performance.

### 5.1 Calibration

Since we want to add and remove displays from the tiled display as needed, we implemented a calibration software that support any number of displays to calibrate. We chose a server-client model to maintain displays and cameras and to run all parts of our calibration software on different nodes within a network. A server will handle all clients that registers at it. It also handles and directs messages to the right clients. See chapter 10 for a detailed description of the implementation of the server and its clients. Each client represents one of the three groups, a user, a projector or display, or a camera. The user client processes user inputs and invokes the different calibration steps just on the user's demand. A projector client represents a display within the render cluster, without any restriction of size, brightness, or location. Thus allowing a display node to drive more than one display at a time. The camera client handles the video stream from a camera and performs the computations for the needed projection matrices and blending masks using [Raskar et al.02]'s approach with building homographies. The homographies are extracted using OpenCV's pattern recognition techniques. A simple chessboard pattern is projected for each display and captured by the camera. The feature points of the pattern are extracted using OpenCV's `FindChessBoardCornerGuesses()` function and the homographies are compute with OpenCV's `FindHomography()` function. Figure 5.1 shows the found feature points during a calibration step. These homographies are extended to  $4 \times 4$  matrices to be usable in a common graphics pipeline. After

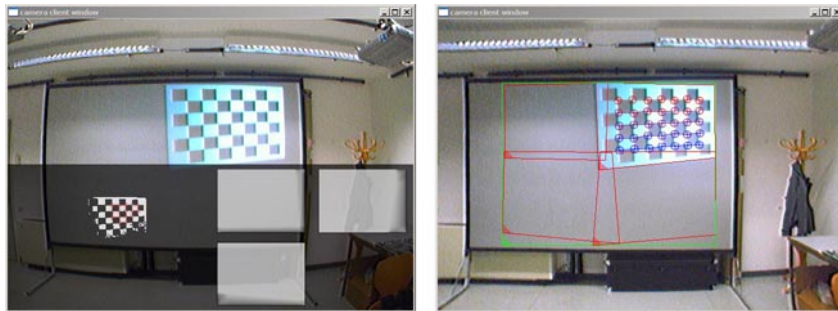


Figure 5.1: Detecting feature points of a projected chessboard pattern during calibration (left). All feature points are detected and the convex hull of each calibrated projector is shown in red (right).

the extension the matrices have to be corrected to approximate depth-buffer values as described in [Raskar00, Raskar99] due to depth value distortion when the homographies are applied to common view frustum or projection matrices. A detail description on homography and blending mask computations and how to approximate the distorted depth values can be found in section 3.1.3). This approach scales well and maintenance is low if the installed system is not changed very often, i.e. if displays that belongs to a node are not removed or added. So if a render cluster is once installed and only the displays are rearranged, the only step done is to run the calibration software parts and recompute the projection matrices and blending masks. If a display, or camera is removed or added to the render cluster, a short configuration file must be written to describe the parameters of the display, or camera respectively. See chapter 9 for a description of the configuration files.

Since we want to calibrate a tiled display and use it within our Studierstube framework, our calibration software can not only be used to calibrate the tiled display, but also to generate Studierstube specific files to run the Studierstube after the calibration step (see section 5.2.3).

## 5.2 Studierstube

### 5.2.1 Data distribution

A distributed version of Open Inventor was developed previously, so that data distribution is not an issue of this thesis. But we will shortly describe how it works. Since Studierstube is based on Open Inventor, data distribution is

based on Open Inventor's scene description philosophy. A scene is described by a scene graph with various kinds of nodes. A node can contain geometry, do transformations, bundle a group of other nodes, or act with a specific behavior when it is visited by an action or an event. The run-time system of Open Inventor takes care of event, action, and even node generation, so it is easy to catch modifications to the scenegraph and distribute them to all slaves. If a node is created or removed from the master's scene graph, it is also created or removed in all slaves' scene graphs. Any action or event that occurs on the master host will be serialized and transmitted to the slaves. At the slaves the same event will be generated to guarantee equal behavior on all machines.

This also includes user input, since every input from the user is converted into an Open Inventor event. This event then visits all nodes in the scene graph. So equal scene nodes on the different cluster nodes show the same event specific behavior.

The replication of node or subgraph generation and destruction and the serialization of events saves a lot of network traffic since only changes in the shared scene graph and occurring events are transmitted. In other approaches (like primitive distribution) every time a frame is rendered all scenery data is transmitted to the display nodes. In our implementation we chose the synchronized execution approach. This approach scales very well, since the main network traffic are user, application and synchronization events (which are mostly very short message packets less than one KB). Scene graph data is only transmitted when a part of the scene is changed. Although the intended use of the Distributed Open Inventor was a collaborative setup for multiple users, it fits perfectly to build a tiled display. For each tile within the tiled displays we setup a "user" that only watches, and renders the scene without any interaction. So we can easily implement the semantics of a tiled display without implementing extra code to support multiple displays in our framework.

### 5.2.2 Display synchronization

To synchronize the display content of all displays within the tiled display, we decided to take a synchronization point when a render node has finished rendering. Thus we can easily synchronize the application's execution path, since it will stop as soon as the synchronization point is reached, and synchronize the display content at the same time. It is the advantage of Open Inventor's visiting pattern philosophy that application execution synchronization is so easily achieved. In Open Inventor events and actions visit all nodes in the scene graph invoking a node specific behavior. The Open Inventor action

that is responsible for rendering is a `SoGLRenderAction`. This action visit all nodes in the scene graph during rendering invoking the nodes' rendering semantics. We implemented one new node that when visited by this action can synchronize the display nodes within the render cluster. To achieve synchronization we first implemented the abstract base class *SoRenderSync*. Based on this class the actual synchronization node, reacts only when it is visited by a `SoGLRenderAction`, thus all other actions are ignored and not delayed by this node. So during rendering the code execution on a display host within the render cluster is suspended after a `SoGLRenderAction` triggered the *SoRenderSync* subclass' action behavior. As soon as all display hosts have finished the rendering process and performed their render action behavior, they leave the suspended mode and continue code execution. This also leads to a application execution synchronization, because Open Inventor will collect all events that arrives during the suspended process and will trigger them just before the next `SoGLRenderAction` takes place.

As mentioned we developed a node to react when a `SoGLRenderAction` traverses the scene graph. As a base for synchronization nodes we implemented the *SoRenderSync* class as an abstract class to provide a simple thread generation to run the server thread. A virtual function is provided to implement the server's thread behavior saving the developer from thread generation peculiarities. All subclasses of this node synchronize several display nodes within the render cluster when a `SoGLRenderAction` arrives at the subclass node (see the following section for details).

We have implemented the *SoRenderSyncUDP* class as a subclass. This class performs synchronization using an UDP server and a multicast group where the clients joins. When a render action passes this node, the client sends a packet to inform the server that it has finished rendering. As soon as all clients have sent their packets to the server, the server will sent a broadcast packet to the multicast group to allow the clients to switch to the next frame and continue their code execution.

Each display node that should be synchronized should have a *SoRenderSyncUDP* node at the end of its scene graph. Any one of the display nodes acts as server and client at the same time to synchronize all other display nodes, respectively clients. This special node starts a server thread to process its clients. As by design it is also allowed to run its client unsynchronized from the rest of the clients. We chose this design to be able to use one of the stations within the render cluster as a monitoring rendering machine. This is important when large and complex scenes are viewed which can not be rendered at interactive frame rates. The monitoring machine is used to show a draft of the scene which renders much faster than the actual output on the display wall. Think of a view of an isosurface mesh or a complex simulation

---

```
typedef struct
{
    unsigned long   frameNumber;
    unsigned short  clientNumber;
    unsigned short  retransmit;
} Packet;
```

---

Table 5.1: The message packet transmitted between the synchronization server and its clients.

output where millions of triangles have to be rendered on the tiled display. When rendering a low resolution model on the monitoring node, we are able to render the scene in real time on this node since it is not synchronized with the other render nodes. The tiled display renders a high-detail model at non-interactive frame rates instead. This allows the user to interactively change the point of view or change other elements (like simulation constraints) on the monitoring node, while watching a high detail model on the tiled display.

### Code fragments and message protocols

Now we will look at the code and the messages we have implemented to synchronize the render cluster. A simple packet is used to perform communication between the synchronization server and its clients. The packet holds information about the rendered frame and the clients. The packet is eight bytes long and contains the frame number that was actually rendered by the client. Each client must have a unique number which is also transmitted to the server. An additional field is used when a client did not meet the synchronization constraints. The packet is a `C-struct` to easily access the information (see table 5.1). An second `C-struct` was defined to maintain the clients and can be used by the server thread to log clients' statistics (see table 5.2).

We implemented the abstract base class *SoRenderSync* to support thread generation. This class has one virtual function `void serverCode(void)` that must be overwritten to support sever semantics. Since this class is derived from an Open Inventor node it also has the virtual function `void GLRender(void)`. Reimplementing this function, the synchronization functionality of each single render client can be achieved. We will have a look at some parts of the *SoRenderSync* class to show how it works and how it can be extended by subclassing.

---

```

typedef struct
{
    unsigned long joinedAtFrame;
    unsigned long framesDropped;
    unsigned long frameNumber;
    bool         active;
} ClientStatistic;

```

---

Table 5.2: The ClientStatistic struct to store information about clients within the server thread.

```

/** Number of client (setting to -1 forces this node to act as
 * server). */
SoSFInt32  clientNum;

/** Determines synchronous/asynchronous rendering for server
 * nodes. Has no effect on client nodes' rendering.*/
SoSFBool   synchronized;

```

As a second step we have to subclass the SoRenderSync class. The *SoRenderSyncUDP* class derived from SoRenderSync uses an UDP server collecting packets from its clients. In the other direction each client joins a multicast group used by the server to send a reply packet that informs the clients to switch their frame buffer and to continue code execution.

We present here a short guide how to subclass the SoRenderSync in three steps:

*First* subclass the SoRenderSync class.

```
class STBAPI_API SoRenderSyncUDP: public SoRenderSync {
```

*Second* add some fields to specify sever and multicast addresses, or whatever fields are needed to build synchronization server-client model.

...

```
public:
```

```

    /** Multicast address and port clients join. */
    SoSFString multicastAddress;
    SoSFInt32  multicastPort;

```

```

    /** Interface address for outgoing client messages. */
    SoSFString interfaceAddress;

```

```

    /** Server address and port all clients will send a
        * message to when this node is rendered. */
    SoSFString serverAddress;
    SoSFInt32  serverPort;
    ...

```

*Third* overwrite the following two virtual functions to implement synchronization semantics and make synchronization work.

```

public:
    virtual void GLRender(SoGLRenderAction *action);
    ...

protected:
    virtual void serverCode(void);
    ...

```

When the render function `GLRender()` is invoked, the client sends its packet with its unique client number and actually rendered frame number to the server and wait for the server's response. This in turn suspends the client from executing code. The server collects all packets from its clients and compare their frame number with its own predicted frame number. If any client does not send its package within a certain time interval, the server exclude this client from the render cluster and sends the frame switch message, a packet with a client number of -1 and the actual frame number. If a client fails it can join the server at any time and gets synchronized again. If the server fails all clients reset their frame number and try to connect to the server. This gives us the possibility to remove and add nodes at any time, even the server code executing node.

The `serverCode()` function is invoked once the server thread starts. It runs and processes its clients requests infinitely.

