Device-Free Interaction and Cross-Platform Pixel Based Output to Display Walls

Daniel Stødle

June 2009

A dissertation for the degree of Philosophiae Doctor

UNIVERSITY OF TROMSØ
Faculty of Science
Department of Computer Science

Abstract

Interacting with computers can be accomplished with or without devices. Interaction using devices like mice, keyboards, gloves, or gesture recognition based on tracking passive or active markers, can be restrictive and impractical when used to interact with high-resolution, wall-sized, tiled displays. Devices must be carried around, or markers must be attached to a user's clothes or body before interaction can be detected. Device-free approaches that recognize gestures without requiring markers are limited either in the size of the area they can cover, their ability to track objects in 3D, the number of simultaneous users or the number of computers they can target.

Extending a computer's display area using additional displays can be accomplished using the built-in graphics hardware or using software. The graphics hardware requires tethering the displays and limits the number of available pixels. However, the performance is good. Software approaches can remove the tethering requirement, but are either limited to mirroring an already existing display, limit the number of displays, limit the number of pixels or do not extend the display area in a transparent manner to applications. Their performance is lower than using the built-in graphics hardware.

This dissertation presents two concepts: (i) Multiple Interaction Spaces; and (ii) the Pixel Space. An interaction space is a volume within which interaction is detected. It is not bound to any given user or computer. Instead, the interaction space is shared between users and computers. The size of an interaction space is variable, and interaction within it is detected in up to three dimensions. The space can be used alone or in complement with other interaction spaces. The Pixel Space is a collection of pixel resources on which computers can display pixels. Through the pixel space, cross-platform sharing of pixel output can be realized. Pixels are made available over a network, and are shared between computers.

Several systems are realized based on these two concepts. Three interaction space systems are built using cameras and microphones to detect and locate objects and sound sources: (i) Camera-sense: 16 cameras and 8 computers detect objects and determine their location and extent in 3D; (ii) Snap-detect: 4 microphones and 1 computer detect snap- and clap-like sounds and determine their location in 2D;

(iii) Arm-angle: A single, steerable camera is used to determine the angle at which a user's arm or other straight object is pointing. Together, these systems realize device-free, marker-less gesture- and sound-based interaction in two and three dimensions for arbitrary applications on a display wall. Several applications are built or modified to utilize input from the systems, including an image viewer, a genomic microarray visualization, a virtual billboard and two games.

Two pixel space systems are built and used with up to 30 displays and computers: (i) The 22 megapixel laptop system extends a computer's display area by utilizing 30 network accessible displays, ranging from handheld displays to a display wall; (ii) the De-centralized VNC (DVNC) system improves pixel sharing performance on tiled display walls by delegating some pixel distribution tasks from a VNC server to VNC viewers.

Three principles are formulated based on the concepts and systems developed: (i) Orthogonal interaction mechanism: The interaction mechanism is realized independently of the computers one wants to interact with, making the process of detecting interaction into a property of the environment; (ii) "where, not what:" Determining where something is, rather than what that something is, is sufficient to enable interaction; and (iii) pixels as network-available resources: The number of pixels available to a computer is determined by the environment, and not only its local pixel resources.

The Camera-sense system is evaluated by conducting experiments to measure its end-to-end latency, accuracy and precision. The system determines object locations with a latency of 113.66 ms, an accuracy of 1.24 cm and a precision of 0.72 cm. The two pixel space systems are evaluated by conducting experiments to measure their pixel update rate, bandwidth and CPU load. The 22 megapixel laptop can drive a single display at 25 frames per second, and a 22 megapixel display wall at 1.03 frames per second. DVNC improves the pixel update rate of VNC on display walls by a factor of 11.88 for certain update operations, while reducing VNC server bandwidth usage and CPU load. The bottleneck preventing better performance in both systems is a scarcity of local CPU and memory bandwidth resources.

Acknowledgements

It's been a wild ride.

I thank my advisor, Professor Otto J. Anshus, for his discussions, patience, persistence and continued motivation that kept urging me to build and research ever more interesting systems, and for his guidance, critical comments and insights in distilling the results into the papers that form the foundation of this dissertation. I also thank him for the numerous cups of coffee we have shared, and the many ideas we developed while drinking them.

I also thank my co-advisor Associate Professor John Markus Bjørndalen, for his ideas, discussions and numerous comments, and Professor Tore Larsen for his help and support. I thank Professors Kai Li and Olga Troyanskaya for inviting me to join them for a year at the Department of Computer Science at Princeton University, and for supporting my research with their continued enthusiasm and fruitful discussions. Thanks to their support, my camera-based interaction system is in use with an application supporting genomic researchers.

I thank my fellow graduate students, Tor-Magne Stien Hagen and Espen S. Johnsen, for their discussions and ample distractions throughout the work on my Ph.D. Their presence in the lab made the long days interesting when the systems – on very rare occasions – were not. I'm also grateful for the discussions I've had with Lars Ailo Bongo both in Tromsø and in Princeton, and my conversations with Karl Gunnar Aarsæther which have been a source of inspiration and ideas. I'd like to also thank my friends and family for supporting me throughout the work on my dissertation.

I deeply acknowledge the help and assistance I have received from the technical and administrative staff at the Department of Computer Science in Tromsø: Jon Ivar Kristiansen, Kai-Even Nilssen, Maria Wulff Hauglann, Svein Tore Jensen and Jan Fuglesteg. I would especially like to acknowledge the efforts made by Ken-Arne Jensen, which range from drilling holes in the wooden camera mounts in Tromsø, to building strips of infrared LEDs by laboriously soldering several hundred individual LEDs, and for fixing the coffee machine on more than one occasion.

I gratefully acknowledge the funding I have received from the Norwegian Research Council as part of the following projects: (i) 159936/V30, SHARE - A Distributed

Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices; (ii) 155550/420, Display Wall with Compute Cluster.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

NAD	Network Accessible Display
VFB	Virtual Framebuffer
VNC	Virtual Network Computing
DVNC	De-centralized VNC
RFB	Remote Framebuffer Protocol
PPI	Pixels per inch
IR	Infrared
LED	Light Emitting Diode
FTIR	Frustrated Total Internal Reflection
DI	Diffuse Illumination
MPx	Megapixel (1000^2 pixels)
GPx	Gigapixel (1000^3 pixels)
FPS	Frames per second
FOV	Field-of-view
MB	Megabyte (1024^2 bytes)
GB	Gigabyte (1024^3 bytes)
EDID	Extended Display Identification Data
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
VRAM	Video RAM
1D, 2D, 3D	One, two and three dimensional
VGA	Video Graphics Array (analog display connector)
DVI	Digital Visual Interface (digital display connector)
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SDL	Simple DirectMedia Layer
RGB, BGR	Red-Green-Blue or Blue-Green-Red (order of color components in a pixel)
AD	Analog-digital

FIR	Finite Impulse Response
RLE	Run-Length Encoding
DSD	Display Sharing Daemon
MAC	Media Access Control (Ethernet or BlueTooth hardware address)
API	Application Programming Interface
ID	Identity
m	Meter
cm	Centimeter
s	Second
ms	Millisecond

Chapter 1

Introduction

The research presented in this dissertation is motivated by the assumption that displays will become ubiquitous and at some point cover almost any surface. As the number of displays keep growing, the collection of pixels from each display form a larger and larger pixel space. Making use of the pixel space requires that one can interact with it, and that computers can provide it with content. One environment that already provides a large amount of pixels is created by a display wall. A display wall is a wall-sized, high-resolution display. The resolution of a display wall currently ranges from about 10 to 286 megapixels [1, 2, 3], with the highest-resolution wall measuring approximately ten meters wide by four meters high [3] – about the size of a large cinema screen. Since current displays are not manufactured with neither the resolution nor physical size demanded by a display wall, display walls are constructed by tiling a set of flat-panel displays or projectors. Due to the large number of pixels, each display is driven by a separate computer in a display cluster [4], although more compact configurations are possible using multi-head graphics cards. Display walls are described in more depth in Chapter 2.

Interacting with displays of this size can be accomplished either with or without devices. Device-based approaches for interaction include using mice, keyboards, gloves [5, 6], the Nintendo Wii Remote (Wiimote) [7], or gesture recognition using markers [8]. These approaches are restrictive and impractical when used to interact with display walls. A mouse usually requires a table surface to work, and wireless keyboards are impractical to carry around. For public installations, devices are problematic, as they may get lost, stolen or have their batteries run out. Some users might also be averse to directly touching a possibly dirty input device or dirty public display. Devices further do not scale to many users, since each user must have his own device. The battery problem can be resolved using passive devices fitted with markers to enable their tracking [9, 10].

Active or passive markers are also used to enable gesture recognition. Some examples include the gesture-based interface developed by Oblong Industries, where

users must wear gloves fitted with markers [5], a multi-touch tracking system built using a Wiimote [7] and the colored markers used to detect interaction in the "Sixth Sense" project [11]. Device-free approaches employing marker-less gesture recognition are either limited to tracking in 2D [12, 13] or to performing gesture recognition in 3D based on a depth map of the scene [14, 15, 16]. All of these systems limit the size of the area users can interact with and the number of supported users, and are further limited to interacting with a single computer at a time.

A computer's display area can be extended with additional displays, either by connecting the displays directly to the computer's graphics hardware, or by extending the display area in software. Using the graphics hardware, displays must be physically tethered to the computer, and the number of displays and pixels are limited by the graphics card's available display cable outlets and the hardware's capabilities. However, the resulting performance is generally good with high frame rates. Using software, the display area can be extended to additional displays without directly tethering the computer to each display, at the cost of lower performance. Existing solutions, however, do not provide a transparent way of accomplishing this [17, 18], limit the maximum resolution [19] or are limited in the number of additional displays they can support [20].

This dissertation presents two concepts: (i) Multiple Interaction Spaces; and (ii) the Pixel Space. An interaction space is a physical volume in which interaction can take place and be detected. The interaction space is not bound to a given user or a given computer. Instead, an interaction space can be used by several users simultaneously to interact with one or a collection of computers at the same time. The size of an interaction space is variable, covering small to large volumes, and interaction can be detected in one, two or three dimensions depending on the underlying hardware and software implementation. An interaction space can be used alone or in complement with other interaction spaces.

The pixel space is a collection of display-backed pixel resources shared on a network that can be utilized by nearby computers to show pixels on their behalf. The pixel resources enable computers to share visual data amongst each other and across platforms. The displays that are part of the pixel space vary in number, resolution and pixel density, and range from handheld displays to tiled display walls.

Using these concepts, three interaction space systems and two pixel space systems have been built. The interaction space systems, discussed in Chapter 4, comprise:

Camera-sense: A system to detect multiple objects, typically but not limited to hands or fingers, simultaneously in front of a display wall using 16 cameras and 8 computers [21, 22, 23] (Figure 1.1). The system determines an object's location and extent in 3D without requiring users to carry devices or wear markers, resulting in a skeleton three-dimensional representation of the detected object. The resulting 3D objects are used to enable multi-touch or gesture-based interaction with applications on a display wall. The system's

latency, accuracy and precision are evaluated. The system's end-to-end latency is 113.66 ms, its accuracy is 1.24 cm and its precision is 0.72 cm.

Snap-detect: A system to detect specific sounds and determine the source location of the sounds in 2D using four microphones connected to a single computer [24]. The system detects the source location of snap- and clap-like sounds, typically from users snapping their fingers or clapping their hands. The resulting 2D sound locations are mapped to a display wall, and interpreted by applications for different purposes like moving windows or zooming in and out of images.

Arm-angle: A system to detect the angle of dominant, straight lines within a steerable camera's field-of-view [24]. The camera is pointed towards a user. Images from the camera are then analyzed to determine the angle of dominant lines, under the assumption that a user's straight arm will be detected as the dominant lines. This enables the system to determine the angle at which a user's arm is pointing, which is then used to select items on a display wall that would otherwise be out of the user's reach. One example is to select an object on the far left side of the display wall, when the user himself is standing on the far right side of the wall.

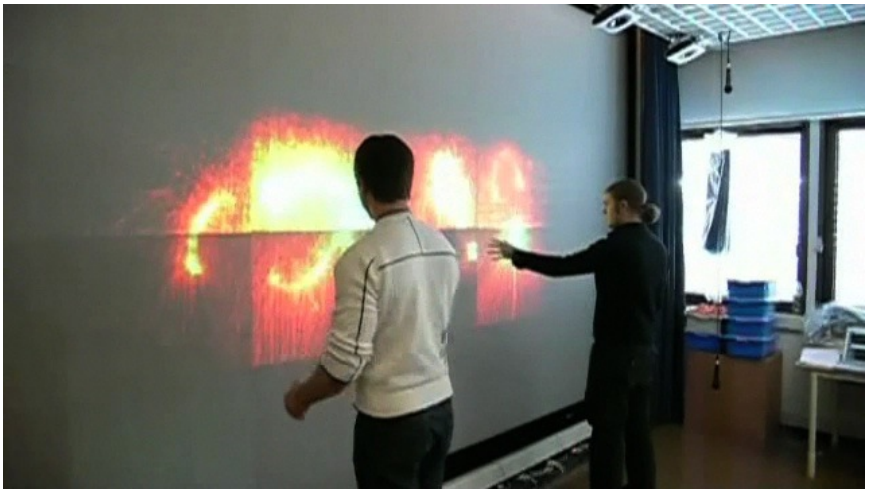


Figure 1.1: Using the Wallfire application and the Camera-sense interaction space to "paint fire" on a display wall.

Several applications have been implemented to utilize input from the different interaction spaces. Figure 1.1 shows an application demonstrating the use of the Camera-sense interaction space. Figure 1.2 shows the Wallview image viewer displaying all the pages of a late draft of this dissertation on a display wall. MASpace, a visualizer for genomic microarray data on display walls has been implemented in collaboration with researchers at Princeton University, and is currently in use by computational biologists for studying the relationship between different genomic

datasets [21]. Additional applications have also developed. All the applications are listed in Section 1.4.3, and further discussed in Chapter 5.

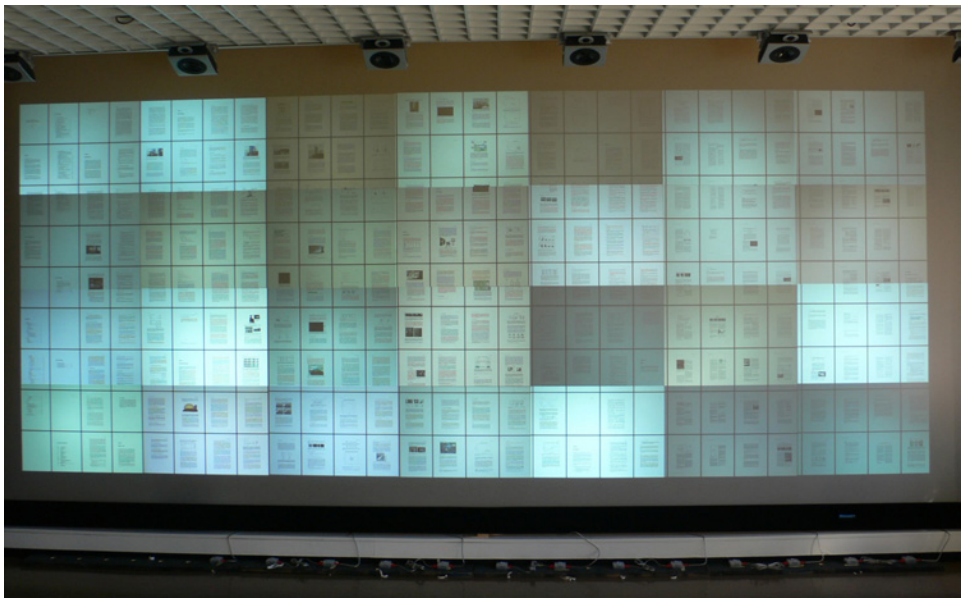


Figure 1.2: Using the Wallview application to visualize every page from a late draft of this dissertation on a display wall.

The Pixel Space is discussed in Chapter 6. Two pixel space systems have been built, based on two models also developed as part of this dissertation. The two models are: (i) The Network Accessible Display (NAD) model; and (ii) the De-centralized VNC (DVNC) model. The NAD model was developed to enable use of the pixel resources in the pixel space. The model assumes that eventually all displays will have some computational and network resources built-in. Using these resources, a display's pixels can be made available over a network, for other computers to utilize.

The DVNC model is based on the existing VNC (Virtual Network Computing) [17] model, but modified to improve performance when using VNC with a tiled display wall. The display wall's tiled architecture makes the VNC pixel sharing system perform sub-optimally. By enabling VNC viewers to exchange pixels amongst each other, the VNC server's load and bandwidth usage is reduced by delegating a VNC update operation from the VNC server to the VNC viewers. Based on these models, two systems have been implemented:

The 22 megapixel laptop: A system based on the NAD model. The system enables a laptop to utilize the pixel resources made available by displays ranging from handheld to a display wall, by creating virtual displays locally, and then push the pixels from the virtual displays to the remote NADs [25]. The resulting system can update a single NAD at a rate of up to 25 frames per

second, and 28 NADs on a 22 megapixel display wall at a rate of 1.03 frames per second.

De-centralized VNC: A system based on the DVNC model. The system modifies an existing, open-source VNC implementation [26], changing both the server and viewers to incorporate the ability for the VNC viewers to exchange pixels amongst each other. The resulting system improves the performance of the unmodified VNC implementation by a factor of up to 11.88 [27].

The 22 megapixel laptop and DVNC systems document how the resources of the pixel space are collected, utilized and shared amongst computers. DVNC further demonstrates how the pixels on a display wall can be kept up to date with improved performance by delegating work from the VNC server to a set of VNC viewers.

1.1 Display ubiquity and the pixel space

Since the invention of the transistor in the middle of the 20th century [28], technology has continued to evolve at a rapid pace. In the three years from 2000 to 2003, the amount of information in the world doubled [29]. Computers have diversified from the single-purpose, room-filling computers of the 50s and 60s, into computers spanning everything from battery-powered, portable computers to petaflop clusters [30]. Moore's law [31] states that the transistor count on a die doubles every 18-24 months. This exponential growth has resulted in an increase in processing speed, network bandwidth, storage and memory capacity by several orders of magnitude in the time since Moore stated the law in 1965, while at the same time making the constituent components diminish ever more in size and cost.

Failing to track Moore's law, display resolution and pixel density have not seen the same growth. 0.76 megapixel displays were available in the eighties, with the Three Rivers PERQ's 768x1024 display as one example [32], and started gaining widespread use during the early nineties. As of late 2008, the highest resolution commodity display available supports 2560x1600 pixels, or roughly 4.1 megapixels[1] [33]. With an increase in resolution and pixel density by factors of five and two[2], the growth over the past thirty years is underwhelming.

Instead of an exponential increase in resolution and size, displays have gone from being bulky and relatively uncommon, to becoming small, power-efficient, cheap and pervasive. Combined with ever cheaper, faster and smaller components, this miniaturization has enabled new classes of devices. Displays now exist everywhere

[1] While there exist even higher resolution displays, they are not very common and generally very expensive. Examples include the ViewSonic 3840x2400 display, introduced in 2004, and the NEC MultiSync MD205MG grayscale display introduced at the end of 2008 geared towards medical imaging with resolution at 2560x2048.

[2] The PERQ had a 8.5x11 inch 768x1024 display in 1980, which amounts to 90 pixels per inch (ppi). The Apple iPhone has a ppi of 163 and the Amazon Kindle a ppi of 167.

in different form factors. Mobile phones, music players and portable gaming systems all carry a tiny display. Laptops are more popular than ever, enabling users to carry not only a small computer with a one to two megapixel display, but also their applications and data around everywhere. Large displays are already ubiquitous, from the flat-panel TV sets in people's homes, to plasma displays in shop windows. Projectors are becoming cheaper and smaller, with the tiniest projectors on the market fitting in the palm of one's hand [34]. Large companies have demonstrated a range of displays that give an idea of where the world is going, with Sony's super-thin, bendable, foldable displays [35], LG.Philips' low-power, low-weight e-paper [36], and E Ink's paper-thin displays with built-in touch sensing [37]. The Amazon Kindle 1, 2 and DX e-book devices are examples of commercialized low-power displays [38].

At some point in the future, pixels may line the walls of people's homes, cover tables, shelves and other objects. Each surface becomes a display of its own, capable of displaying pixels and accepting user input. Walls lined with pixels are not close to reality in the consumer space yet, but a range of custom display wall installations already exist [1, 2, 39, 40]. With display walls covering the walls in users' homes or in public places, the future pixel space surrounding a user at any given time would accumulate a resolution on the order of gigapixels. This makes research both on ways to drive the pixels and interact with the pixels in the pixel space important.

1.2 Interaction Spaces

The computer mouse turned 40 on December 9., 2008 [41]. Since its public debut during the "mother of all demos" at the Fall Joint Computer Conference in San Francisco in 1968, it and the keyboard have become the de facto input devices to almost every personal computer shipped since the early eighties. At the same time, a recurring dream in the realm of science fiction has been to interact with computers using nothing but thoughts, natural language and gestures.

In Asimov's Foundation series [42], scientists manipulate equations on a large display wall-like surface using only their minds. Natural language is used throughout the Star Trek science-fiction series and portrayed in Stanley Kubrick and Arthur C. Clarke's "2001: A space odyssey" [43, 44], and an example of gesture-based interaction was shown in the 2002 movie Minority Report [45]. New input devices like the multi-touch screen of the Apple iPhone, the Nintendo Wiimote [7], and ever-more ubiquitous availability of small web cameras attached to mobile phones, laptops and displays makes the dream of unencumbered natural interaction with computers seem more within reach now than ever before.

Mind control is a growing trend in the toy industry [46]. Mattel recently introduced Mind Flex [47], a game where the participants control a ball using their mind as

the input device. Speech recognition is an established feature in both Apple Mac OS X and Microsoft Windows, with additional commercial alternatives available [48]. Gesture-based interfaces are becoming more common, as demonstrated by the sale of millions of iPhones [49], but so far no input devices have gained the same adoption as the mouse. The mouse and keyboard are not going away; their continued presence to this day is a testament to both their utility and a lack of better alternatives.

The iPhone's interface is almost entirely based on multi-touch gestures; the only exception is a few hardware buttons that among other things control sound volume and device power. Some laptops, like the Apple MacBook Pro, support multi-touch gestures on their trackpads. In this case, the gesture-capabilities are more limited, with the trackpad acting mainly as the pointing device in the regular keyboard/mouse combination. Tablets and smaller hand-held computers have had various kinds of touch- or pencil-style input for years [50]. More recently, Hewlett-Packard (HP) has been selling its TouchSmart range of computers, which uses a touchscreen capable of detecting multiple touches to provide an alternative to the keyboard and mouse. Microsoft is researching multi-touch input on the Surface [13] and TouchWall [51], and is expected to introduce OS-support for multi-touch in Windows 7 [52].

Gestures-based input can span devices and device-classes. One characteristic of the gesture-based interfaces of the iPhone and the TouchSmart range of computers is that their use does not require any setup on the user's part, with the possible exception of a calibration step. There are no devices to keep track of, no devices to carry around and no devices to lose.

1.3 Problem statement

Interaction with computers ranging from handheld to workstations is possible because each computer has the necessary interaction mechanism already built-in or available as external devices in very close proximity – such as the built-in touch screen of the iPhone, or the mouse and keyboard wired or wirelessly connected to a workstation. Since the interaction mechanism is available all the time, interaction can begin immediately when the decision to use one of these computers is made. In contrast, enabling interaction with a display wall is a challenge. A display wall does not have a built-in interaction mechanism, and it is not immediately apparent how and where to approach the wall in order to interact with it. A display wall's architecture is parallel, which makes applying existing input devices to display walls non-trivial. Input must be shared amongst the computers driving the wall, but a device can only connect physically to one computer at a time. Even if this problem were solved, one would still be left with input devices, such as mice or keyboards, designed for regular workstations.

Movement of the user – and thus the interaction mechanism – is another challenge. With handheld computers, the computer moves with the user as he moves around, which has the side-effect of moving the interaction mechanism as well. Laptops have the same property, since the interaction mechanism is built-in to the laptop and thus goes with it wherever the laptop goes. Workstations are usually not moved around while they are in use, even though their input devices may be moved occasionally. For instance, moving a mouse is intrinsic to its operation. In contrast, the combination of the display wall's large size and high resolution makes it possible for users to walk around in front of the wall to study fine details, or step back to get an overview of the contents being displayed. This makes tethered devices impractical in a display wall context, since they limit the movement of users in front of the wall to the length of the device's cable. Wireless devices remove the tethering issue, but the size and weight of possible input devices are still limited to what users can reasonably carry and operate at the same time. Some devices may not work well without a table for support, such as most mice[3] and keyboards. When interacting with a display wall, the user's physical location also becomes significant, while most input devices have been designed for location independence.

Regular workstations are rarely used by more than one user at a time, due to lacking application and hardware support, and the small physical size of the attached displays. Most applications are designed for a single user at a time. Systems and applications that support input from more than a single mouse are not common, although they do exist [53, 54]. Physical size also plays a role. Handheld computers can not practically be shared by two users, and there is usually limited space around a regular size display for several users to interact simultaneously. The display wall's large size changes this, providing sufficient room for several users to stand close to the wall and potentially interact at the same time. However, additional users interacting simultaneously requires additional input devices, as well as changes to the underlying applications to accommodate multi-user and multi-cursor input. As the number of input devices grow, they become increasingly hard to keep track of.

Instead of applying devices to interact with a display wall, the display wall's interaction mechanism can be gesture-based, using the same interaction paradigm as employed by the iPhone, TouchSmart and Microsoft Surface. However, current approaches for gesture-based interaction with wall-sized displays either require users to wear markers [7], are limited to recognizing input only when users stand very close to the display wall (and in some cases further limited to only recognizing touch) [55, 13], or limit the size of the walls they can cover [56].

To employ a gesture-based interaction mechanism for display walls, the mechanism should be:

[3] Gyroscopic mice do exist, however, such as the Gyration Air Mouse, which do not require a table in order to move the cursor.

1. **Always available:** There should be no setup associated with interacting with a display wall, just as there is no setup associated with interacting with a handheld or regular computer.

2. **Unencumbered:** Users should not have to carry devices or wear markers to interact with a display wall.

Further, to accommodate the large size and high resolution of a display wall and go beyond the approaches that already exist on handheld and touch-screen computers, the interaction mechanism should further be:

3. **Multi-user:** Several users should be able to interact with the display wall at the same time. This requires that the interaction mechanism is able to receive multi-user input.

4. **Room-wide:** Users should be able to interact both when they are close to the wall, and further away. The interaction mechanism should be available "everywhere" within the same room as the display wall.

1.3.1 Pixel resources

The range of different hardware architectures, operating systems and applications being used by different classes of devices, from handheld computers, laptops, and workstations to display walls, makes sharing resources and data amongst them over a network a challenge. One common trait, however, is that they often have a built-in or external display. While the displays vary in size and resolution, nearly all of them use pixels to display visual data. The pixel resources represented by these displays have so far been shared primarily using remote desktop systems that only *mirror* the contents of one display onto another remote computer's display [17]. Instead, a computer can utilize the pixel resources afforded by such remote displays by *extending* its display area to encompass them, giving the computer a higher total resolution on which to display pixels. Existing systems that accomplish this limit the number and resolution of the remote displays [20, 19].

A display wall consists of multiple displays driven by multiple computers. The displays are usually arranged in a grid, and together represent a very large number of pixels. An application can utilize the display wall's shared pixel and compute resources in three ways: (i) Sharing the pixels generated by the application by transferring them to the display wall's computers, and have the computers draw them; (ii) sharing the rendering commands generated by the application by transferring them to the computers, and have the computers interpret them; or (iii) parallelizing the application to directly utilize the pixel *and* compute resources of the display wall. Each approach has different implications for the complexity of the application's design, implementation, portability across platforms, and for its performance, including frame rate, CPU load and bandwidth consumption.

Sharing pixels does not require changing an application's design or implementation. Since the representation of pixels is shared across displays, the approach is cross-platform. However, the performance of sharing pixels may be too low and the cost too high for some application classes, such as games, videos and animations.

Sharing an application's rendering commands may require changes to its design and implementation. The applications must be written targeting specific graphics libraries [18, 57, 58], which limits them to platforms where these libraries are available. This also limits the number of applications whose rendering commands can be shared. The approach can provide better performance by utilizing the display wall's graphics acceleration hardware. However, the performance may still not be acceptable for a given application or different application domains.

Parallelizing a sequential application is non-trivial and may involve smaller and larger changes to its design and implementation. Parallelizing all applications is not feasible, due to the amount of work involved in re-designing and re-implementing them, and further made more difficult since not all applications have their source code available. Different operating systems and hardware platforms also make cross-platform parallel applications challenging to write. However, parallelizing an application makes it possible to utilize not only the pixel resources, but also all the local compute and graphics acceleration resources on each computer [23, 59, 60].

To enable computers to utilize remote pixel resources, the approach should be:

1. **Cross-platform:** Computers should be able to: (i) Use the pixel resources on a range of computers, including handhelds, laptops and display walls; and (ii) share its own pixel resources with other computers.

2. **Transparent:** The user should be able to manage his desktop environment as usual, except with a larger display area at his disposal. Applications should be able to utilize the additional display area without any application-specific changes.

3. **Dynamic:** The collection of displays surrounding a user at any given time changes as the user moves from place to place. This should be accommodated without requiring that the user restarts his desktop environment or his applications.

Further, to accommodate the parallel architecture of display walls, the approach should provide:

4. **Structure:** The individual displays that comprise a display wall are arranged in a grid. While the displays are individual, the approach should view them as a collection of displays where each display's position in relation to the others is taken into account.

5. **Performance:** It is non-trivial to drive the large number of pixels on a display wall with high frame rates. To improve performance over existing ap-

proaches, the parallel architecture should be exploited.

1.4 Scientific contributions

This section lists the main contributions claimed by this dissertation, and describes the dissertation's impact. The contributions made are founded on a systems approach where architectures, designs and implementations are developed, before experiments are conducted to document the resulting systems' characteristics. The contributions made are organized into a set of *principles*, *models* and *artifacts*.

1.4.1 Principles

The following principles have been formulated based on the research presented in this dissertation:

Orthogonal interaction mechanism: The interaction mechanism that detects input from a user is realized independently of the computers the user wants to interact with. The mechanism is separate from and orthogonal to the computers it provides input to. It creates an interaction space that can span multiple computers as well as different classes of computers. The resulting interaction spaces can be used alone or in concert, and targeted towards a general or application specific domain.

"Where, not what:" In an interaction space, it is sufficient to determine where an object is, rather than what it is, to enable interaction. Existing interaction mechanisms work because they make assumptions regarding how interaction takes place. The principle is applied in the three interaction space systems presented in this dissertation. The use of multiple cameras in the Camerasense system is made possible since the processing associated with *locating* an object is simpler than the processing required to *recognize* an object. The Snap-detect system does not determine what produced the snap; instead the system only detects where the snap occurred, and on the assumption that a user made the sound, gives applications the ability to respond to the sound. The Arm-angle system does not try to detect arms, but instead looks for straight lines in images captured by a camera.

Pixels as network-available resources: The environment determines the number of pixels available to a computer. Since the environment is dynamic, so is the number of pixels a computer can utilize at any given time. Existing pixel sharing systems like VNC have shown that pixels are suitable for cross-platform sharing, but are restricted in that they only mirror existing pixels, and require the number of pixels to be pre-determined. The systems developed in this dissertation show that rather than just mirror existing pixels, the

area represented by the pixel resources can be used to grow a computer's available resolution by an order of magnitude.

1.4.2 Models

The following models have been developed:

The Network Accessible Display model: A model where displays are made accessible to other computers on a network by incorporating some network and processing capabilities into the display. Using this model, computers can extend the resolution of their built-in display with the pixel resources provided by nearby displays, increasing the total number of pixels available to the desktop environment running on the computer.

The De-centralized VNC model: A model that delegates some of the work usually done by the VNC server to the VNC viewers connected to it, giving the viewers responsibility for exchanging some pixels amongst each other when possible. This increases the rate at which pixels can be updated when the server is used to create the desktop environment for a tiled display wall.

1.4.3 Artifacts

The following artifacts have been produced:

1. **The Camera-sense system,** which creates an interaction space that detects and locates multiple objects to enable gesture-based interaction with display walls. The Camera-sense system is:
   - *Scalable:* The system can be scaled to cover narrow and wide display walls by varying the number of cameras and computers.
   - *Unencumbered:* Users do not need to wear markers to interact using the system.
   - *Device-free:* Users are not required to carry any devices to interact using the system.
   - *3D:* An object's location and extent is determined in 3D.
   - *Multi-user:* The system detects multiple objects simultaneously, from one or more users. It does not distinguish different users from each other.
2. **The Snap-detect system,** which creates an interaction space that enables users to interact with a display wall by snapping their fingers and clapping their hands from anywhere within the same room as the display wall.

3. **The Arm-angle system,** which creates a *movable* interaction space that enables users to select targets that are out of the user's physical reach on a display wall.

4. **The 22 megapixel laptop,** a pixel space system that enables a regular laptop to transparently utilize the pixel resources provided by a handheld device, a workstation and a display wall.

5. **A De-centralized VNC** implementation, which is documented to improve the performance of the original VNC implementation by a factor of up to 11.88 for some operations when used to create the desktop environment on a display wall.

6. **Several applications** that make use of the interaction space systems:

   - *Wallview,* an application to view many high-resolution images simultaneously on a display wall.
   - *Wallboard,* a multi-user application that makes it possible to bring content from the real world into the display wall's pixel space by briefly holding the object in front of the display wall.
   - *Wallfire,* an application that allows users to "paint fire" on the display wall using the Camera-sense system (Figure 1.1).
   - *Angle-snap,* an application utilizing all of the three interaction space systems to select windows far away, move them closer and then interact with them using hand movements.
   - *MASpace,* an application to visualize several genomic microarray datasets on display walls. The application utilizes the Camera-sense interaction space, and can also accept input from a version of the application developed for the Apple iPhone and iPod touch.
   - *Quake 3 Arena and Homeworld,* two existing games that were modified to utilize the Camera-sense and Snap-detect interaction spaces for input, and to run on a display wall with good performance. The purpose of the work was to document the Camera-sense system and show that it is possible to use the system to play games. Games were chosen since they are a class of application that requires low latency and accurate input.

In addition, a number of videos have been produced, which demonstrate various aspects of the systems and applications that have been developed. The videos are available on the disc accompanying this dissertation (the disc is further described in Appendix E); some are also available online:

Hybrid vision- and sound-based interaction on display walls: A video demonstrating the three interaction space systems, published in conjunction with

[24]. The video also shows the Wallboard and Wallfire applications, as well as the Quake 3 Arena game. The video is available online [61] and on the disc.

Three years of the comic "M": A video demonstrating the Camera-sense system being used to enable interaction with the Wallview application. Wallview is used to browse 950 strips – three years – of the Norwegian comic "M" [62]. The video is available online [63], as well as on the disc.

Device-free interaction spaces: A video demonstrating the 3D capabilities of the Camera-sense system. The video was published in conjunction with [21], and is available online [64] and on the disc.

Microarray visualization: A video that demonstrates the Camera-sense system being used to enable interaction with the MASpace application, as well as the MASpace application running on an iPod touch. Available on the disc only.

The 22 megapixel laptop: A video demonstrating the 22 megapixel laptop. Available on the disc only.

De-centralized VNC: Two videos that demonstrate the difference in performance between the original VNC implementation, and the De-centralized VNC implementation. The first video shows a playback of a trace of moving an image on the original VNC implementation, and the second video shows the same trace being played back on the De-centralized VNC implementation. Available on the disc only.

1.4.4 Impact

The concepts and systems researched and developed as part of this dissertation have had impact by: (i) contributing to the state-of-the-art in gesture-based systems and pixel sharing systems through peer-reviewed publications; and (ii) by being used to enable and support several different activities.

The following lists the different systems developed and their impact:

1. **The Camera-sense system:** Is in use by computational biologists at Princeton University to perform genomics research using the MASpace (Section 5.5) microarray visualization, which has also been developed as part of this dissertation. The system has been featured in the news [65], and also been used to support teaching university level courses, talks, presentations, outreach and demonstrations for the public.
2. **The Snap-detect system:** Like the Camera-sense system, the Snap-detect system has been used during outreach and demonstration activities, and received coverage in the news [65].

3. **The Arm-angle system:** Is used to demonstrate a device-free approach to select targets that are physically far away from a user, and in so doing documents a movable interaction space.

4. **The 22 megapixel laptop:** Has been featured in news coverage at CNET [66]. The 22 megapixel laptop [25] introduced the NAD model, which has since been used in further research by other members of the Tromsø display wall group [67].

5. **De-centralized VNC:** The DVNC system has driven the Tromsø display wall desktop environment for 2-3 years. It has been used to support a variety of applications and users, including meteorologists from Tromsø using the display wall to do weather forecasting [68]. It has also been used in activities like teaching, talks, presentations, outreach and demonstrations.

The following is a non-exhaustive list of some outreach activities and demonstrations in which the systems presented in this dissertation have been used or presented:

- 2005, 2006, 2007, 2008: Outreach and demonstrations during the Supercomputing-day at the University of Tromsø.
- 2005, 2006, 2007, 2008: Demonstrations for high-school classes.
- 2006: Demonstrations for Dell, Total Oil and Gas Company, Norwegian Research Council, National Center for Telemedicine, Candidata – the Computer Science alumni association at the University of Tromsø, a local e-science workshop and others.
- 2007: Best poster award for the poster titled "Hybrid vision- and sound based interaction," presented at the VERDIKT conference organized by the Norwegian Research Council in October 2007. Demonstrations for the Norwegian branch of the Fulbright Program, the Norwegian Meteorological Institute and others.
- 2008: Demonstrations during the Norwegian Science Week, for the Technology Transfer Office at the University of Tromsø, the Research and Development department at the Norwegian Meteorological Institute, and others.

1.5 Summary of papers

This dissertation builds on a collection of published, peer-reviewed papers that have all been written towards completion of the Ph.D. project presented herein. The full papers appear in Appendix A. This section gives a brief overview of the papers, organized by the topic they cover.

1.5.1 Interaction Spaces papers

Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays

This paper introduces the Camera-sense system, and demonstrates its use with the two games Quake 3 Arena [69] and Homeworld [70]. The two games are modified to accept input from the Camera-sense system, and had to be parallelized to run with high frame rates on a display wall. The purpose was to document the Camera-sense system's latency, and demonstrate that its latency was sufficiently low to meet the response time characteristics required by games.

Citation: Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays. In *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007*, pages 75–83, June 2007.

Revised: Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays. *Journal of Virtual Reality and Broadcasting*, 5(10), November 2008.

Lessons learned using a camera cluster to detect and locate objects

This paper further documents the Camera-sense system and presents a detailed evaluation of its end-to-end latency.

Citation: Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus. Lessons learned using a camera cluster to detect and locate objects. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 71–78. IOS Press, 2008.

A System for Hybrid Vision- and Sound-Based Interaction with Distal and Proximal Targets on Wall-Sized, High-Resolution Tiled Displays

This paper introduces the Snap-detect and Arm-angle interaction space systems, and documents how they along with the Camera-sense system can be used to enable interaction with both far-away and nearby targets on a display wall.

Citation: Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. A system for hybrid vision- and sound-based interaction with distal and proximal targets on wall-sized, high-resolution tiled displays. In *Proceedings of the IEEE International*

Workshop on Human-Computer Interaction 2007, volume 4796 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2007.

Blurring the line between real and digital: Pinning objects to wall-sized displays

This paper introduces the Wallboard application, and demonstrates how the (at the time) rudimentary 3D capabilities of the Camera-sense system can be used to bring content from the real-world onto the display wall by emulating the actions one would take to pin a document to a billboard.

Citation: Daniel Stødle and Otto J. Anshus. Blurring the line between real and digital: pinning objects to wall-sized displays. In *IPT/EDT '08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologies*, pages 1–5, New York, NY, USA, 2008. ACM.

Tech-note: Device-Free Interaction Spaces

This paper documents and describes how the Camera-sense system is extended from only being able to track objects in 2D, to tracking objects in 3D using a skeleton 3D object representation. It also introduces the Camera-sense system's use with the custom genomic microarray visualization, and documents the system's accuracy for locating stationary objects in a single 2D plane.

Citation: Daniel Stødle, Olga Troyanskaya, Kai Li, and Otto J. Anshus. Tech-note: Device-Free Interaction Spaces. In *3DUI '09: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 39–42, March 2009.

1.5.2 Pixel Space papers

The 22 Megapixel Laptop

This paper describes the Network Accessible Display model and its associated realization: The 22 megapixel laptop.

Citation: Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. The 22 megapixel laptop. In *EDT '07: Proceedings of the 2007 workshop on Emerging displays technologies*, pages 1–4, New York, NY, USA, 2007. ACM.

De-centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays

This paper describes the De-centralized VNC model and its associated implementation. The paper documents how the De-centralized VNC model improves the performance of VNC when it is used to create the desktop environment on display walls.

Citation: Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. De-Centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays. In *NIK '07: Norsk Informatikkonferanse*, pages 53–64. tapir akademisk forlag, November 2007.