### 5.2.3 Tiled Display

With the Studierstube API and its underlying Distributed Open Inventor API, we can easily build up a tiled display since it provides all needed methods to support shared scene data and event serialization. In section 3.1.3 we talked about rough overlap displays. These display are formed by oblique projectors, so we have to correct the display scenery of each projector in respect to the user's view. The computed projection matrices and blending

masks are bound into the scene graph of each display to form a seamless tiled display. We extended the Studierstube API by some classes to be able to render to our tiled display. The *SoAlphaMaskKit* class is responsible to blend pixels in overlapping regions of adjoining projectors, so that these pixels are as bright as pixels in non-overlapping regions. It contains a field `texture` where the blending mask can be stored. Our calibration software will build a *SoAlphaMaskKit* for each projector as an output of the calibration step. The *SoOffAxisCamera* class was extended by a field called `matrix`. It stores a collineation matrix (homography) for the specific tile to correct the scenery of the scene graph the camera will render. This matrix maps pixels from projector coordinate frame to pixels in display coordinate frame. To be useable for visibility culling and clipping, it also corrects depth-buffer values which will be distorted by the collineation. This matrix is set by the calibration software for each projector and stored in a common *UserKit* to be useable in the Studierstube environment. An additional class, the *SoStuberenaKit*, was developed as a startup application when the Studierstube system is launched. It has a *SoSeparator* field called *postScene* that can contain any subgraph. As the name of the field suggests this subgraph is attached at the very end of the scene graph to a local part of the graph that is not distributed. In the most cases this class is used to contain synchronization nodes. This is crucial, because distributing a synchronization node to the shared scene graph will lead to non predictable result. Each display node will start a *SoStuberenaKit* which contains a subclass of *SoRenderSync* mentioned above for synchronization and a *SoAlphaMaskKit* for blending overlapping regions. Since the *SoStuberenaKit* is an application, we do not want to run it forever and waste resources of Studierstube. After the subgraph of this kit is linked to the unshared part of each node's scene graph, the *SoStuberenaKit* application is closed leaving a blank Studierstube framework ready to work.

### 5.2.4 Summary

We extended the Studierstube API by five classes to be able to render and display Studierstube applications to our tiled display. These five classes are:

- The *SoRenderSync* class is an abstract base class all synchronization nodes working within the StubeRenA tiled display should be derived from.
- The *SoRenderSyncUDP* class is responsible to synchronize the display content (see table 5.3).



- The *SoAlphaMaskKit* class is responsible to blend pixels of adjoining projectors (see table 5.4).
- The *SoOffAxisCamera* class was extended by the field `matrix`, which corrects the oblique projection of its projector's area (see table 5.5).
- The *SoStuberenaKit* was developed to link the synchronization nodes to the part of the local scene graph copy on the display machines which will not be distributed and of course will be at the end of the whole scene graph (see table 5.3).

```

#Inventor V2.1 ascii

DEF STBERENA SoApplicationKit {
  readOnly TRUE
  classLoader SoClassLoader {
    className "SoStuberenaKit"
    fileName  "../apps/stuberena/stuberena_stb"
  }

  contextKit DEF stuberena SoStuberenaKit {
    clonePipSheet FALSE

    # !IMPORTANT! Do not remove the following lines.
    # The calibration program will not set these values if the lines are removed.

    postScene SoSeparator {
      File { name "//Dartagnan/Documents/Overlap/renaMask[00].iv" }
      SoRenderSyncUDP {
        multicastAddress "224.0.0.1"
        serverAddress "128.131.167.145"
        clientNum 0
      }
    }
  }

  appGeom Separator {
    Texture2 { filename "../apps/stuberena/stuberena.gif" }
  }

  info Info { }
}

#
# renaStartKit[00].iv generated automatically: Fri Aug 29 10:57:45 2003
#
# StubeRenA calibration software (c) 2003 Vienna University of Technologie
#

```

Table 5.3: The SoStuberenaKit to start a tile within the render cluster and append a SoAlphaMaskKit and a SoRenderSyncUDP node at the end of the local scene graph.

```

#Inventor V2.1 ascii
SoAlphaMaskKit {
  texture SoTexture2 {
    wrapT CLAMP
    wrapS CLAMP
    model MODULATE
    image
      128 128 4
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0x4b 0x5e 0x6b 0x75 0x7c 0x83 0x88 0x8c 0x90 0x93 0x93 0x94 0x94 0x94 0x94
      0x94 0x94 0x94 0x94 0x94 0x94 0x95 0x95 0x95 0x95 0x95 0x95 0x95 0x95 0x95
      0x95 0x95 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x96 0x97 0x97
      0x97 0x97 0x97 0x97 0x97 0x97 0x97 0x97 0x97 0x97 0x98 0x98 0x98 0x98 0x98
      ...
      ...
      ...
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x1c 0x2f 0x3d 0x47 0x4d 0x69 0xff
      0xff 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
      0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
  }
}

#
# renaMask[00].iv generated automatically: Fri Aug 29 10:57:45 2003
#
# StubeRena calibration software (c) 2003 Vienna University of Technologie
#

```

Table 5.4: The SoAlphaMaskKit that is loaded from a SoStuberenaKit application. A value of 0xff fades the corresponding pixel black and a value of 0x00 means that the pixel is illuminated at full brightness.

```

#Inventor V2.1 ascii

SoUserKit {
  # !IMPORTANT! Do not remove the following line.
  # The calibration program will not set this value if the line is removed.

  userID 10

  display SoDisplayKit {
    station 2

    # !IMPORTANT! Do not remove the following lines.
    # The calibration program will not set these values if the lines are removed.

    stereoCameraKit SoStereoCameraKit {
      camLeft SoOffAxisCamera {
        matrix 1.01263334 0.00690716901 0 0.0224769952
              -0.013100665 1.02769573 0 -0.0135801814
              0 0 0.636612232 0
              1.01919947 -1.0478436 0 1
        viewportMapping ADJUST_CAMERA
        nearDistance 0.01
        farDistance 100
        focalDistance 2
        orientation 0 1 0 0
        size 0.35 0.27
        eyepointPosition 0 0 0.5
      }
    }

    cameraControl SoTrackedDisplayControlMode {} # used to hide examiner viewer buttons
    headlight TRUE
    headlightIntensity 1.0
    backgroundColor 0 0 0
    transparencyType BLEND
    windowBorder FALSE
    showMouse FALSE

    # !IMPORTANT! Do not remove the following lines.
    # The calibration program will not set these values if the lines are removed.

    xoffset 0
    yoffset 0
    width 1024
    height 768
  }
}

#
# Projector 00's display mode: mono
#

#
# renaUserKit[00].iv generated automatically: Fri Aug 29 10:57:45 2003
#
# StubeRena calibration software (c) 2003 Vienna University of Technologie
#

```

Table 5.5: A common user kit to run a tile as a passive user to render the scene.

# Chapter 6

## Results

In this chapter we present our results and show how our system performs. We compare rendering results of distributed rendering and rendering on a single computer. We run multiple copies of our Studierstube environment within the render cluster and synchronize the application's execution path with system-level synchronization. This as describe in chapter 3 is a viable way to synchronize the application as long as they run single-threaded. If the Studierstube framework is supporting multi-threaded execution in the future, then this synchronization model only works if only one single thread interacts with the environment while the other threads compute the scenes without affecting the internal states of the application.

### 6.1 Performance

Performance is the vital part of every virtual or augmented reality system. The system should perform at interactive frame rates to guarantee the illusion for the user that every action he takes directly manipulates his environment. This is important to mimic realistic behavior since the user's experience in real life is that every action performed changes the environment at any time.

Our system performs well under all circumstances and all configurations. We run our system on a single computer as a reference when running it on our tiled display.

With the tiled display system there are two possibilities to perform rendering and object culling. *Tile specific rendering* culls an object that fall totally outside the tiles viewing volume before its primitives are generated and rendered. This should raise the performance of the whole system and should allow interaction in real-time. The overhead made to compute the parts of the scene graph that are, partly or totally, visible and that are invis-


ible at all is negligible compared to the rendering time. *Total scene rendering* on the other hand is the easiest way to let the system render scenes. Already existing applications can be used as they are and tile-specific culling and rendering is performed by the graphics accelerator. But as one can expect there more complex scenes must be rendered the more the frame rate drops. At a certain point when too many primitives must be rendered interaction in real-time is no more guaranteed.

But we must state here, that even under tile specific rendering the frame rate can fall below real-time restrictions. Think of a polygonal model of an isosurface extraction of a 3D volume set. This normally generates over a million of polygons to be rendered. And even if we only render the part of the model that is visible in one tile, we still have to compute hundreds of thousands of polygons. This still is not possible to be rendered in real-time. But with our StubeRenA tiled display we can run one node as a monitoring machine, rendering a low resolution model to make real-time interaction possible for the user. The user then can perform his actions in real-time and see the results on the low resolution model, while the other nodes within the cluster render the model at high-detail on the tiled display. We will now compare and show the results of our StubeRenA framework.

## Non-tile specific rendering

In the first test we take normal Studierstube applications that are not aware of tiled specific rendering. Thus the whole scene will be rendered on each display node and visible parts are culled with view frustum clipping techniques by the render hardware. In this scenario we expect equal frame rates on all machines. And the frame rate compared to the single reference rendering computer should even match with slightly lower frame rates on the tiled display originating from synchronization. We run two to four applications in parallel and moved the application's windows to compare even and uneven render loads on the nodes and the render performances. As expected in a non-tile specific rendering the render times only slightly differ from each another. We choose two setups where each render node drives a single display and one setup where each node drives two displays.

First, we setup a *two-display* tiled display with one master control node and two slave rendering nodes driving the displays. When rendering scenes with less polygonal complexity the slaves are slightly slower than the single rendering machine. This difference comes from synchronization when one node is waiting for the other to refresh the display. If we run one application both systems (tiled display and single computer) show the same frame rate of approximately 30 frames per second. If we run two applications concurrently



	uneven load		even load		heavy load	
render time	53ms	47ms	50ms	50ms	0.1s	0.1s
synch. time	0.1ms	10ms	0.1ms	0.05ms	0.2ms	1ms
frames/s	18 fps		20 fps		10 fps	
fps(single)	21 fps		21 fps		10 fps	

Figure 6.1: Non-tile specific rendering with normal Studierstube applications. A two display setup driven by two nodes with uneven, even, and heavy load balance. In the even load scenario synchronization time is  $\leq 0.1ms$ , in the other scenarios synchronization time is  $\geq 1ms$  which slows down the performance compared to the single rendering computer.


we gain 21 frames per second on the single rendering machine, while the tiled display system renders the scenery in 18 frames per second if the render load is uneven distributed, i.e. only one node of the both renders the scenery to the frame buffer, while the other drops all polygons since they are not visible. In an even load scenario where both nodes has to render to their frame buffers, the frame rate gets in reach of the single rendering machine. If we increase the render load starting some more applications the synchronization boundaries become less noticeable and the frame rates equals on the tiled display and the single computer. Figure 6.1 shows the results in comparison.

In the second setup we build a *three-display* tiled display. The master takes the rule of both a control node and a rendering node. In this scenario the same situation arises like in the two display setup. If the render load is not evenly distributed, i.e. if one node must render the scenery to its frame buffer, while the other two cull all polygons, the frame rates drops again slightly compared to the single rendering computer, see figure 6.2.

In the third case we setup a *four-display* tiled display with one master control node and two slave rendering nodes driving two displays each, see figure 6.2.

## Tile specific rendering

Now, we will turn to the more interesting part of driving a tiled display. If scenery is rendered in a tile-specific fashion, i.e. that parts of the scene can be clipped before their primitives are generated and rendered, then we exploit the advantages of a tiled display when multiple instances running



	three display setup			four display setup			
render time	32ms	24ms	24ms	40ms	40ms	25ms	25ms
synch. time	0.1ms	8.5ms	9.5ms	0.1ms	9ms	2ms	2ms
frames/s	30 fps			25 fps			
fps(single)	30 fps			30 fps			

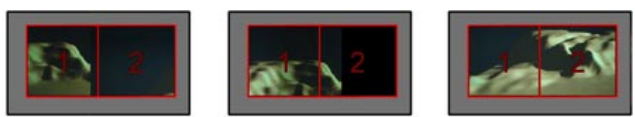
Figure 6.2: Non-tile specific rendering with normal Studierstube applications. A three display setup driven by three nodes and a four display setup driven by two nodes. Frame rates are comparable to single computer rendering. The four display setup shows worse results, because each node must render two frames at a time. Displays 1,2 and 3,4 contributes to node 1 and node 2 respectively.

concurrently within the render cluster. Each instance of the application can perform pre-culling steps before the scene graph is traversed and rendered. Thus we can prune the scene graph and only render a part of it. This in turn improves overall performance of the tiled display. As in the non-tile specific rendering, we choose two setups where each render node drives a single display and one setup where each node drives two displays. We run a view dependent scenery with over 500.000 polygons to compare the tiled display's performance with that of a single machine.

First, we setup a *two-display* tiled display with one master control node and two slave rendering nodes driving the displays. Figure 6.3 shows the results in comparison. If the scenery is moved into the display area of the tiled display, the frame rates drops below the single computer rendering's frame rate. This is of the higher-resolution of the tiled display against the single computer's resolution, since in this scene also a level of detail dependent of the screen resolution is computed. But if the scene is moved into the middle of the tiled display the render performance of the tiled display matches the single machine's performance.

In the second setup we build a *three-display* tiled display. The master takes the rule of both a control node and a rendering node. And in the third case we made a *four-display* tiled display with one master control node and two slave rendering nodes driving two displays each. Figure 6.4 depicts the results of these setups. It shows clearly that the tiled display approach





	uneven load		slightly uneven		even load	
render time	40ms	5ms	60ms	20ms	50ms	50ms
synch. time	0.1ms	34ms	0.1ms	35ms	0.1ms	1ms
frames/s	25 fps		16 fps		20 fps	
fps(single)	30 fps		15 fps		12 fps	


Figure 6.3: Rendering a scenery that performs tile-specific culling before primitives are generated and rendered by the hardware. A two display setup driven by two nodes with uneven, slightly uneven, and even load balance. In the even load scenario synchronization time is  $\leq 0.1ms$ , in the other scenarios synchronization time is  $\geq 3ms$ . In the uneven load scenario the frame rate is below the single computer’s frame rate due to a higher level of detail when the scenery is rendered on the tiled display.

matches the single node rendering even though the tiled display system renders at a higher level of detail generating more polygons than the single machine.

## 6.2 Visual accuracy

One vital part of a tiled display is its visual accuracy. Miscalibration of a projector’s collineation matrix leads to a wrong calculated projection matrix. This influences the visual accuracy since misaligned tiles within the tiled display show discontinuities in geometry. The human eye is trained to find geometric relations like lines and shapes. Even if shapes are only partially seen, the human perception reconstruct their origin shapes. Therefore non-continuous shapes, objects, or geometry at adjoining tiles must be avoided. Otherwise this leads to a visual disruption. Thus makes a tiled display unusable since the discontinuities are disturbing the illusion of a large seamless high-resolution display.

The most important part for visual accuracy are of course the overlapping regions where multiple projectors displays the same geometry. We have implemented the solution shown by Raskar et al. [Raskar et al.02]. We project a calibration pattern, an ordinary chessboard, with each projector. A camera takes a snapshot of each projection and try to find the feature points of the calibration pattern. Since we now the feature points position in normal-



	three display setup	four display setup
frames/s	18-20 fps	15 fps
fps(single)	10-12 fps	10-12 fps

Figure 6.4: Tile specific rendering performed on a three and four display setup. These results show that, if we render in a distributed way and perform tile-specific scene graph pruning, the frame rate always stays higher compared to a single computer rendering the same scenery even at a lower level of detail. And to top the tiled display approach, in the four display setup each node must render two frames at a time, but also in this case the tiled display system performs better than the single computer rendering. Displays 1,2 and 3,4 contributes to node 1 and node 2 respectively.

ized projector space, we can find a collineation between the detected feature points and the normalized feature points. This homography describes the relation between each projector space and camera space. This technique is even good enough to show subpixel accuracy while the resolution of the detecting cameras is 3 to 5 times lower than the resolution of the tiled display, see figure 6.5. Although with the camera we used the algorithm does not always find the best solution for calibration. It sometimes turn out that even if all feature points of the patterns are detected the resultant homography and therefore the resultant projection matrix does not meet the accuracy needed to project continuous geometry in the overlapping regions. This can be mostly avoided by (i) raising the number of feature points, i.e. raising the resolution of chessboard squares on the chessboard pattern, (ii) tuning the camera's distortion parameters and the lighting conditions, and (iii) preventing the camera from choosing auto-leveling the brightness. The last point is most critical since off-the-shelf webcams often exhibit the behavior to over-tune the contrast and brightness levels. If the projection pattern is taken with a standard digital camera under the same environmental circumstances the contrast levels are much better and do not exhibit strong escapees. Figure 6.6 shows the two pictures taken by both cameras and the corresponding histograms. While the standard digital camera shows smooth an equal levels, the off-the-shelf camera exhibits very jagged and bright levels. This can make feature detection difficult, because alternating bright and dark fields, like a

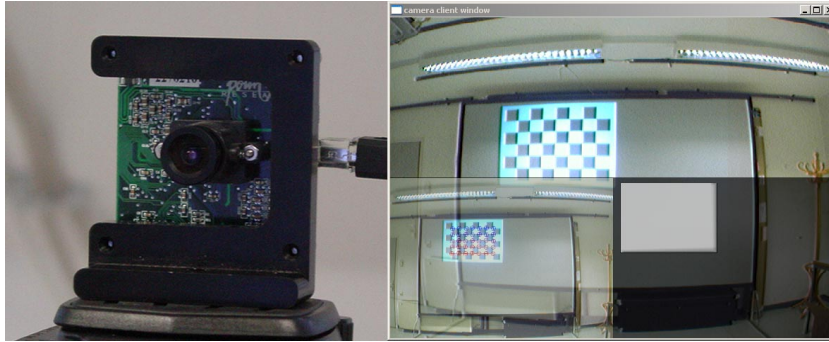


Figure 6.5: An off-the-shelf camera with a resolution of  $640 \times 320$  pixels (left) detects the feature points of a projected calibration pattern (right).

chessboard exhibits, crossfade and the well defined intercept points of the fields can not be detected. This shows directly to use an accurate camera to get sharp and well-defined points of interception. In the end, this influences the detection and calibration step, and the attained results are the most accurate. Finding the intercept points of the chessboard at their real position is most critical to get a good result for the homographies. Figure 6.7 shows two projected patterns and the found positions of crossing fields. On the left the points are all detected in nearly their real positions, whereas on the right some positions of the points are wrong detected. Wrong position leads to slight errors when finding the homographies between the feature point in projector and in camera space. And this in the end mess up the continuity of geometry when a projection matrix is compute from the wrong homography.

With accurate feature point detection we achieve subpixel accuracy on images of the tiled display. Figure 6.8 to figure 6.9 shows render results and the achieved pixel accuracy and geometrical continuity.

## 6.3 Scalability

In this section we discuss the scalability of our system concerning the different part of the tiled display.

### Calibration

First, while our calibration algorithm is scalable, the presented implementation only supports one camera at the moment. This restricts the actual size of the tiled display at the moment. At the time of investigation the number

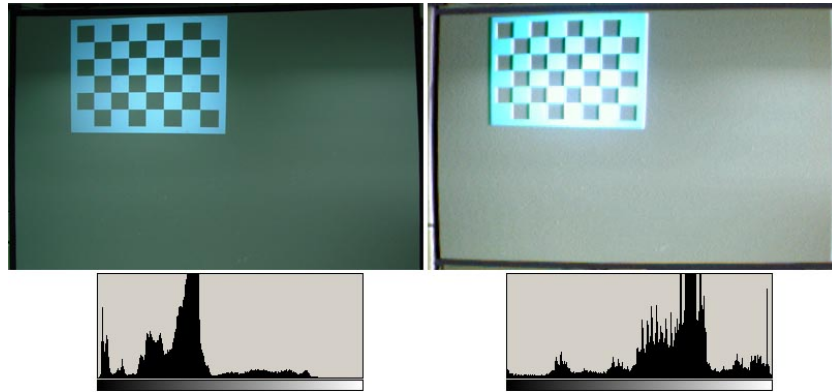


Figure 6.6: Photographs and histograms of a standard digital camera (left) and an off-the-shelf camera (right) for the same shot under equal environmental circumstances. The off-the-shelf camera has a jagged histogram predominantly in bright regions. This makes feature detection difficult.

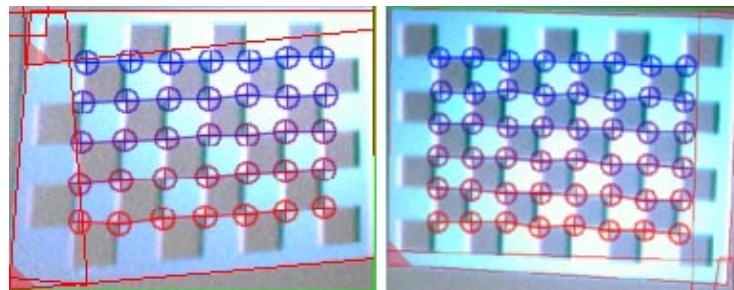


Figure 6.7: These two snapshots show a good result of detected feature points on the left and a bad result on the right. The detected positions of crossing fields are not on their real positions. This leads to wrong computed homographies and to not matching geometries in overlapping regions.



Figure 6.8: A rendered interaction panel used for user input. Geometry and literals remain continuous even in overlapping regions.

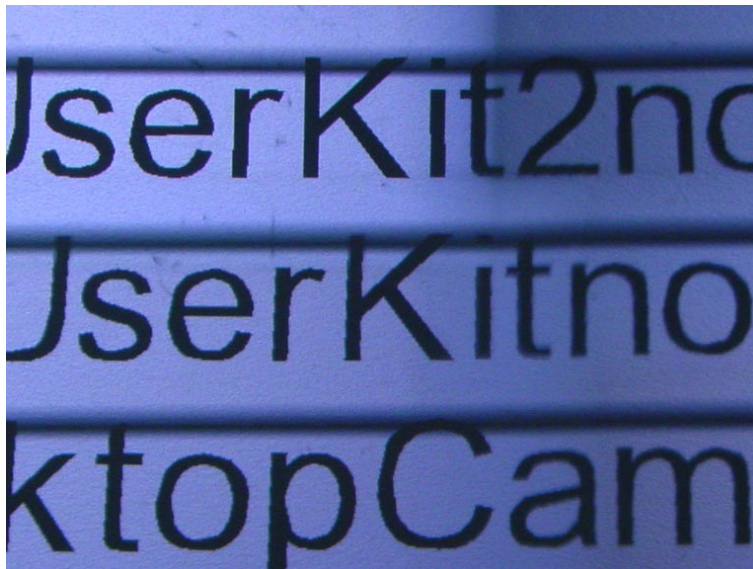


Figure 6.9: A magnified section where all projectors overlap. Pixels from different projectors match in overlapping regions.

of hardware available to drive the tiled display did not exceed the maximum amount that can be calibrated by the current version of our software. But it should be made clear that the actual version should be extended to be prepare to scale up our tiled display in the future.

## Synchronization

When designing a scalable system, we also have to look at our synchronization scheme. It will not scale infinitely, but since we use a dedicated network for synchronization only and the sent synchronization message are very small (8 bytes raw data), the traffic load on the network is very low. In our experiments the overall load of the synchronization network was always less then 0.01% of the total bandwidth. This is a very good indication that the synchronization will not be a limiting factor concerning network load.