1.6 Organization

The remainder of this dissertation is organized as follows. Chapter 2 details the design of current display walls, and some approaches to having applications output to a display wall. Chapter 3 details the methodology employed to conduct the research presented in this dissertation. Chapter 4 presents the work on building a set of interaction spaces for wall-sized, high-resolution displays, while Chapter 5 presents a number of applications developed or modified to use the different interaction space systems. Chapter 6 details two systems developed to drive the pixels of a display wall. Chapter 7 discusses the dissertation, and Chapter 8 draws conclusions based on the research presented. Chapter 9 outlines some areas of future work.

Appendix A contains a chronological listing of the papers on which this dissertation is built, including complete copies of the papers. Appendix B documents the Shout event system, Appendix C documents the network discovery mechanism used by the 22 megapixel laptop and DVNC, and Appendix D documents the memory bandwidth of the two servers used in the DVNC evaluation. Appendix E gives a brief overview of the contents of the CD-ROM accompanying this dissertation.

Chapter 2

Display walls

A display wall is the scientific instrument that this dissertation builds upon. Display walls are the closest approximation to the future's gigapixel-scale displays that can be built today. Two display walls have been used, one located in Tromsø, Norway and one located in Princeton, New Jersey, USA. This chapter gives an introduction to display walls, their construction and application models. It also details the specifications for the Tromsø and Princeton display walls, as well as the specifications for other hardware used in the research presented in this dissertation.

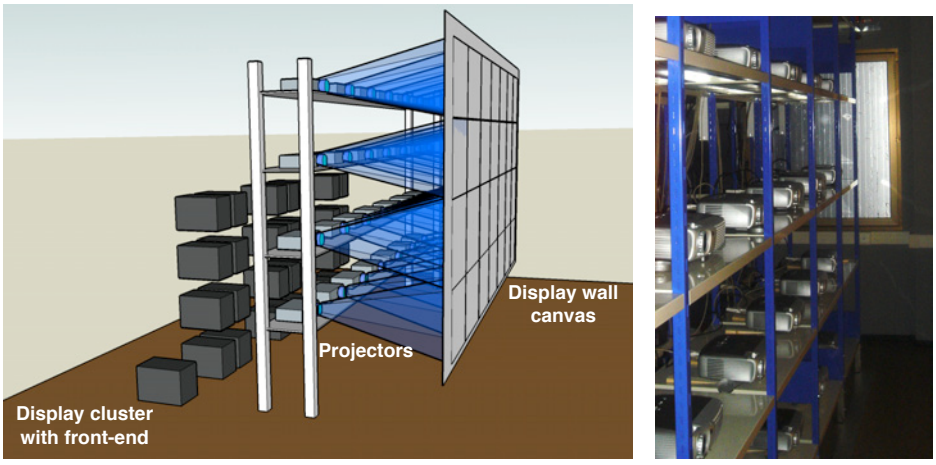
The Videoplace was an attempt in the late 1970s and early 1980s at creating an artificial reality by covering walls and tables with projected displays that respond to nearby users [71]. Research activity on display walls and large-format immersive display systems increased in the early nineties with the development of the CAVE [72, 73], a fully immersive display system where all the walls surrounding a user are active displays. CAVEs did not have very high resolution, on the order of 1-3 megapixels. The work on Tangible Bits [74] presents a vision of a world where every surface is active, along with a number of prototypes. The Interactive Workspaces project at Stanford [75] shared similar goals in trying to build a meeting space with many large displays integrated with portable devices. In 1995, the Infinity Wall[1] was shown for the first time [39], where the focus was on creating a large-scale, high-resolution projected display. Later, the focus shifted towards using commodity components to construct display walls, while at the same time aiming for higher resolutions and better performance [1].

Display walls are characterized by providing high resolution over a large area. Such high resolution displays are not available off-the-shelf. Instead, they must be constructed by tiling several smaller displays, typically flat-panel displays or projectors [1]. Each display is connected to the output from a graphics card mounted in

[1] The Infinity Wall was itself derived from the Power Wall and the earlier CAVE systems by the same research group.

a computer. Depending on the number of displays involved, the workload for driving each display can either be handled by a single computer using one or several multi-head graphics cards, or by dividing the workload between several computers in a cluster.

To avoid flicker, displays have a refresh rate of at least 60 Hz. To run the display at full frame rate, applications and the graphics card must co-operate to produce new pixels at the same rate as the display's refresh rate. The state-of-the-art NVIDIA GeForce GTX 295 is able to drive 8.1 megapixels in total, or two 2560x1600 displays at 60 Hz [76]. To maintain 2D and 3D performance, the card is equipped with an internal memory bus that delivers data at 223.8 GB/sec, about an order of magnitude faster than the memory bandwidth between the CPU and main memory in the current state-of-the-art Roadrunner petaflop supercomputer [30]. Due to limited processing power and bus bandwidth, a single computer is currently not able to drive a large collection of displays with acceptable performance. Instead, a parallel approach can be taken, whereby several computers cooperate to produce a coherent, high resolution image across the multitude of individual displays. This is the approach taken by most contemporary display walls [1, 77, 78], including one of the currently highest resolution display walls in the world: The HIPerSpace [2, 3]. The HIPerSpace has a resolution of 286 megapixels (35840x8000 pixels), and is constructed by tiling seventy Dell 30" displays in a 14x5 configuration. In total, the wall covers an area measuring approximately 9.66 by 2.34 meters. It uses 18 Dell workstations that in total have 72 CPU cores with two NVIDIA graphics cards to drive its 286 megapixels, interconnected using a 10 gigabit Ethernet.



(a) (b)

Figure 2.1: (a) An illustration of the display wall in Tromsø, with 28 projectors back-projecting onto a large canvas. The projectors are driven by a 29-node display cluster. (b) Some of the projectors driving the Tromsø display wall.

Since driving a gigapixel-scale display wall using a single computer is unfeasible

in the near-term, the research in this dissertation assumes that display walls are driven by a cluster of computers. Figure 2.1 illustrates how the Tromsø display wall is constructed.

The two most common display technologies suitable for tiling are off-the-shelf flat-panel displays and projectors. The advantages of tiling flat-panel displays are: (i) Flat-panels are cheaper than projectors compared to the amount of pixels they provide; (ii) they have a higher number of pixels per inch (ppi) – that is, more pixels packed in a small area; (iii) they are easy to tile; (iv) do not necessarily require external cooling; (v) are even in color reproduction and brightness; and (vi) do not require a separate canvas. The primary disadvantage of tiling flat-panel displays is the thick mullions – the borders – that appear between the displays when they are tiled. The borders around each display break the illusion of a large, continuous display, and instead leaves the user with the impression of looking through a mullioned window. Text that spans different displays becomes difficult to read.

While the mullions can be made less jarring in software by creating "virtual space" in between the different flat-panel displays (essentially imbuing the display wall with a greater total resolution than there actually are pixels) [79], a better solution is to tile an array of projectors, and have them rear-project on a canvas. Using rear-projection, the light path from the projectors to the canvas is not obstructed by users. The projectors can be hidden behind the canvas, giving the impression of a display floating on the wall, with no projectors visible to users. The main advantage of using projectors is that they can be tiled seamlessly [80, 81, 82, 83]. Disadvantages include the need for a separate, climate-controlled room to hold the projectors, additional noise (both from external cooling and internal projector fans), color and brightness differences (which to some extent can be mitigated through color correction [84]), limited lamp life, cost, the distance required from projectors to canvas, lower resolution and the more involved mounting required.

Commodity projectors can cover fairly large areas, but do not provide the same level of resolution as flat-panel displays. A typical commodity projector has a resolution of 1024x768 pixels. With such a projector covering a canvas of 3x2.25 meters, the resulting ppi would be about 8.6. In contrast, typical displays have a ppi ranging from 72 to about 130. The two display walls used in this dissertation have a ppi of 19 and 30, which while not astounding, is still far more than what is possible with a single commodity projector alone covering the same area.

2.1 Application models

There are three main application models that can be adopted to drive the pixels of a high-resolution display wall: (i) Centralized application logic and rendering; (ii) centralized logic and parallel rendering; and (iii) parallel logic and parallel rendering.

The first approach matches how most applications are written today, where the assumption is that the application runs on a single computer with a locally attached display. When applying this model for running applications on a display wall, the application's pixel output must be captured, and then streamed to the display wall, which then displays the pixels. In this case, the computational resources of the display wall cluster are not utilized beyond simply copying incoming pixel data into each tile's local framebuffer, making the central pixel source into a bottleneck in driving the display wall with good performance. Transferring pixels is also very costly in terms of network bandwidth usage.

The second approach is to have the application run on a single computer, but distribute its rendering to the individual tiles of a display wall. This relieves the central computer of the rendering load, without introducing the complexities inherent in parallelizing an application; the application logic and data can still remain on a single computer. The display cluster's resources are utilized more efficiently, since the cluster's graphics resources are used for more than just copying pixel data. However, the display cluster's CPU and memory resources still remain under-utilized, and the resulting performance may still not be acceptable for all applications.

The third approach is to parallelize both the application itself and its associated rendering to run directly on the display cluster. In this case, the resulting performance is ideally close to or better than[2] the performance one would get running the application on a single computer with a single display, while enabling the application to display one to two orders of magnitude more pixels than usual. The resulting applications are able to fully utilize the computational resources made available by the display wall cluster. The drawback is that applications often need to be either modified or substantially rewritten in order to parallelize them. This complicates the implementation and can be difficult to retrofit to already existing applications.

There are many existing systems that follow the first approach, including the 22 megapixel laptop [25] and De-centralized VNC [27] systems presented in this dissertation's Chapter 6. The second approach, used by the Wallboard application described in Chapter 5, is taken by systems like Chromium [57], WireGL [85] and Xdmx [18]. Chromium is a derivative of WireGL, and enables existing OpenGL [58] applications to use the display wall's rendering resources by substituting the normal OpenGL library with Chromium's implementation. No recompilation is necessary. Chromium redirects OpenGL commands to a set of slaves running on the display cluster, which in turn execute them locally. Each slave configures its viewport to match the tile on which it is running, resulting in each tile rendering a different part of the scene. Xdmx (Distributed, Multi-head X) enables existing X-based applications [86] to transparently utilize the displays on the display wall. Xdmx is a proxy X server. It redirects X commands from clients to X servers

[2] Parallelizing may make the performance improve, but due to synchronization and communication overhead, the resulting performance is not necessarily better or on par with the original sequential application.

running on the display cluster, modifying the commands as necessary to make applications treat the display wall as one, very large display.

The third approach is taken by several of the applications presented Chapter 5, including the Wallview image viewer, the MASpace microarray visualization, and the Homeworld and Quake 3 Arena games. CGLX (Cross-Platform Cluster Graphics Library) [59] is a system that also follows the approach of parallelizing applications to run on a display wall. CGLX lets OpenGL-based applications use a display wall by running parallel, synchronized copies of the application on each node in the display wall cluster. CGLX requires that applications are recompiled in order to use the distributed OpenGL context created by the system.

2.1.1 Display wall desktop environment

In contrast to the parallel architecture of existing display walls, traditional desktop applications written for current window systems like those of Mac OS X, Windows or Linux are written with the assumption that they will run on a single computer connected to at most two or three displays. When a user has two or more displays, applications generally leave it to the window system to abstract the details of display management, and treat the displays as a large, continuous surface. This way of handling displays is not well-suited to the parallel display wall architecture. However, there is a constant tension between the need for backwards compatibility and forward progress. Having a way to make existing applications run on a display wall is important to make "the long tail" of existing desktop applications available to users. Due to the parallel architecture of most current display walls, it is not possible to simply take an existing application and "run it on the display wall."

Further, it is in practice impossible to rewrite or parallelize all existing applications to run on a display wall. A large number of applications ship without their source code readily available, and the sheer effort of porting applications would be too much to reasonably undertake. However, people still use these applications – such as web browsers or text editors – so being able to run them on a display wall is important. One way to accomplish this is by creating a traditional desktop environment on the display wall wherein existing applications can run as normal, albeit with lower performance compared to a desktop environment spanning just a single computer.

To achieve this goal, one can use the window system's remote desktop features[3] to bring a replica of one's current desktop up on the display wall or using "desktop projector" software such as the one presented in [67]. While this might be sufficient

[3] Mac OS X 10.5 ships with built-in support for sharing a user's desktop using VNC [17]; similarly, Microsoft Windows has long had support for sharing the user's desktop using Microsoft's Remote Desktop Protocol [87]. On Linux, desktops and applications can be shared using a variety of techniques, ranging from X server based mirroring [88], to moving applications between X servers [89] and to plain VNC.

in some cases, it fails to utilize the added resolution and pixel resources afforded by the display wall; instead, one remains constrained to the resolution provided by the host computer. Thus, the benefit of the display wall's order-of-magnitude higher resolution is lost.

Another approach is to use either Xdmx [18] or Virtual Network Computing (VNC) [17] to create a desktop environment with the full resolution of the display wall. Informal experiences with Xdmx prompted the choice of VNC to drive the desktop environment on the Tromsø display wall, due to Xdmx' comparatively lower performance. The desktop is created using the Xvnc component of RealVNC [26]. The Xvnc component is a combined X and VNC server. It is built as a regular X server [86], but instead of driving a real display, creates a virtual framebuffer which is then shared using VNC. Regular X-applications can then use the Xvnc server's virtual framebuffer as any other X display, with the caveat that some modern features like hardware 3D acceleration and overlays are not supported. This way of driving the display wall follows the first approach outlined earlier, where a centralized application sends pixels to the display wall tiles, which in turn only receive and display them.

The VNC server shares the framebuffer using the Remote Framebuffer (RFB) protocol [90] with one or several viewers. A viewer requests the region of the VNC server's framebuffer it wants to receive updates from. The viewer only receives updates from the server that affect the region it is covering. The viewer keeps requesting updates, and the server responds in turn until the viewer closes the connection.

Figure 2.2 illustrates how VNC is normally used, and how this changes when using VNC to drive a display wall. The normal way to use VNC is to use it as a remote desktop system, where a user accesses his computer remotely by connecting a VNC viewer to a remote VNC server. The viewer displays the remote server's associated framebuffer, which can either represent a real display or a virtual one, with the latter being the case with Xvnc. When using VNC to provide a desktop environment for a display wall, there is no longer a single viewer that connects to the server and displays the entirety of the remote server's framebuffer. Instead, one viewer runs on each tile of the display wall, and requests a smaller region of the remote server's framebuffer to match the pixel area the tile displays. Further, the server is no longer a "remote" server – instead, it is located on the same, local network.

When using VNC to drive a display wall, the VNC server's framebuffer is given a resolution that matches the total resolution of the display wall. To drive the Tromsø display wall, Xvnc is configured with a resolution of 7168x3072 pixels, since there are 7x4 tiles with each tile having a resolution of 1024x768 pixels. The color depth used is 16-bits, to keep the performance of the system acceptable. To show the server's framebuffer on the display wall, each tile is configured to run a single viewer filling that tile's entire display. The viewers are further configured to limit the region of the remote framebuffer that they access, so that they request

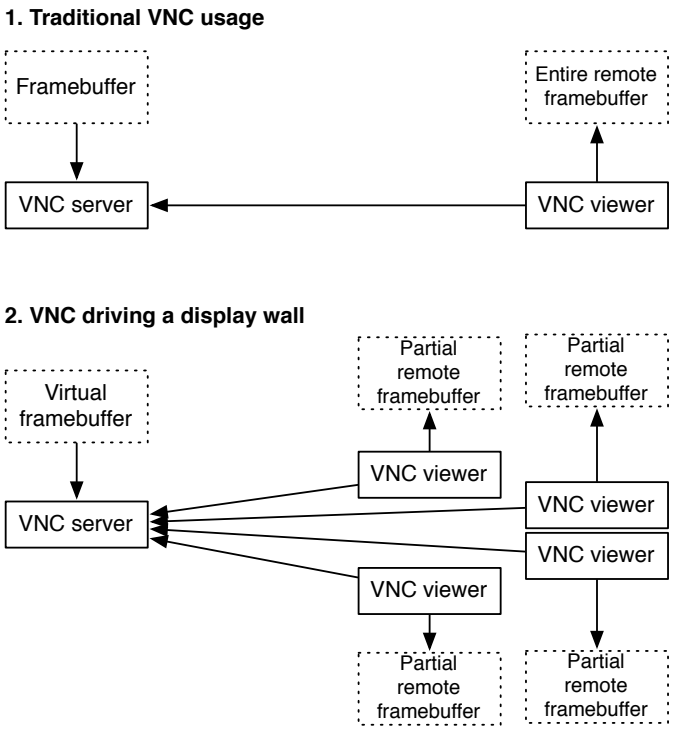


Figure 2.2: (1) Traditional use of VNC. A single viewer connects to the server, and displays the server's entire framebuffer. The framebuffer usually represents a real display, but a virtual framebuffer can also be created. (2) When using VNC to drive a display wall, two changes occur: (i) the VNC server's virtual framebuffer is made large enough to match the display wall's resolution; and (ii) each tile on the display wall request only the relevant portion of the VNC server's virtual framebuffer.

only the part of the remote framebuffer that corresponds with the tile on which they are running. For instance, the top-left corner tile would display 1024x768 pixels, offset 0x0 pixels into the remote framebuffer, while the bottom-right corner tile would display the same number of pixels, but offset 6144x2304 pixels into the remote framebuffer.

In [91], a different approach for creating a large desktop environment is presented. Using a 3D compositing window manager, knowledge of the geometry of the different displays being used and Xmove [89], windows from one display can be moved to other displays driven by other computers. However, this architecture prevents windows from spanning more than one display driven by separate X servers, which makes it unsuitable for systems with a large number of displays driven by separate servers such as the Tromsø display wall.

The 22 megapixel laptop can also be used to create a desktop environment on display walls, as shown in Section 6.3, much like the VNC approach. The difference is that the 22 megapixel laptop is able to transparently integrate the laptop's built-in

display with the virtual displays it creates to match the available network accessible displays. However, the performance of the VNC-based approach is better than the 22 megapixel laptop due to the modifications made to de-centralize it, as presented in Section 6.4.

2.2 Tromsø and Princeton display walls

To conduct the research presented in this dissertation, two different display walls have been used, listed in Table 2.1. The first is located at the Department of Computer Science at the University of Tromsø, Norway. This wall, which is referred to as the Tromsø display wall, is comprised of 28 Dell 4100MP projectors with a resolution of 1024x768 pixels. The projectors are arranged in a 7x4 grid, yielding a display with a total of 7168x3072 pixels in resolution (22 megapixels). The projectors back-project onto a canvas, resulting in a 6x3 meter, 220" display. The Tromsø wall provides a resolution of 30 ppi.

Where	Resolution	Geometry	PPI	Size	Network	Hardware
Tromsø	7168x3072	7x4	30	600x300 cm	Switched, gigabit Ethernet	29 Dell Precision 370 workstations
Princeton	2048x1536	2x2	19	272x200 cm	Switched, gigabit Ethernet	4 Mac minis, 1 workstation

Table 2.1: Specifications for the Tromsø and Princeton display walls.

The Tromsø wall is driven using a cluster of 29 computers. 28 computers are connected to one projector each, with the 29th computer acting as a front-end to the cluster. The computers run the Rocks cluster Linux distribution, version 4 [92]. The computers are all Dell Precision 370 workstations (Intel Pentium 4 EM64T at 3.2 GHz, 2 GB RAM, HyperThreading enabled, NVIDIA Quadro FX 3400 with 256 MB VRAM on a PCI Express x16 bus) and are interconnected using a switched, gigabit Ethernet.

The second wall is located at Princeton University, New Jersey, USA, and is referred to as the Princeton display wall. It consists of four Synelec projector "cubes," tiled in a 2x2 arrangement. It provides 19 ppi over a total resolution of 2048x1536 pixels, and measures 2.7x2 meters. Two instances of this display wall are present at respectively the Department of Computer Science and the Lewis-Sigler Institute for Integrative Genomics[4]. Four Mac minis (Intel Core 2 Duo at 1.83 GHz, 1 GB

[4]Only one of the walls are operational at a time, since the same computer and camera hardware is used for both.

of RAM, integrated Intel GMA 950 GPU with 64 MB shared VRAM) running Mac OS X 10.5.5 each drive one of the wall's four projectors, and are interconnected using switched, gigabit Ethernet. A Dell workstation (Intel Pentium D at 3.0 GHz, 2 GB of RAM, NVIDIA GeForce 7900 GS with 256 MB VRAM on a PCI Express x16 bus) acts as a front-end to the Mac mini cluster. The Princeton display wall was upgraded with the four Mac minis as part of the Ph.D. project presented in this dissertation.

2.2.1 Camera-sense system specifications

For the Tromsø wall, the Camera-sense system makes use of 16 Unibrain Fire-i cameras [93] using manual focus with specifications as shown in Table 2.2. The Princeton wall makes use of 8 Unibrain Fire-i cameras, with identical specifications but a new metallic grey casing, as compared to the translucent casing of the Tromsø cameras.

The Camera-sense system in Tromsø further makes use of 8 Mac minis (Intel Core Duo at 1.66 GHz, 512 MB RAM, integrated Intel GMA 950 GPU with 64 MB shared VRAM) running Mac OS X 10.4.9 to capture and process images from the cameras. In Princeton, the system makes use of the same Mac mini cluster as is used to drive the projectors.

Resolution	FOV	FPS	Depth	Bus	Trigger	Cost
640x480	42° horizontal, 32° vertical	30	8-bit grayscale[a]	FireWire 400	No	∼ $100

Table 2.2: Specifications for a Unibrain Fire-i web camera. FOV is the camera's horizontal and vertical field-of-view; FPS is the maximum camera frame rate; Depth is the bit-depth used to acquire images; Bus is the interconnect used to connect the camera to a computer; Trigger indicates whether several cameras can be triggered externally to acquire images simultaneously.

[a] The camera supports other color modes as well, however 8-bit grayscale is the mode that is actually used.

2.2.2 Snap-detect system specifications

The Snap-detect system uses four Behringer microphones connected to an AD converter (Behringer Ultragain Pro-8 Digital ADA 8000). The AD converter in turn is connected to a Hammerfall HDSP 9652 sound card, which is mounted in a Power-Mac G5 (Dual-processor PowerPC G5 at 2.5 GHz, 4 GB RAM, ATI Radeon 9600 XT with 128MB VRAM) running Mac OS X 10.4.11. The Snap-detect system is only installed at the Tromsø display wall.

2.2.3 Arm-angle system specifications

The Arm-angle system makes use of a Canon VC-C4R steerable "pan-tilt-zoom" camera connected to a framegrabber card (Philips SAA7134/SAA7135HL). The computer is a Dell Precision 370 workstation with specifications identical to the computers in the Tromsø display wall cluster. The Arm-angle system is only installed at the Tromsø display wall.

2.2.4 22 megapixel laptop specification

The 22 megapixel laptop is an Apple MacBook Pro (Intel Core 2 Duo at 2.33 GHz, 3 GB RAM, ATI Radeon X1600 with 256 MB VRAM) running Mac OS X 10.4.9. The NADs utilized include the projectors and workstations as part of the Tromsø display wall, and a Nokia N800 Internet Tablet running OS 2007.

2.2.5 De-centralized VNC server specifications

Two computers are used to run the VNC server in the evaluation of the De-centralized VNC system. The two computers are: (i) A Dell Precision 370 workstation with specifications identical to the computers in the Tromsø display wall cluster; and (ii) a Dell server (Dual-processor Intel Xeon at 3.8 GHz, 8 GB RAM, HyperThreading enabled, ATI Radeon 7000/VE with 32 MB VRAM on a PCI bus). Both computers run RedHat Enterprise Linux 4.

Chapter 3

Methodology

A systems approach is taken to conduct the research presented in this dissertation. Since the interplay between hardware and software is highly complex, it can not be fully understood through analysis alone. Instead, the interplay is characterized by conducting experiments that measure the aspects of a system one is interested in learning more about. The methodology to do this is to develop architectures, then design and implement actual, working systems. Experiments are then designed and conducted, with an emphasis on experiments whose conduct and results can be reproduced by other researchers to an as large extent as possible.

The results are then analyzed and ideally compared to the state of the art[1]. Through the process of constructing the system, a deeper and more detailed understanding of its underlying model, design, architecture and implementation is obtained, all of which contribute to explaining the system's characteristics and the results obtained through the experiments. On the basis of this knowledge and the experimental results, conclusions about the system are drawn. These conclusions can give rise to new questions that can be answered by further research, which restarts the cycle. The system is refined based on the lessons learned, and yet new experiments conducted.

The purpose of the experiments presented in this dissertation is to measure and document various performance aspects of the different systems that have been developed. The performance metrics used are: (i) Latency; (ii) accuracy and precision; (iii) CPU load; (iv) bandwidth; and (v) pixel update rate.

[1] In practice, making an "apples to apples" comparison between two systems is hard to achieve. Some reasons for this is that systems from the state of the art are not always open, and the hardware platforms used to conduct experiments usually differ.

3.1 Latency

Latency is measured in milliseconds. It characterizes the delay between some action happening, and its effect becoming visible to some other part of a system or to a user. Low latency is important in order for a system to appear responsive to end users. The methodology used to measure latency varies. One approach used is to measure the mean running time of a piece of code, to determine the latency incurred by that piece of code. To measure the time, the code is instrumented by surrounding the interesting block of code with calls to `gettimeofday()`[2]. This is illustrated in Listing 3.1. When measuring the latency incurred by a network, the roundtrip latency is determined by sending a packet containing a local timestamp (from `gettimeofday()`), and then waiting for a response containing a copy of the earlier local timestamp. This is the same way that the standard ping [94] program works. The roundtrip latency is then divided by two to arrive at the one-way latency.

```
external_event_occured:
  start        = gettimeofday()
  time_consuming_function()
  stop         = gettimeofday()
  mean_latency = (mean_latency + (stop - start)) * 0.5

time_consuming_function:
  .. do something time consuming ..
```

Listing 3.1: Pseudo-code demonstrating how the latency incurred by a block of code is measured. The code updates the mean latency every time an external event occurs.

3.2 Accuracy and precision

Accuracy and precision are measured in centimeters. The two metrics are used to characterize the accuracy and precision with which the Camera-sense system is able to locate object. Accuracy is the distance from a target location to an actual, observed location as reported by the Camera-sense system. Precision indicates the degree to which the Camera-sense system is consistent in locating an object at a given location, and is calculated as the standard deviation of several accuracy measurements.

The accuracy is measured by having users point at different targets, and then measure the difference between the target's location and the location as observed by the system. For each target, a number of measurements are made. The resulting accuracy is reported as the mean of the difference between the target locations, and

[2] All time measurements are made using the `gettimeofday()` system call on either Mac OS X or Linux. This call returns time as a tuple of seconds and microseconds since the beginning of Unix time (January 1. 1970).

the observed measurements. The mean can be calculated either for all targets, or for each individual target. The lower the mean, the better the system's accuracy – at least for that user. The system's precision for a given target is then calculated as the standard deviation of the difference between target and observations.

The accuracy experiments are the only experiments that directly involve users; this generally makes them harder to reproduce since their movements may not be entirely consistent across several experiments. However, the alternative would be to build a very elaborate "accuracy measurement machine" that could repeat the same movements over and over, which is outside the scope of this dissertation.

3.3 CPU load

CPU load is measured as CPU time, or a percentage thereof. The operating system is constantly scheduling different processes to run when they are ready. The CPU load for a process over a given amount of time is the amount of CPU time the OS has allocated to that process at both kernel and user level since the process began running. The CPU load is measured using the `getrusage()` system call, which returns the calling process' cumulative CPU time at kernel and user level. The function is called either once at the beginning and end of an experiment, or at regular intervals during the experiment (for instance, once every second), depending on the level of detail that is desired. Its output is logged along with a timestamp. When an experiment ends, the process' total CPU load as a percentage can be calculated as $\frac{CPU\_time}{end-start}$, where CPU_time is the CPU time returned by `getrusage()` at either kernel or user level, end is the last timestamp and $start$ is the first timestamp. Listing 3.2 demonstrates how the CPU load is recorded. A separate thread is used to record the amount of CPU time consumed by the process at regular intervals, and is otherwise sleeping, thus not incurring much overhead.

3.4 Bandwidth

Bandwidth is measured as the number of bytes transferred for the duration of an experiment, and then optionally calculating the mean number of bytes transferred per second. Measuring bandwidth helps determine whether the network connection between two parts of a system may represent a bottleneck, and may also aid in characterizing how changes to a system's architecture affects its bandwidth usage. The bandwidth measurements in this dissertation focus only on application level bandwidth, and does not consider the additional bandwidth required to manage the lower-layer protocols such as TCP or UDP. Listing 3.3 shows pseudo-code demonstrating how bandwidth is measured. Two counters are initialized to 0, representing the number of bytes sent and received. The values of the counters may be logged

```
start:
  start_cpu_load_thread()
  do_work()

do_work:
  .. do something ..

cpu_load_thread()
  while true:
    record_cpu_load()
    sleep(N seconds)

record_cpu_load:
  timestamp      = gettimeofday()
  resource_usage = getrusage()
  log(timestamp, resource_usage)
```

Listing 3.2: Pseudo-code demonstrating how the CPU load for a process is measured.

at regular intervals, or just the final values at the end of an experiment. As data is sent or received, the counters are incremented with the number of bytes sent.

3.5 Pixel update rate

Pixel update rate is measured as the number of pixels updated per second. The purpose of measuring the pixel update rate is to determine how quickly and how often a system can refresh its output on a display wall. By measuring an existing system's performance and then comparing it to the performance of a modified system, it is possible to determine if the modified system has increased its performance. By combining this metric with knowledge about CPU load and bandwidth usage, knowledge of bottlenecks and how they change in the modified system can be gained. Listing 3.4 shows how the pixel update rate is measured. As with CPU load and bandwidth, a counter keeps track of the total number of pixels updated for the duration of an experiment. As an application updates areas of the display, the number of pixels inside the rectangle being updated are added to the counter. The value of this counter is either obtained at the end of an experiment, or logged periodically along with a timestamp. Logging can be done using a separate thread or by incorporating the necessary logging code into the application's main loop.

An alternative metric is frames per second, which is the number of full display refreshes per second[3]. This metric is more intuitive to understand than the number of pixels updated per second, but is not always applicable. For instance, it does not make much sense to measure the frame rate of a text editor, since the text editor usually updates only a small part of the display and not necessarily at a high rate.

[3]Frames per second is also used to describe the rate at which images are delivered from the cameras used in the Camera-sense system to the computer they are attached to.

```
send_data:
  bytes = send(socket, buffer, bytes_to_send, 0)
  if bytes > 0:
    .. send ok, update counters and return bytes sent ..
    return bytes
  else:
    .. error condition or socket closed ..
    return 0


recv_data:
  bytes = recv(socket, buffer, max_bytes_to_receive, 0)
  if bytes > 0:
    .. handle new data ..
    return bytes
  else:
    .. error condition or socket closed ..
    return 0


do_something_bandwidth_consuming:
  in_bytes  = 0
  out_bytes = 0
  start     = gettimeofday()
  last_log  = 0
  while not done:
    out_bytes += send_data()
    in_bytes  += recv_data()
    if last_log + 1 < gettimeofday():
      last_log = gettimeofday()
      log(in_bytes, out_bytes, last_log)
  in_bandwidth  =  in_bytes/(gettimeofday() - start)
  out_bandwidth = out_bytes/(gettimeofday() - start)
```

Listing 3.3: Pseudo-code demonstrating how bandwidth is measured.

In these cases, a low frame rate could be interpreted as bad performance, when in reality it just indicates that the application is smarter about when and where to update the display. However, if it is known that the pixel update rate reflects full display updates, the two metrics can be used interchangeably. The pixel update rate (pixels/second) can be converted to frames per second by dividing the pixel update rate by the size of the display in question. For instance, a pixel update rate of 110 megapixels/second on the Tromsø display wall (which has a 22 megapixel resolution) would reflect a frame rate of $\frac{110}{22} = 5$.

3.6 Overhead of instrumentation

There will always be some level of overhead associated with instrumentation. Many of the statistics listed above are obtained through some level of instrumenta-

```
main_loop:
  total_pixels = 0
  start        = gettimeofday()
  start_log_thread()
  while true:
    .. wait for pixel updates ..
    received_pixel_update(pixels, rectangle)

received_pixel_update(pixels, rectangle):
  draw_pixels(pixels)
  total_pixels += rectangle.width * rectangle.height

log_thread:
  while true:
    log(total_pixels, gettimeofday())
    sleep(1)
```

Listing 3.4: Pseudo-code demonstrating how the pixel update rate is measured. Note that thread safety is not considered in the pseudo-code.

tion. No effort has been made to establish the cost of this instrumentation, however given the frequency at which the instrumentation code is invoked and the small amount of data it collects, the cost of the instrumentation is expected to be negligible. The overhead would be of greater importance if the experiments aimed at determining the exact cost of some operation with a very high degree of precision. For instance, measuring the execution time of a piece of code down to microsecond or better accuracy would require a more careful examination of the impact of instrumenting the code – such as cache effects, register spills and so on – than determining it with millisecond precision. The approach to measuring the execution time would also likely change, for instance to using the time stamp registers directly rather than the `gettimeofday()` system call[4]. Such a high degree of precision is not required for the experiments presented in this dissertation.

3.7 Definition of a megapixel

This dissertation defines one megapixel as one million pixels. In the papers detailing the 22 megapixel laptop[5] [25] and De-centralized VNC [27], a different definition was used, defining one megapixel as $1024 * 1024 = 1048576$ pixels. In presenting results from these papers, the results have been converted to the correct definition of a megapixel by multiplying existing data points by a factor of 1.048576.

[4] This in turn would introduce other issues, such as having to control the core that the process runs on, and compensating for any power saving measures the operating system should choose to implement.

[5] The name “the 22 megapixel laptop” has not been changed in accordance with this, however.

Chapter 4

Interaction Spaces

This chapter documents the Interaction Spaces concept, as well as the design, implementation and evaluation of three Interaction Space systems. This chapter is based on the following peer-reviewed, published papers: [23, 22, 24, 21, 95]. The papers have all been written towards fulfillment of the Ph.D. project presented in this dissertation.

Interacting with computers can be done with or without devices. Device-based interaction, using mice, keyboards or any other passive or active device, has several downsides when being used to interact with display walls. The large size and high resolution of display walls makes it possible to move around in front of the wall to explore a dataset, rather than sit passively behind a desk. The high resolution makes it possible to walk closer and see text and other fine details, or step back to see the larger context of a dataset. Some devices, such as keyboards, are sufficiently large that they can not be conveniently carried around or used without a table. A regular mouse does not work without a tracking surface. Another disadvantage is that devices don't scale to many users. Unless the users can agree on taking turns using the device to interact, supporting many users requires that each potential user has his own device. Devices may easily get lost, and in public settings they are more likely to get stolen. Since the devices have to be carried around, they must be wireless. An active, wireless device will require a battery, which must be replaced or recharged at regular intervals. There is also learning associated with devices, and some users may be averse to using public devices for hygienic reasons.

Interaction can also be accomplished without devices, by recognizing speech, gestures or other properties of the persons trying to interact. However, these approaches are not yet fully developed. Speech recognition is a challenging problem, and many gesture-based approaches require that users wear gloves [5, 6] or markers [7, 8]. The approaches that don't require the use of markers are limited to detecting input in 2D, gesture recognition in 3D based on depth maps, or in scalability. 2D-approaches do not provide depth information, and are limited to either

recognizing specific gestures [55, 96], or detecting multi-touch input on a 2D surface [12, 13, 97]. Most marker-less 3D approaches detect gestures by analyzing a depth map of a scene [14, 15, 56, 16]. Both 2D and 3D approaches are limited in scalability, since they restrict the area within which interaction can take place.

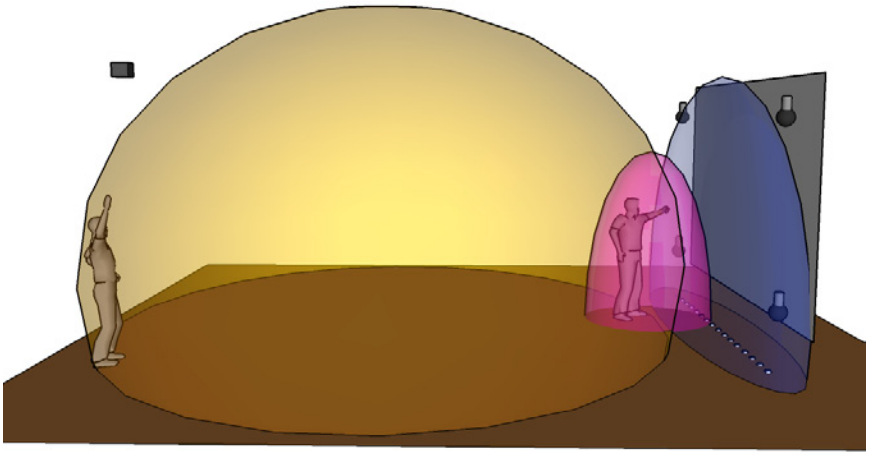


Figure 4.1: Interaction spaces are large and small volumes within which user input is detected. The interaction spaces illustrated above do not require that users use devices or wear markers. The smallest, magenta-colored interaction space is created using a ceiling-mounted camera at the back of the room. The medium-sized space in blue is created using a number of cameras along the floor in front of the display wall, and the largest, yellow space is created using four microphones. Credit for the human models: Google SketchUp.

Whether with or without devices, interaction is still restricted to a single computer, and often a single user as well. This dissertation presents the Interaction Spaces concept. An interaction space is a volume within which user interaction is detected. As illustrated in Figure 4.1, different interaction spaces can cover differently sized and overlapping areas. The interaction spaces exist orthogonally to the computers they act on, and can support one or several users at a time. The input mechanism is no longer bound to a single computer, but instead becomes a property of the space users occupy. An interaction space is always switched on, and may be used alone or in concert with other interaction spaces.

Three systems have been built based on the Interaction Spaces concept. Each of the three systems create a separate interaction space. Taken together, the three systems enable unencumbered, device-free interaction with display walls using both gestures and sounds detected and located in both two and three dimensions. The three systems are: (i) Camera-sense: A scalable, multi-user and multi-"touch"[1] camera-

[1]The system is touch-free in the sense that touching a surface is not necessary in order for inter-

based system that detects and locates objects in 3D; (ii) Snap-detect: A system capable of locating the origin of the sound made by a user snapping his fingers or clapping his hands in 2D; and (iii) Arm-angle: An on-demand camera-based system that recognizes the direction in which a user's arm is pointing.

System	Approach	3D	Multi-user	Volume	Fixed	Hardware
Camera-sense	Gesture	Yes	Yes	7 m^3	Yes	16 cameras, 8 computers
Snap-detect	Sound	No[a]	No	144 m^3	Yes	4 micro-phones, 1 computer
Arm-angle	Gesture	No	No	3 m^3	No	1 camera, 1 computer

Table 4.1: Characteristics of the three interaction space systems.

[a]Sounds are located in 2D, but the detected sounds can emanate from anywhere within a room-sized 3D volume.

The characteristics of the three systems are listed in Table 4.1. The columns are: (i) System: The system being characterized; (ii) Approach: Whether the system detects gestures or sounds to enable interaction; (iii) 3D: Whether the interaction space supports 3D detection; (iv) Multi-user: Whether the interaction space supports several users at the same time; (v) Volume: The approximate extent of the interaction space in cubic meters[2]; (vi) Fixed: Indicates whether the interaction space is fixed, or if it can be moved around; and (vii) Hardware: A very brief summary of the hardware used to create the interaction space.

The Camera-sense system creates the first interaction space in front of the display wall. In this space, the location and extent of objects such as hands, fingers, pens or other items is determined and subsequently interpreted by applications. It is the only space that can locate objects in 3D. The space allows several users to interact simultaneously. It is wedge-shaped and can not be moved, but is scalable in that larger areas can be incorporated into the interaction space by adding more cameras. A parallel approach is taken by utilizing several cameras and computers to create the interaction space.

The Snap-detect system creates the second interaction space in the same room as the display wall. The space detects the sound made by users snapping their fingers, clapping their hands or making other snap-like sounds, and determines the location of the individual sounds' origins in 2D. It can only detect a single snap at a time. However, due to the short duration of a snap or clap, it is possible for several users to interact within a few seconds of each other. The Snap-detect system's space has

action to be detected.

[2]The volume is estimated based on measurements of the Tromsø display wall and the room it is in.

the largest volume of the three interaction spaces, filling an entire room. However, the space is still fixed due to the use of fixed microphones to detect sounds.

The Arm-angle system creates the third interaction space. Like the Camera-sense system, the space is created in front of the display wall. Within the space, the angle of a user's arm is detected without requiring users to wear markers, giving applications information about where a user is pointing. It can only be used by a single user at a time. The space covers a smaller volume than the other two, by capturing the immediate surroundings of a given user. Unlike the other two spaces, the Arm-angle space is created on demand by utilizing the Snap-detect interaction space. By snapping his fingers, a user prompts the creation of the Arm-angle space around him, after which he may proceed to use the space. The space is thus not fixed like the other two, but movable. This is achieved using a steerable, ceiling-mounted camera to pan, tilt and zoom in on the user's location, with the Snap-detect system acting as an enabling technology to provide the user's approximate location.