## Distributed Open Inventor

Turning towards the distribution layer of the Studierstube framework, the Distributed Open Inventor, can prevent scaling up the system. Distributed Open Inventor uses multicast messages which slow down overall performance as other researches have shown in the past. Using multicast messages in a tiled display system can impact the network bandwidth. Since we use the Studierstube framework in a parallel execution fashion with system-level call synchronization, the main data that has to be transmitted is user input and other application state changes (e.g. node generation or field alteration). If the scenery is not altered to often the disadvantages from the multicast approach in Distributed Open Inventor is negligible and only impacts the system for short time periods. This is remarkable when high-detail scenery with thousands of polygons is loaded and then distributed to all instances of the Studierstube framework. This scenario arose in the experiments with tile specific rendering, where a scenery was loaded with over 500.000 polygons. Distributing the dataset take nearly two seconds, but after the scenery was transmitted to all nodes it rendered smooth and without any interruption.

The advantage of this approach is that when the shared scene graph is once distributed to all render nodes, altering states do not impact the network load. But, during scene graph distribution the system slows down, or even stagnate if a scenery with high detail is loaded. Thus having a copy of the shared scene on the slave's harddisk and load it instead of distributing it will improve the render cluster's performance.

## 6.4 Maintenance

As a last point to interrogate for, we should examine our solution with respect to maintenance. Since we designed our system to perform calibration automatically and without any manual fine tuning, maintenance is low. The only thing that must be maintained is the setup of the actual tiled display configuration. As described in chapter 9 the user of the tiled display has to write some configuration files to setup each part of the system, including cameras, projectors, and the user himself to be able to start the calibration process.

After a setup is described by its configuration files it can be reconfigured and moved as the user wants it as long as no system parts are added or removed. Then, the only step to drive the tiled display again after a rearrangement is to run the calibration software to compute the new blending masks and projection matrices. After the calibration step the tiled display is ready to run.





# Chapter 7

## Future work

### 7.1 Display and event synchronization

As mention in section 4.4 and 6.2 there is no synchronization between events and display refreshes. An event related to a certain frame should be delivered to all slaves before the display refresh occurs. Thus the *session manager* and the *render synchronization* should be aware of each other not only running concurrently but in a synchronized manner. To be clear a frame refresh should not occur before its related events have arrived at the client sides. This should be supported by new versions of the tiled display, while in the current version late arriving events are noticeable as disjoining scenery between different tiles.

### 7.2 Synchronized execution

We use high performance display nodes within the render cluster allowing us to run the same application code on every display node. We have to guarantee that every copy of the program runs synchronized, so we used frame buffer switch synchronization to realize system-level synchronization (see 3.4.1). This simple technique guarantees system-level synchronization for single threaded programs, like the current version of Studierstube. But future versions of Studierstube will be based on a multi-threaded implementation. With a multi-threaded Studierstube, we have to synchronize every thread, known as application-level synchronization (see 3.4.1). This is fairly done with a simple `SynchronizeResult()` method that keeps shared data synchronized for every single thread. This affects the code of Studierstube applications to meet synchronized results at defined points and to synchronize program execution.

### 7.3 Very large displays

In our tests we only used one camera to calibrate the whole display. For very large displays this would become more and more inaccurate if more displays are added to the render cluster. So we have to use more than one camera each taking a partially look at the display. Then each camera tries to calibrate as many displays as possible while displays it will see partially or not at all will not lead to a calibration result and do not count for this camera. After each camera has tried to calibrate each display, a virtual camera will assemble the results from all cameras combining the homographies of each camera space to one virtual camera space and calibrate the whole display in the virtual camera space. An implementation of this approach was introduced in [Chen et al.01a].

### 7.4 Detecting the screen wall

Our solution for calibration forces the user of our tiled display to put the camera in front of the screen wall during calibration. Its detection plane must be parallel to the screen and the horizontal and vertical axis has to be aligned with the ones of the display to compute exact projection matrices for the displays in the render cluster. With region growth algorithms it is also possible to detect the screen wall and to put the camera anywhere in front of the screen wall. A method to detect an oblique screen wall in respect to the camera's detection plane and to compute a projection matrix to auto-correct the projection areas of the projectors is described in [Sukthankar et al.00a].

### 7.5 Other synchronization models

We implemented just one model of synchronization, a software buffer swap synchronization. For future research there are many more synchronization models we can think of to implement and support for our tiled display, see section 3.5.

# Chapter 8

## Acknowledgement

First of all I would like to thank Dieter Schmalstieg for offering me this project, making me familiar with the people of our institute, and always handing me to the right person when a problem arose. I would thank all people who helped me to fix the code and spent their time and skill to get things done. Namely, in alphabetical order, Hannes Kaufmann giving me advice in distributed *Studierstube* and to make things running, Joseph Newman for helping me ordering hardware, Gerhard Reitmayr who spent a lot of time for discussion on implementation design to extend and enhance the *Studierstube* software, Thomas Pintaric who wrote a wrapper class to easily access Microsoft®'s DirectShow® which makes camera and video stream access very uncomplicated and totally independent from my tasks, Daniel Wagner for supporting me and giving me introductory advice using cameras under Microsoft®'s DirectX® and DirectShow® and who helped me to debug a lot of code.

I also thank the people at MERL (Mitsubishi electronics research labs) for their code of their Mosaic software. We used it as a base for our calibration software. We extracted some parts of their software to integrate it into ours.

# Index

- abutted display, 18
- collineation, 22, 51, 63, 64
- control node, 30, 33, 40, 52, 55
- data distribution
  - bucketing, 41
  - control node, 33
  - display node, 33
  - render cluster, 33
  - render node, 33
  - task, 33
    - application, 33
    - display, 33
    - render, 33
    - user interface, 33
- display node, 30, 33, 39, 40, 43, 44, 46, 52, 55, 57, 60, 64, 70, 81
- display type, 17
  - abutted display, 18
  - regular overlap display, 19
  - rough overlap display, 20
- homography, 22, 51, 57, 63, 64, 82
- regular overlap display, 19
  - blending, 19
- render cluster, 30
  - control node, 33
  - display node, 33
  - render node, 33
- rough overlap display, 20
  - blending, 21, 51, 57, 64
  - collineation, 22
  - corrected projection matrix, 27
  - depth-buffer approximation, 26, 27
  - homography, 22, 23
  - non-planar surface, 21
  - projection matrix, 26
- SoAlphaMaskKit, 64, 65
- SoGLRenderAction, 60
- SoOffAxisCamera, 64, 65
- SoRenderSync, 60, 61, 64
- SoRenderSyncUDP, 60, 62, 64
- SoStuberenaKit, 64, 65
- synchronized execution, 37
  - application-level sync., 39
  - system-level sync., 37

## Part II



# Chapter 9

## User Manual

### 9.1 Introduction

The Overlap software package is a calibration software designed to run on a cluster of workstations to meet the need for a large tiled display. Its intended use is to calibrate a tiled display composed of various displays connected to a computer cluster. A computer cluster is a set of networked computers which share data of the different overlap software parts. Each computer within the cluster can drive one or more displays. All displays together form a huge display termed a tiled display. And each display will be referred to as a tile of the tiled display. The output generated by the calibration software is used to run a tiled display (examples for outputs used by other applications are blending masks, projection and collineation matrices). See figure 9.1.

The display assembly can be homogeneous (using the same or a similar display device for each tile of the tiled display), or heterogenous (using different devices to form a multi-screen display). In the homogeneous case using projectors one can build a seamless tiled display with as many devices as needed to form a Megapixels screen wall. Alternatively one can use regular monitors for displaying a large scenery with high detail and high resolution. The drawback are the seams between adjoining displays. In the heterogenous case any kind of displays may be used building a mixture of seamless displays and information sideboards. Figure 9.2 depicts these three configurations.

In our implementation we used a  $3 \times 2$  projector array to build a display wall with a total resolution of 4.7 Megapixels.

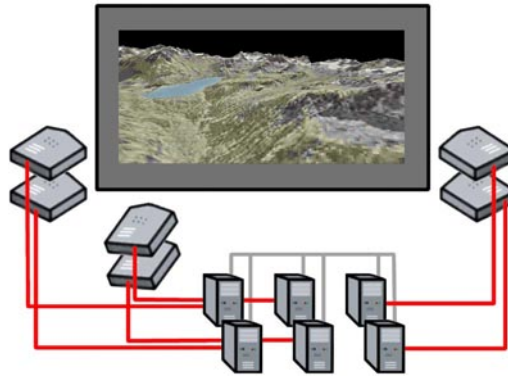


Figure 9.1: A tiled display composed of various displays connected to a computer cluster.

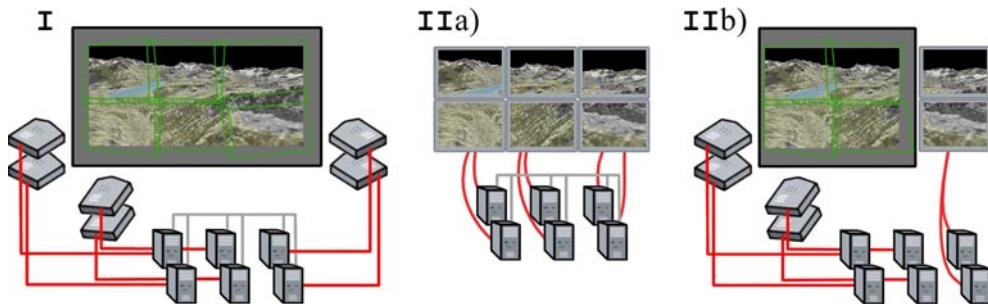


Figure 9.2: The three different display configurations. I) homogeneous display assembly: a seamless tiled display using projectors. II) heterogeneous display assembly: a) a tiled display using multiple monitors, b) a tiled display using a seamless display part and an information part with seams.



## 9.2 Application usage

The Overlap software package contains four applications to calibrate each display in the tiled display, namely a server application, a camera client, projector client, and user client application. The server application manages the communication between the three other applications. The server must be the first application to run. All other applications can be started arbitrarily. The camera client application takes care of calibration. A projector client application will be launched for each display within the render cluster that contributes to the tiled display. The user client application is the interface for an user to start the calibration and change various calibration parameters. Each application will be started with a configuration file holding a brief description, see the following paragraphs for detailed information.

### 9.2.1 Server application

The server application will be launched just with a single parameter telling it on which port to listen to. The server is only responsible for delivering the concurrent clients with the right information. Thus every client application is able to communicate with each other.

Usage: `server [port]`, if no port is supported a default port number is used.

### 9.2.2 Camera client application

The camera application should be started with a configuration file telling the application which camera to use and at which resolution. The application will run a preview window showing the current area seen by the camera. The camera should be installed in a way keeping its image plane roughly parallel to the screen surface with the camera's field of view covering the whole area of the projection screen. In the current implementation of the Overlap software only one camera is used to calibrate the tiled display. So, all displays must be seen by the selected camera. Once running the calibration step the camera will take a snapshot of each display finding its relative position and orientation to all other displays and in respect to the display screen.

#### Configuration file

The configuration file contains information about the server and the camera. There are four values to set in the configuration file, see also table [9.1](#).

<i>server</i> : <i>string</i> : <i>integer</i>	<i>server</i> : 169.128.128.1:23451
<i>size</i> : <i>integer</i> , <i>integer</i>	<i>size</i> : 640,480
<i>name</i> : <i>string</i>	<i>name</i> : 1394
<i>param file</i> : <i>string</i>	<i>param file</i> : camparams.txt

Table 9.1: Camera configuration file specification (left), example (right).

<i>float float float float</i>	-0.200 -0.300 0.000 -0.006
<i>float float float</i>	640.000 0.000 320.000
<i>float float float</i>	0.000 480.000 240.000
<i>float float float</i>	0.000 0.000 1.000

Table 9.2: Camera parameter file specification (left), example (right).

**server**: Specifies the host name or the IP-address where the calibration server is running. The port can be additionally set. Write the port after the name with a colon separated

**size**: Describes the resolution of the used camera in pixels. Horizontal resolution first, vertical second.

**name**: The camera's name, or a part of the name (like 1394, Quickcam, Sony, ...).

**param file**: Specifies a parameter file that contains the distortion parameter of the camera. See next section for a brief description on parameter files.

### Parameter file

The parameter file contains the distortion coefficients used by OpenCV to undistort images taken by a calibrated camera. Within the file the parameters are written as floats and separated by spaces. The four floats in the first line representing the four tangential and radial distortion parameters. Where the next three lines containing three floats each line represent the camera intrinsic matrix. Table 9.2 shows the file specification and an example.

### Usage

`camera [config-file]`, if no configuration file is supported default values are used for the camera without a guarantee to run the application.

`camera -calib images fieldX fieldY fieldSize time`, use camera client as stand-alone application to find distortion parameters. *Note:* a calibration pattern (chessboard) is needed to compute the parameters. Move and rotate the calibration pattern in front of the camera to find good results for the distortion parameters. After all images are taken the undistorted camera view is shown. The computed result can be accepted and the application quits. If the result is not accepted by the user the camera calibration process starts again. The result can be copied to a parameter file. Set `images` to the number of images that should be taken to find the distortion parameters. Set `fieldX` and `fieldY` to the number of fields on the chessboard in horizontal and vertical direction. `fieldSize` is the real size of a chessboard field in centimeters. `time` is the elapsed time in milliseconds between consecutive snapshots.

`camera -preview parameter-file`, use camera client as stand-alone application to manually tune distortion parameters. The output can be copied to a parameter file. Set `parameter-file` to a distortion parameter file. Its content will be read and used for fine adjustment.

### 9.2.3 Projector client application

The projector application will be started for each display in the render cluster respectively in the tiled display. With the information taken from the configuration file one can add as many displays as needed to form the tiled display. The configuration file contains a description of the display like resolution, gamma correction value, and its brightness. But it delivers also the opportunity to build a mono tiled display, or a stereo tiled display. In the latter case all displays have to be either a stereo left, or stereo right display. If a mixed usage of mono and stereo displays are used, the Overlap software will change to mono mode. All displays with a stereo left attribute forming the tiled display for the left eye, and all displays with a stereo right attribute forming the tiled display for the right eye. Each tiled display will be calibrated like a mono tiled display would do, but to achieve a stereo tiled display, both should overlap. Figure 9.3 depicts the situation for a mono display and a stereo display.

#### Configuration file

The configuration file contains information about the server and the projector or display used. There are six values to set in the configuration file, see also

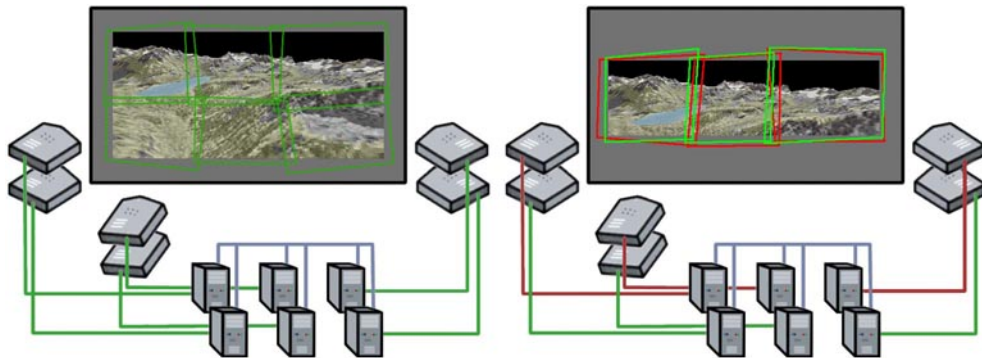


Figure 9.3: A mono tiled display on the right and a stereo tiled display on the left. Note the overlap of two projectors at a time from the left eye group (red) and the right eye group (green).

table 9.3.

**server:** Specifies the host name or the IP-address where the calibration server is running. The port can be additionally set. Write the port after the name with a colon separated

**origin and size:** Describes the origin and the size of the used display in pixels. The origin is measured with respect to the computer's desktop or framebuffer resolution (e.g. using a  $2048 \times 768$  pixels desktop resolution with two displays at a resolution of  $1024 \times 768$  pixels leads to 0,0 for the first and 1024,0 for the second display).

**mode:** This value is one of `mono`, `stereo_left`, or `stereo_right` to describe the intended use of the display. If `mono` is used a mono display wall will be built. If `stereo_left` or `stereo_right` is used for the used displays, a passive stereo display will be built. If a mixture of `mono` and `stereo_*` is used, a mono display wall is constructed.

**gamma and brightness:** Specifies an exponential gamma correction value within the alpha mask generation parameter, and a linear brightness factor with respect to all other used displays (range between 0 and 1)

## Usage

`projector [config-file]`, if no configuration file is supported default values are used for the display without a guarantee to run the application or representing the display correctly.

<code>server: <i>string:integer</i></code>	<code>server: 169.128.128.1:23451</code>
<code>origin: <i>integer,integer</i></code>	<code>origin: 0,0</code>
<code>size: <i>integer,integer</i></code>	<code>size: 1024,768</code>
<code>mode: <i>string</i></code>	<code>mode: mono</code>
<code>gamma: <i>float</i></code>	<code>gamma: 1.9</code>
<code>brightness: <i>float</i></code>	<code>brightness: 1.0</code>

Table 9.3: Projector configuration file specification (left), example (right).

### 9.2.4 User client application

The user application represents the interface for the user to start and change the calibration steps. It runs in a console window and the user controls the application with the keyboard. See table 9.4 for a description of the keys' intended use.

#### Configuration file

The configuration file contains information about the Overlap server and contains information necessary to build Open Inventor files for the Studierstube environment. There are four values to set in the configuration file, see also table 9.5.

**server:** Specifies the host name or the IP-address where the calibration server is running. The port can be additionally set. Write the port after the name with a colon separated

**render sync server address:** This specifies the host name or IP address where the synchronization server runs. This server is used in the StubeRenA environment to synchronize framebuffer switches between the all render synchronization clients. Clients are instantiate by the subclass of SoRenderSyncUDP and join a multicast group to listen to the server's synchronization packet.

**render sync multicast address:** The multicast IP address where the render synchronization clients can join. This multicast group is used by the render synchronization server. A packet is sent to all members of this group to force a framebuffer switch. This guarantees synchronized output on all tiles of the tiled display.

**shared directory:** A shared directory destination, where generated files are stored. Files that are generated during the calibration step are

---

Calibration control keys	
CTRL-X	Stops all overlap applications connected with the server (including all clients and the server)
X	Stops the user application
G, . .	Terminates a client group (P-projector, C-camera, A-both projector and camera clients)
H	Shows help (command keys)
S	Starts calibration
ESCAPE	Stops calibration
R	Runs Studierstube on display wall
T	Runs Studierstube on display wall and terminates all calibration programs

---

Projector control keys	
N	Sets new resolution of chessboard
P	Shows/hides pattern on each projector client's display area
B	Shows/hides blank screen on each projector client's display area
RIGHT/LEFT	Increase/decrease pattern brightness of each projector client

---

Camera control keys	
V	Switches between camera preview modes
C	Shows/hides camera client's preview window
SPACE	Resets calibration values (alpha masks and display wall area)
UP/DOWN	Scrolls through detected pictures or computed alpha masks

---

Table 9.4: Keys and their meanings.

---

```

server: string:integer
render sync server address: string
render sync multicast address: string
shared directory: string

```

---

```

server: 169.128.128.1:23451
render sync server address: 169.128.128.1
render sync multicast address: 224.0.0.1
shared directory: //Blue/shared/overlap

```

---

Table 9.5: User configuration file specification (top), example (bottom).

store to this directory. This directory contains all files needed for the StubeRenA to run (e.g. alpha masks, correcting projection matrices, startup informations).

### Usage

`user [config-file]`, if no configuration file is supported default values are used for the user application without a guarantee to run the Studierstube software correctly afterwards.

## 9.3 Installation and Compilation

Overlap consists of four program parts. The files `overlap.exe`, `camera.exe`, `projector.exe`, and `user.exe`. Additional components are the DLLs from Intel's Image Processing Library (IPL) and the DLL from the DsDxVideoWrapper which must also be installed (extend PATH variable with `'./overlap/bin/DLLs'`) or in the same directory as the `camera.exe` file.

The original Overlap directory structure looks as follows (unzipping the overlap software package)

Executables and DLLs

```

./overlap/bin
./overlap/bin/DLLs

```

Source code

```

./overlap/src
./overlap/src/all
./overlap/src/lib
./overlap/src/renaCameraClient

```

```
./overlap/src/renaClient  
./overlap/src/renaProjectorClient  
./overlap/src/renaProtocols  
./overlap/src/renaServer  
./overlap/src/renaUserClient
```

### 9.3.1 Before compiling the Overlap package

- Install DirectX8.1 SDK (for the DsDxVideoWrapper)
- Install OpenCV (for camera calibration routines)
- Set an environment variable OPENCV to full path of `./Opencv/cv` directory
- Compile and build the `CV1.lib` (for release code) and/or `CV1d.lib` (for debug code)
  - Open `./OpenCV/_dsw/cv.dsw`
  - Switch to 'Win32 Release Static' or 'Win32 Debug Static' within the Build bar.
  - Start compile
- Compile Overlap package
  - Switch to project 'all files' in the workspace window to compile the whole package
  - or switch to any project of 'rena\*Client', 'renaServer' to compile individual parts of the package

### 9.3.2 Before running any part of the Overlap software

- Add to PATH variable the full path to `./Overlap/bin/DLLs`
- Install DirectX8.1 SDK (for the DsDxVideoWrapper)

## 9.4 Running the applications

- See section [9.3.2](#) before proceeding.



- Run the server application first – `server.exe` file (usage: `server [port]` see section 9.2.1).
- Run any of the client applications on any node within the network.
- You have to run at least one camera and two projector clients to be able to start the calibration.
- The server does not allow more than one user client. The first user client that register at the server will be served.
- In the server and user console window you will see which clients are connected to the server.
- Hit the key 'H' in the user console to get the list of control commands to use the calibration software (see also table 9.4).
- In the camera *preview window* you will see the actual process of the calibration and the camera's view.
- The camera *console window* shows information about the calibration progress.

## 9.5 Known problems

- Run the server and user application on the same machine. Since very small buffers are used for communication and no care about message fragmentation is done, this should avoid any problem. If the user client fails start it again (this is because of fragmented messages, maybe we will fix this problem in future versions of Overlap).



# Chapter 10

## Developer Manual

### 10.1 Introduction

The Overlap software package is a calibration software designed to run on a cluster of workstations to meet the need for a large tiled display. Its intended use is to calibrate a tiled display composed of various displays connected to a computer cluster. A computer cluster is a set of networked computers which share data of the different overlap software parts. Each computer within the cluster can drive one or more displays. All displays together form a huge high-resolution display termed a tiled display. And each display will be named a tile of the tiled display.

This chapter describes in a short way the main steps taken when running one of the software components and the used classes to do so. It should be remarked that display and projector is used interchangeably, because of the intended use of Overlap to calibrate a seamless tiled display built up with projectors. But it should be mentioned here that any display can be bound into the tiled display.

In the following section we will show the hierarchy of classes used and their relationship one another. We use ACE, the Adaptive Communication Environment, as a base to implement platform independent code for all classes in respect to network communications. But some parts are still platform specific and may be ported to other platforms in future extensions.