The three interaction spaces are used to interact in different ways with a number of applications on the display wall. The Camera-sense space can be used to either recognize gestures directly using the location and skeleton 3D object representation detected by the space, or as a multi-touch gesture interface. Unlike many existing multi-touch input systems [12, 13, 97], the Camera-sense space does not require users to actually touch the display in order to interact. This has several advantages: The interaction mechanism is physically separated from what it acts on. This enables the display to be moved away from the user, for instance behind a shop window. Further, users need not worry about touching a possibly dirty public display, and conversely, the display itself does not get dirty from being repeatedly touched. For the case of the Tromsø display wall, the display wall's canvas itself is flexible. This makes touching it inconvenient, since the canvas inevitably starts to wobble in response to touch, which distorts the content being shown on the display wall. The Camera-sense space has been used to interact with a number of new and existing applications, including a custom image viewer for the display wall, a custom microarray visualization, and the two games Quake 3 Arena [69] and Homeworld [70]. The applications are further detailed in Chapter 5.

The Snap-detect space has been used in several ways. It is used by the display wall's desktop environment to allow window movement. When two snaps are detected (a "double snap"), the currently focused window on the display wall is moved to the average location of the two snaps. It is also used to control the aforementioned image viewer, where a double snap results in the image viewer zooming in on the location of the two snaps. A triple snap resets the image viewer's view, if the images being displayed should disappear from the user's view. The Snap-detect space is also used as an alternative firing mechanism in Quake 3 Arena, and is instrumental in enabling the creation of the Arm-angle space.

The Arm-angle space is used to select objects on the display wall by having users

point at them. It has been used to augment the Snap-detect window movement functionality by enabling users to point at the windows they wish to move, where snapping previously could only be used to move the currently focused window. It has also been used to select geometric objects in Wallboard, a custom whiteboard-style application. The Arm-angle space is unique in that it is movable, and in that it demonstrates how additional interaction spaces may be built based on already existing interaction spaces.

4.1 Limitations

This chapter documents the development of three interaction space systems. It is not focused on the development of user interfaces for display walls[3], nor is any attempt made at determining whether a particular user interface or interaction approach is "good," "user friendly" or "efficient." The systems have been built in order to gain an understanding of the overarching Interaction Spaces concept, to demonstrate the ideas and possibilities that stem from it, and to enable unencumbered, device-free interaction.

None of the three systems presented are able to distinguish different users. This capability is found in relatively few device-free interaction systems, with the DiamondTouch [97] as an example of a system that can distinguish different users. Of the three systems, the Camera-sense system might be extended to distinguish different users by linking objects detected in 3D to each other; at present, however, no such effort is made. The Snap-detect system can not be reasonably extended to distinguish different users, and the Arm-angle system is presently single-user only.

A further limitation of scope is object recognition. None of the systems attempt to determine what exactly an object is. The Camera-sense system enables users to interact using any object, but no attempt is made at determining if the object is a pen, a finger, a hand or something else. The Snap-detect system only tries to detect snap- and clap-like sounds; no attempt is made at identifying what made the sound – it could be someone snapping their finger, but it could also be a mechanical clicker, or a book slamming into a table. The Arm-angle system – despite its name – does not actually identify a user's arm. Instead, it is based on analyzing an image to detect straight lines, and on the assumption that straight lines represent a possible arm, use the angle of those lines to enable interaction. The principle behind the systems is "where, not what," and applying this principle to simplify the design and implementation of device-free, marker-less interaction spaces.

[3] Although some interfaces are developed, no claim is made towards their efficacy.

4.2 Related work

Interactive spaces and interactive or "smart" rooms are discussed in the work on Tangible Bits [74] and the Stanford Interactive Workspaces research on an "interactive room operating system" [75]. These papers are discussed in Chapter 2. The "interaction space" term is very broad, and has been used fields ranging from art and multimedia [98] to pervasive [99] and mobile [100] computing. This dissertation's research on interaction spaces is focused on creating device-free interaction spaces to enable interaction with display walls. As the size of the display in question increases, different technical approaches are used to provide gesture-based input. Small to regular-sized displays such as those found on the Nintendo DS or the Apple iPhone usually employ either resistive or capacitive [101] touch-screen technology to enable gesture-based interaction. Resistive touch-screens work by detecting contact between a resistive and a conductive layer as they are pushed together. A capacitive touch-screen detects changes in the voltage distributed in a capacitive layer when a finger or other capacitive device is in close proximity. When in close enough proximity, a voltage drop on the capacitive surface can be measured and used to determine the location of the touch. As displays grow larger, most current approaches including the approach taken by the Camera-sense system are camera-based. Other approaches to implementing touch screens include using a grid of sensors mounted along the display's frames to determine the point of contact; this approach is taken by the CarrollTouch frame [102].

One way to use the Camera-sense system is for multi-touch interaction, by having applications consider input from only one of the multiple planes in which objects are detected. Some of the earliest work on multi-touch input was conducted in the late seventies and early eighties by Myron Krueger in the Videoplace work [71]. The Videoplace is an art project where a cameras are used to detect the silhouettes of users. It did not require users to wear special gloves in order to enable interaction, and it was one of the first examples of many multi-touch gestures that are now common across nearly all multi-touch capable devices, such as the pinch-to-zoom gesture. Another early example of multi-touch input appears in [103], where the development of a capacitive multi-input touch tablet with limited 3D-sensing capabilities is shown.

The DiamondTouch [97], developed by Mitsubishi Electric Research Labs, is a multi-touch table. The table detects touches using electric capacitance. When the user touches a region of the table, a signal from an antenna beneath that region is transmitted through the user into the chair the user sits on, completing an electronic circuit. This enables the DiamondTouch to distinguish between different users touching the table. Due to the electronics used, the DiamondTouch requires the use of front-projection. There are several differences between the DiamondTouch and the Camera-sense system: (i) The DiamondTouch is a 2D only system, whereas the Camera-sense system locates objects in 3D; (ii) the DiamondTouch can only detect

human fingers or hands, while the Camera-sense system can detect any object; (iii) the DiamondTouch only reports which antenna was touched, and not where on the antenna the user touched. Further, the DiamondTouch requires users to touch the table in order to interact. The Camera-sense system can detect user interaction without requiring users to touch the interactive surface.

The technology used by the DiamondTouch has so far only been used for multi-touch tables, and not for enabling multi-touch interaction on display walls. In 2005, Jeff Han demonstrated a system based on using frustrated total internal reflection (FTIR) of infrared light to provide a multi-touch capable display [12]. The technology works by shining infrared (IR) light from a series of IR light emitting diodes (LEDs) into the sides of a rigid, plastic canvas. At points where a user touches the canvas, the internal reflection is "frustrated," resulting in infrared light escaping from that location on the opposite side of the canvas. A camera behind the canvas equipped with a filter to block visible light sees the infrared light. By applying background subtraction and thresholding to the camera image, the location of different touch points is determined by locating bright spots in the processed image. The FTIR multi-touch technology differs from the Camera-sense system in that: (i) FTIR multi-touch is limited to locating touches in 2D; (ii) the accuracy of FTIR multi-touch declines over time as the canvas gets progressively dirtier from being touched by users [12], a problem not faced by the Camera-sense system; and (iii) the scale of the system: The system is only used on a small projected display. The FTIR multi-touch approach has been commercialized by Perceptive Pixel, and was used extensively to cover the 2008 Presidential Election in the US on CNN [104].

The TouchLight [105] is a system that uses two cameras to capture stereo images of a transparent projection surface[4]. The images from the two cameras are processed to create a depth-map of the scene covered by the two cameras. The system's goal is to enable gesture-based interaction by detecting objects at approximately the same depth as the projection surface. The result is a "touch image" which can be interpreted by applications to allow interaction. The TouchLight differs from the Camera-sense system as follows: (i) The TouchLight is restricted to 2D gestures, at least in its current implementation; and (ii) the approach is not easily scalable, whereas the Camera-sense system can be scaled to cover larger displays by adding more cameras. A similar stereo camera system is presented in [106], where cameras are used to enable multi-touch interaction with a flat surface.

Another stereo camera based approach is the WaveScape system [56], in which a depth map of the scene in front of a plasma-TV sized screen is computed using two IR cameras and an infrared illuminator. The illuminator projects a semi-random dot-pattern onto the scene, which aids the stereo algorithm in more robustly determining the depth of different parts of the scene. The stereo camera output is then used to recognize human shapes which are mapped onto a human model for gesture

[4]The surface is composed of transparent acrylic plastic overlaid with a commercially available holographic film called the DNP HoloScreen.

recognition. The Wavescape system differs from the Camera-sense system in that: (i) The Wavescape system recognizes gestures from humans, whereas the Camera-sense system detects and locates arbitrary objects. The Camera-sense system does not attempt to recognize gestures, but instead give applications information about 2D and 3D objects which the applications themselves may use to recognize gestures; (ii) the Wavescape system models the human based on assumptions made from the computed depth-map, while the Camera-sense system detects not only the tip of objects in 3D but also their extent further back into the scene (even if occluded by the tip of the object, which the WaveScape system can not do); and (iii) the Wavescape system, like the TouchLight, is not scalable without changing the system's architecture, while the Camera-sense system's architecture is inherently scalable and can scale to wider walls by adding more cameras and computers.

The Microsoft Surface [13] is a multi-touch capable table, similar in function to the DiamondTouch. A projector under the table creates the visual display, and cameras mounted with the projector enable positioning of up to fifty-two simultaneous points of contact. The Surface uses five cameras and diffuse illumination (DI) of infrared light to detect points of contact. Infrared light is emitted from below the table, and reflected off of objects through the table's projection surface which is transparent to infrared light. When objects come closer to the table, the amount of IR light reflected increases, enabling touches to be detected. The basic image processing required is similar to that used by the FTIR approach, which uses background subtraction and thresholding to determine where objects are. This is the same approach taken by the Camera-sense system. The Surface only detects interaction in 2D; no attempt is made at building a depth-map or extract the geometry of 3D objects. In the Camera-sense system, a skeleton 3D representation of objects is constructed.

The SecondLight [107] is an extension of the Microsoft Surface. It is based on the Surface, but with two additions: Images can be projected on surfaces held above the table surface, and interaction can be detected above the table. For instance, a piece of paper could be held at a small distance from the table surface, and a different image from the one shown on the regular table could be projected onto the paper. At the same time, users can interact both with the primary table surface, and a secondary surface held above the table. The SecondLight combines both FTIR and DI to enable interaction. It differs from the Camera-sense system in that: (i) the SecondLight's scale is small, whereas the Camera-sense system is designed for large display walls; and (ii) the detected gestures are still 2D only, although the authors speculate on extensions to 3D.

The Microsoft TouchWall [51] is a prototype projected display with multi-touch capabilities. There are few technical details available about the TouchWall. Three infrared lasers are mounted below the canvas, and a camera is situated behind it. The camera is equipped with an IR-pass filter, like the cameras used by the FTIR multi-touch walls and the Microsoft Surface. When a user comes close to touching

the canvas, the infrared laser light's beam is diffused by the finger and reflected towards the display. This can be detected by the camera, and used to enable multi-touch interaction. The TouchWall has one characteristic that the FTIR multi-touch wall does not: It can detect interaction without necessarily requiring that users touch the display. Compared to the Camera-sense system, however, it is still limited to 2D.

A different DI approach was used to build the Duke multi-touch wall [108]. The approach uses 8 cameras to track touch-input over a 4.1x1.5 meter display wall. The basic technique is similar to the approach taken by the Microsoft surface. The display wall's canvas is illuminated with infrared light, and Touchlib [109] is used to detect and track touches from the individual cameras. A "composition" process (their terminology) is used to merge touches from different cameras. The system differs from the Camera-sense system in its design, and that it only detects objects in 2D, and not 3D. However, it demonstrates that DI approaches to multi-touch can be made to scale to display walls.

Another approach for multi-touch interaction is using range finders. In [110], researchers at IBM developed a system that makes use of a scanning laser rangefinder to enable multi-touch interaction with display walls. The rangefinder is mounted in one corner of the display, and rapidly scans the area covering the display wall to determine the location of objects. In the configuration presented, it suffers from occlusion issues, although it is suggested that this could be resolved by adding additional rangefinders. The use of lasers makes it possible to move the plane where interaction can occur away from the actual display. This is an advantage that it shares with the Camera-sense system. However, the range finder approach is still limited to 2D interaction. A range finder was also used to enable interaction with the FogScreen [111].

The SMART board [112] is an interactive whiteboard. It detects users touching the display using a set of custom cameras (called "corner blocks") mounted in the corners of the display [113]. The cameras perform on-board image processing to detect objects intersecting a narrow plane in front of the display. This is similar to the approach taken by the Camera-sense system, where the location of objects in a given plane is determined using triangulation. The scalability of the system is limited to displays where the four cameras can be mounted, their resolution and field-of-view. In a 2009 tech report [114], the authors present a "continuous interaction space" which is created by combining a commercial marker-based tracking system [115] with a SMART board. The work differs from the research presented here in its requirement that users wear markers, and in the resulting system's limited scalability. The work is further focused on defining and evaluating gestures, rather than building the underlying interaction space systems.

The HoloWall and HoloTable [116] are two realizations of a system that enables device-free interaction with a projected display. The system detects hands and other objects using DI of infrared light and a single camera behind the display.

Objects are detected up to 30 cm away from the display. The system differs from the Camera-sense system in that it: (i) Uses only a single camera to detect objects, whereas the Camera-sense system makes use of several cameras; (ii) the HoloWall relies on IR light to more easily detect objects, while the Camera-sense system takes a more traditional approach to segment foreground objects from background objects in the visual light spectrum; (iii) the HoloWall attempts to distinguish different objects and classify them, while the Camera-sense system makes no such efforts. Finally the Holowall, while being able to detect objects some distance away from the display, does not detect their 3D location.

GestureTek is a company that offers both 2D and 3D input solutions based on high-end cameras [117]. High-end cameras typically provide higher framerates and resolution, as well as support for synchronization, a feature that is not available when using commodity web cameras. The systems can provide 3D gestural input using depth cameras, such as the ones being developed by the company 3DV Systems [14]. A depth camera detects not only the color values for pixels in an image, but also associates a depth value with each pixel. Depth cameras can also be emulated using a stereo camera setup, as used in some of the approaches discussed previously [105, 56]. The main differences between the systems provided by GestureTek and the Camera-system are: (i) The GestureTek systems, while making use of multiple cameras, all have the cameras cover the same limited region of interest from different angles. In contrast, the Camera-sense system uses several cameras with partially overlapping or completely disparate regions of interest to cover a larger area; (ii) the GestureTek systems rely on expensive high-end cameras, whereas the Camera-sense system is designed for cheap commodity web cameras.

In [15], a depth camera from 3DV Systems is used to determine the occupancy state of the space in front of the camera. This is used to enable coarse-grained detection of objects, and use the results to enable interaction. The system is used to augment a display showing a real-world model with simulated content. Users can interact with the model by moving their hands over the real-world model, and see the results on the display. The work is similar to the Camera-sense system in that it merely aims at detecting the presence of objects, and not determine what the objects actually are. However, is is not used for interacting with a display wall, and the system's scalability has not been demonstrated. The particular 3D sensing technology used may also preclude the use of more than one camera[5].

There are a number of device-based gesture interfaces that share similarities with the Camera-sense system. When Nintendo introduced the Wii game console in 2006, they simultaneously made a very versatile input device – the Wii Remote ("Wiimote") – available to a large number of users. The Wiimote is equipped with

[5]The camera senses depth by sending out pulses of infrared light, and then measure the time it takes for the pulse to be reflected and return to the camera. However, the camera is not able to distinguish its own IR light from other IR sources. Thus, it is possible that several such cameras would interfere with each other, even if no other IR sources were present.

accelerometers and a small IR camera in addition to a number of buttons. The IR camera is used to detect the IR LEDs on the Wii's "sensor bar[6]." In [7], the author presents several different ways of utilizing the Wiimote's built in camera to create a 2D multi-touch input system by illuminating the scene with infrared light, and then having users wear IR reflectors on their fingertips. The Wiimote is limited to tracking just four dots however, and users are required to wear either passive IR reflectors or active IR LEDs for the Wiimote to track the user. The Camera-sense system does not require the use of passive or active markers.

The system being developed by Oblong industries [5] enables interaction with collection of displays in a way very similar to the interaction portrayed in the movie Minority Report [45], which is not surprising given that their system was the inspiration for the technology shown in the movie. Users wear gloves that are tracked by a vision system, and the system then recognizes gestures based on the hand pose and pointing direction. Another marker-based system developed by the German company Advanced Realtime Tracking makes use of IR reflectors mounted to users' hands or heads [8]. One goal of the Camera-sense system is to not require the use of markers or devices, and this is what sets the Camera-system apart from these systems.

The iPoint Presenter [118] is a system that tracks a user's fingers in 3D using cameras and infrared illumination. It can track up to eight fingers without requiring users to wear markers. The system uses two cameras in a stereo configuration to detect the location of fingers. Few technical details on the system are available. The system differs from the Camera-sense system in that it only detects fingers, and not arbitrary objects. This can be both an advantage and a disadvantage, depending on the application. Further, no information about an object's extent in 3D is provided; only the finger location (not its extent) is tracked. This differs from the Camera-sense system where a skeleton 3D object is constructed.

Instead of using gestures to interact, one can also use sounds. One approach is to use speech recognition to detect spoken commands. Speech recognition is built into many modern operating systems. The main focus in speech recognition is accurate detection of speech rather than determining where the speaker is located. In [119], an interface is built where continuous vocal sounds are used to control an interface. A similar approach is taken in [120], where vocal sounds are used to control the location of a cursor on a display. Users vary the pitch of their voice to move a cursor up, down, left or right, and use spoken commands to change direction or perform a click. Unlike the Snap-detect system, neither of these two systems attempt to determine where the sound originates. Further, the Snap-detect system does not attempt to do speech recognition at all.

In [121], a 3D interface is created that responds to loud noises above a dynamic

[6]The name sensor bar is somewhat misleading, since the bar itself does not do any sensing; it merely creates two IR dots which can be seen by the camera built into each Wiimote.

threshold. The system creates virtual boxes in space within which the user can clap his hands. The boxes act as buttons that control a media player. Instead of creating fixed areas where the system is sensitive to sound, the Snap-detect system makes the entire room sensitive to users snapping their fingers or clapping their hands and then making the 2D location available to applications. The systems also differ in the approach to detecting sounds: The Snap-detect system matches audio with a template sound clip, whereas the authors of [121] just detect any loud noise above a given threshold.

In [122, 110], an interface is developed where users can interact with large displays by knocking on its surface. The implementation shares the use of multilateration to locate the knocking sound's origin with Snap-detect. However, the system inherently requires that users actually knock on the surface; the system does not detect sounds that do not travel through the surface itself. In Scratch Input [123], any surface – such as a table or a wall – is made active by placing a small microphone on it. The system detects scratching noises, and uses them to interpret simple gestures such as drawing a line, circle, triangle or square on the surface. Unlike the Snap-detect system, no effort is made to determine where the scratching occurs; instead the focus is on disambiguating a set of gestures. As with the knock-system [122], Scratch Input requires that users actually touch the surface, while the Snap-detect system can locate sounds emanating from anywhere within the room.

In [124], sound source localization is used to locate the source of sounds in a 3D environment. 8 microphones mounted on a robot pick up sound, which is cross-correlated to determine the time delay of arrival of a sound between the different microphones. This is the same technique as employed by the Snap-detect system. The systems differs in that the Snap-detect system only detects snaps, while the robot navigation work is based on detecting all "interesting" sound sources and then navigating the robot towards such sounds. The systems further differ in the number of microphones, and the resulting dimensionality of the sound source location: 2D for Snap-detect, and 3D for the robot system.

The Arm-angle system was built to make it possible for users to select targets that are far away from their current location. In [125], the authors present a system that allows distant freehand pointing for interacting with wall-sized displays. Users can be located far away from a display wall, and select targets by pointing at them. The system requires users to wear markers in order to detect the direction in which they are pointing. In [126], laser pointers are used to enable interaction at a distance. Another alternative is to track a marker-equipped wand in 3D [127] and uses the resulting location information to allow users to point at objects from afar. In contrast to these systems, the Arm-angle system does not detect neither the user nor the arm: Instead, the angle of straight lines in images captured by a camera are used to infer the direction in which the user's arm is pointing. The assumption made is that the arm is sufficiently straight so as to be detected as a small number of lines with similar angles, and that few other straight lines appear in the image. A further

difference is that the Arm-angle system does not require markers. The Arm-angle system however is limited to detecting the pointing angle when the user is close to the display wall; no support is provided for distant pointing.

There have been a number of other techniques for selecting items that are physically far away from the user. These techniques include drag-and-pop [128], the Vacuum [129] and the Frisbee [130]. These techniques generally work by temporarily moving a target located far from the user closer. In drag-and-pop, initiating a drag-operation causes relevant drop-target to move closer to the location of the drag. When the Vacuum is invoked, it moves remote objects within an "arc of interest" closer to the user. The Frisbee works by creating a "telescope" on the display wall which displays the pixels located at a far-away target. This enables users to see and interact with pixels that are physically unreachable to the user. The "tablecloth" is another approach that lets the user scroll the desktop much as he would scroll a window [131]. These systems are generally complementary to the Arm-angle system, since they do not apply any detection of gestures to enable interaction.

4.3 The Camera-sense system

The idea behind the Camera-sense system is to use many cameras mounted in a line along the floor to detect objects, and then combine information about the detected objects from the different cameras to determine each object's location and extent in 3D. No effort is made to identify what the particular objects are, or how they are related to each other. The system instead only tries to determine if objects have entered the interaction space, and determine where they are. This makes it possible to use any object to interact, including hands, fingers or pens. However, it also has the potential downside that different users can not be distinguished from each other.

4.3.1 Architecture

Figures 4.2 and 4.3 show the overall architecture of the Camera-sense system. The system uses several cameras and a cluster of computers to detect the presence of moving objects in each camera's field-of-view. The Camera-sense interaction space is created inside the wedge-shaped volume outlined in the figures. The wedge shape is due to the different cameras' partially overlapping field-of-views. Each computer in the cluster captures images from the cameras attached to it at the maximum frame rate supported by the camera. The maximum frame rate depends on the camera, and is usually between 15, 30 or 60 frames per second (fps) for commodity web cameras. The cameras used by the Camera-sense system have a maximum frame rate of 30.

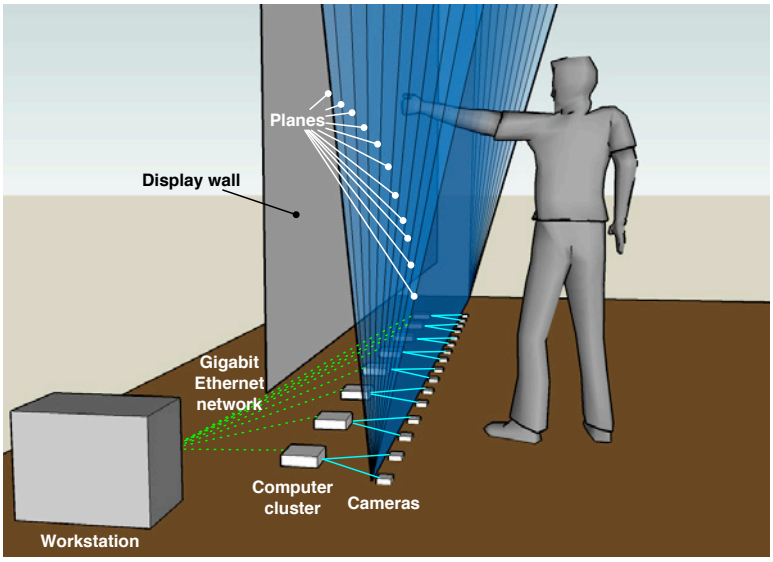


Figure 4.2: The architecture of the Camera-sense system. A collection of cameras is connected to a cluster of computers. The cluster processes images captured from the cameras, and sends information about detected objects to a separate workstation over a local area network. The workstation runs an object locator which locates objects in each plane in 2D using triangulation. An object's 3D representation is then inferred from its corresponding 2D location in the different planes. Credit for the human model: Google SketchUp.

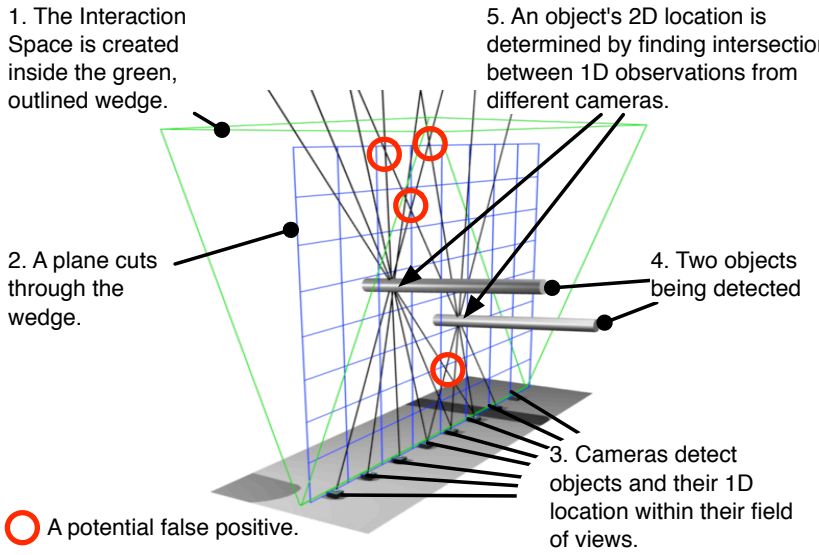


Figure 4.3: The Camera-sense interaction space is created by using several floor-mounted cameras to detect objects and then triangulate their locations. Objects are detected in planes inside the (approximate) wedge-shaped volume created by joining each camera's field-of-view with the other cameras.

The cameras are mounted in a line along the floor in front of a display wall with their lenses pointing towards the ceiling. The ceiling is expected to be brightly and uniformly lit, in order to create high contrast between the background and the darker objects to detect. The cameras are spaced evenly 32 cm apart. The number of cameras can be scaled up or down as necessary to cover wide or narrow display walls. The two implementations used by the Tromsø and Princeton display walls use sixteen and eight cameras respectively.

Each camera's field-of-view is divided into an equal number of slices. Corresponding slices from all the cameras together create a plane in front of the display wall in which objects are detected and located in 2D. An object's skeleton 3D representation is then assembled using one 2D location from each of the different planes.

To determine an object's 2D location, it must first be detected in several corresponding slices from different cameras. Once detected in a slice, its location and extent in 1D along the horizontal axis of the slice is determined. The 1D locations and extents are sent using the Shout event system[7] to an object locator running on a workstation.

The object locator triangulates the 2D location of objects within a plane using the 1D locations and extents from that plane. To obtain a skeleton 3D representation of an object, the object locator selects a single 2D object location from each slice and adds it to the object's 3D representation. The first 2D location is chosen from the outermost plane, which is farthest from the display wall. The location from every subsequent plane is chosen by selecting the one closest to the location from the previous plane.

Once the object locator has created or updated its 3D representation of an object, events containing the object's 2D location within the different planes and an event containing its 3D representation is broadcast to applications using Shout. Applications then use the events to interpret gestures.

4.3.2 Design

Figure 4.4 shows the design of the Camera-sense system. The design consists of an image processing component that runs on each computer in the image processing cluster, and an object locator that runs on a workstation[8].

The system uses commodity web cameras to achieve its goal of locating objects within the Camera-sense interaction space. Each camera is assigned an identifier, ranging from 0 up to the number of cameras. Cameras that are physically adjacent

[7] The Shout event system is described in Appendix B.

[8] The object locator may also run on one of the image processing computers, which is the case for the system deployment at the Princeton display wall.

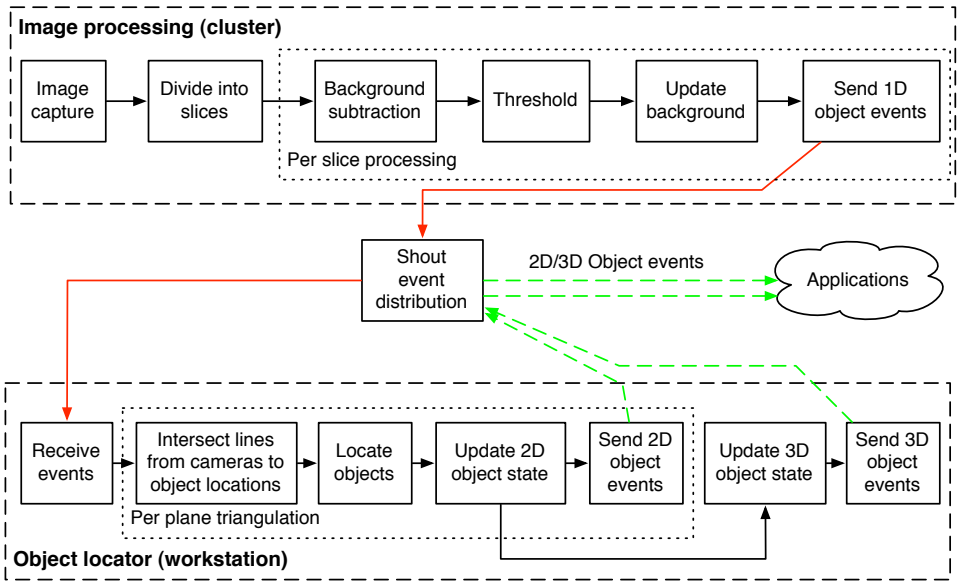


Figure 4.4: The design of the Camera-sense system. Images are captured and processed, before the results are sent via the Shout event system to the object locator. The object locator determines where objects are in 2D and 3D, and sends 2D and 3D object events to Shout, which distributes them to applications.

are assigned successive camera IDs, i.e. camera i is physically located between cameras $i-1$ and $i+1$.

The cameras used by the system are Unibrain Fire-i cameras with a resolution of 640x480 pixels delivering images at a rate of 30 frames per second. The full specifications are listed in Section 2.2.1. The cameras connect to the image processing cluster using FireWire 400 [132].

Commodity web cameras differ from high-end cameras in a number of ways, including the rate at which images can be captured, image resolution, hardware triggering support, noise levels and lens optics. Of these factors, the one with the biggest implications for the design of the system is the lack of triggering support. Triggering makes it possible to synchronize the time at which all the cameras begin to acquire an image through their imaging sensor. An external source sends a pulse to the cameras through a wire that connects all the cameras to each other. The pulse "triggers" the cameras, resulting in nearly simultaneous image acquisition.

Without triggering, the object locator must handle the fact that images can be captured at different points in time, which impacts both the latency and accuracy of the system. Latency is increased because the object locator must wait for data from all cameras before it can triangulate object locations. The lack of triggering makes the triangulation of *moving* objects less accurate, since the object will have moved slightly between the time two cameras have acquired an image. These issues are

discussed further in Section 4.7.

Image processing

The image processing component's goal is to detect the 1D location and extent of objects appearing in images captured from the camera(s) connected to the computer it is running on. An image is captured from a camera, then divided into 25 slices. Since the first part of the image is covered by the actual display wall assembly, the first slice is located near the vertical center of the image at an offset of about 240 pixels[9]; the exact offset is configured manually for each camera based on the camera's physical placement and orientation. The slices are the same width as the captured image, but only three pixels tall. A five-pixel gap is placed between each slice, making each slice consume 8 lines of pixels from the image. With 25 slices, 200 vertical pixels are consumed from the image, which leaves some room for adjusting the vertical offset of the first slice without exhausting the camera's vertical resolution of 480 pixels. Figure 4.5 shows an example image captured from one of the cameras and its division into slices.

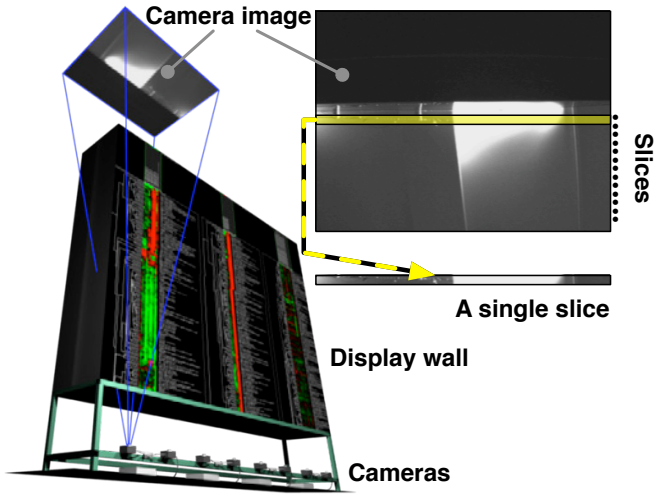


Figure 4.5: The view from one of the cameras mounted along the floor, and the slices the image is divided into. Each slice cuts through a plane tilting out from the display wall's surface and into the surrounding space, as illustrated in Figure 4.3.

The slice height of three pixels was chosen based on the camera's resolution and field-of-view. With a resolution of 640x480 and a vertical field-of-view of 32°, three pixels cover $\frac{32^\circ}{480} = 0.2^\circ$ of the vertical field-of-view (without adjusting for

[9]The first slice could be moved up if the cameras were rotated so as not to have the display wall obscure the first part of the image. This has not been done due to the complexity of mounting the cameras in this fashion. It is far simpler to make all the cameras lie flat on the ground.

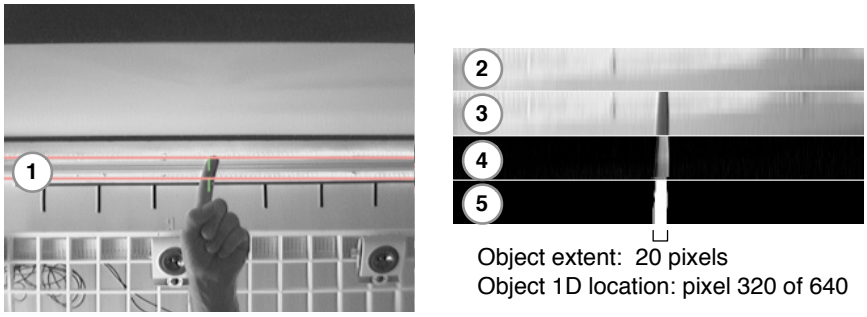


Figure 4.6: (1) An image captured from a camera. (2) The background slice. (3) The current slice being processed. (4) The slice after background subtraction. (5) The slice after thresholding, where the object's extent and 1D location have been determined.

camera distortion). At a distance of 3 meters, the plane formed using 0.2 degrees is 1.04 cm thick. This is about the width of an index finger.

Each slice is processed using two standard techniques from computer vision: Background subtraction [133] and thresholding [134]. Background subtraction removes the background (which is usually the ceiling) from the slice, by subtracting the new slice from a background slice. This yields a delta slice, where negative values indicate that the corresponding pixel in the new slice is darker than the corresponding background pixel, and positive values indicate that the corresponding pixel in the new slice is brighter. The background slice is initially set to the first slice from a camera when the image processing begins. Every subsequent slice is then slowly averaged into the background slice, to compensate for camera noise and changing lighting conditions.

For pixels in the new slice that closely match the background slice, the resulting pixel value lies close to 0. By selecting a suitable threshold value, foreground objects can be isolated in the current slice. Since the image processing component assumes that the background is bright and uniformly lit, only negative values are used, since any foreground objects would hide the bright background and thus appear darker than the background. Figure 4.6 illustrates how a finger is detected in a single slice from one camera. A finger enters the slice (1). The background slice (2) is subtracted from the current slice (3), yielding a temporary delta slice (4). Any pixels with a negative value less than the threshold value is set to white and considered part of the current foreground (5).

Figure 4.7 illustrates how slices are analyzed to detect objects and determine their 1D location. Due to camera noise, random pixels may sometimes exceed the threshold and be considered part of the foreground when in reality they are not. To lessen the impact of noise, solitary pixels in the thresholded slice are ignored, but only if they do not have an adjacent foreground pixel either two positions to the left or right of the solitary pixel. Camera noise may also have the opposite effect, where a single pixel fails to be detected as part of the foreground. When

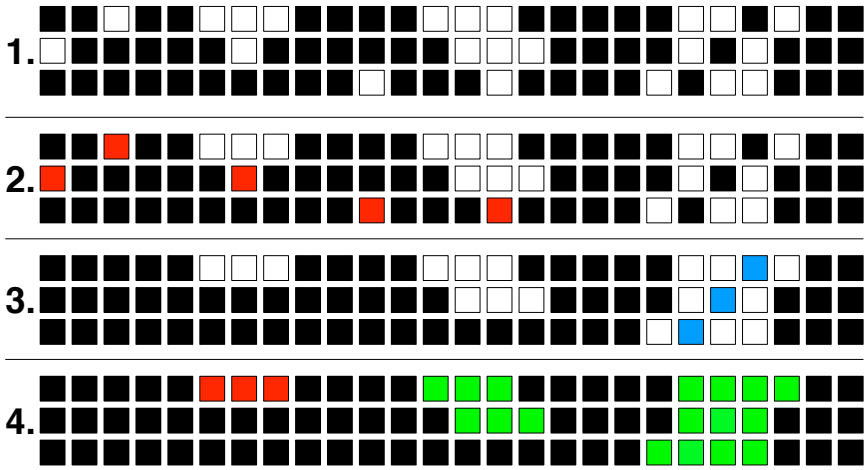


Figure 4.7: (1) A slice after foreground pixels have been found using thresholding. (2) Solitary foreground pixels (red) are considered noise and ignored from further consideration as long as there are no foreground pixels either two positions to each pixel's left or right. (3) Solitary background pixels with foreground pixels neighbouring both sides (blue) are considered part of the foreground. (4) Foreground pixels that span two of the three lines (green) are considered detected objects; other foreground pixels (red) are ignored.

this pixel falls in between two pixels that are considered part of the foreground, the system might end up detecting two objects when in reality there is only one. To compensate for this, single-pixel gaps between two foreground pixels are closed by considering the gap as part of the foreground.

Next, groups of adjacent foreground pixels are located in each individual line of the slice. Each group represents a potential object. An object is defined as detected if groups from two of the three lines in a slice overlap, and one of the overlapping groups appear in the center line. The object's 1D location is then set to the center pixel index of the group from the center line, and its extent set to the number of pixels in the center group.

If one or more objects are detected by the image processing, their 1D location and extent is sent to the object locator. If no objects are detected, an event notifying the object locator of this is sent instead. This event is only sent once when no objects are detected. If subsequent images from the same camera result in no objects being detected, no further events are sent until objects have again been detected in an image from that camera.

Object locator

The object locator detects and locates objects in 2D and 3D. Objects are first detected in each of the different planes, by using 1D object events from the image processing cluster to determine an object's 2D location. The resulting 2D locations

are then used to update the current state of the different 3D objects, before both 2D and 3D object events are sent to applications using Shout.

Since image acquisition is not synchronized across cameras, the object locator must wait to receive 1D object or no detect events pertaining to all the cameras. Once it has received the events, the object locator proceeds to triangulate the 2D location of objects. The triangulation is illustrated in Figure 4.8(a), where three objects are detected. To make triangulation possible, the field-of-view, lens distortion, physical location and orientation of each camera must be known. The field-of-view is configured based on the lens capabilities reported by the vendor. The lens distortion is initially assumed to be zero, and the initial value for the camera's location is set by measuring its physical offset from the floor and the left side of the display wall. The initial orientation is assumed to be horizontal. All the parameters may change after the system calibration, as detailed in Section 4.3.3.

To triangulate the 2D location of an object, lines are projected from a camera's location through the 1D object locations derived from an image from that camera. Potential objects lie at the intersection of lines projected from different cameras. An object is detected and located in 2D if it is observed by at least three cameras, and the resulting intersection points are in close proximity, as detailed in Section 4.3.3.

To prevent false positives when triangulating an object's 2D location, the object must be detected in images from at least three different cameras, as shown in Figure 4.8(b). Potential false positives are also highlighted in Figure 4.3. Two cameras are sufficient to determine the location of a *single* object in 2D, but if more objects are to be located at the same time, false positives may occur. This scenario is common when attempting to use the system for multi-touch or multi-user interaction. Using three or more cameras, the false positives can be removed, leaving only the 2D locations of actual objects.

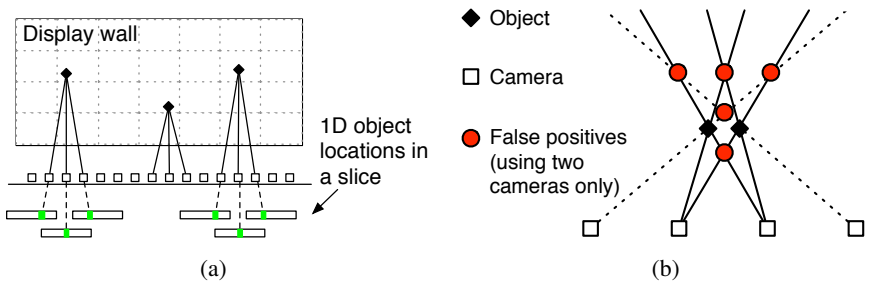


Figure 4.8: (a) Triangulating 2D object locations based on the 1D locations of objects detected after image processing. The rectangles below the cameras illustrate the output from the image processing. The highlighted green areas within the rectangles show where an object has been detected in 1D. (b) When attempting to locate more than one object using observations from just two cameras, false positives (red circles) occur. By incorporating observations from at least one more camera, the false positives can be eliminated, leaving only the true object locations (black diamonds).