### 10.2 Class hierarchy

Figure 10.1 shows the classes used to run the server side, while figure 10.2 and figure 10.3 shows the classes for the client side. All following sections will use this diagrams to illustrate how the individual components and classes

interact with each other.

Table 10.1 depicts the structure of protocols that are sent between clients and server or clients and clients. It also shows a part of the used RACOM class which stores all header strings. This structure allows future extensions of protocols and the behavior they should invoke at the client side. A protocol has one or two parts and can be simple or complex to invoke different behaviors. The first part is the header in the form `<...|` where the dots stand for specific protocols. A valid protocol header would be *registration ok* (`<rgok|`), or *register user client* (`<regU|`). The last character of the header is a client specifier and can be one of U, C, or P for one of different client types. If it is none of the three then this header is a general message that can be sent to any client or to the server.

A simple protocol only consists of the header and would be used to inform the client that its registration was successful as an example (`RACOM::REGISTER_OK ⇒ <rgok|`). Simple protocols are used for a direct communication between the server and a client.

A complex protocol consists of the header and a tail, a detailed message, delimited by a special character. As an example a complex protocol would be to inform the camera client which image it has to take (e.g. take image 3: `RACOM::TAKE_IMAGE + RACOM::RACOM_FULL + 3 + RACOM::EOM ⇒ <timC|full|3>`). Complex protocols are always delimited with a `>` character (`RACOM::EOM`).

As a last option protocols can be combined. This allows to send protocols from one client to another. So, if the user client wants to tell all camera clients to take picture three, it will send the server a combined protocol that tells the server to forward it to all camera clients (e.g. user to camera take image 3: `RACOM::SEND_TO_CAM + -1 + RACOM::TAKE_IMAGE + RACOM::FULL + 3 + RACOM::EOM ⇒ <sdC|full|3>`). Note the two client specifiers at the end of each header (U and C). This emphasizes that some protocols are only used by specific clients. As a last example, if the user client wants a specific camera, let's say number 0, we have to replace the -1 with a 0. If -1 is used in a `RACOM::SEND_TO...` protocol, all mentioned clients of that group (CAM,PROJ) are delivered with the subsequent protocol.

### 10.3 Overlap server code

The server application consists of a handful classes to establish a connection oriented peer to peer communication to a client.

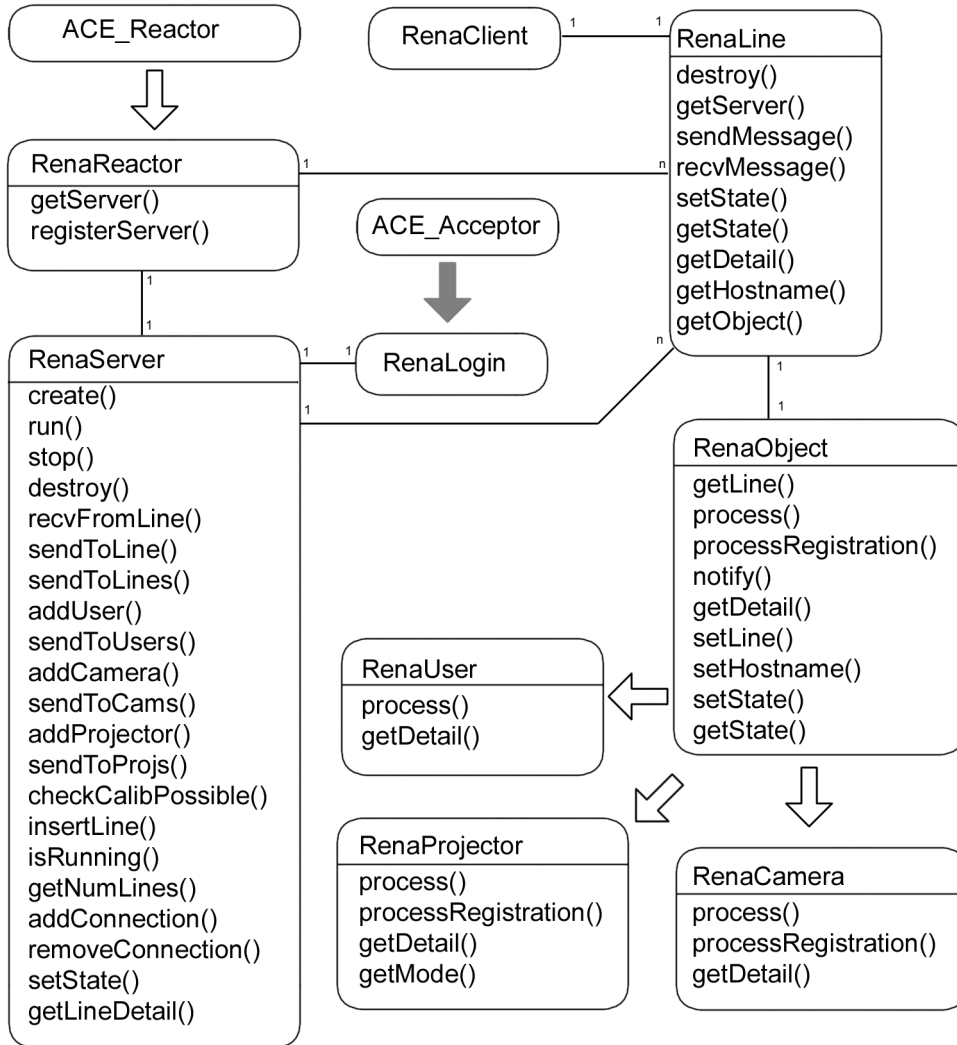


Figure 10.1: The classes and their relationship on the server side.

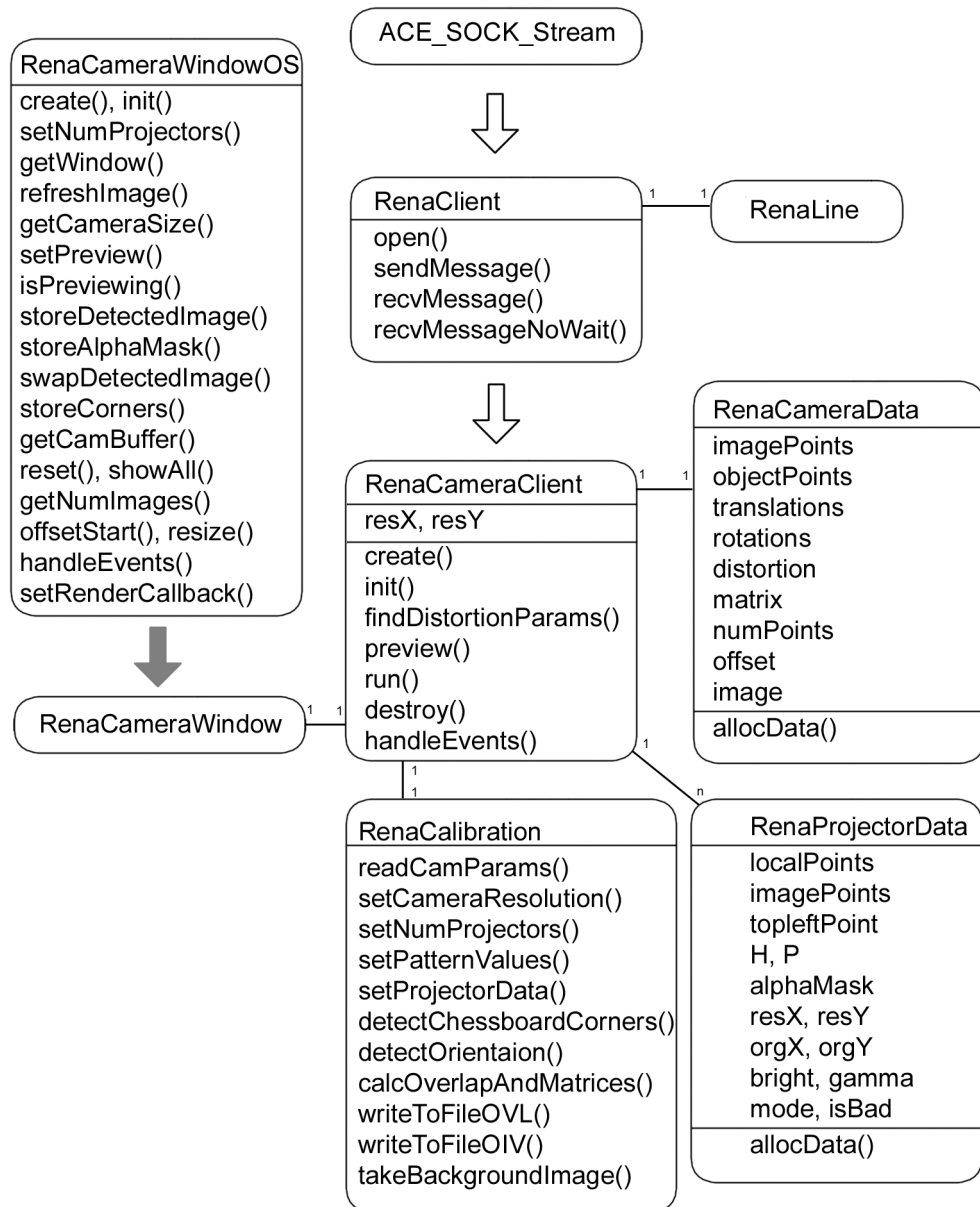


Figure 10.2: The classes and their relationship on the camera client side.

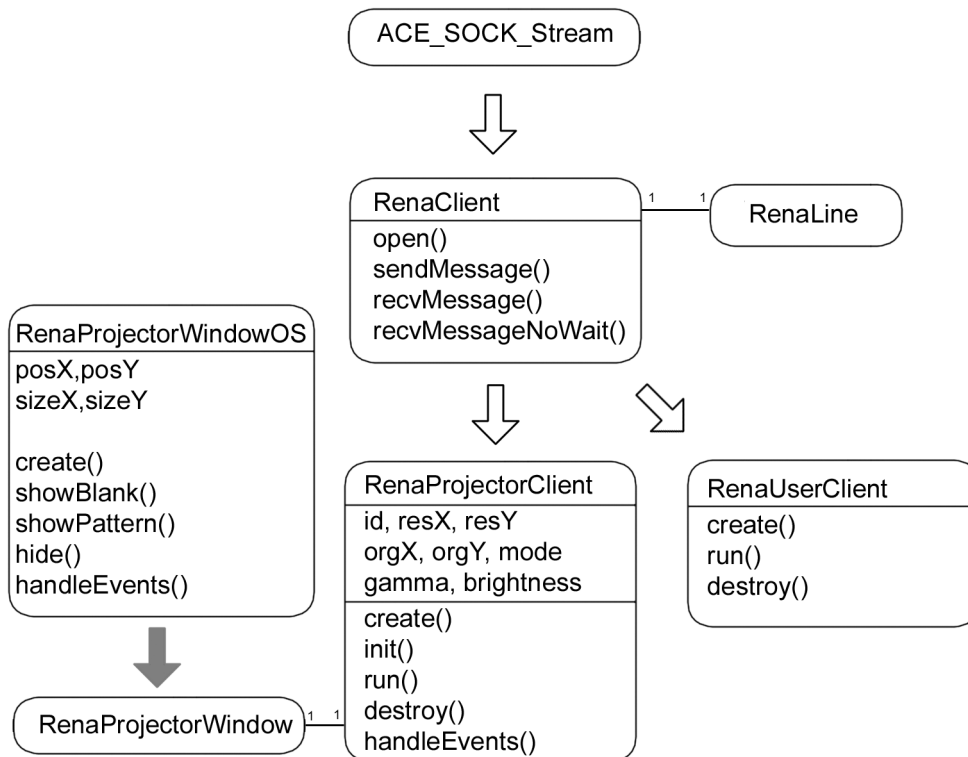


Figure 10.3: The classes and their relationship on the projector and user client side.

---

A **Header** contains 6 chars including < and |. The **Body** can be of arbitrary size and contains additional information.

<header|body>

A simple protocol uses a header only:

<c1rd| – client ready

A Complex protocol uses header and body:

<stbP|//Blue/shared/ 196.0.0.1> – start Studierstube on projector client!

Combined protocols use simple and complex protocols:

<sdpU|-1><stbP|//Blue/shared/ 196.0.0.1> – send to all projectors to start Studierstube!

---

Table 10.1: The schematic structure of a protocol.

### 10.3.1 RenaServer

`RenaServer.h`, `RenaServer.cxx`

This class contains an ACE reactor and ACE acceptor for managing incoming clients that want to connect. Each accepted client will be stored in a list of `RenaLine` instances. The reactor and acceptor are derivatives of ACE reactor and acceptor built-in classes.

It is the application base class. An instance is created in the `main()` function and will start the Overlap server waiting for clients to connect.

### 10.3.2 RenaReactor

`RenaServer.h`

This class is a subclass of the `ACE_Reactor` built-in class to handle opening connection from clients. See ACE's documentation for a more descriptive essay. It contains additionally to the ACE functionality the server instance it correlates to, to make sever data access (especially for client information and communication through the `RenaLine` object) easily.

### 10.3.3 RenaLogin

`RenaLogin.h`

A subclass of the `ACE_Acceptor` template class to handle clients that try to connect to the server via a socket stream.

### 10.3.4 RenaLine

`RenaLine.h`, `RenaLine.cxx`

This is a subclass of the `ACE_Svc_Handler` template class. It handles incoming and outgoing messages via a socket stream connection. Each `RenaLine` instance is a unique object representing an opened socket stream with the server on one end and a client on the other end supporting message transfer between the peers.

When a client first connects to the server, an unspecified line will be opened containing a non specified client. Until the client does not register at the server telling its behavior (camera, projector, or user), the line remains unspecified.

Each line contains therefore a unique `RenaObject` instance with information about the client and its dedicated protocols for communicating with the server and the other clients. When messages are passed to a client the `RenaObject` will handle the message's content and reply accordingly.



### 10.3.5 RenaObject

`RenaObject.h`, `RenaObject.cxx`

Each `RenaObject` correspond to a client application. Each subclass of `RenaObject` class represents a specified client with dedicated information about the client. The base class `RenaObject` supports basic communication methods to send and receive messages via a socket stream using its `RenaLine` instance.

The two fundamental methods of the `RenaObject` are the `process()` and `processRegistration()` methods. Both methods waiting for incoming messages and deciding on the content of the incoming message what message should be replied.

The `process()` method allowing client to register at the server and telling the server its intended use as camera, projector, or user client. If the client does not send the correct protocol to register it will be removed after four tries. If on the other hand the client send the correct protocol for registration it metamorphoses to one of the appropriate subclasses of `RenaObject` (`RenaCamera`, `RenaProjector`, or `RenaUser`).

The three next paragraphs describes the subclasses of `RenaObject`. The only difference between `RenaObject` class and its subclasses are additional information that are extracted when the client registers calling `processRegistration()` method and the dedicated communication protocols that each client must understand calling `process()` method.

### 10.3.6 RenaCamera

`RenaObject.h`, `RenaObjectComm.cxx`

This class containing information about the camera client's resolution and supporting the evaluation of the communication protocol the client sends to the server (`process()` method).

### 10.3.7 RenaProjector

`RenaObject.h`, `RenaObjectComm.cxx`

This class containing information about the projector client's resolution, origin, mode, gamma correction value, and its brightness in relation to other display devices. It also supports the evaluation of the communication protocol the client sends to the server (`process()` method).

### 10.3.8 RenaUser

`RenaObject.h`, `RenaObjectComm.cxx`

This class evaluates of the communication protocol the client sends to the server (`process()` method), i.e. the demanded action the user wants to take place.

### 10.3.9 main

`server.cxx`

The main function starts the server application.

### 10.3.10 Summary

All classes of the Overlap server application supporting either the basic network communication using ACE or the protocols that are used by both the client and server sides to run the whole Overlap software. None of the classes perform work concerning the calibration itself. See the next sections for a brief description on how the calibration is done by the clients.

## 10.4 Camera client code

The code has three parts. The first part is the camera display window to show the camera's view and the progress of the calibration. The second runs the client side and handles communication and protocols. And last but not least the third part. It is the main part of the calibration supporting all the needed routines to calibrate the tiled display. See the following three sections for a brief description.

### 10.4.1 RenaCameraWindow

`RenaCameraWindowWin32.h`, `RenaCameraWindowWin32.cxx`,  
`RenaCameraWindow.h`

The `RenaCameraWindowWin32` class has the capability to generate a preview window for a camera connected to the computer. Once it has found the demanded camera a window is created to show the camera's view. When starting the calibration the window also displays the progress of the calibration step, showing the detected chessboards and calculated alpha mask.

## 10.4.2 RenaCameraClient

`RenaCameraClient.h`, `RenaCameraClient.cxx`

This class provides the methods to establish and end a communication with the server (`create()` and `destroy()` method), handles its own protocols (`run()` method), and starts the client's preview window (`init()` method). It also stores the resolution of its camera on the client side and has an instance of *RenaCalibration* class to calibrate the displays.

## 10.4.3 RenaCalibration

`RenaCalibration.h`, `RenaCalibration.cxx`

This class does all the stuff for calibrating the displays, generating blending masks for overlapping regions, and writing the results to disk.

But before it can calibrate the displays it has to know the camera's distortion parameters defined in OpenCV. The parameters are 4 lens distortion parameters and a  $3 \times 3$  distortion matrix (i.e. a rotation and translation matrix). See OpenCV documentations for a brief description. Look for the keywords camera calibration, lens distortion or 3d reconstruction. The `readCamParams()` method reads these values from a file, see chapter 9 for a description of this file. The `setCameraResolution()` method sets the used resolution of the camera.

All other parameters are shipped by the user client to the camera clients and/or projector clients and used during calibration.

`setProjectorData()`, `setNumProjectors()`, and `setPatternValues()` are the methods of use to store the appropriate values for each projector and the values for the projected chessboard resolution. For each projector a variety of parameters are stored including its resolution, origin, brightness, and gamma correction values, and the mode parameter. In the calibration step we need also to store its homography and its detected chessboard points. Further we need to store the calculated alpha mask. These values are stored in an instance of `RenaProjectorData` class, see next section.

`detectChessboardCorners()` and `detectOrientation()` are the methods to detect the corners of the chessboard's grid elements. These corner points are stored to calculate the homography between camera space and display space. `detectOrientation()` is used to determine the order of the points, since the used OpenCV function `cvFindChessBoardCornerGuesses` does not sort the points from top to bottom or left to right, it does it also vice versa from bottom to top or right to left. So we have to determine in which direction the function detected the points.

`calcOverlapAndMatrices()` is the main method to calculate the homo-

graphies and their related projection matrices, the maximum projection area that can be build with the tiled display, and the alpha masks to generate a seamless display with smooth and seamless overlapping regions. For maintainability and easy reading the code, this function is divided into four sub-functions. `calcHomographies()` computes the homographies using the detected feature points of the chessboard calibration pattern. To calculate the maximum projection area that is possible for the current tiled display configuration `calcMaximumProjectionArea{Stereo}()` is used. `calcAlphaMasks()`, and `calcGLMatrices()` computes the blending masks and the corrected projection matrices to drive each tile in the tiled display.

There are two methods to save the computed values to disk, `writeToFileOVL()` and `writeToFileOIV()`. The former one stores the matrix in the alpha mask in raw data form to disk. The latter one generates some Open Inventor files to store the matrix and the alpha mask for later use in the Studierstube project.

There are some helper methods left which are self explanatory and which are called as subroutines from the other methods. Step through the source code to get a hint or look at the method documentation of the Overlap software package.

#### 10.4.4 RenaProjectorData

`RenaCalibration.h`, `RenaCalibration.cxx`

The calibration software must know about the projectors' parameters. An additional class stores these values, `RenaProjectorData`. Each instance representing one display within the render cluster containing its resolution, origin, brightness, and gamma correction value. In the calibration step it stores the homography and detected chessboard points in normalized projector space and in normalized camera space. As a postprocessing step an OpenGL projection matrix and alpha mask is calculated to build a seamless tiled display with all used displays.

#### 10.4.5 RenaMath

`RenaMath.h`, `RenaMath.cxx`

These two files containing some vector algebra to access and calculate vectors and matrices easily. The code is very short and self explanatory. See the method documentation of the Overlap software package to get a hint.

### 10.4.6 main

`camera.cxx`

The `main()` function starts the camera client application.

## 10.5 Projector client code

The code has two parts. The first part is the projector client display window to show a chessboard or a blank screen during calibration. The second runs the client side and handles communication and protocols. See the following two sections for a brief description.

### 10.5.1 RenaProjectorWindow

`RenaProjectorWindow.h`, `RenaProjWindowWin32.h`,  
`RenaProjWindowWin32.cxx`

This class is used to display a chessboard with a given number of fields and a given border size. The methods `showBlank()`, `showPattern()`, and `hide()` are used to show a blank (black) screen, the desired pattern and to hide the display window to show the desktop content.

### 10.5.2 RenaProjectorClient

`RenaProjectorClient.h`, `RenaProjectorClient.cxx`

This class provides the methods to establish and end a communication with the server (`create()`, `destroy()` methods), handles protocols (`run()` method), and creates the client's display window (`init()` method). It also stores the resolution, origin, mode, gamma correction value, and brightness of its display on the client side.

### 10.5.3 main

`projector.cxx`

The `main()` function starts the projector client application.

## 10.6 User client code

There is only one class to provide the user client application.

### 10.6.1 RenaUserClient

`RenaUserClient.h`, `RenaUserClient.cxx`

This class supports the methods to communicate with all other clients and to start the calibration step. The main function is the `startCalib()` method. It provides all actions to step through the calibration cycle. Showing the chessboard on each projector one after the other and sending the appropriate protocols to the camera client to take a snapshot of the tiled display, or to calculate the calibration parameters.

The `startCalib()` method is written in that way that more than one camera client can participate to the calibration cycle. Although the functionality for more than one camera neither in computing a global space for all cameras to determine homographies between camera spaces nor in supporting protocols to do so is not written yet. But it holds the opportunity to easily extend a future version of the Overlap software to keep scalable in the number of used display and arranging them (e.g. think of a huge panorama vision with say  $8 \times 4$  displays, one must use more than one camera to get a good shot and accurate projection matrices for the displays).

### 10.6.2 main

`user.cxx`

The `main()` function starts the user client application.

## 10.7 Additional code

### 10.7.1 RenaClient

`RenaClient.h`, `RenaClient.cxx`

This class provides the basic communication method using socket stream with ACE. There are four methods.

- `open()` opens a socket stream using ACE.
- `sendMessage()` sends a message to the server side using a socket stream.
- `recvMessage()` receives a message from the server side.
- `recvMessageNoWait()` waits for a message until a demanded time period expires.

## 10.7.2 RACOM

`protocols.h`, `protocols.cxx`

This class stores all protocol keywords to ease up programming and avoid redundant copies of the keywords within the code.

The class supports also a `copy()` and `concat()` method to copy or concat multiple buffers into one. Both methods are declaration are the same `copy/concat(char *dest, int number, ...)`, `dest` is the destination buffer where the other buffers are copied or concatenated to, `number` is the number of buffer that follows this argument.

## 10.7.3 Defined macros

The following macros are defined to help programming in a convenient way. They intended use is to send or receive messages, convert messages to string terminating them with a zero byte, to send headers or check if a message contains a specific header. It frees the developer from calling dedicated function and handling the returned value.