Whenever a plane has been processed, the current 3D object state is updated. The 3D object state consists of a set of skeleton 3D objects. Each 3D object consists of an object ID, as well as 25 2D locations; one for each of the 25 planes. With each 2D location, the originating 2D object's ID is stored, as well as a flag that indicates whether the 2D location is valid or not[10]. The coordinate system uses metric values to represent the object's location in front of the wall along the X and Y axes. The Z axis' values are computed by mapping the index of each plane linearly to a value ranging from 0 to 1, where 0 represents the plane closest to the wall, and 1 represents the plane farthest away from the wall.

To update the 3D object state using 2D locations from a newly processed plane, the object locator first attempts to update existing 3D objects using the new 2D locations, before it may use any leftover objects to potentially create a new 3D object. First, the object locator compares the 2D object IDs in the plane with the 2D object IDs stored in each 3D object. If a match is found, the 3D object is updated using the matching 2D object, and the matching 2D object is removed from further consideration.

If there are 2D objects left to process, they are added to existing 3D objects as follows. Each 3D object that has not yet been updated is processed in turn. If the 3D object has a 2D location that is in close proximity of the new 2D location, the 2D location is assigned to that 3D object, and then removed from further consideration. The object locator scans the 3D object's 2D locations starting with the 2D location from the same plane as the new 2D location originates, and then expands the search to the planes closer to and further away from the wall.

Figure 4.9 illustrates the process. In step 1, a 2D location from plane 4 is being compared to the current state of a 3D object. In step 2, the locator checks the 2D location in plane 4 of the 3D object, and finds that it does not have a valid location for this plane. It then expands the search in step 3, finding that neither plane 3 nor 5 have valid locations. In step 4, the locator finds valid locations in both plane 2 and 6. It compares the distance between the 2D locations, and determines that the distance between the 2D location in plane 2 of the 3D object and the new 2D location is below the proximity threshold. In step 5, the locator updates plane 4 of the 3D object with the new 2D location and marking the plane as containing a valid location.

When all existing 3D objects have been updated or found not to match any of the new 2D locations, new 3D objects are created for each of the remaining 2D locations. Two different events are then generated for each of the 3D objects that contain at least five valid 2D object locations[11]. The first event includes the ob-

[10] For instance, an object that just barely has entered the interaction space will not have valid 2D locations for the planes closest to the display wall.

[11] An object must appear in five planes to be considered "reliably" detected in 3D, on the assumption that it is unlikely that something could be detected in five planes at almost the same location and still be noise.

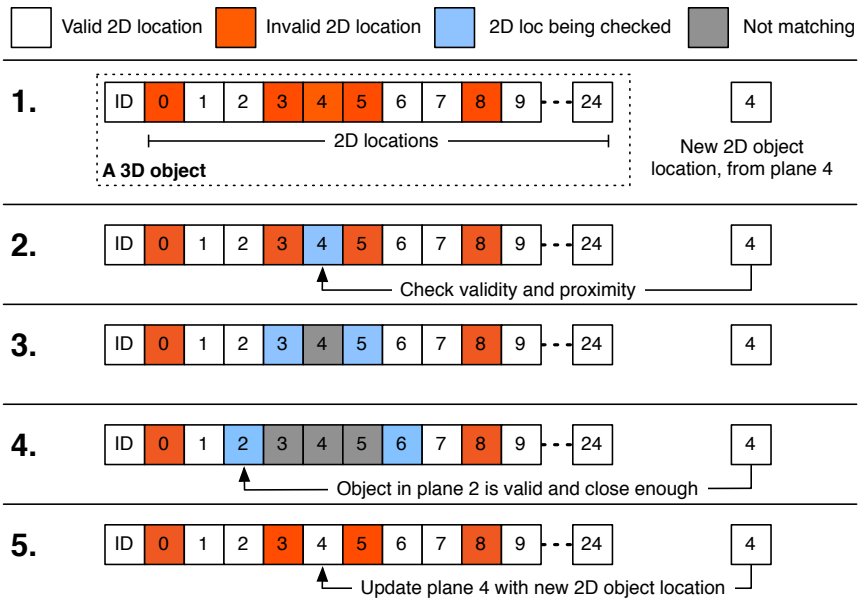


Figure 4.9: The steps taken to incorporate a 2D object location from a plane with a 3D object. See the main text for description of the different steps.

ject's ID and a "simple" 3D location, that only references the 2D location of the innermost plane in which the 3D object has been detected, and using that plane's index to generate the Z axis coordinate. The second event contains a serialized description of the 3D object, including the object ID and all of its state regarding valid and invalid 2D object locations in different planes. Both events are sent to applications using Shout.

4.3.3 Implementation

The Camera-sense system has been implemented and is currently being used with the Tromsø and Princeton display walls. For the Tromsø display wall, the system employs sixteen commodity web cameras connected in pairs to eight computers using FireWire 400 [132]. For the Princeton display wall, eight cameras and four computers are used. The image processing component and object locator have both been developed in C and Objective-C for Mac OS X. The cameras are mounted as shown in Figure 4.10. The specifications for the cameras and image processing cluster are given in Section 2.2.1.

Image processing

A screenshot of the image processing application is shown in Figure 4.11. It captures images from the cameras attached to the computer using libdc1394 [135],

(a) (b)

Figure 4.10: The camera deployment in (a) Tromsø and (b) Princeton. In Tromsø, the cameras are mounted along the floor on wooden boards with holes drilled to cup the cameras, and held in place using rubber bands. In Princeton, the eight cameras are mounted to the display wall's frame using plastic straps and the vendor-supplied camera mount.

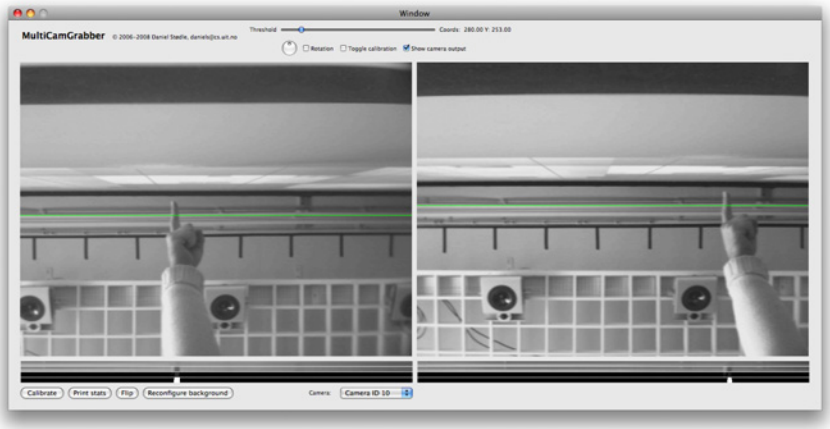


Figure 4.11: The image processing application capturing images from two attached cameras. The green line in the two camera images indicates where the innermost slice is located (the one closest to the display wall), and must be configured manually. The buttons and sliders were used during development and are not necessary for normal operation.

an open source library for communicating with FireWire (IIDC 1394) cameras. The application handles each camera independently. To capture and process images from a camera, two threads are used. The first thread runs an infinite loop that waits for new images to be delivered by libdc1394. When a new image is available, it is copied to a buffer and queued for processing by the second thread. Queuing the new image has the effect of notifying the second thread that a new image is available.

The second thread dequeues images from the queue. Each image is processed, before object events that result from processing the image are sent. The image is divided into slices, and each slice processed independently. Four pixel buffers are

used to process each slice: (i) current: Contains the pixels that are going to be processed from the current image; (ii) background: The background pixels; (iii) delta: The result from subtracting the current pixels from the background pixels; (iv) average delta: The average of the delta buffer over the last eight images. A pixel is marked as part of the foreground if it is less than a constant threshold of -10, and either (i) less than the three times the average delta, or (ii) less than five times the average delta for the entire slice. When all the pixels have been processed, the background buffer is updated using values from the current buffer. Each background pixel is updated by retaining 99.5% of the background pixel's value and adding 0.5% of the current pixel to it.

The threshold factors and the background retention factor were all determined empirically when tuning the system. For the threshold factors, the goal was to detect foreground pixels without generating false positives. The background retention factor controls how quickly the current pixels are incorporated into the background, and thus controls how quickly the system can adapt to sudden changes in lighting. However, with a low background retention factor, the current pixels (and any foreground objects appearing in them) will replace the background too quickly. This results in foreground objects being incorporated into the background if they are stationary for too long; for instance, pointing at the same location for more than five to ten seconds.

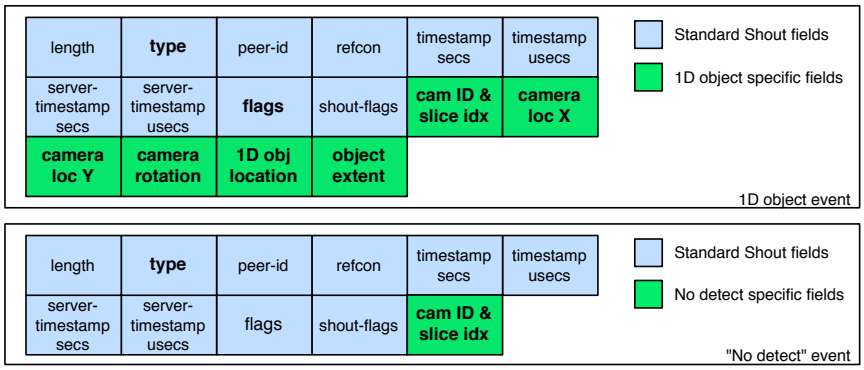


Figure 4.12: The 1D object and no detect event formats. Each field is 4 bytes wide. The fields in **bold** are custom to the 1D object and no detect event types; the remaining fields are Shout specific and further detailed in Section B.2.

The image processing application then analyzes the set of foreground pixels to find groups that comprise objects, as discussed in Section 4.3.2. Once objects have been located in all slices, their 1D location and extent are sent to the object locator as 1D object events using Shout. If no objects are detected, a "no detect" event is sent instead. The event formats are illustrated in Figure 4.12. The figure shows both the standard Shout event fields (which are discussed in more depth in Appendix B), and the event specific fields. For the 1D object event, the fields and their usage are: (i) Type: Set to the 1D object event type; (ii) Flags: Indicates whether this is the

first or last detected object in a slice[12]; (iii) Camera ID and slice index: Contains the camera's identifier, and the index of the slice in which this object was detected; (iv-v) Camera location X and Y: The camera's location relative to the floor and the display wall's far left side, in centimeters; (vi) The camera's horizontal rotation in degrees; (vii) 1D object location: The object's pixel location (ranging from 0 to 640); and (viii) Object extent: The object's width in pixels. The no detect event is like the 1D object event, except that only the camera ID and slice index are included.

Object locator

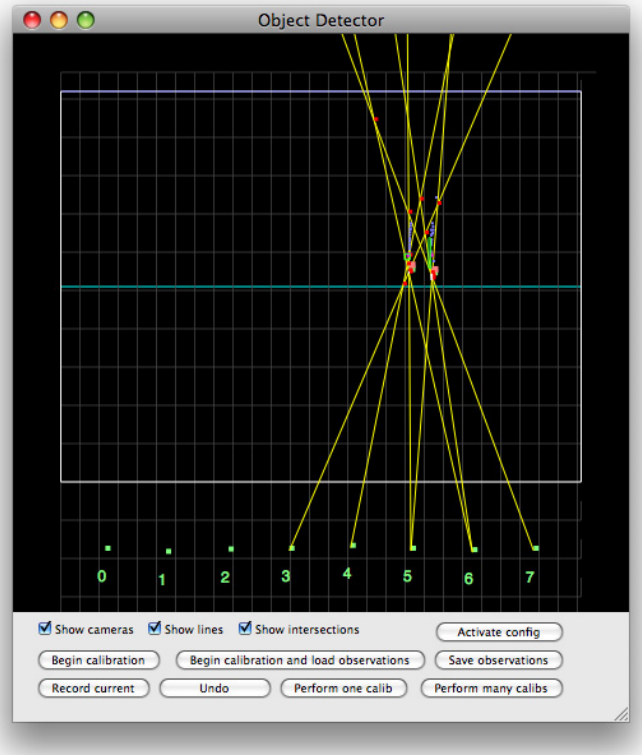


Figure 4.13: The object locator being used to detect objects on the Princeton display wall.

A screenshot of the object locator is shown in Figure 4.13. The object locator has two functions: (i) Determine 2D and 3D object locations; and (ii) calibrate the system. Determining object locations is done as follows.

[12]This is necessary since the object locator does not know a priori how many objects have been detected in a given slice. Thus, the event must include information that enables the object locator to determine if it has received all detected object events pertaining to a given slice from a given camera.

The object locator runs a loop that continuously accepts events. Once it has received 1D object events pertaining to all the cameras in a given plane, the object locator runs a triangulation step for that plane based on the new events. The object locator maintains state on previously detected objects in different planes, as well as state on current 3D objects. Each object, whether 2D or 3D, is assigned an object ID, which persists while the object remains detected. Objects are removed if their location is not updated for three seconds. This is enough to enable applications to recognize gestures such as tapping the display wall by tracking the object ID, without having to keep state around indefinitely.

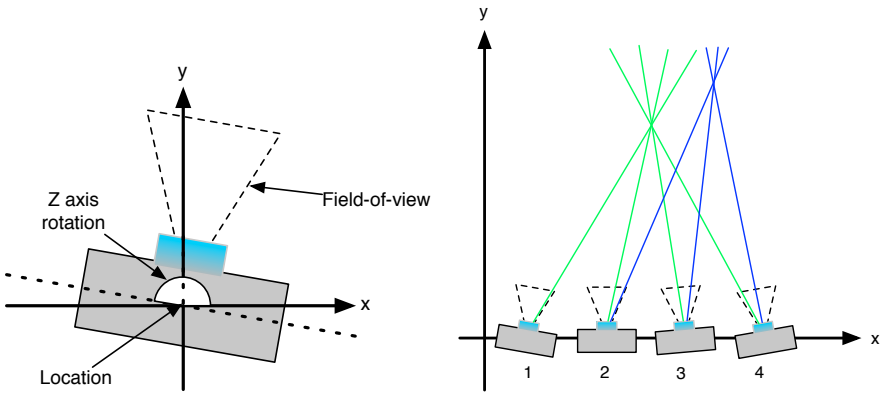


Figure 4.14: (a) The camera parameters. The camera's physical location is given in centimeters, offset from the floor and the display wall's left side. It may be rotated left or right about the Z axis (which is not shown, but runs perpendicular to the X and Y axes). The rotation has been exaggerated for illustration purposes. The camera is assumed to have no rotation about the Y axis. Rotation about the X axis is handled by configuring the offset for the first slice index. (b) Four cameras, with different rotations, are used to triangulate the locations of two different objects, as marked by the green and blue lines. Table 4.2 shows an example of events that could lead to the above figure.

Cam ID	Cam loc X	Cam loc Y	1D object loc	Cam rotation	Flags
1	0.00 cm	0.10 cm	630 px	10°	First and Last
2	0.32 cm	0.10 cm	400 px	0°	First
2	0.32 cm	0.10 cm	635 px	0°	Last
3	0.64 cm	0.10 cm	300 px	-5°	First
3	0.64 cm	0.10 cm	550 px	-5°	Last
4	0.96 cm	0.10 cm	50 px	-10°	First
4	0.96 cm	0.10 cm	315 px	-10°	Last

Table 4.2: Example 1D object events. The values are rough estimates (and for the rotation, exaggerated) and may not match the illustration in Figure 4.14(b) exactly. The slice index and object extent are omitted since they are respectively constant and not relevant to the example.

Figure 4.14 shows how the object locator uses the information about a camera's

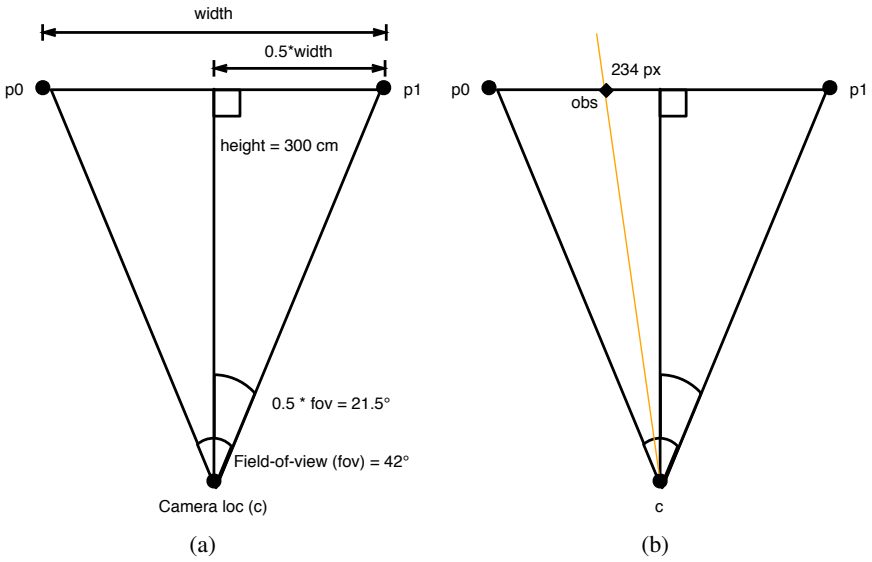


Figure 4.15: Calculating a line segment based on a 1D object event for a given camera. (a) To obtain the points $p0$ and $p1$, the width is calculated using standard trigonometry. The location of the observed point obs in (b) then lies somewhere along the line between $p0$ and $p1$. The resulting orange line segment from c to obs is then intersected with line segments rooted at other cameras.

location and rotation to triangulate 2D object locations in a plane, with the accompanying 1D object events shown in Table 4.2. Line segments are created by projecting a line from the camera's location, through the point at which an object is detected. This is done by applying standard trigonometry, as illustrated in Figure 4.15. The figure shows a camera's field-of-view, divided in two to create a right triangle. The $height$ is set to 300 cm (which is approximately where the top of the display wall is located). Since the field-of-view (fov) is known, the $width$ can be calculated based on the relationship between the $\tan$ trigonometric function and the lengths of a right triangle: $0.5 * width = height * \tan(0.5 * fov) = 118.17$ cm. At a distance of 300 cm, one camera thus covers a horizontal distance of $2 * 118.17$ cm $= 236.34$ cm. The locations of the points $p0$ and $p1$ subject to the camera's rotation then become $p0 = \text{rotate}(\{-0.5 * width, height\}, r) + c$ and $p1 = \text{rotate}(\{0.5 * width, height\}, r) + c$, where $c = \{cx, cy\}$ is the location and r is the rotation of the camera. The rotate-function is defined as $\text{rotate}(pt, angle)$: $\{pt.x = pt.x * \cos(angle) - pt.y * \sin(angle), pt.y = pt.x * \sin(angle) + pt.y * \cos(angle)\}$.

A line segment going through an observation at pixel pix can now be created by defining two points on the line. The first point is at the camera's location c. The second, obs, is located somewhere between $p0$ and $p1$: $obs = p0 + \frac{pix}{640} * (p1 - p0)$. For example, if a 1D object were detected with center at $pix = 234$ from a camera with no rotation and located at $c = \{0, 0\}$, $p0$ and $p1$ would be at $\{-118.34, 300\}$

and $\{118.34, 300\}$ respectively. The second point obs would then lie at $obs = \{-118.34, 300\} + \frac{234}{640} * (\{118.34, 300\} - \{-118.34, 300\}) = \{-118.34, 300\} + 0.365 * \{236.68, 0\} = \{-31.95, 300\}$.

The resulting line segments are then intersected with each other, yielding a set of intersection points. The intersection points are divided into clusters of points in close proximity of each other. If at least two intersection points originating from three different cameras are in close proximity, a 2D object is defined as detected. The system considers points in close proximity if the distance from their bounding box' top-left corner to its bottom-right corner is less than 20 cm. This threshold was set based on observations of the actual distances between intersection points, while weighing the trade-off between the desire to detect objects even in the face of inaccurate data from the cameras and not make the resulting object locations too inaccurate.

Clusters of points are isolated as follows: Start with all points, and calculate the current point cluster's bounding box. Then remove the point which reduces the distance from the bounding box' top-left coordinate to its bottom-right coordinate the most. Repeat this step until the distance is less than 20 cm, or only two intersection points remain in the point cluster. The resulting points are excluded from further consideration and considered as either a new object, an update to an existing object, or discarded if the distance is greater than 20 cm. Then the process is repeated with a now smaller number of points, until no points remain in the point cluster. For each cluster of points, the mean of the points is taken as the object location. The system then selects the closest, not-yet-updated 2D object and updates its location as long as the existing object is less than 4.5 cm away. Otherwise, a new 2D object is created.

If the system could rely on perfect detection of objects from the cameras, the above approach would be sufficient to detect and track all objects. However, since the image processing will never achieve perfect accuracy in detecting foreground objects, a foreground object may sometimes be observed only by two cameras in a given triangulation round. To mitigate this problem, the object locator will attempt to use a single intersection point to update an existing 2D object's location if the intersection point is less than 32 cm from the existing object[13]. Only the closest such intersection point is used, and the 2D object must already exist – that is, it must have been observed by at least three cameras in a previous round.

Once a plane has been processed, the object locator updates its 3D object state using the approach outlined in Section 4.3.2. Figure 4.16 shows how the Z coordinate of user's finger progresses as it moves closer to the wall. The proximity threshold used to associate new 2D object locations with an existing object is cur-

[13]This value corresponds to an object moving at about 9.6 meters/second. Smaller values would reduce the maximum speed of objects that could be tracked using only one intersection point. The trade-off in choosing this value is the possible incorporation of false positives versus a better detection rate for existing objects.

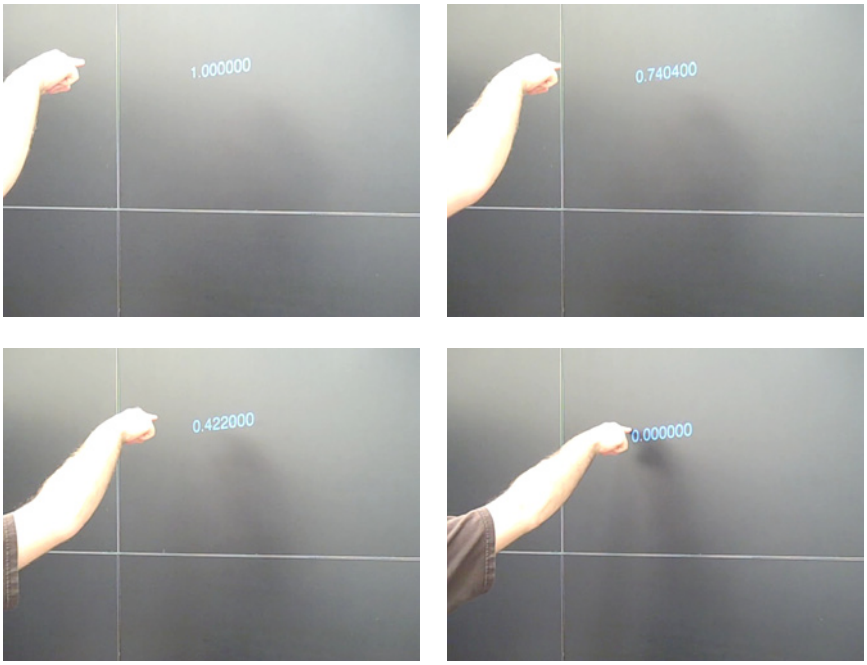


Figure 4.16: The depth penetration of a user's finger is shown on the display wall as the hand moves closer to the wall.

rently set to 20 cm. This threshold is higher than other similar thresholds used until now. Since the detection of objects is not perfect, an object may not be detected in all the planes where it actually appears. If this object were additionally pointing at an angle (so that successive 2D locations from the outermost to the innermost plane are significantly offset), the proximity threshold must be high in order to still enable the construction of skeleton 3D objects when the angle of the object as it approaches the wall is either very acute or very obtuse.

Camera calibration

The Camera-sense system currently uses a limited camera model that must be calibrated prior to using the system. The camera model incorporates the location, Z-axis rotation, field-of-view and distortion of the camera image. The goal of calibrating the system is to refine the initial estimates of each camera's parameters. To estimate the parameters, the system records the 1D object events from all cameras as a user touches a large number of calibration points on the display wall. Each point has a known, physical location. When it is touched, an operator records the 1D object events coming from the image processing cluster. With the location of the physical object known, the error for each camera that detects the object is estimated by measuring the shortest distance between the known location and the line segment projected from a camera through the observed 1D object.

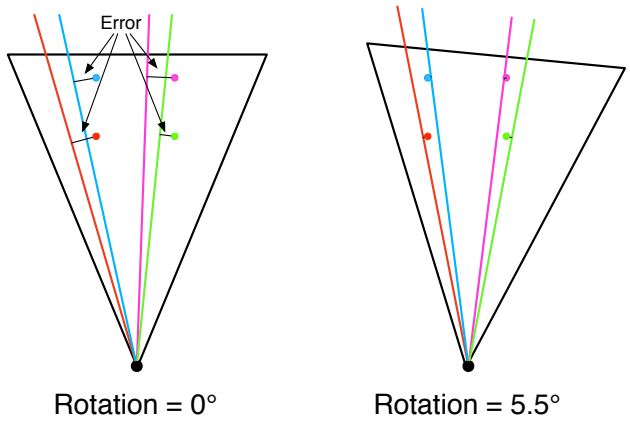


Figure 4.17: An example calibration of a camera. The red, blue, magenta and green circles indicate the known locations of four calibration points. The colored lines are the corresponding line segments resulting from observations made by the camera when the calibration points were touched. In this case, adjusting the camera rotation reduces the error.

When all the calibration points have been touched, each camera will be associated with a set of 1D object events with corresponding calibration points. The parameters of each camera are then automatically adjusted by iteratively changing different parameters in a greedy fashion. The goal is to minimize the sum of the error, as measured by summing the errors for each calibration point and its corresponding 1D object event. An example is shown in Figure 4.17, where the camera's rotation is adjusted to reduce the error between the known locations and the resulting line segments.

4.4 The Snap-detect system

The idea behind the Snap-detect system is to detect snap- and clap-like sounds by continuously comparing audio from four microphones with a pre-recorded sample of a user snapping his fingers. When a user snaps his fingers, the sound travels from the source of the sound (which is usually the user's hand or hands) to each of the four microphones. When the sound is detected in the audio stream from at least three of the four microphones, the location of the sound's origin is estimated by measuring the sound's arrival time to the different microphones. The location of the sound is determined in 2D, but the sound can originate from anywhere within the room in which the microphones are placed. This makes the Snap-detect interaction space a pseudo-3D interaction space, since it can be used in all three dimensions, but the resulting input is mapped down to two dimensions.

4.4.1 Architecture

Figure 4.18 shows the architecture of the Snap-detect system. Four microphones are mounted in a rectangular fashion around the Tromsø display wall, with two microphones close to the ceiling, and two placed close to the floor on either side of the display wall. The location of each microphone is known. The microphones are connected individually using equal-length audio cables to an Analog-Digital (AD) converter, which takes the analog audio signal from each microphone, digitizes it and then passes it on to a workstation. The workstation detects the sound using an approach inspired by cross correlation[14] [136], and locates the sound using multilateration. The cross correlation-like approach is employed when processing the audio signal from each microphone to compare the incoming audio stream with a pre-recorded template sound. If the sound is detected in at least three of the four audio streams, multilateration can be used to determine its location. Multilateration works by measuring the time difference of arrival (TDOA) of the sound to the different microphones, and then intersecting the resulting set of conic sections.

4.4.2 Design

The Snap-detect system is designed using a single process to capture and analyze audio from the four microphones. Audio from the four microphones are split into four audio streams. These streams are processed independently with the goal of detecting a template sound in each of the audio streams. The template is a short recording of a user snapping his fingers.

To detect the template sound in a stream of incoming samples, three steps are taken. First, each sample in the incoming audio stream is replaced with the absolute value of itself, making all the samples in the incoming audio stream positive. Further, sample values that are very close to zero are rounded to zero to avoid amplifying noise when the samples are normalized[15]. Second, normalization is performed by first determining the maximum sample value in the incoming audio stream, and then dividing each sample value by the maximum sample value, resulting in an audio stream where the sample range is between $[0., 1.]$: $sample[i] = \frac{|sample[i]|}{\max|sample|}$. The pre-recorded template sound is processed in the same way once, when the Snap-detect application starts up.

The fourth step is to compute the product of each incoming sample with its corresponding template clip sample, sum the products and divide by the number of samples: $c = \frac{1}{n}\sum_{i=0}^{n} sample[i+j] * template[i]$ where c is the result, n is the

[14] An earlier implementation used standard cross correlation, but for performance reasons this approach was simplified to the one described in Section 4.4.2.

[15] This also avoids introducing denormal floating point values.

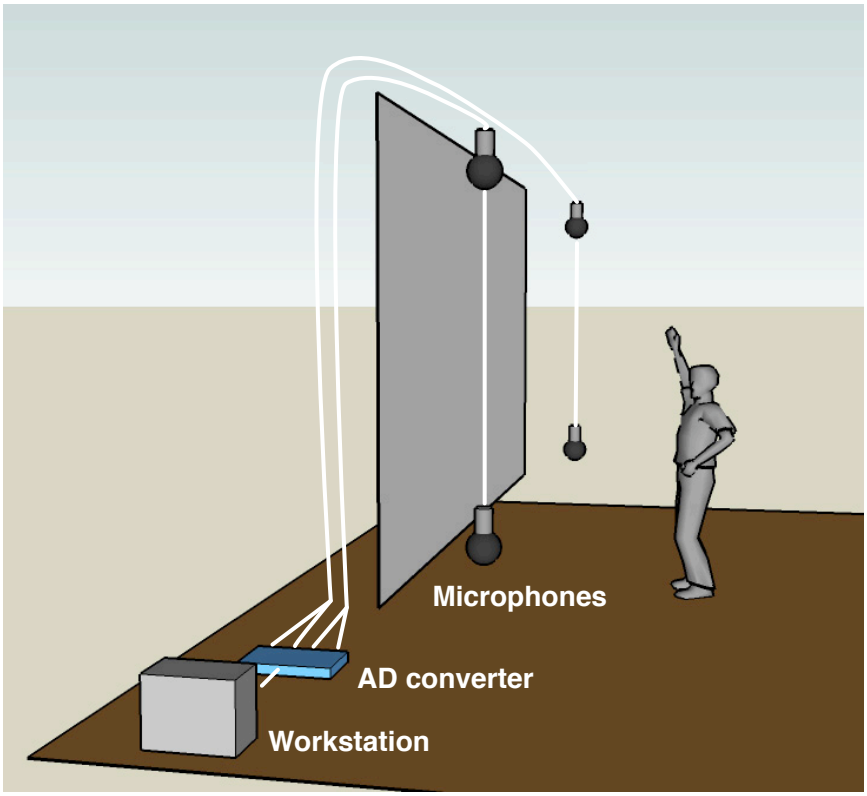


Figure 4.18: The architecture of the Snap-detect system. Four microphones connect to an AD converter, which digitizes the audio and sends it to a workstation. Credit for the human model: Google SketchUp.

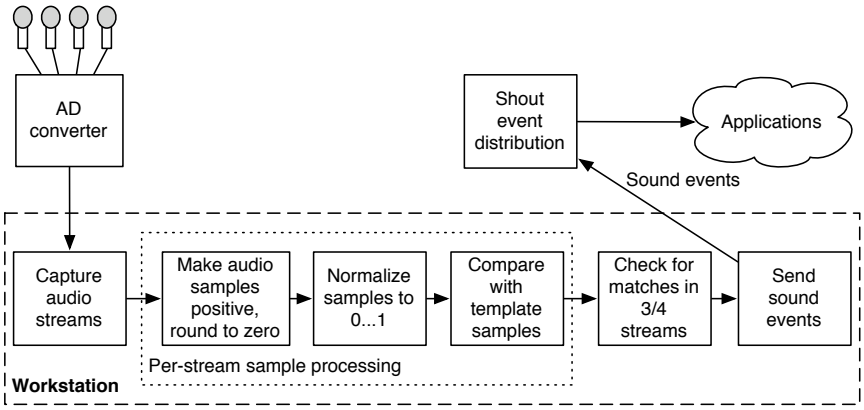


Figure 4.19: The design of the Snap-detect system.

number of samples in the template, j is the index into the incoming sample stream to use as the possible starting point for the snap, $sample$ is the incoming samples and $template$ the template samples. Then a weighted mean is updated as

$\mu_{weighted} = 0.99 * \mu_{weighted} + 0.01 * c$. If c is less than an experimentally determined threshold and the difference between the weighted mean and c is greater than the same threshold for more than 20 samples in a row[16], a snap or clap is defined as detected in the given stream. If the snap is detected in at least three of the four incoming audio streams, the source of the sound can be determined using multilateration.

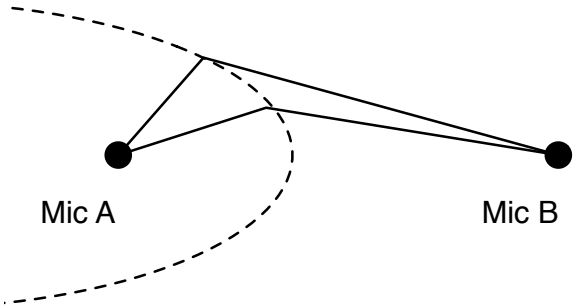


Figure 4.20: The difference in time from when the sound arrives at microphone A and microphone B is constant. The possible locations where the user may be located lie on the resulting conic section with focal point at the microphone which detected the snap first of the two microphones being examined.

Multilateration works by calculating the time difference between the first arrival of a sound, to its detection in the other audio streams. It has seen use in many different applications, from the ship navigation systems Decca and Loran-C to determining the location of aircraft using several radar sites. For the case of the Snap-detect system, multilateration is used to determine the location of the sound. Each pair of microphones where the snap has been detected gives rise to a conic (hyperbolic) equation where the focal point is at the location in space of one of the microphones, as illustrated in Figure 4.20. The curve indicates the potential locations of the sound source given the difference in arrival times between the two microphones. By determining the intersection of several such hyperbolic curves between different pairs of microphones, the source of the sound, and thus the location of the user's hand(s), can be estimated.

4.4.3 Implementation

The Snap-detect system has been implemented using the hardware listed in Section 2.2.2. The four microphones used are shown in Figure 4.21. The software is written in C using some inline PowerPC AltiVec (vector) intrinsics. Audio samples are read from the sound card using the open-source PortAudio library [137]. On startup, the software loads a template clip that will be matched against incoming

[16] At a sample rate of 48000 Hz, this corresponds to 0.00041 seconds.

audio samples. The template clip is 2048 samples long, or 0.04 seconds. The clip is pre-processed by replacing each sample with the absolute value of the sample, and then normalizing the result. To match the template clip against the incoming samples, the incoming samples are processed in the same way, taking the absolute value and then normalizing them.

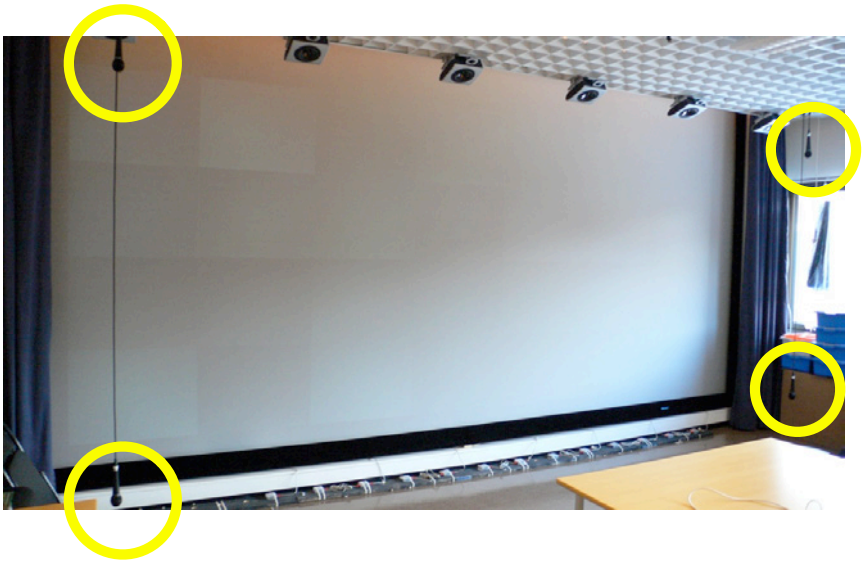


Figure 4.21: The deployment of the Snap-detect system for the Tromsø display wall. The yellow circles highlight the four microphones in use.

```
global detect_countdown, weighted_mean
.. after sample positivization and normalization ..
for i = 0 to window_size
  sum = 0
  for j = 0 to length(template_samples):
    sum += template_samples[j] * incoming_samples[i+j]
  c = sum / length(template_samples)
  weighted_mean = (weighted_mean * 0.99) + (c * 0.01)
  if c < threshold and (weighted_mean - c) > threshold:
    if detect_countdown == 0:
      .. mark channel as detecting a snap ..
    else:
      detect_countdown -= 1
  else:
    detect_countdown = 20
```

Listing 4.1: Pseudo-code to detect snaps in a single channel.

Matching the incoming samples to the template samples is done as shown in Listing 4.1. The incoming samples are slid over the template samples one sample at a time in a window sized at `window_size` samples. For each iteration of the outer loop, the samples from the template and incoming stream are multiplied with each other, and the result added to a total, before the total is divided by the number of samples

in the template. The `window_size` is the number of new samples being processed in the audio callback. The `threshold` was set to 0.0005 by trial and error.

Once a snap has been detected in one stream, subsequent detects in other streams are timestamped using the number of samples since the first detect. Since the sample rate is known, the distance the sound has traveled after being detected by the first microphone can be calculated as $num\_samples * \frac{speed\_of\_sound}{sample\_rate}$. Conics are fitted to observations from different pairs of microphones, before the resulting conics are intersected. The location of the snap is then determined by finding intersection points that are clustered within a short distance of each other.

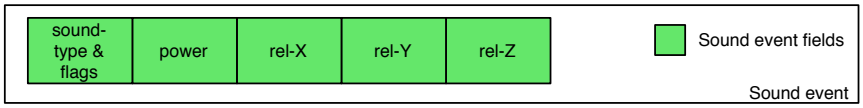


Figure 4.22: The sound event type. The standard Shout event fields have been left out of the figure. Each field is 4 bytes long.

The location of the sound is then sent as a sound event to applications using Shout. The format of this event is shown in Figure 4.22. The sound-type and flags field is used to store a 24-bit sound-type. This field is included for future expandability, if the system were ever to detect sounds other than snaps or claps. The remaining 8 bits of the sound-type and flags field are used for flags. The flags are used to indicate which of the relative X, Y and Z locations are valid. In the current implementation, the flags are always set to have valid X and Y locations. The relative X, Y and Z location fields indicate the relative location of the sound in front of the display wall, all ranging from 0 to 1. A value of $0, 0$ indicates that the sound occurred to the lower-left hand corner of the display wall, while a value of $1, 1$ indicates the upper-right hand corner of the display wall. The power and Z coordinate fields are currently unused.

Additionally, the raw sample timestamps are sent using Shout to an application that visualizes the calculations done by the Snap-detect system. Figure 4.23 shows a screenshot of the Snap-detect visualizer. The Snap-detect application itself runs as a user-interface-less process that sends raw events, while the visualizer can run on any (Macintosh) computer. The yellow dots indicate a cluster of intersection points whose mean location is used as the location of the snap.

4.5 Arm-angle system

The Arm-angle system was developed based on the idea that the direction in which a user's arm is pointing can be determined by looking for straight lines in images captured by a steerable camera. Straight lines are located using the Hough transform [138]. The resulting information can be used to enable users to point at targets

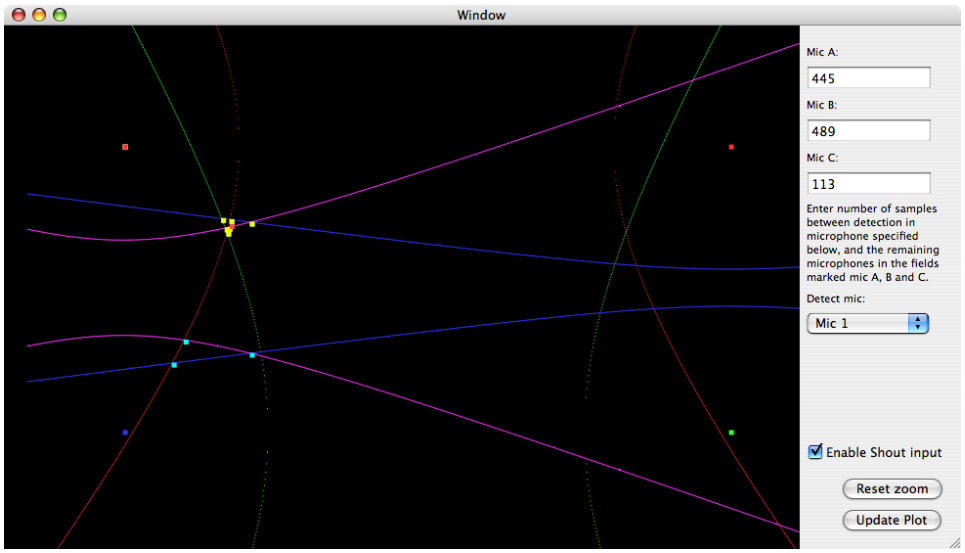


Figure 4.23: A screenshot from the Snap-detect visualizer as it detects a snap. The cluster of yellow dots indicate where the system has determined that the sound came from. The four microphones can be seen as colored dots in the four corners of the figure.

that are far away from the user on the display wall. Making use of the Snap-detect interaction space, the user "calls" the Arm-angle interaction space by snapping his fingers, which prompts the camera to be panned and tilted towards the location of the snap.