Macros for blocking waiting until a message arrives.

```

_get    number of bytes received
_msg    message buffer
_size  size of message buffer in bytes
_term  bool flag if true terminate received message with zero
RENA_RECV_MSG_GET_RET(_get,_msg,_size)
RENA_RECV_MSG_RET(_msg,_size)
RENA_RECV_MSG_TERM_RET(_msg,_size,_term)

```

Macros for non blocking waiting for messages.

```

_get    number of bytes received
_msg    message buffer
_size  size of message buffer in bytes
_call  call function void _call(void) when no message is ar-
       riving (i.e. in idle case)
RENA_RECV_MSG_NOWAIT_RET(_msg,_size)
RENA_RECV_MSG_GET_NOWAIT_RET(_get,_msg,_size)
RENA_RECV_MSG_GET_IDLE_CALL_RET(_get,_msg,_size,_call)
RENA_RECV_MSG_IDLE_CALL_RET(_msg,_size,_call)

```

Macro for sending a message header.

```
_header  message buffer containing header
RENA_SEND_HEADER_RET(_header)
```

Macros for sending messages.

```
_get     number of bytes sent
_msg     message buffer
_size    size of message buffer in bytes
RENA_SEND_MSG_GET_RET(_get, _msg, _size)
RENA_SEND_MSG_RET(_msg, _size)
```

Macro for sending a message to the user client.

```
_buf     message buffer (size of _buf must be at least the message
         size plus header size plus two additional bytes)
_msg     message to user
RENA_SEND_MSG_TO_USER_RET(_buf, _msg)
```

Macro for testing a message buffer if it contains the demanded header.

```
_header  header to search for
_msg     message buffer
RENA_MSG_HAS_HEADER(_msg, _head)
```

Macros for information output in debug mode.

```
RENA_OUTPUT_COMM(_x)    prints message _x in debug mode
RENA_ERROR_RETURN(_x, _y) prints the ACE error message _x
                        in debug mode, returns error code
                        _y in release and debug mode.
```

## 10.8 Remarks for future extensions

### 10.8.1 Extension of Overlap

The use of macros and dedicated protocols makes it very simple to extend the Overlap software. See section [10.7.3](#) to get a hint what the different macros are for.

The most viable and easy way to extend Overlap is to extend the protocols to support the demanded new functionality. You just have to add a new header and maybe some sub-protocols. The new functionality must be supported by the client which you want to extend and the client's `run()` method must care for the new protocol. When the protocol is detected in the run method, just call the appropriate methods for the new functionality



you want.

Especially the server need not to be change. Since it only acts as a moderator between the clients. In other words, each client sends a message to the server telling it with which other client it want to talk.

### 10.8.2 Known bugs

At the moment no care about message fragmentation is done. Message fragmentation occurs sometimes when more messages are stored in the message buffer than it can hold. At the end of the buffer a message can be cut. The rest of the message will be written in the buffer when it is read and cleared by its application. The user client receives a lot of data containing all informations about the used displays and cameras. This sometimes leads to message fragmentation when the user client does not read the buffer quick enough. So the user client is sensitive for message fragmentation and sometimes fails if more messages are stored in its buffer and message cutting occurs.



# Bibliography

- [Bishop00] Gary Bishop, Greg Welch, *Working in the Office of the “Real Soon Now”*, IEEE Computer Graphics and Applications **20(4)** (2000), pages 76-78.
- [Boden et al.95] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, Wen-King Su *Myrinet – A Gigabit-per-Second Local-Area Network* IEEE MICRO **Vol. 15** (February 1995), pages 29-36.
- [Chen et al.00a] Han Chen, Kai Li, Bin Wei, *A Parallel Ultra-High Resolution MPEG-2 Video Decoder for PC Cluster Based Tiled Display Systems*, International Parallel and Distributed Processing Symposium (April 2002).
- [Chen et al.00b] Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, Kai Li, *Software Environments For Running Desktop Applications On A Scalable High-Resolution DisplayWall*, Technical Report TR-619-00, Department of Computer Science, Princeton University (April 2000).
- [Chen et al.00c] Yuqun Chen, Han Chen, Douglas W. Clark, Zhiyan Liu, Grant Wallace, Kai Li, *Software Environments For Cluster-based Display Systems*, First IEEE/ACM International Symposium on Cluster Computing and the Grid (May 2001)
- [Chen et al.00d] Yuqun Chen, Douglas W. Clark, Adam Finkelstein, Timothy C. Housel, Kai Li, *Automatic Alignment Of High-Resolution Multi-Projector Displays Using An Un-Calibrated Camera*, IEEE Visualization (2000), pages 125-130.
- [Chen et al.01a] Han Chen, Rahul Sukthankar, Grant Wallace, Tat-Jen Cham, *Calibrating Scalable Multi-Projector Displays Using Camera Homography Trees*, Computer Vision and Pattern Recognition Technical Sketch (2001).

- [Chen et al.01b] Han Chen, Yuqun Chen, Adam Finkelstein, Thomas Funkhouser, Kai Li, Zhiyan Liu, Rudrajit Samanta, Grant Wallace, *Data Distribution Strategies for High-Resolution Displays*, Computers and Graphics **25** (2001), pages 811-818.
- [Cruz-Neira et al.93] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, *Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE*, Computer Graphics **27(Annual Conference Series)** (1993), pages 135-142.
- [Friesen00] Jerrold A. Friesen, Thomas D. Tarman, *Remote High-Performance Visualization and Collaboration*, IEEE Computer Graphics and Applications (2000).
- [Gamma et al.94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley (1994).
- [Gotz01] David Gotz, *Design Considerations for a Multi-Projector Display Rendering Cluster*, Technical Report TR01-25, University of North Carolina at Chapel Hill (2001).
- [Hereld et al.00a] Mark Hereld, Ivan R. Judson, Rick L. Stevens, *Introduction to Building Projection-based Tiled Display Systems*, Computer Graphics and Applications **20(4)** (2000), pages 22-28.
- [Hereld et al.00b] Mark Hereld, Ivan R. Judson, Joseph Paris, Rick L. Stevens, *Developing Tiled Projection Display Systems*, Proceedings of Fourth Immersive Projection Technology Workshop (2000).
- [Hesina et al.99] Gerd Hesina, Dieter Schmalstieg, Anton Fuhrmann, Werner Purgathofer, *Distributed Open Inventor: A Practical Approach to Distributed 3D Graphics*, Proceedings of the ACM Symposium on Virtual Reality Software and Technology (1999), pages 74-81.
- [Humphreys,Hanrahan99] Greg Humphreys, Pat Hanrahan, *A Distributed Graphics System for Large Tiled Displays*, Proceedings of IEEE Visualization Conference (1999), pages 215-223.
- [Humphreys et al.00] Greg Humphreys, Ian Buck, Matthew Eldridge, Pat Hanrahan, *Distributed Rendering for Scalable Displays*, IEEE Supercomputing 2000 (October 2000).

- [Humphreys et al.01] Greg Humphreys, Ian Buck, Matthew Eldridge, G. Stoll, M. Everett, Pat Hanrahan, *WireGL: A Scalable Graphics System for Clusters*, Proceedings of SIGGRAPH 2001 (August 2001), pages 129-140.
- [Jaynes et al.01] Christopher Jaynes, Stephen Webb, R. Matt Steele, Michael Brown, W. Brent Seales *Dynamic Shadow Removal from Front Projection Displays* IEEE Visualization (2001).
- [Li et al.00] Kai Li, Han Chen, Yuqun Chen, Douglas W. Clark, Perry Cook, Stefanos Damianakis, Georg Essl, Adam Finkelstein, Thomas Funkhouser, Timothy Housel, Allison Klein, Zhiyan Liu, Emil Praun, Rudrajit Samanta, Ben Shedd, Jaswinder Pal Singh, George Tzanetakis, and Jiannan Zheng, *Building and Using A Scalable Display Wall System*, IEEE Computer Graphics and Applications **20(4)** (2000), pages 671-680.
- [Raskar99] Ramesh Raskar, *Oblique Projector Rendering on Planar Surfaces for a Tracked User*, SIGGRAPH 1999 (1999), Sketch.
- [Raskar et al.99] Ramesh Raskar, Michael S. Brown, Ruigang Yang, Wei-Chao Chen, Greg Welch, Herman Towles, Brent Seales, Henry Fuchs, *Multi-Projector Displays Using Camera-Based Registration*, Proceedings of IEEE Visualization (October 1999), pages 161-168.
- [Raskar00] Ramesh Raskar, *Immersive Planar Display using Roughly Aligned Projectors*, In IEEE Virtual Reality (March 2000).
- [Raskar et al.02] Ramesh Raskar, Jeroen van Baar, Jin Xiang Chai *A Low-Cost Projector Mosaic with Fast Registration*, In Proceedings of Fifth Asian Conference on Computer Vision (January 2002).
- [Raskar et al.03] Ramesh Raskar, Jeroen van Baar, Paul Beardsley, Thomas Willwacher, Srinivas Rao, Clifton Forlines *iLamps: Geometrically Aware and Self-Configuring Projectors*, SIGGRAPH 2003 (2003).
- [Samanta99] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, Jaswinder Pal Singh, *Load Balancing for Multi-Projector Rendering Systems*, SIGGRAPH/Eurographics Workshop on Graphics Hardware (August 1999).
- [Schikore et al.00] Daniel R. Schikore, Richard A. Fischer, Randall Frank, Ross Gaunt, John Hobson, Brad Whitlock. *High-Resolution Multiprojector Display Walls*, IEEE Computer Graphics and Applications **20(4)** (2000), pages 38-44.

- [Sukthankar et al.00a] Rahul Sukthankar, Robert G. Stockton, Matthew D. Mullin, *Automatic Keystone Correction for Camera-assisted Presentation Interfaces*, Proceedings of International Conference on Multimodal Interfaces (2000).
- [Sukthankar et al.00b] Rahul Sukthankar, Robert G. Stockton, Matthew D. Mullin, *Self-Calibrating Camera-Assisted Presentation Interface*, Proceedings of International Conference on Control, Automation, Robotics and Computer Vision (2000).
- [Sukthankar et al.01] Rahul Sukthankar, Tat-Jen Cham, Gita Sukthankar, *Dynamic Shadow Elimination for Multi-Projector Displays*, Proceedings of Computer Vision and Pattern Recognition (2001).
- [Wernecke94] Josie Wernecke, *The Inventor Mentor: Programming Object Oriented 3D Graphics with Open Inventor<sup>TM</sup>*, Release 2, March 1994.
- [Wernecke et al.94] Josie Wernecke and Open Inventor Architecture Group, *Inventor Toolmaker : Extending Open Inventor, Release 2*, March 1994.
- [Yang et al.01] Ruigang Yang, David Gotz, Justin Hensley, Herman Towles, Michael S. Brown, *PixelFlex: A Reconfigurable Multi-Projector Display System*, Proceedings of IEEE Visualization (2001).
- [Chromium] Standford University, DOE labs, *Chromium Project*, <http://sourceforge.net/projects/chromium>.
- [SoftGenLock] sourceforge.net, *SoftGenLock Manual: Software Active Stereo and Genlock for Linux*, <http://netjuggler.sourceforge.net>.
- [ArsBox] Ars Electronica Futurelab, *ARSBOX*, <http://futurelab.aec.at/arsbox>.
- [ChannelSync] CGSD Company, *ChannelSync<sup>TM</sup>*, <http://www.egsd.com>.
- [Panoram] Panoram Technologies, <http://www.panoramtech.com/>.
- [Studierstube] *Studierstube Augmented Reality Project*, <http://www.studierstube.org/>.
- [Trimension] Trimension, <http://www.trimension-inc.com>.