4.5.1 Architecture

Figure 4.24 shows the architecture of the Arm-angle system. A steerable camera mounted in the ceiling streams images to a workstation. The camera is located at the back of the room, pointing towards the Tromsø display wall. The workstation controls the camera's pan, tilt and zoom using a serial interface. When the user snaps his fingers, the camera is steered towards the location of the snap, so as to capture the user in its field-of-view. When the camera has stopped moving, the image analysis proceeds to detect straight lines in the image. If any are found, their angle and location relative to the display wall are sent to applications using Shout.

4.5.2 Design

The Arm-angle system does not attempt to recognize what is being used to point with, but only the direction in which the object points. The object can be anything that is straight, such as a user's arm or a stick. Figure 4.25 shows the design of the Arm-angle system. The design consists of two components: (i) A camera

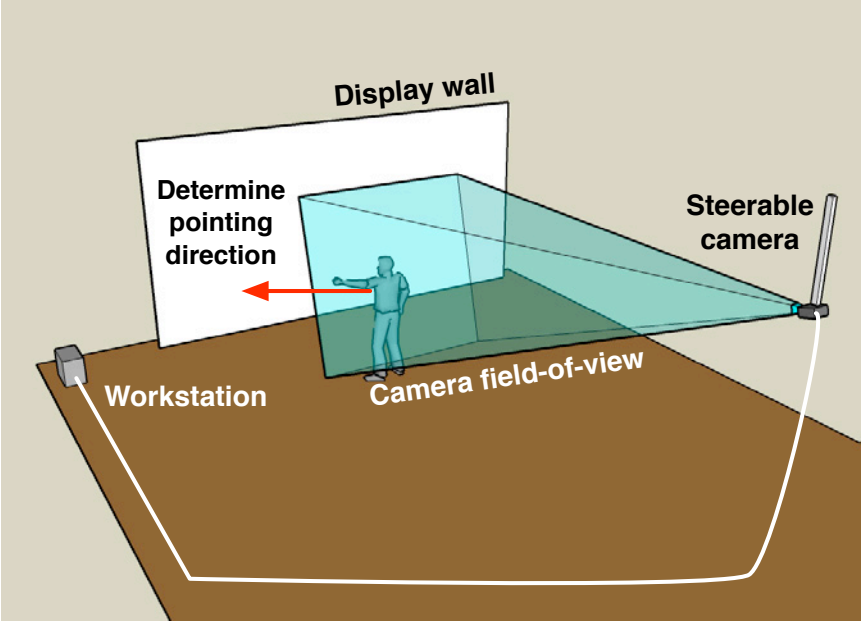


Figure 4.24: The architecture of the Arm-angle system. A steerable camera sends images to a workstation, which uses them to determine the angle at which the user's arm is pointing.
Credit for the human model: Google SketchUp.

control component; and (ii) an image processing component. The camera control component receives sound events from Shout, and in response to such events move the camera towards the location indicated by the sound event. It then waits for the camera to stop moving, before signaling the image processing component.

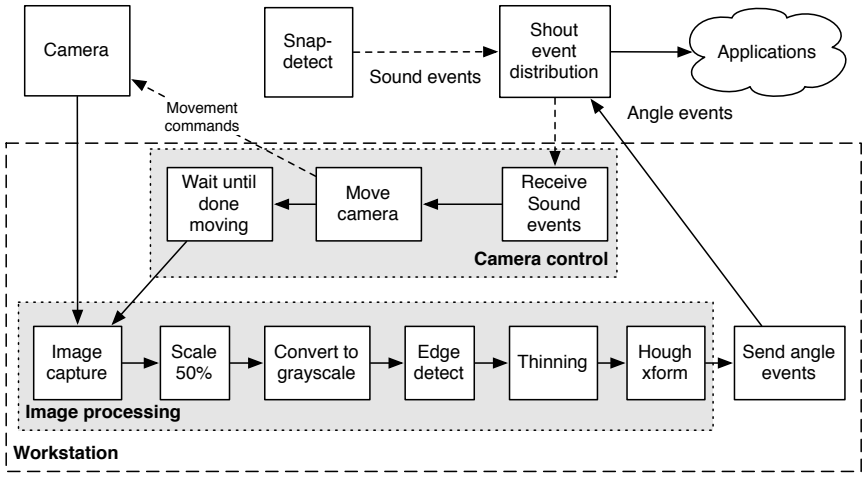


Figure 4.25: The design of the Arm-angle system.

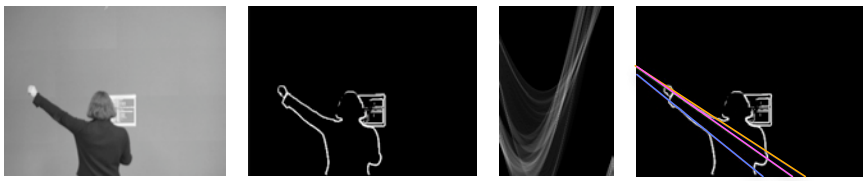
The image processing component captures images from the camera, and then processes the images as follows. First, the image is scaled down by 50%. This is

done because the camera used provides interlaced images, and as a side-effect reduces the amount of data to be processed. Next, the image is converted from color to grayscale, which further reduces the amount of data to process by a factor of three[17]. Edges are then detected in the image using the Sobel edge detector [139]. The resulting edge map is thinned, which reduces the thickness of detected edges and improves the output of the Hough transform.

The Hough transform is used to determine the parametrization of straight lines in the image. The Hough transform works by examining each pixel in the image. If the pixel is an edge (i.e., the pixel is white), a parametrization of all lines through that pixel are plotted in an accumulation buffer. When this is done for all edge pixels, the resulting accumulation buffer contains a set of votes for each possible line in the image. The parametrizations with the highest number of votes are then used to infer the direction in which the user's arm is pointing. If the resulting lines all point in same direction differing only by a few degrees, an angle event is created and sent to applications using Shout.

4.5.3 Implementation

The arm-angle component was implemented in C and using the camera and workstation hardware listed in Section 2.2.3. The camera is controlled using the open-source "devserv" software [140], and is moved in response to incoming sound events from the Shout event system. The camera zoom is fixed, and the location to which the camera should pan is determined by interpolating between the pan-values (in degrees) for the left and right edges of the display wall using the sound component's reported position along the horizontal axis (i.e., length-wise along the display wall).



(a) Input image (b) Edge detected (c) Hough xform (d) Lines super-imposed

Figure 4.26: The steps involved in determining the angle at which the user's arm is pointing. (a) The input image from the camera. (b) The image after edge detection. (c) The result of the Hough transform. (d) The dominant lines superimposed on the original camera image.

Figure 4.26 shows the images at various stages of the image processing. The system captures images from the camera at a rate of 8-9 frames per second. The images have a resolution of 720x540 pixels in interlaced[18] format. The image is scaled down by averaging neighbouring pixels to the left, right, up and down in the

[17] Converting from 24-bit color to 8-bit grayscale.

[18] Every other line of the image is from the previous image captured by the camera.

```
current_line = pixels
next_line    = &pixels[width]
output_line  = thinned_output
for y = 0 to height-2:
  for x = 0 to width-2:
    if is_edge(current_line[x]) and is_edge(current_line[x+1]) and
       is_edge(   next_line[x]) and is_edge(   next_line[x+1]):
       output_line[x] = 0
    else:
       output_line[x] = current_line[x]
  current_line = next_line
  next_line    = &next_line[width]
```

Listing 4.2: Pseudo-code to thin an edge-detected image.

image. Color to grayscale conversion is performed by averaging the red, green and blue color components. The Sobel edge detector is then used to locate edges in the image. The image is thinned using a simple algorithm that removes consecutive edge pixels along the horizontal axis of the image if at least two pixels are in a row at both the current line and the next line of the image, as shown in Listing 4.2. Finally, the Hough transform is applied to the thinned image. The output from the Hough transform is analyzed to locate high-intensity pixels, which indicate many votes for a particular line. The angle of these lines are then sent to applications in angle events.

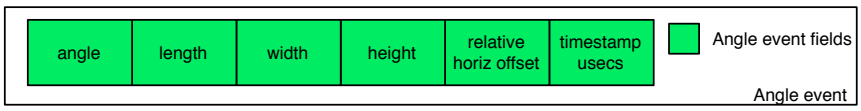


Figure 4.27: The angle event format. The standard Shout fields have been left out of the illustration. Each field is 4 bytes long.

The event format is shown in Figure 4.27. The angle and length values correspond to the line parametrization obtained through the Hough transform, which is the normal form of a line given as $\rho = x * \cos(\theta) + y * \sin(\theta)$, where ρ corresponds to the length field, and θ is given in the angle field. The width and height contain the width and height of the source image in which the line was found (i.e., $720 * 0.5 = 360$ and $540 * 0.5 = 270$). The relative horizontal offset field is the horizontal location of the last snap that caused the camera to move. Using the field, the line's position can be determined in front of the display wall.

4.6 Evaluation

There are several technical metrics that can be evaluated for the systems described so far, including latency, accuracy, precision, failure rates, CPU load, memory usage and bandwidth. Latency is the time between when a user takes an action, until

that action is reflected on the display wall. Accuracy determines how well the system is able to relate an object's physical location with its corresponding location on the display wall. Precision is the system's ability to consistently determine an object's location, regardless of whether the resulting location is accurate or not. Failure rates includes both the rate at which the system fails to detect an object when it is present, and the rate of false detections, i.e. how often the system detects objects that aren't actually present. CPU load, memory usage and bandwidth indicate the resource consumption of the system, and can be used to identify potential bottlenecks, for instance to improve latency by reducing the time spent computing object locations.

Of the three systems, this dissertation presents an evaluation of the Camera-sense system. The Snap-detect and Arm-angle systems have not been evaluated, a task which is currently left as future work. The evaluation will further focus on the Camera-sense system's latency, accuracy and precision. Informal measurements indicate that neither memory usage, bandwidth nor CPU load represent a bottleneck in the Camera-sense system. Memory usage is about 12 MB and 25 MB of real memory for the object locator and image processing applications, CPU load is about 10% for the object locator and 25% for the image processing application, and total bandwidth usage from all the image processing applications to the object locator is about 0.5 MB/second with 4-8 cameras detecting 1-2 objects. These informal measurements were all made on one of the Mac minis in the Princeton display wall, with graphical output from the applications disabled. Neither application has been optimized to reduce CPU load. A rigorous evaluation of these metrics has been left as future work, along with an evaluation of the system's failure rates.

The metrics latency, accuracy and precision were chosen since they represent the two characteristics end-users are most likely to notice when they use the system. If the latency is high, movements made by users will take a long time to be reflected by corresponding reactions on the display wall. This in turn can result in overcompensation or missed targets when users try to interact [141]. If the system exhibits low accuracy or precision, users will be unable to reliably touch the targets they are trying to hit on the display wall, making the system less useful. The evaluation of the Camera-sense system is limited to evaluating its performance for a single plane in 2D. The results presented here appear in [21, 22, 23].

4.6.1 Latency

The end-to-end latency of the Camera-sense system for locating an object in a single plane has been measured. The end-to-end latency is the time from when a camera starts acquiring an image, until data from that and all other cameras have arrived at the object locator, been processed and then sent to applications as object events. To measure the end-to-end latency, the latency introduced by the different components of the system has been measured and summed.

There are four main locations where latency is introduced in the system, as shown in Figure 4.29: (i) Camera image acquisition: The time taken from the camera starts acquiring an image, until the image is available to applications on the computer the camera is connected to; (ii) Image processing: The time it takes to process a single slice from one camera; (iii) Shout: The time it takes to send an event from an instance of the image processing application through the Shout event system to the object locator; (iv) Object locator: The time it takes to triangulate object locations and track the resulting objects in a single plane, including time spent waiting for 1D object locations from the image processing cluster.

Camera image acquisition latency

A custom measurement application was created to measure the image acquisition latency. The application measures the latency of one or several FireWire cameras by setting a computer's display to white, and then determine how much time passes before the white color is observed by the attached camera(s). The measurement application is running on the same computer that the cameras and the display are connected to. The cameras are pointed directly at the computer's display. For every measurement, the display is set to white for a duration of 0.3 seconds, before it is set to black for 0.7 seconds in preparation for the next measurement. This should give a good estimate of the time it takes for a camera to acquire an image, transfer it to the computer, pass through the operating system and FireWire camera library (libdc1394) before it is available to applications for processing.

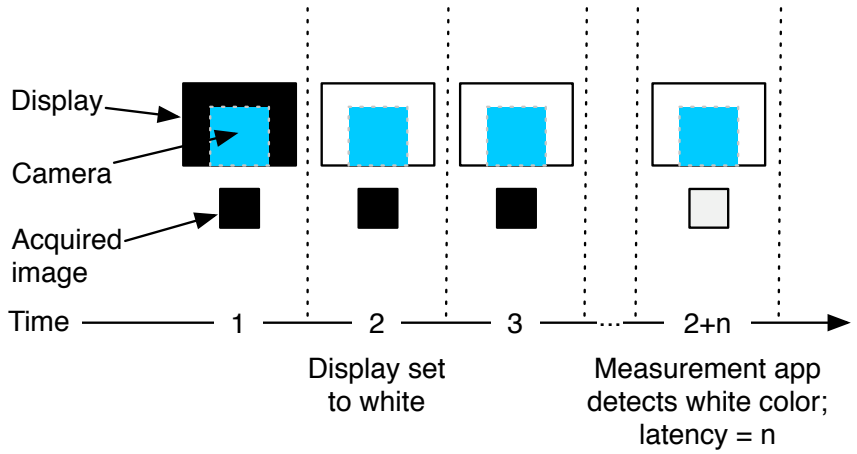


Figure 4.28: Measuring camera image acquisition latency. At time 1, the display is set to black. At time 2, the display is set to white, but the camera has not detected this yet. At time 3, the camera still has not detected the white display. At time 2+n, the image acquired from the camera is sufficiently bright that the measurement application determines that the camera has seen the white display. The image acquisition latency is set to n, and the next measurement is made by setting the display to black and repeating the process.

Figure 4.28 illustrates how one measurement is made. The display starts out being black, and then set to white for a duration of 0.3 seconds. To detect when the camera observes the white display, the measurement application examines a 20x20 square of pixels located at the center of the image acquired by the camera. Once the difference between the current and previous mean pixel values for this square exceeds 150[19], the camera is said to detect the white display, and a latency measurement is taken. After 0.3 seconds, the display reverts to black and stays that way for another 0.7 seconds.

The experiment was conducted with a single camera[20] attached to two different computers: A Mac mini (1.66 GHz Intel Core Duo, 512 MB RAM, Mac OS X 10.4.9) and a Dell workstation (3.0 GHz Intel Pentium 4, 2 GB RAM, Hyper-Threading enabled, Ubuntu Linux 6.10). The camera was mounted so that its lens was completely up against the (LCD) display's surface. The latency measurement application was started, before 1000 measurements were made, by alternating the display between black and white.

In some cases, the approach taken to detecting the white display might fail to measure a sufficiently steep increase in pixel intensity. When this happens, no measurement is made. One possible reason for such failed measurements may be that the camera has acquired an image halfway through when the display is set to white. When the acquisition ends, the resulting image will be gray, but not sufficiently bright to trigger the detection of a white image. In the next image acquired from the camera, the additional brightness would not be sufficient to break the 150 pixel mean difference required to detect the white image.

Image processing and object locator latency

To measure the latency incurred by image processing and the object locator, the two applications were instrumented. The image processing application was instrumented as shown in Listing 4.3. The purpose is to measure the time from when the camera library has an image available, until that image has been processed, and the resulting events queued for sending to the object locator using Shout. 1000 measurements were made for a single slice, with a single camera connected to a Mac mini (specifications as the one used in the camera image acquisition latency measurement).

Listing 4.4 shows how the object locator was instrumented. The purpose is to measure the time it takes from receiving the first 1D object event, until the events

[19] Pixel values range from 0-255. Thus, when the display goes from black to white, a large change in the mean pixel value is expected; where black = 0 and white = 255. The threshold of 150 represents a change in intensity from black by nearly 60%.

[20] In a second experiment, two cameras were attached to the computers, yielding results that were within the standard deviation from the results obtained using one camera. For this reason, the results obtained using two cameras are not included.

```
camera_image_acquire_thread:
  while true:
    image     = wait_for_image_from_camera()
    timestamp = gettimeofday()
    enqueue_image(image, timestamp)

image_processing_thread:
  while true:
    image, start = dequeue_image()
    for each plane:
      detect_objects_in_plane(image, plane)
    send_1D_object_events()
    stop    = gettimeofday()
    latency = stop - start
```

Listing 4.3: Pseudo-code demonstrating how the image processing application was instrumented to measure the latency incurred by image processing. A timestamp is taken when a new image is available from the camera. The image and the timestamp is queued, and then dequeued by a different thread which processes the image. When the processing is complete and any object events have been sent, another timestamp is taken and the latency measurement made.

have been processed, and the resulting 2D and 3D object events have been queued. Latency is introduced since: (i) Not all 1D object events arrive at the same time; and (ii) the object locator must wait to receive events pertaining to all cameras before it can use the events to detect objects. 1000 measurements were made while processing data from 12 cameras connected to 6 Mac minis.

Shout latency

The methodology used to measure the latency for distributing events using Shout appears in Section B.5.1, where both the methodology and results are presented. This Section includes uses the roundtrip latency results from the experiment with an event rate of 1 and 8 pong slaves, divided by two to arrive at the one-way latency. This is the mean latency that the Shout system can deliver for the system's deployment in Tromsø, where there are 8 Mac minis running one instance of the image processing software each. The results for the Shout latency reported in this dissertation differs from the one given in earlier papers [23, 22], as explained in Section B.6.

Results

Figure 4.29 illustrates the four components of the latency chain, from camera image acquisition, to image processing, Shout event distribution and the object locator and their associated contribution to the end-to-end latency. When the object locator

```
receive_events:
  while true:
    event = shout_wait_next_event()
    append_1d_observation(event)

append_1D_observation(event):
  if not have_start_timestamp:
     start = gettimeofday()
     have_start_timestamp = true
  if event is last from camera or event is no_detect:
    mark camera as ready
    perform_object_detection()

perform_object_detection:
  if all cameras are marked ready:
    intersect_lines(); detect_objects(); send_object_events()
    stop    = gettimeofday()
    latency = stop-start
    have_start_timestamp = false
```

Listing 4.4: Pseudo-code demonstrating how the object locator was instrumented to measure its latency. The start timestamp is set on reception of the first 1D object event. The events are processed when all cameras are "ready." 2D and 3D object events are then sent, before the stop timestamp is taken and the resulting latency measurement calculated.

has finished detecting and locating objects, the Shout latency is incurred once more as events are sent from the object locator to applications.

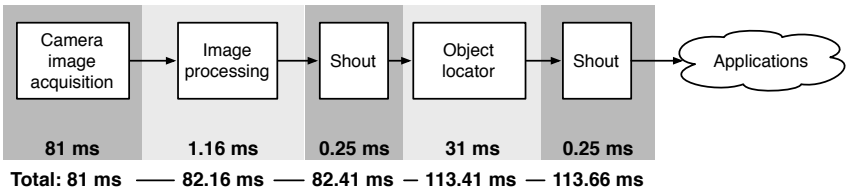


Figure 4.29: The latency measurement results, with individual latency contribution listed within each box, and the cumulative latency shown at the bottom.

Table 4.3 shows the latency measurements in more detail, with the number of samples, mean latency contribution, standard deviation and minimum/maximum values. For the camera image acquisition, the results in parentheses are from the measurements on the Linux workstation. The other experiments were not conducted on Linux since the image processing and object locator applications were written targeting Mac OS X. For the camera image acquisition, the number of samples is lower than the expected 1000 due to some failed measurements, as described earlier.

The Camera-sense system's mean end-to-end latency is 113.66 ms, including the added latency required to deliver the resulting 2D and 3D object events to applications. The biggest contributor to latency in the system is camera image acquisition, which accounts for 71% of the latency. On the average, acquiring images on the

	Cam. img. acq.	Image proc.	Shout	Object loc.	Sum
N	923 (852)	1000	800000	1000	-
μ	81 ms (93 ms)	1.16 ms	0.25 ms	31 ms	113.41 ms[a]
σ	10 ms (9 ms)	0.11 ms	0.11 ms	10 ms	-
min	58 ms (72 ms)	0.97 ms	0.23 ms	0.008 ms	59.21 ms
max	104 ms (114 ms)	3.3 ms	40.88 ms[b]	139 ms	287.18 ms

Table 4.3: Detailed latency measurement results. N is the number of samples, μ is the mean and σ the standard deviation.

[a]The end-to-end latency is 113.66 ms, since the Shout latency must be counted twice: Once to send events from the image processing cluster to the object locator, and once more to send events from the object locator to applications.

[b]These maxima are very rare: Of the 800000 measurements, only 540 (0.06%) are above 1 ms, and 59 are above 10 ms.

Linux workstation incurred about 10 ms more latency than image acquisition on Mac OS X. The second biggest contributor is the object locator, which contributes 31 ms (27%) to the end-to-end latency. Both the camera image acquisition and the object locator incur about one order of magnitude more latency than other parts of the system.

The results are discussed in Section 4.7.

4.6.2 Accuracy and precision

The Camera-sense system's accuracy and precision in 2D within a single plane has been measured. Designing an experiment to measure the system's accuracy and precision in an empirical and objective manner that enables other researchers to consistently reproduce the results is challenging. Measuring the system's accuracy requires that a physical object is placed at a known location, before the system is used to determine its location. The system's accuracy at that point is then defined as the distance from the known (target) location to the observed location. The system's precision is determined by calculating the standard deviation of several accuracy measurements.

To make such an experiment fully repeatable while measuring the system's accuracy and precision over a large area, a mechanical arm or similar would have to be constructed that could interact with the system in a pre-determined and consistent way. This approach is very laborious. An alternative is to have a user do these pre-determined movements. However, such an experiment can never be fully reproduced, since the interactions made by each user would undoubtedly change from experiment to experiment. Users might adapt to the experiment, accidentally do things in a different way from an earlier experiment, or affect the system in a number of other ways which are hard to control for. Further, it is highly unlikely that other researchers would have access to the same set of users in an attempt at

reproducing the experiment.

Experiments involving users are thus unsuited to methodically exploring how changes to the system's various stages affect the total system accuracy and precision. The results can not be used to measure the effect of varying lighting conditions, different calibrations or changes to other parts of the system, as one must have confidence in that the outcome reflects the actual changes to the system, and not changes in the way a given user interacts with the system during the experiment. Instead, such an experiment can only provide an approximation of the system's accuracy and precision for that given user.

Methodology

The Camera-sense system's precision was calculated based on the accuracy measurements. To make the accuracy measurements, the following methodology was employed. The experiment was conducted on the Princeton display wall, which measures 272x202 cm. A display application running on the display wall was written to show a set of 100 targets, one after another, as 10x10 pixel squares spaced evenly in a 10x10 grid. Each target is touched in turn by a user. When a user touches a target, feedback is given in the form of a fountain of particles appearing at the location of the touch. The targets are located within an interior rectangle on the display wall measuring 167x60 cm. This area represents the area where interaction is most likely to occur: Any higher, and users would have to stretch to reach the targets, and any lower would require users to bend. The horizontal extent was chosen based on the fact that at least three cameras must see an object before it can be located: To the far left and far right of the display wall, initial touches are thus not recognized[21].

A master application runs the experiment by instructing the display application to show each target one after another. The display application responds by displaying the target. For each target, the master receives object events for a single plane (the plane closest to the wall) from the Camera-sense system. It extracts the location and object extent from the events, and records them along with the current sample count and target location.

For each target that the user touches, the master discards the initial 15 object events, before it logs data from the following 30 object events. The first 15 object events (which represent about half a second of touching given the cameras' frame rate of 30) are discarded to let the user place his finger on the target, in an effort to get more consistent results. Once the master has logged 30 samples for the current

[21] Touches can be tracked to the far left and right once they have been recognized, as the system in these cases can utilize data from just two cameras to track the object. To extend the system to allow initial detection of objects along the full width of the display wall, additional cameras could be placed to the left and right of the display wall. Another approach would be to increase the spacing between cameras to cover a larger area.

target, it instructs the display application to flash the display and then show the next target. At this point, the master begins a grace period where the user is given time to remove his finger before touching the next target. The purpose of this is to avoid incorporating object events stemming from the previous target into the current target. The grace period lasts for one second after the master detects that the user has removed his finger from the display wall.

The experiment was conducted twice, with a different user in the second experiment.

Results

Tables 4.4 shows the accuracy and precision results from the two experiments. The accuracy is given as the mean distance from the targets to their corresponding observations, and the precision is shown as the standard deviation. The table also lists the mean horizontal and vertical distance from targets to observations, which represents the system's accuracy along the horizontal and vertical axes.

Experiment	μ	σ	μ_{dx}	μ_{dy}
User 1	1.12 cm	0.72 cm	-0.21 cm	-0.47 cm
User 2	1.24 cm	0.69 cm	-0.40 cm	-0.25 cm

Table 4.4: Accuracy and precision results from the two experiments. μ is the mean distance from all observations to their corresponding targets, and represents the system's overall accuracy. σ is the standard deviation, and represents the system's overall precision. μ_{dx} and μ_{dy} are the mean horizontal and vertical distance from an observation to its target.

The system has a mean distance from targets to observations of 1.12 cm for user 1, and 1.24 cm for user 2. These values are well within the standard deviations for the two users, indicating that the system exhibits similar accuracy for the two users. The mean horizontal distance from targets to observations is -0.21 cm and -0.40 cm respectively, indicating that the observations are skewed slightly to the right of their intended target. The mean vertical distance is -0.47 cm and -0.25 cm for users 1 and 2, which puts the mean slightly above their intended target.

Figure 4.30 shows a plot of the results from the experiment conducted with user 1. The X and Y axes indicate the offset from the left and bottom sides of the Princeton display wall, with all measurements in centimeters. The measurements are plotted as dots and the targets are shown as red circles in the figure. The same plot for user 2 is shown in Figure 4.32(a). Ideally, the dots should be centered inside the red circles, indicating perfect accuracy. Perfect precision would be indicated by all dots for each target lying on top of each other (i.e., appearing as a single dot).

Two boxes A and B are highlighted in Figure 4.30. The boxes show areas where the Camera-sense system exhibits low and high accuracy and precision. In box A, the vertical spread of the dots indicates that the observed locations flicker up and down,

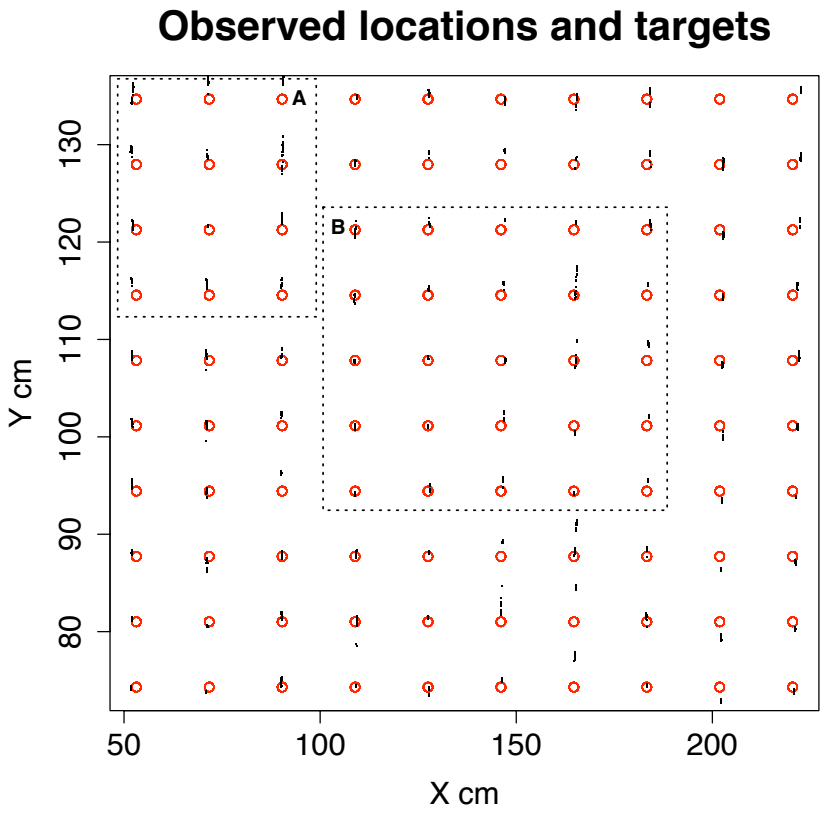


Figure 4.30: 100 targets (red circles) and the locations (dots) detected by the Camera-sense system for each target for user 1. The X and Y axis show the horizontal and vertical location on the 272x202 cm Princeton display wall. Boxes A and B highlight areas where the system exhibits low and high accuracy and precision.

resulting in relatively low precision in this area. The lack of precision also results in lower accuracy in this area. In comparison, the horizontal location remains fairly stable, indicating that the system's precision is better along the horizontal axis than along the vertical axis. In box B, the system delivers better accuracy, with most observations being closer to their targets. The system's precision in this area is also better, with less variation in the measurements.

Figure 4.31 shows the distance from each target to its associated observations for user 1, with the corresponding plot for user 2 shown in Figure 4.32(b). Negative values on the X or Y axis indicate that an observation is above or to the left of the target. Figures 4.33 and 4.35 show histograms of the same data in the horizontal and vertical directions for users 1 and 2. The histograms show how the system's accuracy differs along the X and Y axes, with more samples clustered within a narrow range for the dx values compared to the dy values.

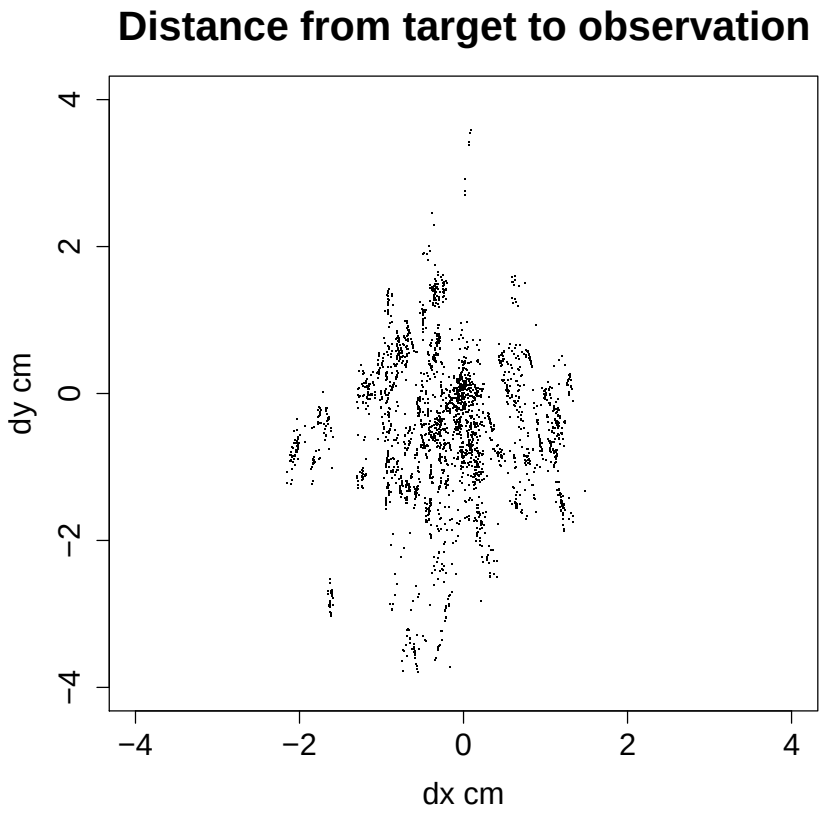


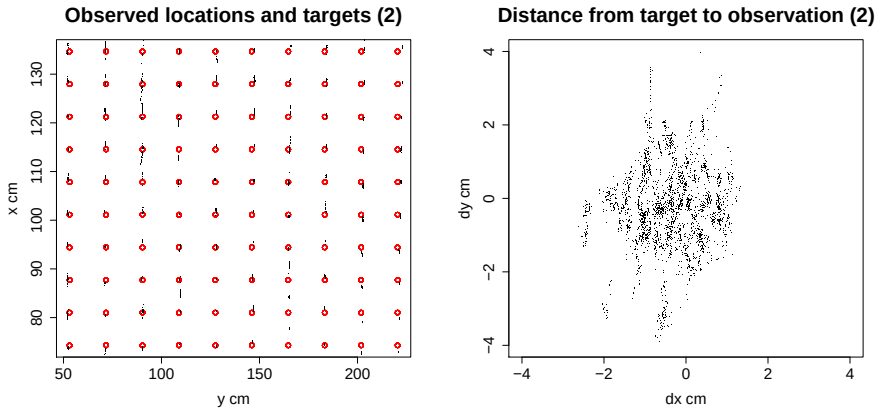
Figure 4.31: The difference between the target location and the actual sample location for every sample, measured in centimeters, for user 1.

Figures 4.34 and 4.36 show the system's per-target precision using histograms of the standard deviation for each target's 30 samples. For user 1, 90% of the targets had a horizontal standard deviation less than 0.08 cm and a vertical standard deviation less than 0.47 cm. For user 2, 90% of the targets had a horizontal standard deviation less than 0.16 cm and a vertical standard deviation less than 0.68 cm.

Table 4.5 shows the mean object extent, its standard deviation as well as the minimum and maximum object extent recorded for the two experiments. The object extent is given in pixels, and is fractional since the object locator calculates the object extent as the mean of the extents from the cameras that detect a given object.

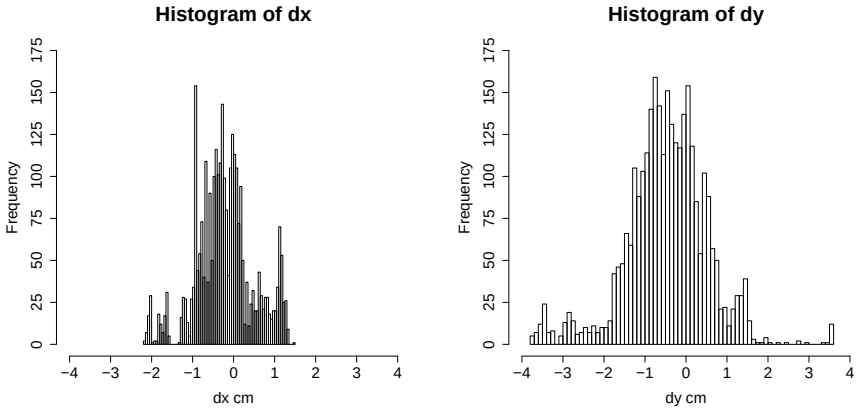
Experiment	μ_{obj}	σ_{obj}	min	max
User 1	4.45 px	0.82 px	2.34 px	6.49 px
User 2	4.26 px	0.72 px	2.00 px	6.49 px

Table 4.5: Results for the object extent recorded as part of the two accuracy experiments.



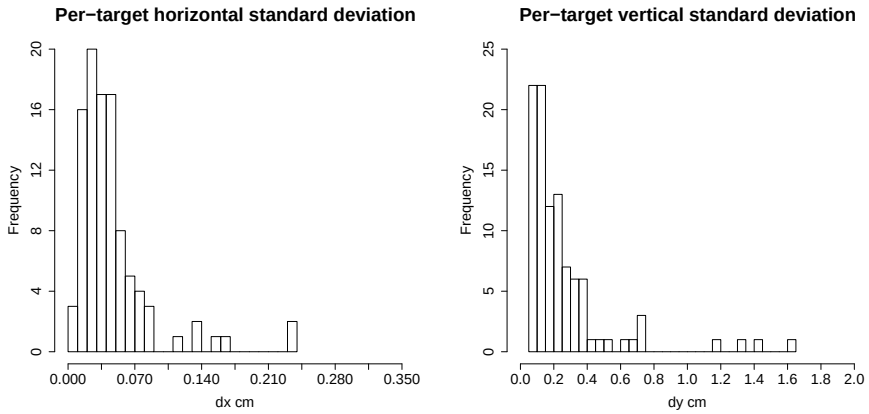
(a) (b)

Figure 4.32: Results from the experiment conducted with user 2. (a) A plot of the measurement results (dots) compared to the target locations (red circles) and the (b) difference from target and observed location are shown.



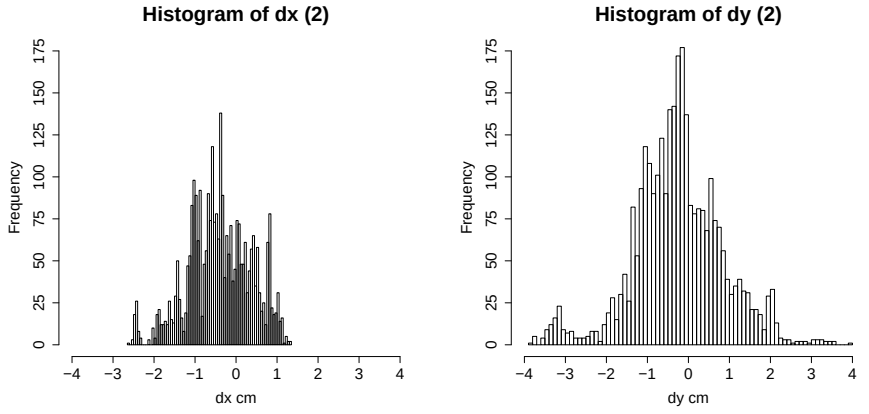
(a) (b)

Figure 4.33: Histograms of the horizontal (a) and vertical (b) distance from each sample to its target for user 1.



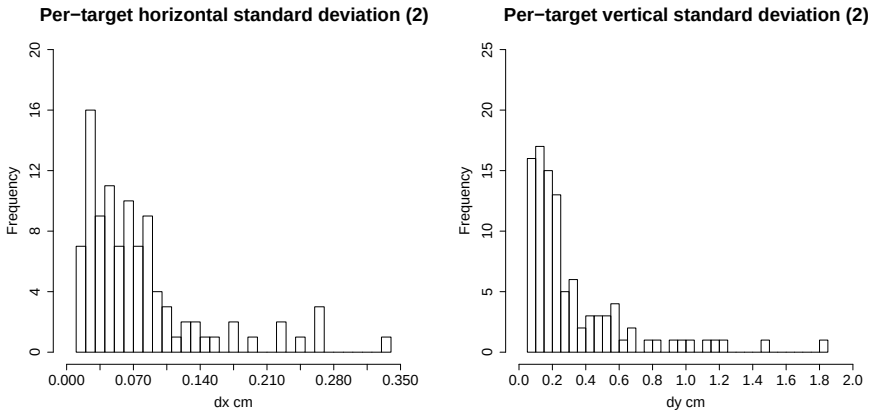
(a) (b)

Figure 4.34: Histograms of the per-target horizontal (a) and vertical (b) standard deviation for user 1.



(a) (b)

Figure 4.35: Histograms of the horizontal (a) and vertical (b) distance from each sample to its target for user 2.



(a) (b)

Figure 4.36: Histograms of the per-target horizontal (a) and vertical (b) standard deviation for user 2.

4.7 Discussion

The Camera-sense system's end-to-end latency was 113.66 ms, of which the majority is due to the time taken by the commodity web cameras to acquire images and transfer them to the computer. With better camera technology, the latency incurred by the cameras could likely be reduced, either as an artifact of using cameras with higher framerates, or smaller on-camera image buffers. The operating system is already thought to have reduced the image acquisition latency by more than 20 ms between the first experiment results presented in [142] and a repeat of the experiment in [22]. This dissertation presents the more recent results from [22]. The camera image acquisition latency experiment when conducted on the Linux workstation yielded a mean latency that was about 10 ms higher than the corresponding results on Mac OS X. Since the computer hardware used to measure the Linux latency differs from the hardware used to measure the latency on Mac OS X, the difference in latency can not be attributed to the operating system directly. The difference comes either from the different hardware used, the differing operating systems, or a combination of the two.

The object locator latency correlates well with the camera frame rate at 30 frames per second, which corresponds to a new frame every 33 ms ($\frac{1s}{30} = 0.033s$). The latency is as expected given that the object locator must wait for all events related to all cameras before processing them. Since the cameras are not synchronized, two frames from two different cameras may be captured as far apart in time as 33 ms. Thus, the worst-case is that the object locator must wait for up to 33 ms to receive events. The measurements show that the object locator on the average waits for 31 ms. Cameras with higher framerates would reduce the latency incurred by the object locator, since the maximum inter-camera image acquisition time difference

would be reduced. This wait could also be eliminated if a different approach to triangulating object locations were used. In this alternative approach, object locations would be updated every time a new 1D object event is received by the object locator. This approach has not yet been explored.

The Camera-sense system was measured to have a mean accuracy of 1.12 cm and 1.24 cm for users 1 and 2, with a precision of 0.72 cm and 0.69 cm. There are several factors that affect the system's accuracy and precision: (i) Object speed; (ii) lack of camera synchronization; (iii) camera frame rate; (iv) lighting; (v) precision of foreground and background segmentation; (vi) object extent; (vii) camera placement and alignment; and (viii) calibration.

Since the system makes use of unsynchronized cameras, moving objects influence the system's accuracy and precision. With unsynchronized cameras, the difference in inter-camera image acquisition time must be taken into account. For instance, with cameras running at 30 fps, two cameras may end up acquiring images as much as 33 ms apart. These images will still be used together to determine the location of objects. If the object is moving, its 1D location in the second image will be slightly different from where it was in the first image.

Using synchronized cameras, the problem of accurately determining the location of moving objects would be resolved. In this case, all the cameras would acquire each image at practically the same moment in time, resulting in moving objects no longer shifting in location in between images from different cameras. The drawback is far more expensive cameras, since camera synchronization (external triggering) is a feature that is only found on high-end cameras, such as the $500 Imaging Source DMK 21BF04.H.

Rather than using synchronized cameras, the system's accuracy and precision can also be improved by using cameras with higher frame rates. For instance, increasing the frame rate from 30 to 60 would halve the maximum inter-camera image acquisition time from 33 ms to 16 ms. A higher camera frame rate also has the benefit of reducing the object locator's latency, and increasing the rate at which 2D and 3D object events can be sent to applications. The drawback to higher camera frame rates is the higher bus bandwidth requirements. For instance, transmitting 640x480 in 8-bit grayscale at 30 fps consumes 8.78 MB/s, not including packet overhead. The maximum capacity of a FireWire 400 bus is about 47.68 MB/s (400 MBit). Of this, at most 80% of the bus capacity can be used for isochronous traffic (38.14 MB/s) [132], thus at most four cameras can be supported at this rate[22]. Doubling the framerate doubles the bandwidth requirements, and reduces the number of cameras that can used simultaneously on the same bus.

The experiment conducted to evaluate the system's accuracy and precision was

[22] In practice, the limit is often three. Most FireWire cards limit the number of isochronous contexts to four, and the operating system may use one of them for its own purposes, such as providing IP networking support over FireWire.

designed to eliminate object speed, lack of synchronization and camera frame rate as factors. The experiment measures the system's accuracy for stationary objects, which makes object speed and lack of camera synchronization less relevant. The camera frame rate impacts latency and the accuracy for locating moving objects; since latency is not an issue in the experiment and the objects to be located are kept stationary, the (constant) camera frame rate should not affect the experiment's outcome.

Since the system is designed with the assumption that the background is static and brightly lit, low lighting will affect the accuracy of the foreground/background segmentation. In lowly lit conditions, the contrast between the foreground object to be detected and the background is lower, making it harder to separate the two. Camera noise also increases under low lighting. The experiments were conducted under fairly good lighting conditions, where parts of the scene were well-lit by two lamps mounted in the ceiling, whereas other regions of the scene were darker, as shown in the sample image in Figure 4.5.

The accuracy of the foreground/background segmentation is important in order to determine where an object appears in an image, which ends up yielding the 1D locations and object extents that are later used to triangulate the 2D object location within a plane. Since the segmentation algorithm uses a dynamic background, stationary objects will eventually be incorporated into the background. The use of a dynamic background should not affect the evaluation to a great extent, since the object is only held in place for about two seconds in total. During this time, the implementation incorporates 0.005% of the current foreground image into the background image about 60 times, or about 0.3% in total for each target point.

The accuracy of the segmentation helps explain some of the results obtained during the evaluation. When the computed 2D location of an object exhibits vertical jitter (i.e., seemingly random changes in location up and down), the root cause is often one or two cameras that oscillate between detecting the object and not detecting the object, or detecting the object at slightly different 1D locations due to random noise or lighting effects. Figure 4.37 illustrates the effect. In these cases, the output from the oscillating camera(s) are inaccurate, since they either fail to detect the object, or when they do detect it, provide an incorrect 1D location for the object.

Object extent also plays a role in the accuracy of the system. The object extent has not been entirely eliminated as a factor in the experiments, since the extent of a user's finger may vary slightly depending on the exact finger pose. While the object extent is not directly used in the triangulation of objects (only the center of an object is used), it does give an indication of how precise the detection of an object is. As part of the evaluation, the object extent was also recorded. It does not vary much, ranging from 2.34 and 2.00, to 6.5 pixels with a mean of 4.45 and 4.26 for the two different users. This indicates that the user is not holding his finger completely steady, that the accuracy and precision of the foreground/background segmentation varies, or both. The end result is that the detected 1D location shifts,

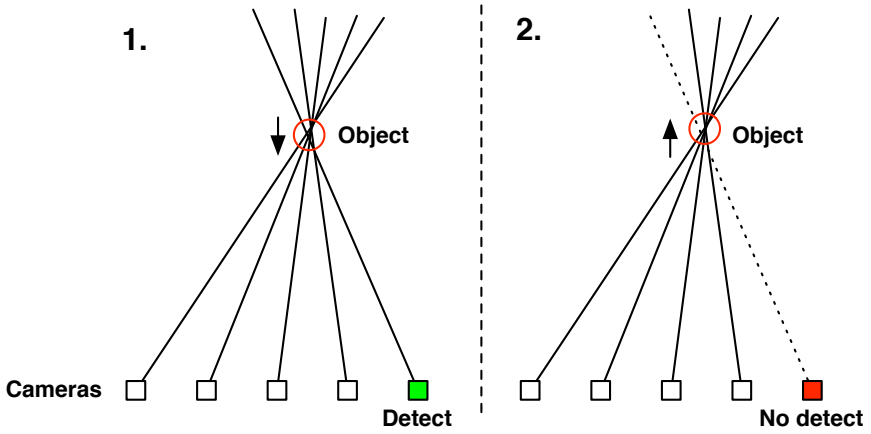


Figure 4.37: (1) The presence of a slightly inaccurate 1D location from the highlighted camera pushes the location of the detected object down. At the same time, the X location is not modified as much, since it has much better "support" from the horizontally spread-out cameras. (2) When the inaccurate camera no longer sees the object due to noise, lighting or simply because the object disappears from the camera's field-of-view, the object's location goes back up to the more "correct" location as determined by the remaining cameras. If the camera repeatedly switches between detecting and not detecting the object, the result is vertical jitter.

which in turn yields inaccurate 2D object locations.

Placement and alignment of the cameras is important to provide even camera coverage within the interaction space created by the system. The spacing between cameras affects the resulting accuracy of the system; wider spacing would reduce the accuracy, while narrower spacing would enable different objects to be seen by more cameras and thus increase the system's accuracy. Spacing also impacts the system's precision.

For three cameras to all see an object, their field-of-views must overlap at the location where the object is. For this reason, the camera spacing also affects the lowest height at which objects can be detected. The placement and alignment of the cameras was kept constant during the experiments, and thus should not impact the results beyond the accuracy at which they were initially mounted (i.e., if a camera was already out of alignment at the time the experiments were conducted, this would clearly affect the resulting accuracy).

Calibration is the final factor that impacts the system's accuracy and precision. Calibration and camera placement represent two interrelated factors. Any time the camera placement or alignment is changed, the system should be recalibrated. A reasonably accurate initial setup of the cameras is also necessary for the calibration to succeed. The evaluation has shown that most measurements had a standard deviation less than about 0.1 cm horizontally, and 0.5 cm vertically for one of the users. This indicates that the system is fairly precise in where it locates the detected

objects, even though the per-target accuracy varies. Since the system has regions where it achieves very good accuracy and precision (such as the region highlighted in box B in Figure 4.30), it is likely that improving the calibration of the system would result in higher accuracy and better precision across the entire display wall. To improve the calibration, a more sophisticated camera model would be necessary. The current model is limited in several ways, including no Y axis rotation and limited handling of lens distortion.

The Camera-sense system can be scaled to cover wider or narrower display walls by adding additional cameras and computers, as has been demonstrated with its deployment at both the Tromsø and Princeton display walls, using 16 and 8 cameras respectively. The informal experiments conducted to measure the CPU load and memory usage of the image processing and object locator applications did not point to any performance bottlenecks. However, with enough cameras, it is possible that the object locator or the Shout event system may become bottlenecks, since they are the centralized components in an otherwise embarrassingly parallel system. Parallelizing the object locator would be one approach to remove the object locator as a potential bottleneck; this could for instance be done by having one locator handle cameras $[1, j]$ while a second locator handles cameras $[j + 1, n]$. A more sophisticated solution would allow some overlap, such that both locators would process data from cameras near the border between the two to avoid introducing "dead zones." At present such a parallelization is future work. If the Shout server were to become a bottleneck, removing it would either require re-architecting the event system and move away from the current centralized architecture, or develop a custom protocol for exchanging information between the image processing applications and the object locator instance(s) independently of the Shout system.

4.7.1 The Camera-sense system and the state of the art

Comparing the Camera-sense system to the state of the art is challenging, since the literature does not always report on the accuracy, precision or latency of the developed systems, such as the Microsoft Surface [13], the Duke Multi-touch wall [108], or the TouchWall [51]. However, informally and judging by videos available of these systems, their accuracy appears to be better than the Camera-sense system. They are camera-based, so the camera image acquisition latency is likely to be similar to or better than the Camera-sense system, depending on the cameras used and the frame rates they operate at.

The accuracy reported for the FTIR multi-touch wall [12] is 1 mm, which is about an order of magnitude better than the Camera-sense system. The latency is likely somewhat lower than the Camera-sense system, since the camera used runs at 30 frames per second and the FTIR multi-touch approach does not employ multiple cameras; thus, there is no need to wait for data from more than one camera before updating the object state.

In the laser-range finder approach taken in [110], the reported accuracy of the laser range finder is about 1.4 cm with an update rate between 20 and 30 Hz. The Camera-sense system has a similar level of accuracy, at about 1.1-1.2 cm. On the assumption that the range finder sends position data continuously, its latency is likely in the 33-50 ms range (depending on the rate at which it scans). However, their use of a 4-tap FIR[23] filter to reduce the impact of noise adds latency to their system, resulting in a total system latency that is a factor two to three higher than the latency from the range finder alone, depending on what kind of filter is used. This places it on par with or slightly worse than the Camera-sense system.

4.8 Lessons learned

Through the design, implementation and evaluation of the interaction space systems, a number of lessons have been learned. The Camera-sense system began as a system to provide 2D multi-touch capabilities for the Tromsø display wall, before being extended to provide 3D input. The early versions of the Camera-sense system employed a "strange" coordinate system, where 1 unit along the horizontal axis was equivalent to the (expected) distance between the cameras, and to make things more complicated, one unit along the vertical axis represented a longer distance than one unit along the horizontal axis. Initially having a coordinate system that was independent of the camera placement seemed like a good idea, but made tasks like properly calibrating the system more difficult and also made it hard to judge the system's accuracy in an informal manner, since coordinates would have to be manually converted to the metric system. Eventually, the metric coordinate system was adopted instead.

Placing the cameras along the floor – rather than along the ceiling – was done due to the ease at which cameras could be moved and adjusted. However, one disadvantage of mounting the cameras along the floor is that they frequently are knocked out of position, due for instance to users bumping into the cameras with their feet or cleaners hitting them with their mop while cleaning the floors.

The event format used by the Camera-sense system could be simplified. The initial design goal for the verbose format (which includes the camera location and rotation in *every* 1D object event sent from the image processing cluster) was to enable different applications to visualize the "raw" 1D object events. However, from a design and efficiency viewpoint, this information should have been kept local to the object locator. A better design to enable visualization of raw data would be to have applications query for the current camera configuration, and let the object locator respond to such queries.

The Camera-sense system's 3D capabilities gives it much potential. While it does

[23] Finite Impulse Response.

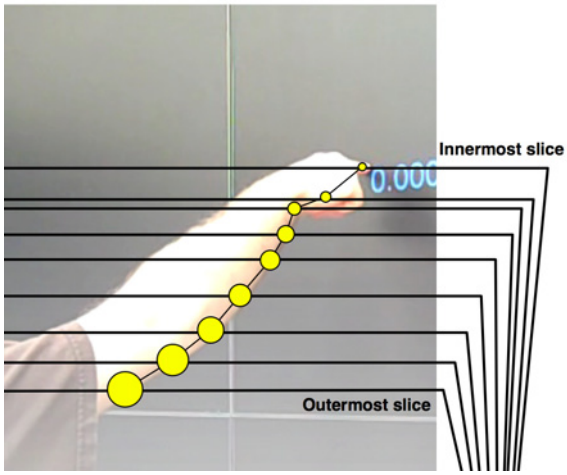


Figure 4.38: An illustration of how the skeleton-based 3D object would appear to an application.

not cover as much area in terms of depth as comparable systems [14, 56], it improves on their capabilities by being able to track objects over a larger area. It can further track not only an object's "tip" in 3D, but also its extent further back in the scene, as illustrated in Figure 4.38. Each 3D object is composed of one 2D object location per plane. These 2D locations are strung together, and could be used to determine the angle at which the object is pointing[24]. The 3D capabilities of the Camera-sense system also mean that it can replace the Arm-angle system, although a component incorporating the Arm-angle functionality into the Camera-sense system has not yet been implemented.

During the work with the Camera-sense system, several different approaches to creating the background lighting have been explored. For the initial 2D-only instance of the system built in Tromsø, two rows of commodity Christmas lights are mounted in the ceiling to generate a bright background for the system, as shown in Figure 4.39. Another approach that has been attempted, also with the 2D-only system, has been to use infrared light. A narrow strip of IR LEDs was built[25], and cheap IR-pass filters using exposed camera film was used to block visible light from the cameras. This approach had potential, but the LEDs used had a too narrow viewing angle which made them invisible when placed to the far left or right of a camera's field-of-view. In Princeton, the ceiling was more brightly lit than in Tromsø, so no additional lighting (visible or IR) was necessary.

[24] One way to do it would be taking the 2D location for the innermost and outermost slices and projecting a line through them.

[25] Courtesy of the Technical Staff at the Department of Computer Science in Tromsø.

Figure 4.39: The Christmas lights being used to create a bright background for the Camera-sense system in Tromsø.

4.9 Further improvements

There is more work left to both improve the systems presented, and to fully evaluate them. The Camera-sense system can be improved in many ways. The evaluation points toward the camera calibration as a likely reason for the observed inaccuracies. Reworking the calibration and using a complete camera model should go a long way towards improving the system's accuracy.

The Z-axis coordinates of 3D objects should be properly converted to real-world (metric) units, and not taken as a direct conversion of the plane index as now. The 1D object event format could also be made more efficient, as discussed earlier, and different approaches to back-lighting should be explored. One possibility beyond the ones already explored, is using more powerful infrared lights (essentially, heat sources) to illuminate the ceiling.

The object locator's approach to isolating clusters of intersection points could also be improved, since it at present does not take into account clusters with a bounding box that is tall, but very narrow. The very first attempt at isolating point clusters calculated the area of the bounding box and used that as the "measure of clustering." However, this approach was highly prone to creating point clusters with very narrow but tall bounding boxes, even more so than the current approach which uses the distance from the top-left corner of the bounding box, to the bottom-right corner of the box. A better solution may be to exclude points based on each point's distance to every other point and remove the point which has the biggest total distance to all the other points. This approach has not yet been explored.

There are also several aspects of the Camera-sense system that should be further evaluated. The latency of the system has been measured when processing just a single plane; it is possible that processing additional planes could lead to an increase in latency which the current measurements do not reflect. Further, the evaluation only measured the accuracy and precision for tracking stationary objects in a single plane. Additional evaluation should include the system's accuracy and precision for tracking moving objects in 2D, and its accuracy and precision when tracking both stationary and moving objects in 3D.

During use of the system, it has been observed that it occasionally generates false positives; that is, it detects objects where there are none. Defining and conducting an experiment to measure the false object detection rate in a controlled manner would further enable a better characterization of the system's accuracy. The system's accuracy and precision should also be characterized when: (i) Changing the spacing between cameras; (ii) changing the resolution of the cameras; (iii) changing each camera's lens characteristics[26]; and (iv) changing each camera's image acquisition frame rate.

Only informal experiences have been made using the Snap-detect and Arm-angle systems. The Snap-detect system's accuracy, precision and false positive rate should be measured. Informal experiences using the system has given the impression that false positives are not a big issue with the Snap-detect system; they do occur, but rarely enough to make the system work well in day-to-day use. The system's accuracy is good enough for it to be used to bring up a menu at the user's location in the Wallboard application. A possible extension to the Snap-detect system could be to locate sounds not only in 2D, but also in 3D by using additional microphones mounted in the room housing the display wall.

The Arm-angle system has not been evaluated. Experiments to evaluate its accuracy and precision should be designed and conducted. Informal use of the system indicates it has a tendency to pick up false positives from content being shown on the display wall. Straight lines on the display wall may be sufficiently prevalent so as to override the lines being detected from the user's arm. This results in the system incorrectly identifying where the user is attempting to point.

Some approaches to resolve this include: (i) Fitting an infrared-pass filter to the camera (thus blocking the visible light from the display wall), and illuminate the scene with IR light (this illumination would also benefit the Camera-sense system); (ii) polarize the light from each projector, and mount a polarization filter on the camera. This would have the effect of preventing the camera from seeing the light generated by the projectors driving the display wall; (iii) utilize knowledge of what is actually being displayed on the wall to separate the foreground (i.e., the person) from the background (content being shown on the display wall); and (iv) simply hide the content in front of the user when the Arm-angle system is called. This

[26] For instance, comparing the use of a fish-eye lens with the current 42° field-of-view lens.

approach is taken in the Wallboard application [95] in order for the display wall's intense light to not shine through the objects being scanned, and works quite well. Finally, the Arm-angle system could itself be implemented using the Camera-sense system.

4.10 Conclusion

This chapter has presented the concept of several Interaction Spaces and detailed the architecture, design and implementation of three interaction space systems. The three systems demonstrate how interaction can be separated from the computers they act on. Interaction is no longer a capability associated with a given computer, but instead a capability inherent to the environment, and in the case of the three systems, associated with a display wall constructed using a cluster of computers. The systems have different characteristics. They can detect one or several users interacting simultaneously, and vary in the size of the areas they cover. The Arm-angle interaction space's construction is enabled by the Snap-detect interaction space. It demonstrates a movable interaction space.

The Camera-sense system's latency, accuracy and precision has been evaluated. The system's end-to-end latency is 113.66 ms. The system locates objects with an accuracy of 1.24 cm, and a precision of 0.72 cm. The results demonstrate that a system built using commodity components provides accuracy that is on par with [110] and lower than [12] the state-of-the-art, while improving on the state-of-the-art by tracking objects in 3D without the need for users to wear markers. The system's latency is sufficiently low to enable interaction with applications. The system is scalable. Its embarrassingly parallel architecture makes it possible to add and remove cameras to cover wider or narrower display walls.

Chapter 5

Applications

This chapter describes a number of applications that have been developed or modified to take advantage of the three different interaction space systems. The applications are listed in Table 5.1, along with whether the applications are multi-user or not, newly developed or modified, and their use of the different interaction spaces. This chapter is based in part on the research presented in the following peer-reviewed, published papers: [21, 95, 24, 23].

Application	Multi-user	New	Camera-sense	Snap-detect	Arm-angle
Wallview	No	Yes	Yes	Yes	No
Wallboard	Yes	Yes	Yes	Yes	Yes[a]
Wallfire	Yes	Yes	Yes	No	No
Angle-snap	No	Yes	Yes	Yes	Yes
MASpace	No	Yes[b]	Yes	No	No
Quake 3 Arena	Yes	No	Yes	Yes	No
Homeworld	No	No	Yes	No	No

Table 5.1: New and existing applications and their use of the different interaction spaces. If an application is listed as New, it has been developed from scratch towards fulfillment of the Ph.D. project presented in this dissertation.

[a] The Arm-angle space is only used when the virtual billboard-functionality of the application is disabled, since the camera used by the Arm-angle system is otherwise in use to support the virtual billboard.

[b] The application has been developed from scratch, but is built to replicate the output of the already existing HIDRA [143] application on display walls.

5.1 Wallview

Wallview is an image viewer developed specifically for display walls, shown in Figure 5.1. The purpose of the application is to show many high-resolution images

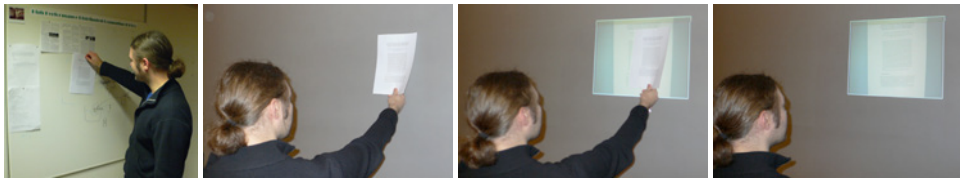
Figure 5.1: The Wallview application being used to navigate a large collection of comics.

at once and enable users to pan, zoom and rotate the images using multi-touch gestures. It has been used to navigate three years (950 images) of the Norwegian comic M [62] and a whole range of other images. The application only supports a single user navigating the images being displayed. It has been developed from scratch in order to provide high-performance image viewing on the Tromsø display wall. It uses the Camera-sense interaction space to pan, zoom and rotate the view on the display wall. The Snap-detect interaction space is used to detect one, two and three snaps in a row. Two snaps zooms the view in at the location of the snap. One snap zooms the view back out, and three snaps resets the view to its origin.

Wallview is a parallel application. A viewer runs on each display wall cluster node. OpenGL [58] is used to enable graphics card accelerated drawing of images, and SDL [144] is used to load images from disk. A separate application controls the pan, rotation and zoom by interpreting events from the Camera-sense and Snap-detect systems. Using one hand, users can move the entire view around. With two hands, the image can be zoomed in or out by controlling the distance between the hands, much like the pinch-gesture used to zoom on the Apple iPhone. Two hands are also used to rotate the image, by moving them in a circular fashion. The Wallview application can also make use of the 3D capabilities of the Camera-sense system. When enabled, the penetration of an object into the Camera-sense interaction space is used to control the viewer's zoom-factor.

5.2 Wallboard

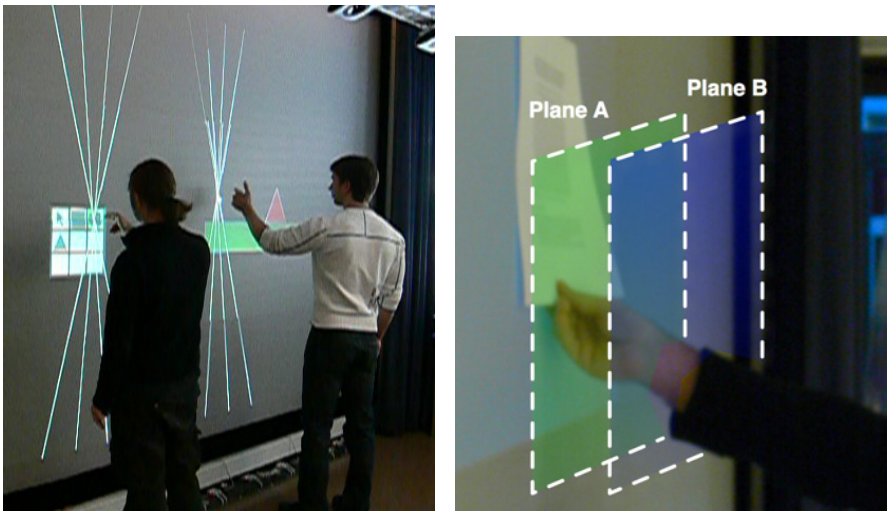
Wallboard is a virtual billboard with some additional drawing functionality. It enables users to "scan" or image content from the real world, and have it appear on



(a) (b) (c) (d)

Figure 5.2: Using the Wallboard application to scan an object, bringing a representation of it from the real world into the display wall's digital domain. (a) Pinning a document to a regular billboard. (b) The document is held in front of the wall for less than two-three seconds. (c) The user removes the document, (d) leaving an image of the document on the wall at the location where the user held it.

the display wall. It also functions as a very simple whiteboard, enabling users to draw circles, squares, triangles and paths. While several users may interact with the application at the same time to create or move objects, the imaging component can only be used by one user at a time. The application was developed from scratch, to demonstrate the abilities of the different interaction spaces, demonstrate the virtual billboard concept and to provide a test application for a system called "The Wall Windowing System" (W^2S) being developed by other researchers in the Tromsø display wall group [145].



(a) (b)

Figure 5.3: (a) Two users using the Wallboard application simultaneously to manipulate geometric objects. The user to the left has brought up the menu by snapping his fingers, and is selecting which object to create. The white lines visualize the output from the Camera-sense system, and can be turned on or off. (b) Wallboard begins imaging an object if it detects a wide object penetrating through the entire Camera-sense interaction space.

Wallboard makes use of all three interaction spaces. The Camera-sense space is used to enable multi-touch style interaction. It is one of a few applications that

makes use of the 3D capabilities of the Camera-sense system. The 3D capabilities are used when imaging a document or other object. Wallboard initiates the imaging component if it detects that a thick object has penetrated through the entire Camera-sense interaction space, as shown in Figure 5.3(b). If an object detected in planes A and B is sufficiently wide (as determined by the extent reported by the Camera-sense system), sufficiently close and remains stationary for more than one second, the system steers the movable camera towards the location where the document is held, and captures an image of it. That image is then placed on the display wall. More details can be found in [95].

The Snap-detect system is used to enable users to call a menu by snapping their fingers. From this menu, they can initiate drawing operations or create circles, squares or triangles. The Arm-angle space was previously used to move objects around, in the same way as the Angle-snap application described in Section 5.4. However, the camera used by the Arm-angle system is now used to provide content for the virtual billboard functionality of the application. Their use is thus mutually exclusive.

5.3 Wallfire

Wallfire, which appears in the Introduction in Figure 1.1, is a demonstration application for the Camera-sense interaction space. It can be used by several users simultaneously to "paint fire" on the display wall. Objects are detected and located using the Camera-sense system, and wherever an object appears, fire is seeded on the corresponding location on the display wall. If no objects are detected, the display wall slowly turns all black. The Wallfire application has also been implemented to run on an iPod touch, as shown in Figure 5.4.

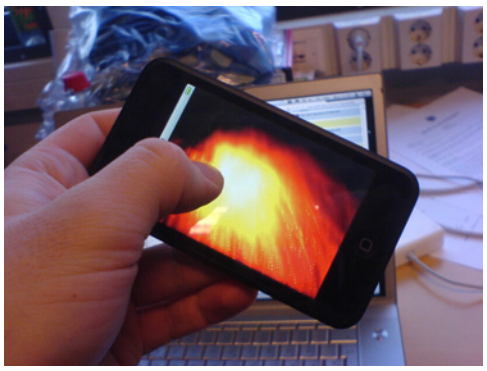


Figure 5.4: The Wallfire application running on an iPod touch.

5.4 Angle-snap

The Angle-snap application was written for the VNC-based desktop environment running on the Tromsø display wall. It enables users to double-snap their fingers in order to move the currently focused window to their location. Windows can also

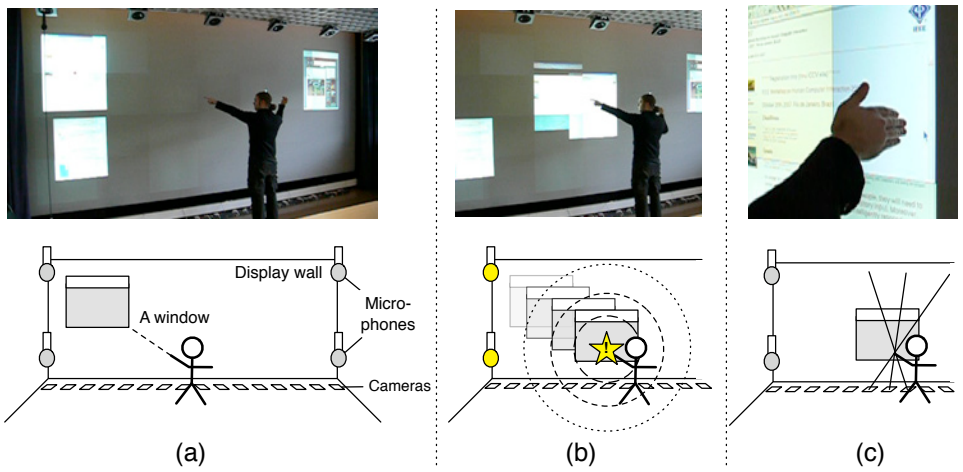


Figure 5.5: The Angle-snap application is used to select, move and interact with a window on the display wall, by combining the capabilities of the three different interaction space systems. (a) The Arm-angle system detects the direction in which the user's arm is pointing, and makes it possible to select a window on the display wall. (b) The Snap-detect system detects the location of the sound made by a user snapping his fingers. This prompts the window to move to the user's approximate location. (c) The Camera-sense system is used to control a mouse pointer on the display wall's desktop environment, enabling the window to be manipulated.

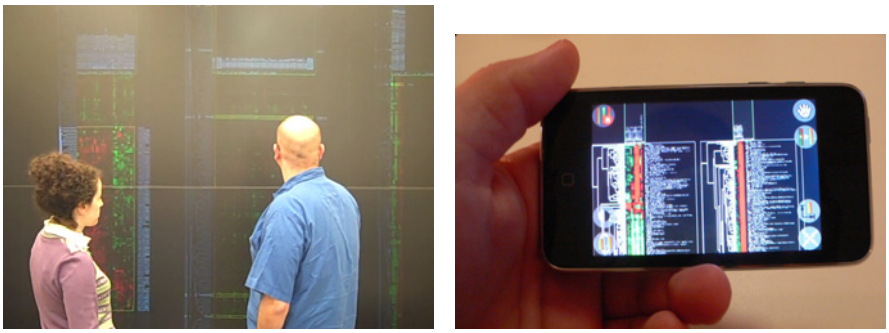
be moved by snapping once, then pointing at the window, and then snap again to move the selected window closer to the user. Once the window is up close, users can interact with the window using their hands. The application only supports a single user at a time, as the desktop environment currently only supports a single user[1]. It makes use of all three interaction spaces. The Snap-detect system enables the window to be moved closer towards the user's location. The Arm-angle system makes it possible for users to point towards the window they wish to move, and the Camera-sense system enables interaction with the window up close. More details about the Angle-snap application appear in [24].

5.5 MASpace

MASpace, short for Microarray Space, is an application developed to visualize multiple genomic microarray datasets on display walls (Figure 5.6(a)). MASpace was developed from scratch to replicate the visualization produced by the HIDRA application [143] on display walls. It is currently being used on the Princeton display wall by computational biologists at Princeton University. On the display wall, only a single user can control the visualization at a time. However, the application

[1]Different approaches could be considered to add multi-cursor support, including the ones detailed in [53, 146, 91, 54].

also has an iPod touch version which displays the same content as on the display wall. The iPod can control the visualization on the display wall, or it can be decoupled from the visualization on the wall enabling navigation of the datasets without affecting the display wall visualization. The iPod creates a separate, mobile interaction space, which also ties into the Pixel Space by displaying the same data as on the display wall.



(a) (b)

Figure 5.6: (a) Using MASpace to compare microarray datasets. (b) The MASpace application running on an iPod touch.

The MASpace application uses the Camera-sense interaction space to enable interaction with the visualization. To control the visualization, users must be able to select genes, and pan and zoom the visualization. To differentiate between selecting genes and panning the visualization, MASpace makes use of the 3D capabilities of the interaction space. If the user touches the wall, genes are selected. Otherwise, the view is panned, or zoomed using a two-handed gesture, like Wallview. The user can "tap" the wall with a closed fist to bring up a contextual menu that enables additional control, such as dropping the current selection, resetting the visualization's view coordinates (if users get lost in the visualization) and rearranging datasets. The use of a closed fist is an example of distinguishing gestures based on an object's extent. Double-tapping with a single finger causes the application to zoom the view in or out, or if the double-tap was on a gene, information about that gene from an online gene database is pulled from the web and shown on the wall.

The visualization can also be controlled using an iPhone or iPod touch, with software that provides a miniature view of the display wall visualization, shown in Figure 5.6(b). The same gestures are used on the iPhone as on the wall, with the exception that the iPhone does not support 3D sensing; for this reason, users must toggle between selecting genes and moving the view. The two different platforms have different strengths. The iPod lets users see and control the display wall from anywhere, but with lower performance and resolution. Using the Camera-sense space, it is possible for several users to gather around the display wall and collaborate on controlling the visualization. The iPod application also gives users the ability to explore the datasets independently of what is being shown on the display

wall, acting as an independent display of the visualization.

5.6 Quake 3 Arena and Homeworld

Games are a class of application that require low latency input. This makes computer games a good application domain to use new input mechanisms with, since games become "unplayable" if the mechanism does not provide sufficient responsiveness. Quake 3 Arena [69] (Q3A) and Homeworld [70] are two commercial games developed by id Software and Relic Entertainment, respectively. Quake 3 Arena is a first-person shooter, and Homeworld is a real-time strategy game. The two games were open sourced in 2005 and 2003. With access to the source code of the two games, it was possible to modify them to: (i) Take advantage of the Camera-sense interaction space; and (ii) run with good performance on the Tromsø display wall.

Figure 5.7: Quake 3 Arena and Homeworld being played on the Tromsø display wall using the Camera-sense interaction space. The players to the left and right play Q3A, and the middle player is playing Homeworld.

Q3A supports several players playing against each other at the same time using the Camera-sense interaction space. The Snap-detect space is used to enable bystanders to affect the game. When they snap their fingers, all of the players' weapons fire at once. Homeworld makes use of the Camera-sense space to emulate the in-game cursor. By distinguishing between a narrow and a wide object, users can perform clicks and drags in the game. Further details about the modifications to Q3A and Homeworld appear in [23].

Chapter 6

Pixel Space

This chapter documents the Pixel Space concept and the design, implementation and evaluation of two Pixel Space systems. The chapter is based on the following peer-reviewed, published papers which were all written towards the fulfillment of the Ph.D. project presented in this dissertation: [25, 27].

Pixels are everywhere. Users carry pixels around on mobile phones, gaming appliances, music/video players, laptops and other portable devices. In June 2005, laptops outsold desktops for the first time [147]. TVs and flat-panel displays are ubiquitous. The Sixth Sense developed at MIT [11] is an example of how pixels may soon be literally anywhere a user decides to go. The user wears a hat embedded with both a tiny projector and a camera. The projector makes it possible to display pixels on any surface that the user is looking at, while the camera enables the user to interact with the pixels by tracking colored markers on the user's fingers and interpreting them as gestures. The growing collection of displays from portable devices, tablets, laptops, workstations and display walls all contribute to creating a large, near infinite pixel space.

While the pixel resources represented by this collection of displays continues to grow, computers are usually only able to utilize a tiny fraction of the resources by directly connecting the computer's graphics card to one or two external displays. This limits the number of displays one can utilize, and makes transparently utilizing the pixels afforded by a display wall from a single computer difficult. The need for tethering a computer to displays is further limiting in the type of displays one can utilize. Small displays like those found on portable devices like the Apple iPhone, iPod touch, and the Nokia N770/N800/N810 Internet Tablets[1], can not be connected to an external computer, and can thus not be utilized by other computers at all.

[1]The iPhone and iPod touch have displays with a resolution of 480x320, and the Nokia Internet Tablet has a resolution of 800x480.

Pixels are an attractive way of sharing visual data for many reasons. First, they are fully cross-platform, with almost any display[2] today being able to show pixels. Further, the processing required to show pixels is very low. Displaying them amounts to copying them to the appropriate location in memory, with little or no pre-processing. In contrast, higher level drawing operations would require more processing power on the display side with the result still appearing as pixels on the display. Sharing pixels does not require sharing the underlying data. When utilizing the pixel resources of the pixel space, this is an advantage, since one might want to utilize "public" pixel resources without potentially leaving one's personal data behind[3]. The drawback to sharing pixels is that any computational resources provided by the displays in the pixel space are not utilized at all. Further, continuously streaming pixels over a network is bandwidth intensive and may become a bottleneck as the size of the area being updated grows.

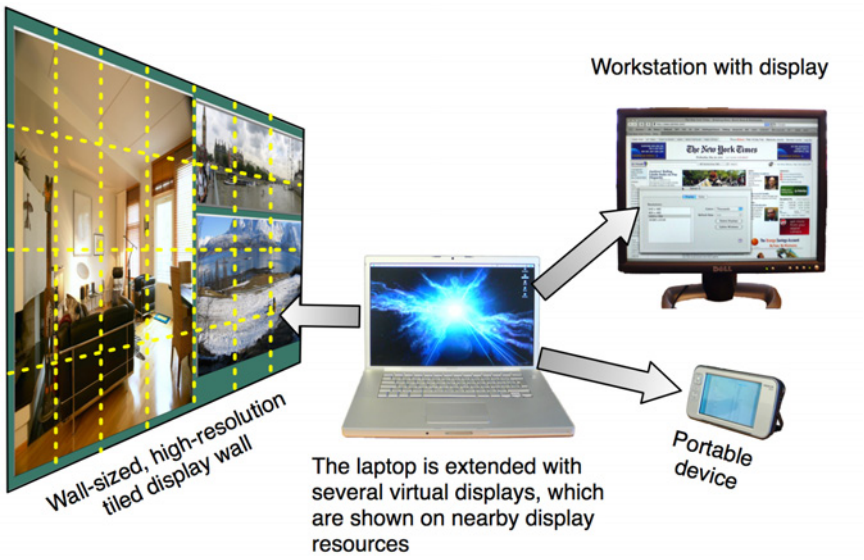


Figure 6.1: The 22 megapixel laptop. The laptop's display area is extended to utilize the pixels provided by a display wall, a workstation and a portable device.

This chapter presents two models with two associated systems implementing the models. The first model is the Network Accessible Display (NAD) model. The associated implementation is called the 22 megapixel laptop, and was built to enable utilization of the pixel resources in the pixel space. In the NAD model, all displays are equipped with some networking and processing capabilities. Using these capabilities, each display shares itself on the network, enabling other computers to use a given NAD to display pixels on its behalf. While most displays still lack

[2]The exception is vector-based displays, which have largely fallen out of use.

[3]This can be circumvented by for instance taking a screen shot of the pixels, or use other approaches to logging what is displayed. However, this does still not expose the actual data behind the pixels, such as the data for a graph.

both a network connection and the ability to run custom code, both a workstation with a display and a portable device such as the Nokia Internet Tablet have the hardware required by a NAD. They are both equipped with displays, have a network connection and can run custom software, which when combined allows for the functionality of a NAD to be emulated. The 22 megapixel laptop is an implementation that demonstrates the model, where the Tromsø display wall, a regular workstation with a display and an Internet Tablet are turned into NADs that can be transparently utilized from a laptop, as illustrated in Figure 6.1.

The implementation of the 22 megapixel laptop does not consider ways of improving the performance of sharing pixels. The second model and associated system presented in this chapter is De-centralized VNC (DVNC). As described in Section 2.1.1, VNC can be used to provide a desktop environment for a display wall. VNC is based on sharing pixels, but when used with a tiled display wall its performance is reduced. The reduction is caused by the VNC server sending redundant pixel data to the VNC viewers running on each tile of the display wall. Redundant pixels must be sent in cases where pixels are moved, but not otherwise updated. Some examples where this happens include moving a window, scrolling the contents of a window or panning an image. DVNC removes the need for redundant pixel updates from the VNC server by enabling the VNC viewers to exchange pixels amongst each other.

The main difference between the 22 megapixel laptop and DVNC is that the 22 megapixel laptop provides a transparent way of utilizing the display wall's pixel space from a user's personal computer, while DVNC provides a desktop environment that is permanently shared amongst different users. Further, the 22 megapixel laptop is dynamic: It can adapt to the available pixel resources without requiring that users restart their applications or their desktop environment. Since the number and resolution of nearby pixel resources isn't known a priori, the laptop discovers the available resources. The user can then decide whether or not to use them, and in which configuration. In contrast, DVNC's resolution is defined when the server is started, and can not be changed without restarting the server (and thus loosing all desktop state).

6.1 Limitations

The 22 megapixel laptop is a research prototype. The user interface and network protocols have been developed towards creating a system that demonstrates the NAD model, and achieves the goal of giving users transparent access to the potentially large number of nearby pixel resources. The system does not provide the highest level of polish and performance. The design and implementation of the system does not consider aspects such as authentication and access control, or

compression[4] and encryption of network data. Further, the 22 megapixel laptop does not allow the available NADs to be shared amongst several users at the same time. Only a single user may use a given NAD at a time.

The De-centralized VNC system builds on VNC, and thus already has support for various kinds of encryption and authentication. However, no effort has been made towards improving VNC's performance by developing better compression algorithms such as the work done in [148], or the caching strategy employed in [149]. DVNC further does not employ any authentication between the different VNC viewers. In the Tromsø display wall setting, this is not a major issue since the viewers are on an internal, trusted network. Finally, the DVNC viewers do not compress the pixel data sent to other viewers at all.

6.2 Related work

In a paper contemporary with the NAD model [150], the authors present ideas that are very similar to those embodied in the NAD model: Namely, displays with networking and processing built-in. The resulting implementations are very different, however. Where their work focuses on sharing windows from a computer to a shared display (similar to the approach taken in [151, 146]), the NAD implementation is built with a focus on transparency: Users should be able to treat the shared pixel resources as a natural extension of their desktop, rather than as a surface that must be managed separately. However, the approach in [150] is able to share the pixel resources amongst several users at the same time, whereas the current NAD implementation is limited to one user at a time using a network accessible display.

The first "network projectors" were announced in the summer of 2007 [152], enabling the projector to be driven using the Microsoft Remote Desktop protocol [87]. Such projectors are the products coming closest to adhering to the NAD model available on the market. Digital photo frames are increasingly also shipping with network capabilities. The iGala Digital Photo Frame [153] is a digital photo frame that incorporates a regular Linux-based OS, a touch-screen and a wireless network connection, making it a nearly ideal NAD. However, such photo frames do not currently allow remote computers to control their pixels[5].

Besides the 22 megapixel laptop, there are other software solutions that provide the ability to create virtual displays and share them with other computers. MaxiVista [20], ZoneScreen and Screen Recycler[6] provide this ability on Windows and Mac OS X. They are limited in their ability to scale to many displays, lacking awareness of the different displays' arrangement – which is necessary to utilize the collection

[4] With the exception of rudimentary compression using Run-Length Encoding.

[5] Except by constantly updating the digital photos stored on the device.

[6] http://www.zoneos.com/zonescreen.htm and http://www.screenrecycler.com (links last visited April 25. 2009).

of displays in a tiled display wall – and the maximum resolution they can support. The 22 megapixel laptop can create an arbitrary number of displays[7], and the displays may have resolutions up to about 100 megapixels, subject to the amount of available RAM on the computer. In [91, 83], the authors present a system which integrates several displays from different computers into a single geometrically aware desktop. The 22 megapixel laptop does not take into account the actual geometry of displays, only their arrangement so as to collect "related" displays (such as those used on a display wall) together.

Sharing and accessing remote displays can be done using a range of different remote desktop systems, from Virtual Network Computing (VNC) [17], to X11 [86], Microsoft Remote Desktop [87], Sun-Ray [154] and THINC [155]. These systems generally differ in: (i) the way they send display updates, either as pixels (compressed or otherwise), or as drawing operations[8] which are interpreted by the remote end and used to draw content; (ii) the way they compress content; and (iii) the degree to which they are cross-platform. None of the systems have been designed for use with display walls, where one to two orders of magnitude more pixels than usual must be provided, and the display wall's parallel architecture must be accommodated.

VNC is a system that enables users to share their entire desktop with other users. The system is based on sharing pixels stored locally in the (possibly virtual) display's framebuffer. There are several implementations of VNC, including RealVNC [26], TightVNC, UltraVNC and many others, and much work has been done towards improving VNC's performance, such as the compression and caching techniques mentioned earlier [148, 149] or adding more advanced features to VNC such as 3D support [156]. Microsoft Remote Desktop uses a drawing operation-based approach to share display content. THINC [155] attempts to improve on remote desktop solutions for thin-client systems using a combination of more efficient pixel coding using the YUV color space, as well as transfer of raw graphics operations intercepted before the native window system has turned them into pixels. The systems further differ in their ability to provide pixels to an entire display wall. VNC can drive the entire Tromsø display wall, while Microsoft Remote Desktop limits the maximum resolution to 4096x2048 pixels, which is about one-third of the resolution offered by the Tromsø wall. THINC has not been applied to resolutions of larger size than a regular workstation display.

The Scalable Adaptive Graphics Environment (SAGE) [40] is a system for streaming pixels from a rendering cluster to one or several display walls. The system assumes very high-bandwidth links from the rendering cluster to the display endpoints. The system allows different windows to overlap on a display wall, where each window's contents is derived from different pixel streams coming from the

[7] Subject to the Mac OS X Window Server's limitation of 32 displays, as discussed in Section 6.3.5.

[8] Example drawing operations include "draw line from A to B," "draw string at X,Y" and so on.

rendering cluster. Some example pixel streams are a VNC-desktop, a movie, or an animated scientific visualization. Contrary to DVNC, no pixel data is exchanged between the nodes driving the display wall. Instead, moving a window causes the pixel streams delivered by SAGE to be reconfigured, so that they "track" the window as it moves across the display wall. SAGE constantly streams pixels, which results in high load even when there are few or no changes to the pixels.

6.3 Network Accessible Displays

The idea behind the Network Accessible Display (NAD) model is to discover nearby displays sharing their pixel resources, and then utilize them to increase the amount of pixel real estate available to a computer. The nearby pixel resources are merged with the computer's own pixel resources, such as those provided by a laptop's built-in display, to create a higher resolution display area that can be transparently used by applications running on the computer.

The NAD model assumes that all displays at some point will be equipped with CPU and networking resources. Using these resources, the display can run code to advertise its existence on a local network, and make its pixels available for use by other computers that are not directly connected to the display. Other computers can then discover the NAD, and utilize it by connecting and streaming pixels to it.

Figure 6.2: The 22 megapixel laptop in use driving both the Tromsø display wall and the Nokia N800 Internet Tablet.

The 22 megapixel laptop is a system based on the NAD model. This section is

based on research presented in [25], and presents the architecture, design, implementation and evaluation of the 22 megapixel laptop. The 22 megapixel laptop is shown in Figure 6.2 driving a display wall and a Nokia N800 Internet Tablet.

6.3.1 Architecture

Figure 6.3 shows the architecture of the 22 megapixel laptop. The system consists of two components: A laptop component and a NAD component. The laptop component is responsible for discovering nearby displays, presenting them to the user, and extending the laptop's display area to include nearby NADs when the user requests it. The NAD component makes the displays attached to the computer it is running on available to other computers.

The laptop component discovers one or more NAD components using a network discovery mechanism based on UDP multicast[9]. The laptop's user then manages the NADs by selecting which NADs to use and how to arrange the NADs. When the user has configured the NADs, a set of virtual displays are created with resolutions and bit depths matching the capabilities of the chosen NADs. The pixels from the virtual displays are then sent to their respective NADs until the user decides to stop using the NADs.

BlueTooth is used to determine if a NAD is in physical proximity to the computer. No networking is done over BlueTooth, however. A NAD must first be discovered on the local network before its proximity can be determined using BlueTooth. BlueTooth proximity detection works by scanning for discoverable BlueTooth devices, and then determine if any of the resulting MAC addresses match the BlueTooth MAC address as part of the NAD properties received over the local network.

The NAD component runs on computers or devices with attached displays. It advertises the display's presence and availability on the local wired or wireless network. When a NAD receives a connection from a computer, it sends its properties to the computer before it begins receiving pixels from it and displaying them on the locally attached display.

6.3.2 Design

Figure 6.4 shows the design of the 22 megapixel laptop's two components. The laptop component consists of three sub-components: (i) A user interface; (ii) a kernel extension; and (iii) a display sharing daemon (DSD).

[9]The network discovery mechanism is further detailed in Appendix C.

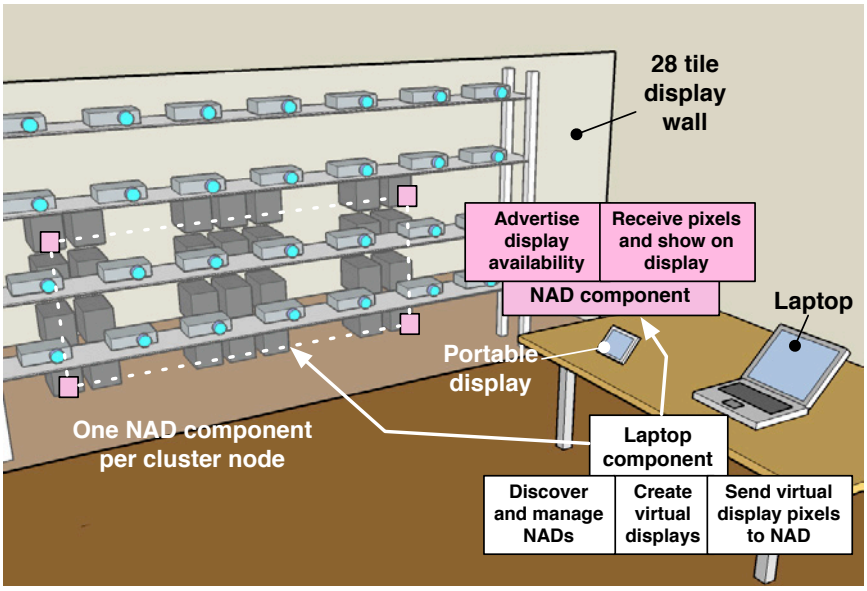


Figure 6.3: The architecture of the 22 megapixel laptop. The computers driving the display wall and the portable display both run the NAD component. The laptop component discovers the NAD components on the local wired and wireless networks.

User interface

The user interface communicates with the kernel extension to configure the number and resolution of the virtual displays. When a user enters an area where NADs are available, the NADs appear in a window in their preferred arrangement, as shown in figure 6.5. To detect nearby NADs, the user interface uses a combination of network discovery and BlueTooth proximity detection. If a NAD is discovered on the network and a BlueTooth MAC address is part of its properties, the user interface scans for that MAC address. If the given MAC address isn't discovered, the display is hidden from the list of NADs under the assumption that the user isn't in physical proximity to the display.

Kernel extension

The kernel extension creates a set of virtual displays in response to commands from the user interface. The virtual displays appear to the computer's window system as regular displays, but without support for hardware accelerated graphics. The virtual displays are detected using the same mechanism as those used when the window system detects that a display is connected to the computer's VGA or DVI port. The window system manages the virtual displays in the same way that it would a real display. This makes using the virtual displays fully transparent to applications. Applications can draw to the virtual displays without requiring that

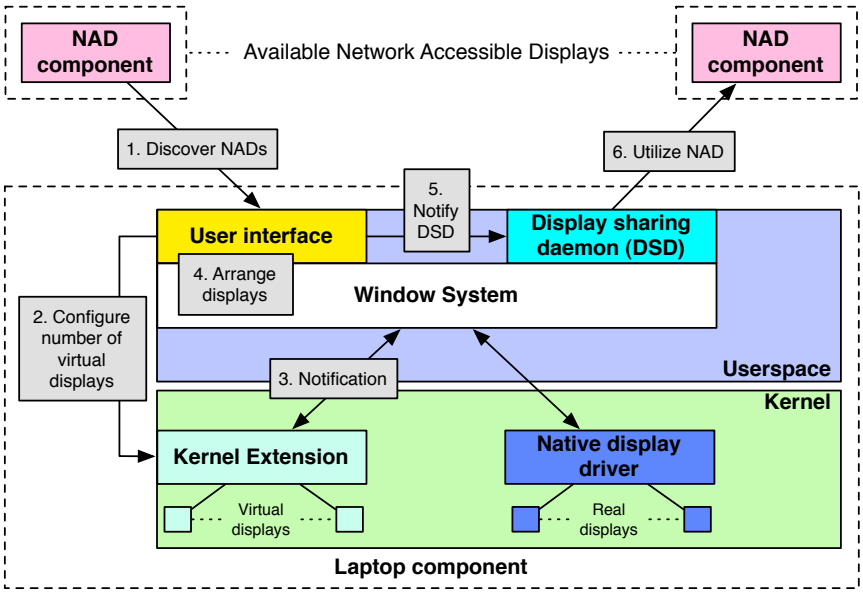


Figure 6.4: The design of the 22 megapixel laptop. (1) The available NADs are discovered by the user interface running on the laptop. (2) In response to the user, the interface instructs the kernel extension to create a number of virtual displays, matching the capabilities of the discovered NADs. (3) The creation of the virtual displays causes the window system to be notified that new displays are available, using the same notification mechanism that is used by the native display driver whenever a real display is plugged in to the laptop's VGA or DVI port. (4) The user interface arranges the geometry of the displays to match the configuration indicated by the NADs. (5) The user interface notifies the display sharing daemon (DSD) that it should connect to a given set of NADs, and send pixels from the corresponding set of virtual displays. (6) The DSD sends pixels to the NADs.

the applications are rewritten, relaunched or otherwise modified in any way.

Display sharing daemon

The display sharing daemon accepts messages from the user interface. The messages can instruct the daemon to either push the contents of a given virtual display to a NAD, or disconnect from a NAD. After connecting to a NAD, the DSD detects changes to the virtual displays and sends them to each virtual display's NAD counterpart as compressed (Run-Length Encoded) pixels. A single display sharing daemon drives all the NADs being used by the computer.

NAD component

The NAD component runs on any computer whose display should be turned into a Network Accessible Display. It is configured with the properties of the locally attached display(s) and accepts incoming TCP connections on a randomly assigned

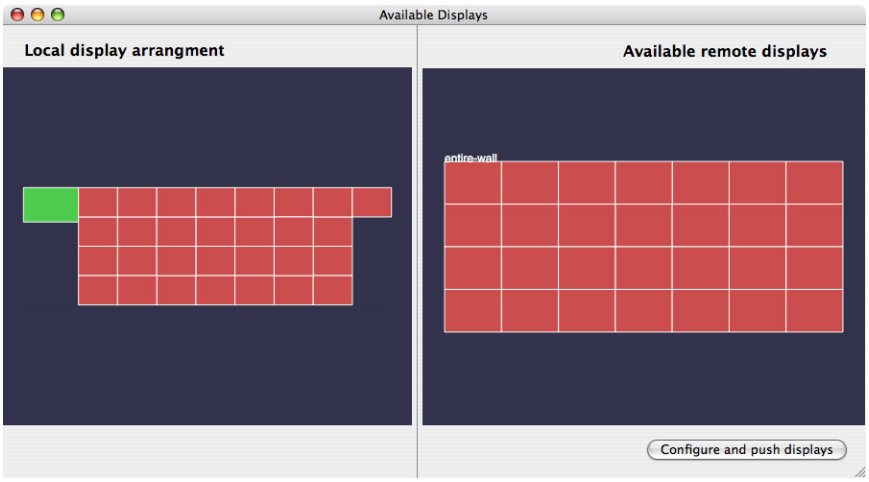


Figure 6.5: The user interface for discovering and utilizing nearby NADs. The left pane of the window shows the current arrangement of the virtual displays, while the right hand indicates available NADs. The user drags a box around the displays that he wishes to utilize, and activate them by pressing the Configure and push displays button.

port. The NAD advertises its existence using the network discovery mechanism described in Appendix C. The advertisement includes the TCP port it is listening on, which is used by clients (such as the laptop component) when connecting to the NAD.

Upon receiving a connection from a client, the NAD's properties are sent to the client. The properties include the NAD's name, resolution, bit depth, BlueTooth MAC address (if available), and display wall geometry (if available). The name is a human readable name for the NAD, which must be assigned manually. The resolution and bit depth are both determined by the NAD component when it starts on a given computer. The BlueTooth MAC address must be assigned manually. If included, the client can scan for that MAC address using its own BlueTooth hardware to determine whether the given NAD is in physical proximity of the client. Finally, display wall geometry must be assigned manually, and is only included if the NAD is considered part of a display wall. The display wall geometry includes the width and height (in number of displays) of the display wall, as well as the NAD's index within this grid of displays. It is used by clients when arranging the NADs, so that the arrangement of virtual displays by the clients match the physical arrangement of the display wall NADs. The screenshot in Figure 6.5 shows how this is used to present the user with a collection of remote NADs that mimic the arrangement of the display wall.

Once the display properties have been sent to the client, the NAD creates a drawing context covering its entire display. The drawing context is used to display pixels received from the client. Only a single client may send pixels to the NAD at a time, since each client expects to control the entire display; the design does not support

interleaving pixels from different clients. When the client disconnects, the drawing context is destroyed, returning the display to its previous state.

6.3.3 Implementation

The laptop component has been implemented for Mac OS X 10.4, and also been confirmed to work on Mac OS X 10.5. The NAD component is cross-platform, and has been implemented on both Linux and Mac OS X. It can run on x86, PowerPC and ARM architectures.

User interface

The application creating the user interface is implemented in Objective-C using the Cocoa APIs [157]. It communicates with the kernel extension using a sysctl-interface which the kernel extension exports, and with the display sharing daemon and NADs using regular BSD sockets. Configuring and arranging the virtual displays is handled using the `CGDisplayConfiguration` APIs exported by Mac OS X' window server[10]. Communication with the DSD and NADs use TCP for transport, and a custom protocol to exchange information about display capabilities. The protocol is used to exchange information between the NADs, the DSD and the user interface.

The protocol uses a message-based abstraction built on top of TCP. Each message is encoded using a 4-byte length field, followed by the message type and the message content. The message type determines the format of the message content. Using this protocol, the user interface sends queries to the NADs, which return a response containing a dictionary describing the given NAD's capabilities.

Kernel extension

The kernel extension is implemented in C++, and follows the standard pattern for kernel extension development on Mac OS X [158], as illustrated in Figure 6.6. On Mac OS X, kernel extensions are loaded on demand during "matching." The contents of an XML property list stored with the kernel extension is matched with the currently connected hardware. The kernel extension with the highest "probing score" is loaded by the kernel. Kernel extensions that don't support specific kinds of hardware can still be loaded automatically by the kernel by matching on the `IOResources` class, which is always present.

[10]The window server internally uses the interfaces exported by the kernel extension to change the resolution of the virtual displays. Display arrangement is handled internally by the window server, and does not involve communication with the kernel extension, although the arrangement is controlled by the user interface.

The kernel extension consists of three classes: A virtual framebuffer (VFB) master, a VFB nub and the class implementing the actual VFB. When the computer boots, the VFB master is loaded and instantiated by the kernel by having it match on the `IOResources` class. The master handles communication with the user interface running in user space by exposing a sysctl-based interface. While designed for the user interface, any application can use the interface by making the appropriate `sysctl()` system call, or from the command line using the sysctl utility.

The master then instantiates a number of VFB nubs. A nub is part of the matching system used by the IOKit [158] in the Mac OS X kernel. When a nub is instantiated, the kernel interprets this as if new hardware was attached and begins looking for a driver for the newly created nub. The VFB class is designed to match on the VFB nub, and is thus loaded and instantiated by the kernel when the nub is created. One VFB instance is created for each VFB nub that the master creates.

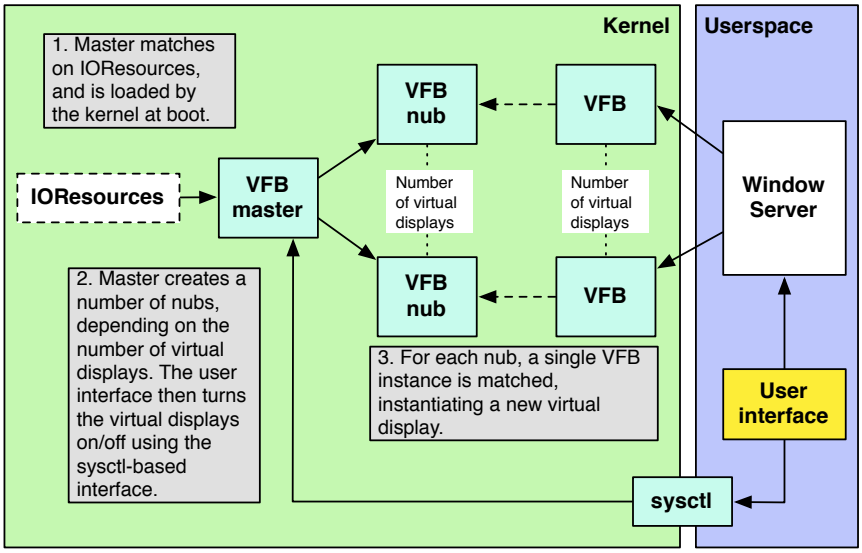


Figure 6.6: The kernel extension. (1) The VFB master class matches on the IOResources "hardware" during system boot. This prompts the kernel to load the VFB master class. (2) The master, once loaded, creates a set of "nubs," which is the Mac OS X term for a device endpoint. (3) The VFB class, which acts as a driver for the device endpoint created by the VFB nub class, is instantiated by the kernel when the VFB nubs are created as part of the standard driver loading mechanism in Mac OS X. The window server discovers the display through notifications from the kernel, and configures them to default settings. At this point, the user interface can manage the virtual displays using the sysctl interface exported by the VFB master class, as well as through the window server itself.

The VFB class is a subclass of the operating system defined IOFramebuffer class. Creating a VFB instance instantiates the actual virtual display and its associated virtual framebuffer. This in turn triggers a notification to the window system. The VFB instance allocates the memory to store the contents of the framebuffer, and

also exposes a pre-determined[11] set of display resolutions. While the resolutions are pre-determined, they may vary from VFB instance to VFB instance. The VFB instance also maintains EDID[12] data [159] which is used to give each of the different virtual displays human-readable names. The human-readable name appears in several locations in the general Mac OS X user interface, including the Displays panel of System Preferences.

Display sharing daemon

The display sharing daemon is implemented in Objective-C and C using the `CoreFoundation` APIs on Mac OS X. It runs in the background as a daemon process and receives messages from the user interface to push different virtual displays to available NADs. The `CGRemoteOperation` API is used to detect changes to the virtual displays' pixel content, and send the resulting pixels as compressed data to connected NADs.

An updated area of pixels is transferred by creating a message containing a rectangle that defines the location and size of the pixel area, followed by the pixels themselves. The DSD uses run-length encoding to compress the pixels. Run-length encoding is a very simple compression scheme where sequences of identical pixels are replaced with the number of identical pixels followed by a single copy of the pixel value itself. Runs of non-identical pixels are encoded as the number of non-identical pixels, followed by the pixels. Two bytes are used to store the run-length, using positive values to indicate that the pixel value is repeated N times, and negative values to indicate the number of non-identical pixels following the run-length. Other compression types (such as gzip or jpeg) are possible, but not implemented.

The functionality of the DSD is similar to what a regular VNC server could provide: Sharing of a display's pixels with a viewer running somewhere else. However, a custom DSD was written rather than building on VNC in order to: (i) allow displays to be pushed to remote NADs; and (ii) handle the large number of virtual displays. VNC is not designed to support sharing more than a single display at a time (however, that display may have an arbitrarily large resolution). Figure 6.7 illustrates the main difference between how a regular VNC server operates, and how the DSD works. A VNC server waits for incoming connections from VNC viewers, before it begins transferring pixels from its local framebuffer to the remote viewer. In contrast, the DSD is controlled by the user interface, and when prompted to do so *pushes* different virtual displays to the selected NADs. Once the connection is established, however, their high-level behaviour is identical: The

[11] There is no documented way to dynamically change the list of resolutions supported by a given virtual display, although it should be possible.

[12] Extended display identification data. EDID is a standard way for displays to report their capabilities to a graphics card, and contains information such as display timings, vendor, supported resolutions, and the display's serial number.

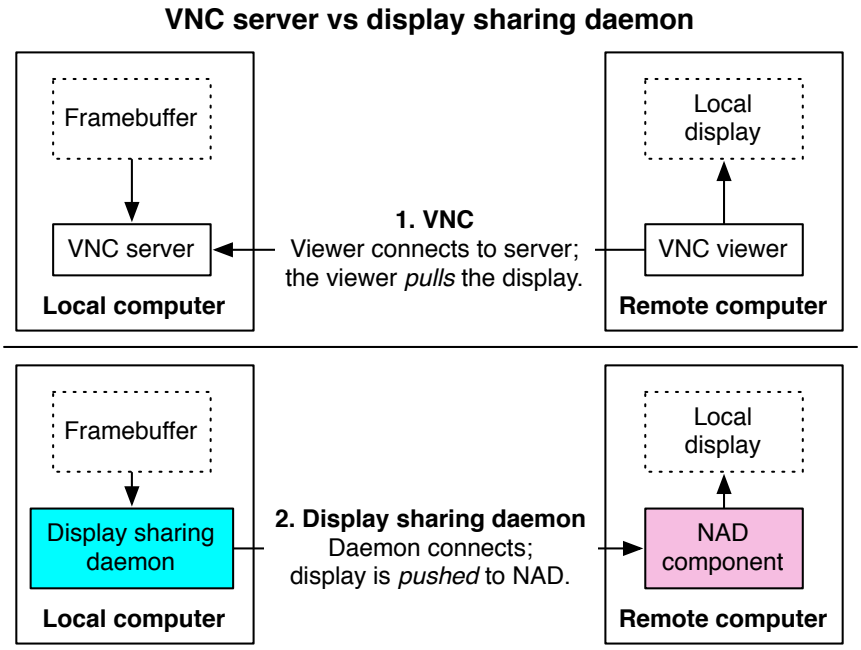


Figure 6.7: (1) The VNC viewer connects to the VNC server, and then displays pixels from the framebuffer shared by the VNC server. The VNC viewer pulls the shared display to it; the action is initiated by the viewer, not the server. (2) In contrast, the display sharing daemon connects to the remote NAD, and starts sharing the framebuffer with it. The DSD initiates the action, pushing a virtual display to the remote NAD.

VNC viewer/NAD requests pixel updates, and the VNC server/DSD responds with new pixels.

NAD component

The NAD component is implemented in C using the SDL library [144] and BSD sockets for communication. SDL is used to create a drawing context in which pixels from remote DSD's are drawn. When it receives a connection from a DSD, the DSD states the pixel format[13], compression type and resolution of the content it wishes to display. If the NAD can accept the requested resolution, it will start accepting pixel updates; otherwise the connection is closed. Each update is rendered to the local display as it arrives from the client.

[13]The pixel format includes the bit depth used to represent each pixel, as well as the ordering of the individual red, green and blue components (for instance, RGB versus BGR).

6.3.4 Evaluation

The 22 megapixel laptop has been evaluated by measuring the system's pixel update rate for different NAD configurations. A MacBook Pro with specifications as given in Section 2.2.4 was configured to utilize increasingly large parts of the Tromsø display wall. A custom application that updated all the virtual displays at a fixed framerate was written to measure the performance of the system.

Methodology

Each tile of the Tromsø display wall was configured to run the NAD component, which advertised a 1024x768 display in 32-bit color. For each experiment, the MacBook Pro was configured to utilize one or more of the NADs, from 1, 2, 4, 8, 16, 24 to 28 (the Tromsø display wall has 28 tiles). Each experiment was repeated five times, which adds additional results to the results initally obtained in [25]. Before starting measurements, a "draw" process running on the MacBook Pro created a window that completely covered all the virtual displays, before updating this window 300 times at an attempted rate of 10 frames per second[14]. When the draw process had finished updating the virtual displays, statistics were gathered from the NADs.

Three different statistics were recorded for each experiment: (i) The total number of pixels updated by the NADs for the duration of the experiment; (ii) the total number of bytes used to send data to the NADs; and (iii) the CPU load on the MacBook Pro for the draw process, display sharing daemon and the Mac OS X window server at both kernel and user level. To sample the CPU load of the different processes, the `mach`-calls `task_for_pid()` and `task_info()` are used. This approach was taken since the window server could not be instrumented, since its source code is unavailable. This way of sampling a process' CPU load is identical to that used by the "top" command line utility on Mac OS X. They return the same information as `getrusage()`, but for a specific process and not just the calling process.

Results

Figure 6.8 shows how the 22 megapixel laptop handles an increasing number of virtual displays. The straight, dotted line indicates the target update rate (a full refresh of all the virtual displays ten times per second), compared to the actual update rate. The laptop is able to track the target rate for one and two virtual displays, before flattening out at four virtual displays. The system also peaks in performance at four displays covering about 3.14 megapixels, delivering 27.38 megapixels/second.

[14] The actual update rate turned out to be lower for most of the configurations, as will be discussed.

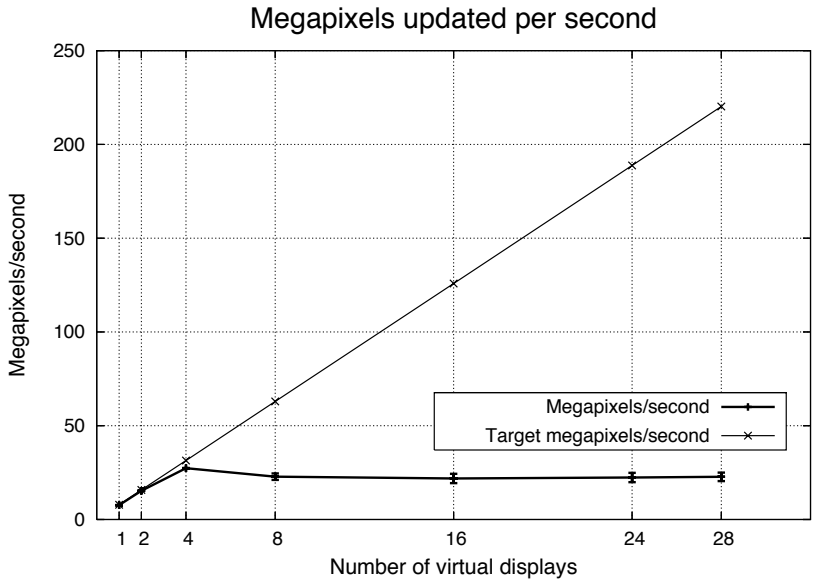


Figure 6.8: The actual number of megapixels updated per second by the virtual displays, compared to the target rate.

The corresponding frame rate at this point is 8.70 fps[15], which is slightly less than the target of 10 frames per second. The frame rate is graphed in Figure 6.9, where the number of pixels refreshed per second is converted to the corresponding frame rate as the number of virtual displays increase.

NADs	MPx/s	$\sigma_{MPx/s}$	MB/s	$\sigma_{MB/s}$	FPS	σ_{FPS}	CPU load
1	7.66	0.00	10.96	0.00	9.74	0.00	39.79
2	15.33	0.02	21.94	0.04	9.75	0.01	75.84
4	27.38	0.12	39.28	0.23	8.70	0.03	163.89
8	22.83	1.70	32.75	2.45	3.63	0.27	177.23
16	21.90	2.49	31.43	3.51	1.74	0.19	179.53
24	22.38	2.47	32.29	3.47	1.19	0.13	180.05
28	22.74	2.26	32.91	3.18	1.03	0.10	180.18

Table 6.1: Results from the 22 megapixel laptop evaluation. NADs is the number of NADs used. MPx/s is megapixels per second, MB/s is megabytes/second and FPS is the frame rate. $\sigma_{\{MPx/s,MB/s,FPS\}}$ is the respective metric's standard deviation. CPU load is the sum of the CPU loads for all of the three processes in percent. The total exceeds 100% due to the use of a dual-core laptop.

Figure 6.10 shows the bandwidth used by the 22 megapixel laptop as the number of virtual displays increase. The bandwidth usage peaks at 39.28 MB/second driving four virtual displays, and correlates well with the pixel update rate in Figure 6.8.

[15]The frame rate is calculated as the update rate divided by the total area (4 virtual displays = 2x2 displays = $(2 * 2) * (1024 * 768) = 3145728$ pixels): $\frac{27.38 MPx/s}{3.145728 MPx} = 8.70$.

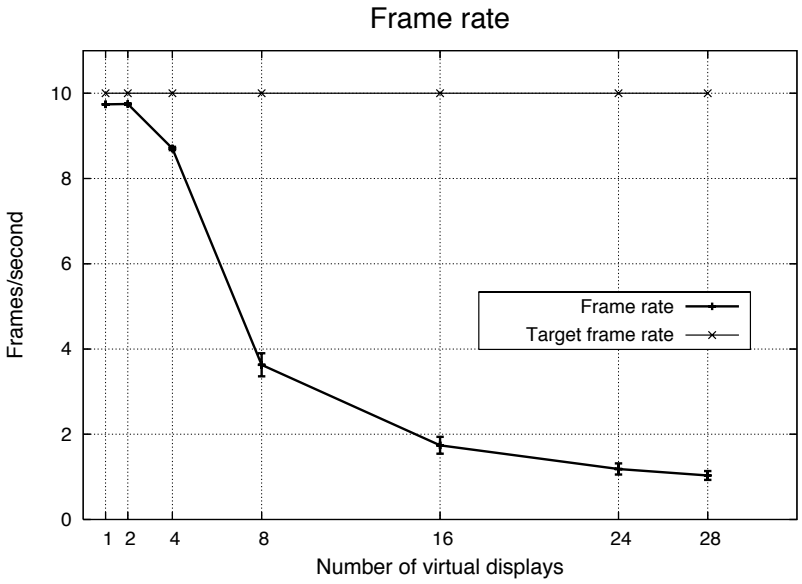


Figure 6.9: The target frame rate (10 frames per second) compared to the actual frame rate achieved by the 22 megapixel laptop. The graph is obtained by dividing the megapixels/second measurement by the combined resolution of the area being refreshed.

Figure 6.12 shows the CPU load incurred by the display sharing daemon, the draw process and the Mac OS X window server, as well as the sum of their loads. The total load peaks at 180%[16] with 24 and 28 displays. Most of the CPU is used by the display sharing daemon. Figure 6.11 shows the CPU load at kernel and user level for the three processes. The draw process spends very little time at kernel level, and most of its time in user space, peaking with a load of 34.7% at 16 displays. The window server and draw process track each other's user level loads closely. The window server's kernel level load is also very similar to its user level load.

6.3.5 Discussion

The experiments demonstrate the tradeoff between the size of the area being updated, and the rate at which the area can be updated. As the size of the area to update grows, the rate at which it can be updated goes down. The results show that the 22 megapixel laptop in its current implementation is able to refresh up to 27.38 megapixels/second. For a single virtual display, this would correspond to a frame rate of $\frac{27.38 MPx/s}{1024*768 pixels} = 34.8$ frames per second. To determine the actual peak frame rate for using a single NAD, a follow-up experiment was conducted, where the draw process was configured to run at a target rate of 30 frames per second, and then measure the number of pixels updated when sending either 16-bit

[16]The maximum load is 200% since the MacBook Pro has a dual-core processor.

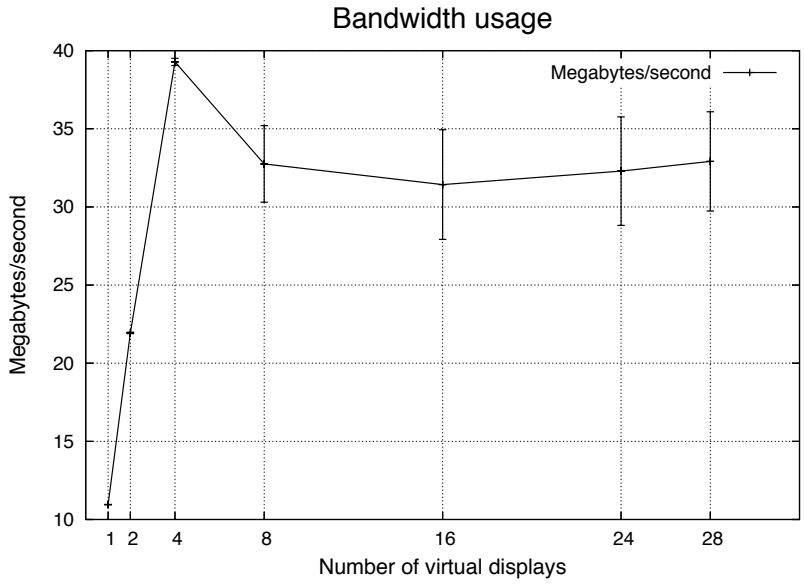


Figure 6.10: The bandwidth used to send pixels to the NADs.

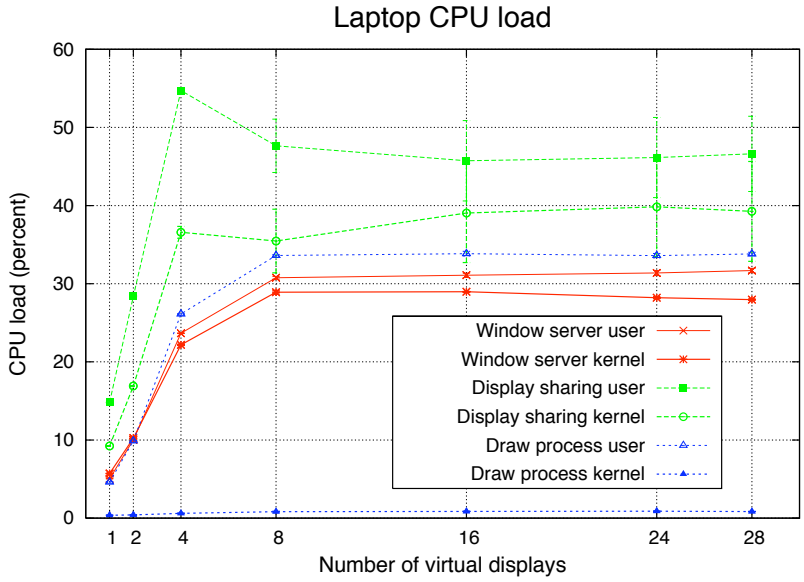


Figure 6.11: The CPU load measured at user and kernel level for the window server, display sharing daemon and draw process. The CPU load is measured in percent.

or 32-bit pixels to a single 1024x768 NAD[17]. At a depth of 16 bits per pixel, a frame rate of 25 was measured, and for 32 bits per pixel, the measured frame rate

[17]Network usage and CPU load was not measured for this follow-up experiment.

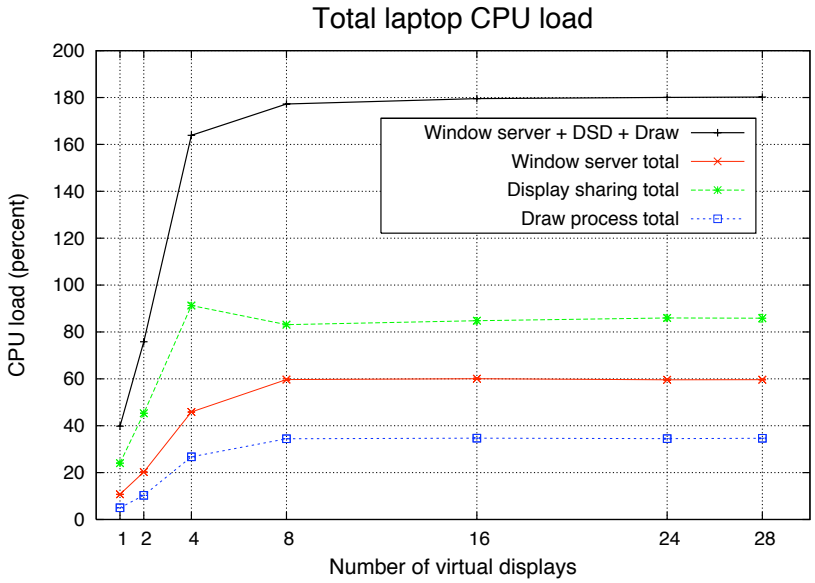


Figure 6.12: The sum of the CPU load at user and kernel level for the three processes, as well as the sum of the three processes' CPU load. All measurements in percent of the CPU.

was 18. Taken together, these performance measurements indicate that the system provides sufficient performance for working with relatively static content – such as displaying documents, images or similarly static data – on the display wall. When a single low-resolution NAD is used, the system has sufficient performance to share dynamic content like videos. However, the motivation for the 22 megapixel laptop was to enable transparent usage of nearby pixel resources, and not to improve on the performance of the state-of-the-art in pixel sharing systems (such as VNC [17], THINC [155] and SAGE [40]).

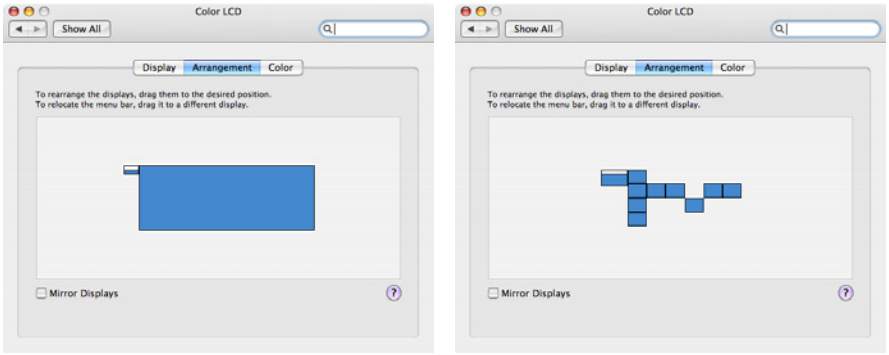
The results show that the network is never saturated by the system. The peak transfer rate of 39.28 MB/second is less than half of the bandwidth available from one computer on a switched, gigabit Ethernet. This rules out the network as the bottleneck. The next candidate is CPU load. The CPU load correlates well with the number of virtual displays used, approximately doubling every time the total resolution of the virtual displays doubles. However, the CPU load peaks at 180% when 4 virtual displays are used, indicating that the CPU is not the main bottleneck either. However, the CPU load measurements still give valuable clues as to where the main bottleneck is.

The draw process incurs little load at kernel level, regardless of the number of virtual displays. This is as expected, since the draw process does all of its work at user level, by drawing to a window covering the virtual displays. This drawing is performed by copying data from one buffer to another, an operation that does not involve the kernel at all. The draw process instead incurs this CPU load at user

level. The time spent at user level is almost exclusively due to copying data from an image buffer to the window covering the virtual displays.

The window server's CPU load tracks the CPU load of the draw process almost perfectly. The reason for this is that all windows on Mac OS X are double-buffered. The draw process draws to the window's back buffer, which the window server then copies to the virtual display's framebuffer. However, the window server appears to be doing twice the work necessary: It incurs almost the same load at both kernel and user level. This indicates that the window's contents are copied no less than three times: First from the draw process to the back buffer, then from the back buffer to some internal window server buffer, and then finally to the virtual display's framebuffer.

The display sharing daemon incurs most of the CPU load, both at kernel and user level. Between 53-62% of the DSD's load is incurred at user level. The time spent at user level is due to copying data from the each virtual display's framebuffer, compressing the data and then queuing it to be sent by the kernel. The kernel level load comes from having to copy the buffers queued by the DSD at user level, and then transferring the data over the network using TCP. All of this indicates that the main bottleneck in the system as the total display resolution increases is copying data in local memory. It is likely that better performance could be achieved by eliminating redundant memory copies. Ideally, the apparently redundant memory copy performed by the window server and the kernel's need for making a copy of buffers sent over TCP should be eliminated. This would move the bottleneck either fully to the local CPU, or to the network.



(a) (b)

Figure 6.13: Two screenshots of the Displays control panel in the System Preferences application on Mac OS X. (a) A single virtual display has been configured with a resolution of 16384x6144 pixels (about 100 megapixels). The laptop's internal display is visible as the tiny box to the left, with a resolution of 1440x900 pixels. (b) Attempting to manage 28 virtual displays using the built-in Mac OS X display management software. Only nine of the 28 virtual displays are visible, for a total of ten displays including the laptop's built-in display.

Figure 6.13 shows two indirect results obtained by implementing the 22 megapixel

laptop. Figure 6.13(a) demonstrates the window server can support a 100 megapixel virtual display with a resolution of 16384x6144 pixels. In Figure 6.13(b), a screenshot of Mac OS X' System Preferences application is shown. At the time the screenshot was taken, the 28 virtual displays were arranged to match the Tromsø display wall configuration. However, the Displays control panel in System Preferences does not anticipate more than 10 displays ever being connected at the same time. The selection of virtual displays further appears completely random. Further investigation uncovered that the "Displays" menu extra (which ships with Mac OS X) is limited to showing the state of up to 16 displays, and that the Mac OS X window server itself limits the maximum number of displays to 32. Of these 32 displays, only 30 slots are actually available for use, since the window server appears to maintain a single "ghost display" sized at 1x1 pixel, and the laptop's built-in display also consumes one slot. The purpose of this invisible and inaccessible ghost display is unknown. Despite the 32-display limit, the window server appears to detect the presence of additional displays beyond this "magic" limit. However, the window server makes no attempt at utilizing them.

6.4 De-centralized VNC

The idea behind De-centralized VNC (DVNC) is to improve the performance of VNC [17] when VNC is used to create the desktop environment on a tiled display wall, by delegating work from the VNC server to the VNC viewers. Section 2.1.1 describes how VNC is used to create a traditional desktop environment on the Tromsø display wall. DVNC improves the performance of regular VNC by enabling VNC viewers to exchange pixels amongst each other. This section is based on research presented in [27].

6.4.1 Model and design

Figure 6.14 shows the VNC model for original VNC and DVNC. VNC is usually used to give a single VNC viewer access to a framebuffer shared by a remote VNC server. When used on a display wall, the remote server's framebuffer is divided between several VNC viewers. However, doing this makes VNC's "Copy Rect" update operation less efficient, since each viewer no longer has access to all the pixels from the VNC server's framebuffer. To resolve this issue, DVNC changes the VNC model by enabling VNC viewers to exchange pixels amongst each other, so that the Copy Rect operation can be executed as if all the VNC viewers had access to all the pixels shared by the VNC server.

VNC uses the RFB protocol [90] to share pixels from a VNC server to one or more VNC viewers. The RFB protocol uses three main operations to update a remote viewer with new pixels: Image Rectangle, Fill Rectangle and Copy Rectangle,

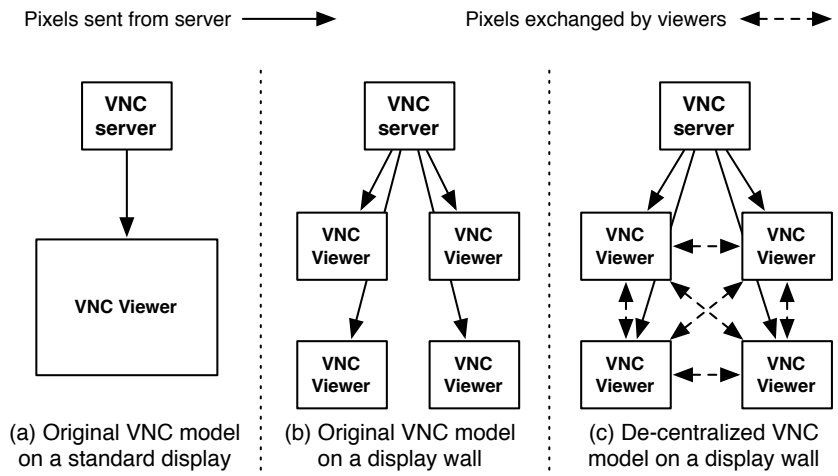


Figure 6.14: The VNC model when used with a regular display, a display wall, and the De-centralized VNC model on a display wall.

shown in Figure 6.15. The Image Rect operation is used to update a rectangle on the remote viewer with a given set of pixels, which are included as part of the operation. The Fill Rect operation is used to fill a rectangle with a specific color, and the Copy Rect operation is used to move the pixels inside a rectangle by a given delta dx, dy, as illustrated in Figure 6.16(a). The delta indicates the number of pixels left/right and up/down the area should be moved. The Copy Rect operation is used when areas of the screen are moved, but not otherwise changed. For instance, moving a window, panning an image or scrolling a document generates Copy Rect operations.

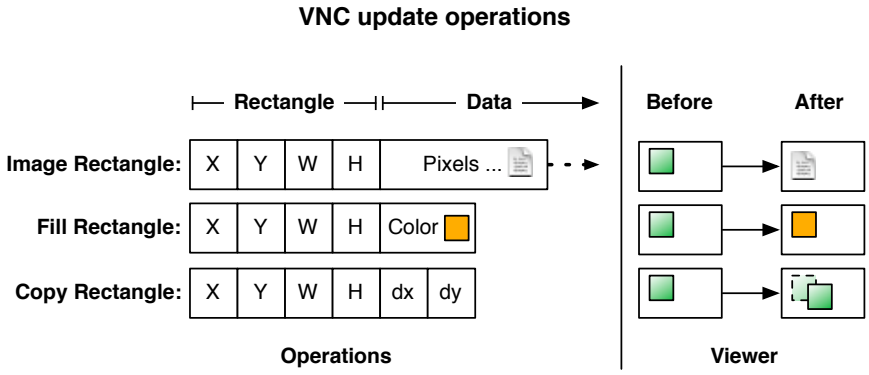
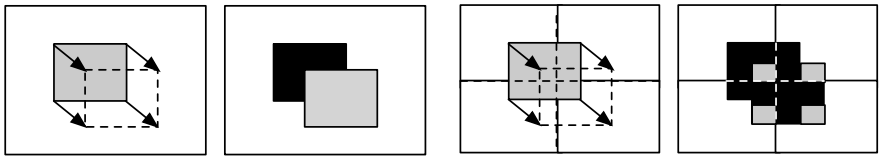


Figure 6.15: The three operations used by the RFB protocol to update a remote viewer with new pixels. The Image Rect operation is variable length, depending on the number of pixels included with the operation. The Fill Rect and Copy Rect operations are fixed in size at 12 bytes. X, Y, W and H represent the offset and size of the rectangle. For the Copy Rect operation, dx and dy indicate the horizontal and vertical amount by which the destination rectangle is offset.

Of the three operations, the Image Rect operation is the most costly to transfer over a network since it requires sending both the rectangle and the pixels to fill the rectangle with. For the Tromsø display wall, a full update of all the pixels using a single Image Rect operation would require transferring 63 MB if the pixels were uncompressed[18]. In comparison, the Fill Rect and Copy Rect operations require only 12 bytes to encode, regardless of the size of the area being updated. The Fill Rect operation specifies the rectangle and a color to fill the rectangle with, while the Copy Rect operation specifies a rectangle and a movement delta.

The bandwidth used by the Image and Fill Rect operations does not change much when using VNC to drive a display wall. There is some added overhead in that the server must send several smaller Image Rect/Fill Rect operations to different viewers, rather than one large Image Rect/Fill Rect operation to a single viewer. However, the VNC server does not send any redundant data to the viewers.



1. The grey area is moved down and to the right by copying the pixels from the source to the destination area.

2. After the grey area has been moved, the black area is exposed, and must be refreshed by the server using Image Rect operations.

(a)

1. On a tiled display, the Copy Rect operation is less useful, as each viewer does not have a complete copy of the framebuffer.

2. Each viewer is only able to complete part of the Copy Rect, resulting in a much larger region that must be refreshed by the VNC server.

(b)

Figure 6.16: The Copy Rect operation used when (a) a single viewer displays the entire remote framebuffer, and (b) several viewers display different parts of the remote framebuffer.

The Copy Rect operation is different. When using several viewers to display the framebuffer exported by the VNC server, the Copy Rect operation is less effective than if a single viewer were used to display the entire framebuffer, as shown in Figure 6.16. When the area is moved, it will expose pixels that the server must update using the Image Rect operation. When a single viewer displays the entire framebuffer, this is not a problem. However, when the server's framebuffer is split amongst many viewers, Copy Rect operations spanning the boundaries between viewers end up exposing a larger area that must be refreshed by the server. This leads to additional load on the server, which would have been avoided if all the remote viewers had access to the entire framebuffer exported by the server. However, giving all viewers access to the VNC server's entire framebuffer is neither

[18] Assuming 3 bytes per pixel * 7168 * 3072 pixels = 63 MB.

practical nor scalable, at least if the server is to be responsible for keeping each viewer updated.

To resolve this problem and improve VNC's performance on tiled display walls, VNC is de-centralized. The Image and Fill Rect operations can not be de-centralized. The Image Rect operation always relies on pixels computed by the server, and the Fill Rect operation's efficiency is not affected much by the move to a display wall. The Copy Rect operation can be de-centralized by having the viewers communicate amongst each other to exchange the pixels necessary for completing the operation *as if* all the viewers involved had access to the entire remote framebuffer. This reduces the server load, since the server spends less resources sending pixels that have already been distributed to the viewers. Figure 6.17 shows how the Copy Rect operation is de-centralized.

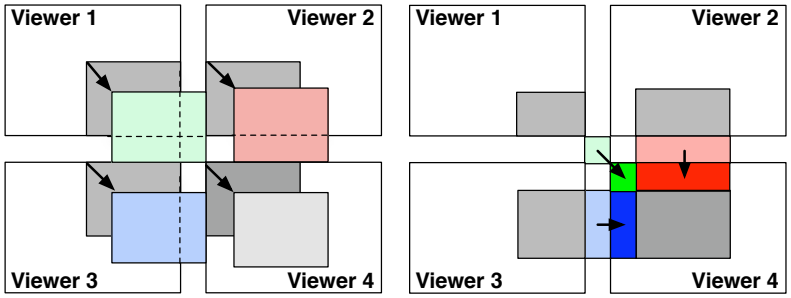


Figure 6.17: De-centralizing the Copy Rect operation. (a) Four tiles receive a Copy Rect operation that overlaps all of them, indicating that the grey area is to be moved down and to the right in the direction of the arrows. The colored areas indicate the destination for viewers 1, 2 and 3. (b) To perform the operation, viewer 4 receives the pixels it does not have locally from viewers 1, 2 and 3, shown as the incoming, colored boxes. Similarly, viewers 2 and 3 receive pixels from viewer 1 (not shown).

DVNC enables viewers to send updates to each other. This introduces a race condition between updates received from the server, and updates received from other viewers. An example of this race condition is illustrated in Figure 6.18. Two viewers A and B receive updates for non-overlapping areas of the server's framebuffer. The server begins by updating the framebuffer using a Copy Rect operation that spans both viewers, and then updates a part of the framebuffer covered by viewer B using an Image Rect operation. If viewer A does not send the necessary pixels to viewer B before viewer B applies the Image Rect operation, the result is a display that is inconsistent with the VNC server's framebuffer.

To avoid this problem, DVNC imposes a total ordering on all updates sent by the server, and a requirement that all viewers must receive the exact same set of Copy Rect operations. Since all viewers receive the same Copy Rect operations, they can make independent decisions about whether or not they participate in the execution of a given Copy Rect operation. If the rectangle covered by a viewer overlaps

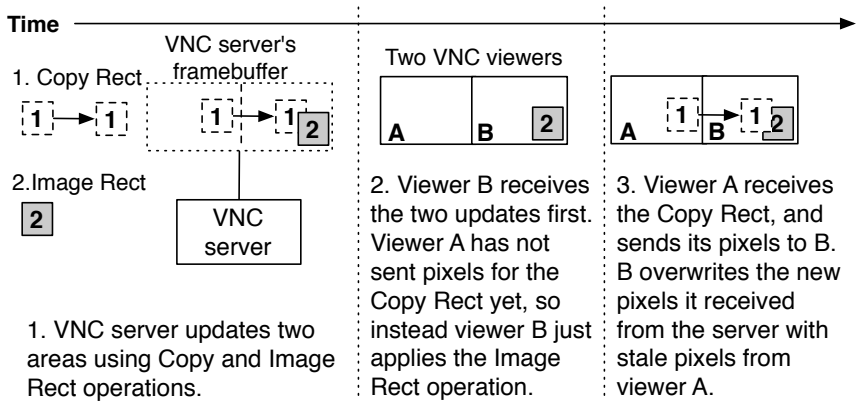


Figure 6.18: Possible race condition in DVNC that must be handled in order to ensure that the pixels displayed by the viewers remains consistent with the pixels in the framebuffer exported by the server.

with either the source or destination rectangle given by the Copy Rect operation, it participates in the Copy Rect operation either as a sender, receiver or both of pixels. Using this knowledge, a viewer can determine if it should delay the execution of an Image or Fill Rect operation to maintain consistency.

To avoid circular dependencies, the Copy Rect operation is split into a pre-phase and a post-phase. During the pre-phase, the viewer sends the pixels that will be moved out of its local framebuffer to the viewers that need them. The Copy Rect is then executed locally, but not completed until the post-phase occurs. The post-phase is executed once the viewer has received pixels from other viewers pertaining to the current Copy Rect operation. At this point the Copy Rect operation is completed and other operations from the VNC server can be processed.

6.4.2 Implementation

DVNC has been implemented by modifying the RealVNC [26] distribution version 4. RealVNC is an open-source VNC implementation. Both the server and the viewer code required modifications, with the majority of changes made to the viewer code.

Server modifications

Four modifications were made to the server, by: (i) adding support for sending identical Copy Rects to all viewers; (ii) not clipping Copy Rect operations to the area covered by viewers; (iii) adding an explicit timestamp (a counter) to the beginning of every framebuffer update; and (iv) adding support for measuring load and bandwidth to evaluate the resulting implementation.

The first change is to the way updates are distributed by the VNC server. The original VNC server accumulates updates for each viewer in a change set, and then sends the set of changes to a viewer once the viewer requests them. The change set is then cleared, before new updates begin accumulating while the server waits for another request from the viewer. This way to manage what has been sent to viewers has the benefit of accommodating both fast and slow viewers, since fast viewers can request updates more often than slow viewers and be serviced accordingly.

In DVNC, this behavior is changed to ensure that all viewers receive the same set of Copy Rect operations. To achieve this, the server is modified to accumulate just one set of changes for all viewers, and prevent the server from sending changes to the individual viewers until all of them have requested an update from the server. The drawback to these modifications is that the server can provide updates no faster than the slowest viewer. However, in the display wall environment, all viewers run on identical hardware using the same local network, thus the difference between the slowest and fastest viewer is expected to be small.

The second change is to no longer clip the Copy Rect operation to the area covered by a viewer. In the original VNC implementation, the VNC server clips all update operations to the area covered by a viewer to avoid sending updates to the viewer that it won't display[19]. In DVNC, this behaviour has been changed to not clip the Copy Rect operation, while the Image and Fill Rect operations continue to be clipped as before.

The third change was to add a 4-byte timestamp to the RFB protocol's framebuffer start message. The purpose of this timestamp is to enable viewers to disambiguate Copy Rect operations covering the same area (that is, the same rectangle and delta values) from each other, and thus maintain a consistent display. The drawback to this change is that the network protocol is modified from the standard VNC protocol, making DVNC incompatible with regular VNC viewers.

The final change was to extend the server with functionality for logging the server's CPU load and bandwidth usage. The purpose of this was to enable experimental insights into the final performance of the system. The instrumentation follows the pattern outlined in Chapter 3. This instrumentation was incorporated both into the DVNC server, and the original VNC server.

Viewer modifications

The viewer was modified in four ways to add support for: (i) viewer discovery; (ii) queueing of incoming update operations; (iii) viewer-to-viewer pixel exchange; and (iv) performance measurements.

[19] For instance, a viewer covering the area $x, y, width, height = \{0, 0, 1024, 768\}$ does not need to receive pixels that do not overlap this rectangle, such as $\{1100, 0, 2, 2\}$.

Viewer discovery is handled by a separate thread using the network discovery mechanism detailed in Appendix C. Each discovery message contains the port on which the viewer listens for incoming connections from other viewers, as well as the region of the VNC server's framebuffer that the viewer covers. When a viewer starts up, it uses the discovery mechanism to connect to all other viewers. Once a connection to a viewer has been established, any Copy Rect operations that involve the two viewers will result in pixels being exchanged between them.

In contrast to the original viewer implementation, incoming update operations from the server are not applied immediately. Instead, they are queued, and then processed when the server signals the end of a framebuffer update. This introduces a queueing overhead that is not present in the original implementation. To reduce this overhead, updates that do not overlap with already queued operations are applied immediately.

Before applying an update, its type is examined. Any Copy Rect operations encountered have their pre-phase executed, before the viewer attempts to execute the Copy Rect's post-phase. This only succeeds if the viewer has received the necessary pixels from other viewers, or if no pixels are needed from other viewers for that Copy Rect operation. Otherwise, the operation is put back on the queue. Other operations from the same framebuffer update may be applied afterwards, but only if they do not intersect with the area covered by any preceding Copy Rect operations.

A separate thread is used to exchange pixel updates with other viewers. The thread sets one outgoing pixel queue for each connected viewer and one incoming pixel queue in which pixels received from other viewers are collected. The queues are used when the Copy Rect pre- and post-phases execute. The Copy Rect pre-phase enqueues pixels from the local viewer to remote viewers, while the post-phase fetches pixel updates from the thread's incoming data queue. The thread communicates with other viewers using TCP. Pixels are sent as they are stored in memory locally by the viewer. For this to work, all viewers must run on hardware with the same endianness and represent pixels in the same way. A fully cross-platform implementation should either byteswap pixels as necessary or compress them to reduce network bandwidth usage.

6.4.3 Evaluation

The De-centralized VNC implementation has been evaluated by measuring its performance for two traces and a set of control experiments, and comparing the result to the performance of the original (but instrumented) VNC implementation. Two different computers were used to run the VNC server. Their specifications are listed in Section 2.2.5. The two computers will be referred to as the Pentium 4 and the Xeon. The experiment was conducted on and using hardware from the Tromsø

display wall.

The evaluation measured the following four metrics: (i) Total number of pixels refreshed; (ii) total number of bytes sent by the server to the viewers; (iii) the server's CPU load; and (iv) the VNC viewer queueing overhead[20].

Methodology

The DVNC and VNC servers were configured to export a 16-bit color desktop with a resolution of 7168x3072, matching the resolution of the Tromsø display wall. They were both instrumented to record their CPU load at kernel and user level. The CPU load is sampled 10 times per second, and sent using Shout to a logging application running on a different computer. The bandwidth impact of this is negligible at less than 500 bytes per second.

The VNC viewers were instrumented to record three statistics: (i) The number of pixels updated; (ii) the number of bytes received from the server; and (iii) the queuing overhead. Each viewer makes its own measurements, before the results are gathered at the end of an experiment. The pixel update count and bytes received are summed, producing a total number of pixels updated by all the viewers, and the total number of bytes sent by the VNC server to all the viewers. The queuing overhead is collected to produce histograms and analyzed to produce min/max/mean statistics.

Three sets of experiments were conducted, listed in Table 6.2. The first two were recorded trace experiments, where the system's performance when a user interacts with the desktop environment was measured. The third was a control experiment, where a custom event generator replaces the user trace with the goal of determining the maximum performance gain using DVNC in a situation where the server's possibility for using the Copy Rect operation is near maximized.

The two trace experiments rely on two traces recorded of a user panning an image and moving a window. The input events generated by the user as the image was panned and the window was moved were recorded and then later played back using the Shout event system. The trace experiments were conducted on both the Pentium 4 and Xeon.

The control experiments use a custom-written event generator to move the image from the Image Pan trace up and down. The event generator uses the `XTestExtension` API to X to post input events to the DVNC or VNC X-server. An illustration of the control experiment's operation is shown in Figure 6.19. Since the image covers almost the entire display wall, this experiment maximizes the VNC server's opportunity for using the Copy Rect operation. Fresh pixels that the

[20]The queuing overhead adds to the latency between when a viewer receives an update, and when that update is drawn to the viewer's display.

Experiment	Image size	Duration	Notes
Image Pan	9372x9372	255 s	A trace of a user panning an image covering nearly the entire display wall[a] is played back.
Window Move	2592x1944	145 s	A trace of a user moving a window around on the display wall is played back.
Control	9372x9372	30 s	The image from the Image Pan trace is moved up and down in a programmed way using a custom event generator, in increments of 8 pixels. The rate of movement is varied for each experiment, to determine when the VNC and DVNC servers' performance break down.

Table 6.2: The three experiments, and their characteristics.

[a] With the exception of the image viewer's window title bar.

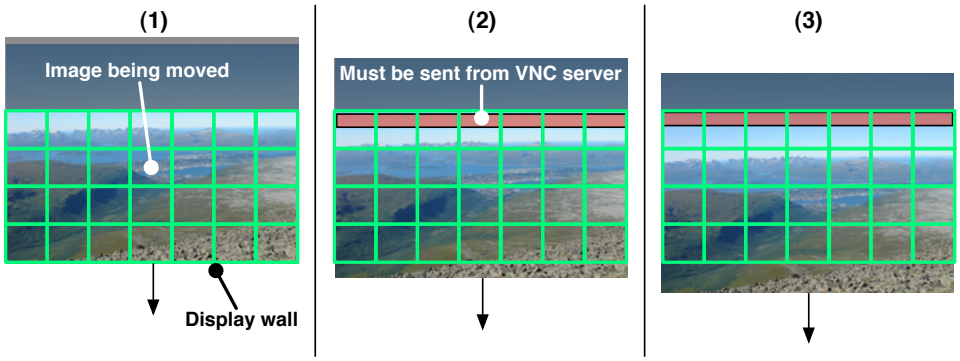


Figure 6.19: An illustration of how the control experiment operates. The green outline indicates the tiles of the Tromsø display wall. An image covers the display wall, with parts of it being outside the area that the display wall can show. The image is moved down at a fixed rate. The VNC server must provide new pixels at the top (as shown by the red boxes in (2) and (3)) when the image is moved down; when the image is moved back up new pixels must be provided at the bottom (not shown).

VNC server must send to the viewers only appear either at the bottom or the top of the display wall, depending on which direction the image is moving in. For each experiment, the rate at which the image is moved is varied, from one movement of 8 pixels per second up to 50 movements (400 pixels) per second. The control experiments were only conducted on the Pentium 4.

To characterize the additional network overhead introduced by the changes to DVNC, and in particular the introduction of a timestamp in the "framebuffer start" message of the RFB protocol [90], a null-benchmark was also conducted. The VNC server's desktop was configured to display a static image. The number of bytes to refresh

all the viewers on the display wall were measured. The original VNC server sent a total of 85688.97 KB, while the DVNC server sent 85707.69 KB, which represents an overhead of 0.02%.

Trace results

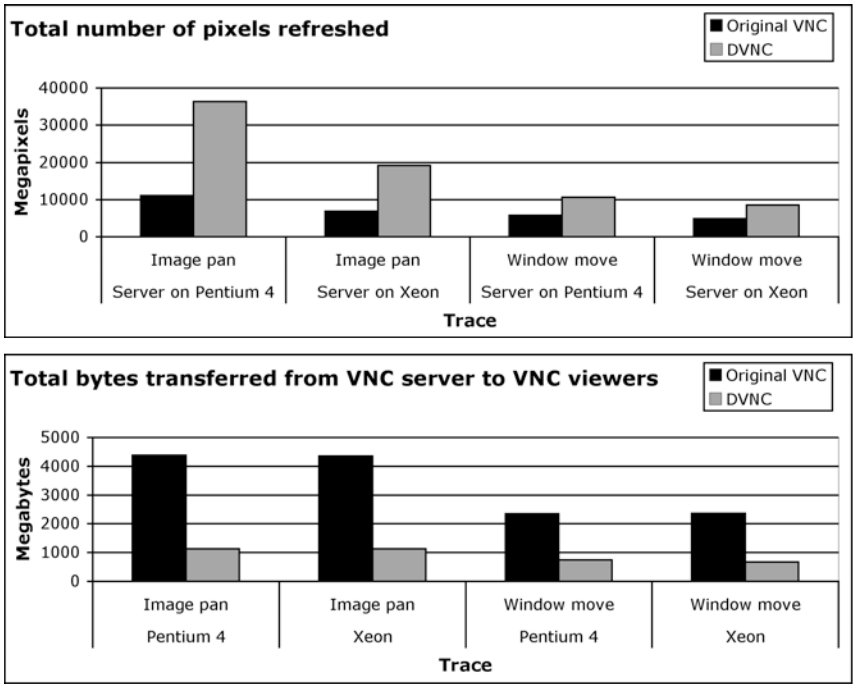


Figure 6.20: Top: Total number of pixels refreshed for each trace by the original and DVNC viewers. Bottom: Total number of bytes sent by the server to the viewers.

Figure 6.20 shows the number of pixels refreshed and the bytes sent for the two trace experiments when conducted on the Pentium 4 and Xeon. The source data appears in Table 6.3. DVNC improves the performance of the original VNC implementation when running on both the Pentium 4 and the Xeon. However, the Pentium 4 refreshes almost a factor of two more pixels than the Xeon for both the DVNC and original VNC implementation. The number of bytes sent does not change much between the Pentium 4 and Xeon.

Figures 6.21 and 6.22 show the cumulative VNC server CPU load for the Image Pan and Window Move traces, respectively, for both the DVNC and original VNC servers. The X axis shows the running time of the traces, and the Y axis shows the CPU time consumed by the DVNC and original VNC server as the experiment progresses. Table 6.4 summarizes the results.

DVNC significantly reduces the CPU load of the server by 20% to 35% compared

Trace	Version	CPU	Pixels refr.	Chg	Bytes sent	Chg
Image Pan	DVNC	P4	36.3 GPx[a]	3.30	1132.06 MB	0.25
Image Pan	VNC	P4	11.0 GPx		4385.00 MB	
Image Pan	DVNC	Xeon	19.1 GPx	2.76	1136.82 MB	0.26
Image Pan	VNC	Xeon	6.9 GPx		4359.29 MB	
Window Move	DVNC	P4	10.6 GPx	1.85	750.24 MB	0.32
Window Move	VNC	P4	5.7 GPx		2347.74 MB	
Window Move	DVNC	Xeon	8.4 GPx	1.71	671.91 MB	0.28
Window Move	VNC	Xeon	4.9 GPx		2357.96 MB	

Table 6.3: The trace results. The Trace, Version and CPU columns indicate which trace was run, the version of VNC used (either DVNC or the original VNC implementation), and the computer used to run the server. The results are given in the Pixels refreshed and Bytes sent columns, with the relative change between DVNC and the original VNC implementations[b] listed next to the results.

[a] Gigapixels. Note that these values differ from the ones that appear in the paper [27] due to conversion from the incorrect megapixel definition to the correct one, as discussed in Section 3.7.

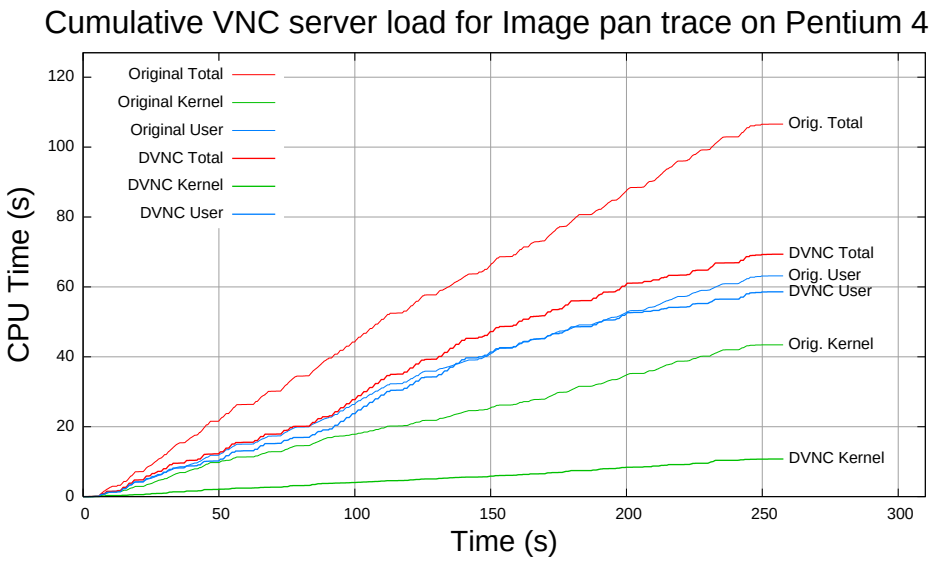
[b] Obtained by dividing the DVNC value by the VNC value.

to the original VNC server. Most of the reduction happens at kernel level, with 76% less load in the Image Pan experiment and a 34% to 46% reduction for the Window Move experiment. The user level load is also somewhat reduced, ranging from 4% to 13%.

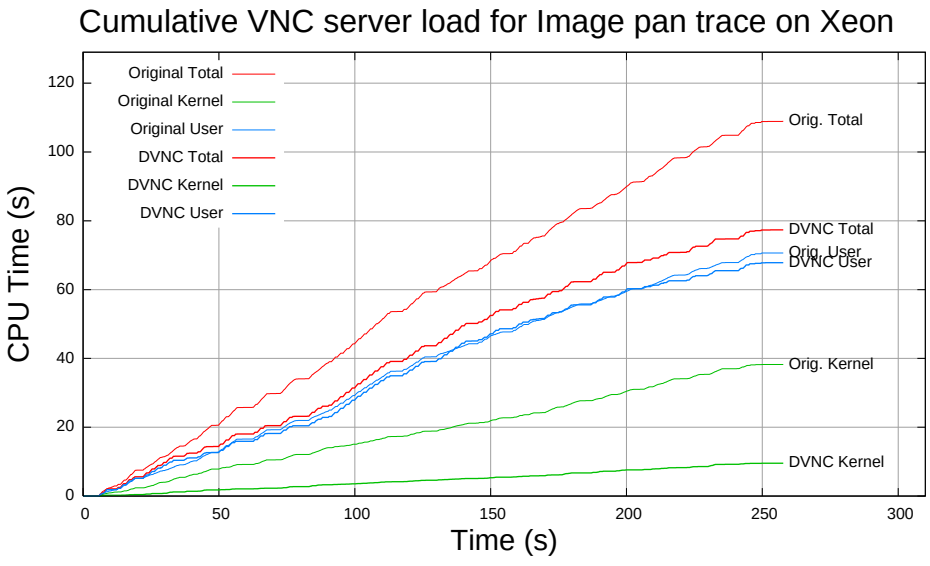
Trace	Version	CPU	Total	Chg	Kernel	Chg	User	Chg
Image Pan	DVNC	P4	69.3 s	0.65	10.7 s	0.24	58.6 s	0.92
Image Pan	VNC	P4	106.5 s		43.4 s		63.1 s	
Image Pan	DVNC	Xeon	77.3 s	0.71	9.5 s	0.24	67.8 s	0.96
Image Pan	VNC	Xeon	108.8 s		38.2 s		70.6 s	
Window Move	DVNC	P4	52.0 s	0.80	14.6 s	0.66	37.4 s	0.87
Window Move	VNC	P4	64.8 s		22.0 s		42.8 s	
Window Move	DVNC	Xeon	56.0 s	0.80	9.5 s	0.54	46.5 s	0.89
Window Move	VNC	Xeon	69.5 s		17.5 s		51.9 s	

Table 6.4: The VNC server CPU load for the trace experiments. The first three columns are as in Table 6.2. The Total, Kernel and User columns indicate the amount of CPU time spent by the server, along with the relative change between DVNC and original VNC[a].

[a] Obtained by dividing the DVNC value by the VNC value.

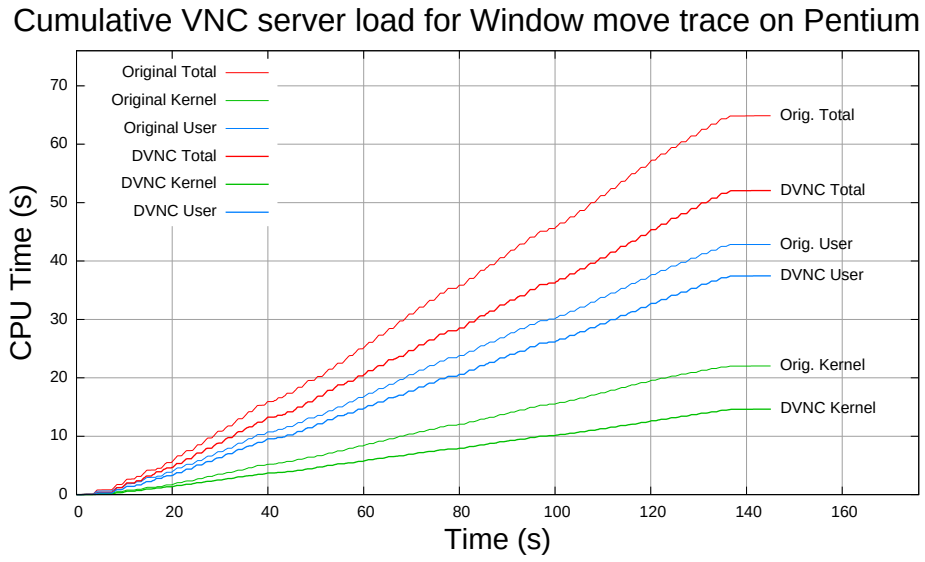


(a)

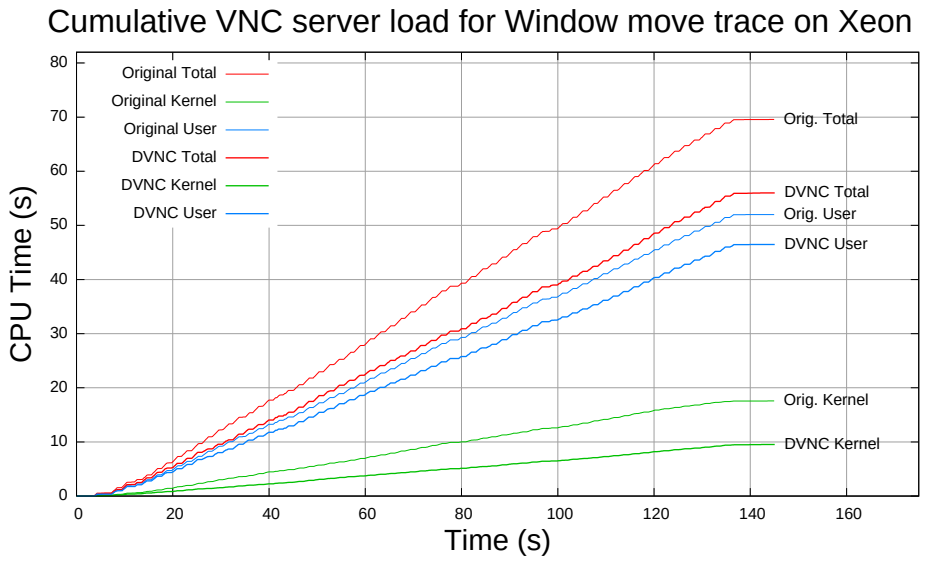


(b)

Figure 6.21: Cumulative server CPU load for the Image pan trace on (a) Pentium 4 and (b) Xeon, with total, user and kernel level load for both implementations.



(a)



(b)

Figure 6.22: Cumulative server CPU load for the Window move trace on (a) Pentium 4 and (b) Xeon, with total, user and kernel level load for both implementations.

Queueing overhead

Table 6.5 shows the queuing overhead for the two traces on the Pentium 4 and Xeon. The minimum overhead is 0.000 seconds, with a mean overhead between 0.01 and 0.02 seconds. The maximum overhead is about 0.64 seconds, which implies that an update operation was queued for more than half a second before being drawn.

Trace	CPU	Min	Avg	Max
Image pan	P4	0.000 s	0.018 s	0.640 s
Image pan	Xeon	0.000 s	0.011 s	0.582 s
Window move	P4	0.000 s	0.012 s	0.512 s
Window move	Xeon	0.000 s	0.011 s	0.504 s

Table 6.5: The queueing overhead, measured in seconds, for the traces on the Pentium 4 and the Xeon. The original VNC implementation does not have a queuing overhead and is thus not included in the table.

Figure 6.23 shows a histogram of the queueing overhead for the Image Pan and Window Move traces on the Pentium 4. The majority of the updates are queued for less than 0.05 seconds, with some outliers contributing to the high maximum queuing overhead values. For the Image Pan trace, 53% of the operations have an overhead less than 0.004 seconds. 70% and 87% of the operations have an overhead less than 0.01 and 0.03 seconds respectively. For the Window Move trace, 54% of the operations have a queueing overhead less than 0.004 seconds, and more than 75% and 91% of the operations have a queueing overhead less than 0.01 and 0.03 seconds respectively.

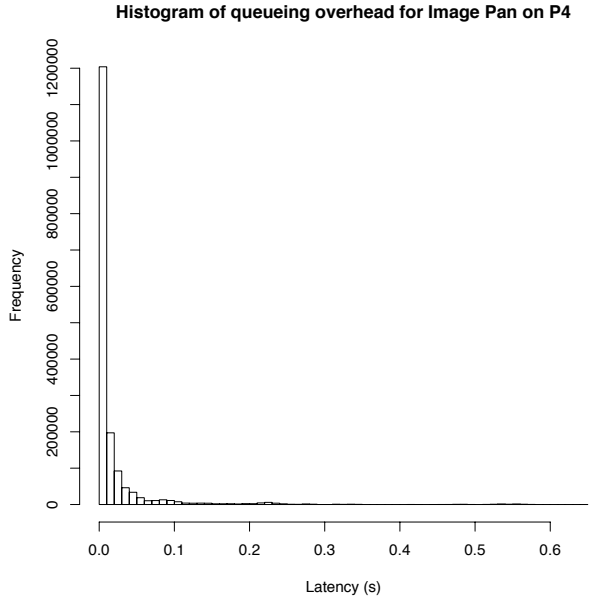
Control results

Figure 6.24 shows the number of pixels refreshed in the control experiments as the rate at which the image was moved was varied between 1 and 50 times per second. The number of pixels for DVNC and the original VNC implementation is compared to a target pixel count. The target update rate is calculated by measuring the number of pixels refreshed when scrolling the image vertically by 8 pixels once, and multiplying that number with the duration of each experiment and rate at which the image is moved.

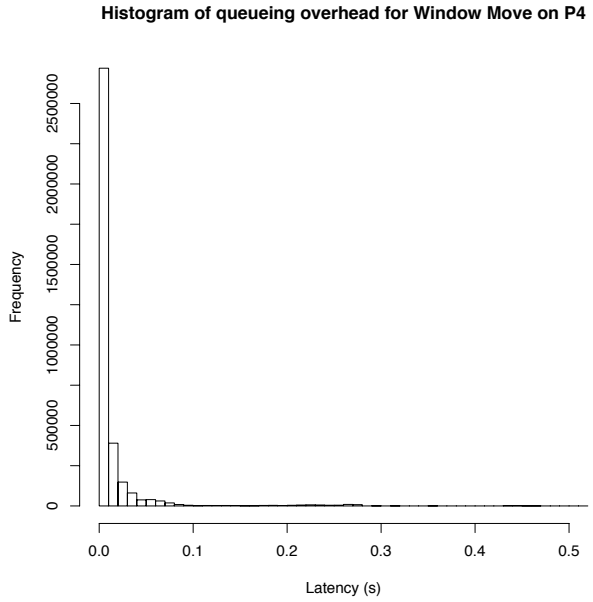
Version	CPU	Pixels refreshed	Chg	Bytes sent	Chg
DVNC	P4	21.4 GPx	11.88	375.89 MB	0.33
VNC	P4	1.8 GPx		1135.40 MB	

Table 6.6: The number of pixels refreshed and bytes sent at event rate 50 in the control experiment, as well as the relative change between DVNC and VNC[a].

[a] Obtained by dividing the DVNC value by the VNC value.



(a)



(b)

Figure 6.23: Histograms of the queueing overhead for the (a) Image Pan and (b) Window Move traces running on the Pentium 4. The X axis shows the overhead in seconds, and the Y axis shows the frequency of VNC server update operations that were executed by the viewers with the given overhead.

The number of pixels refreshed tracks the target pixel count well until the movement rate is 26 movements per second. At this point, the original VNC implementation's performance breaks down, and at a rate of 28 is reduced by 57.8%. DVNC's performance keeps tracking the target until the rate reaches 35, where it too begins to decline. However, the reduction in DVNC's performance happens more gracefully than the original VNC implementation. Table 6.6 shows the results at event rate 50, where DVNC is able to refresh 11.88 times more pixels than the original implementation, while cutting bandwidth usage by two thirds.

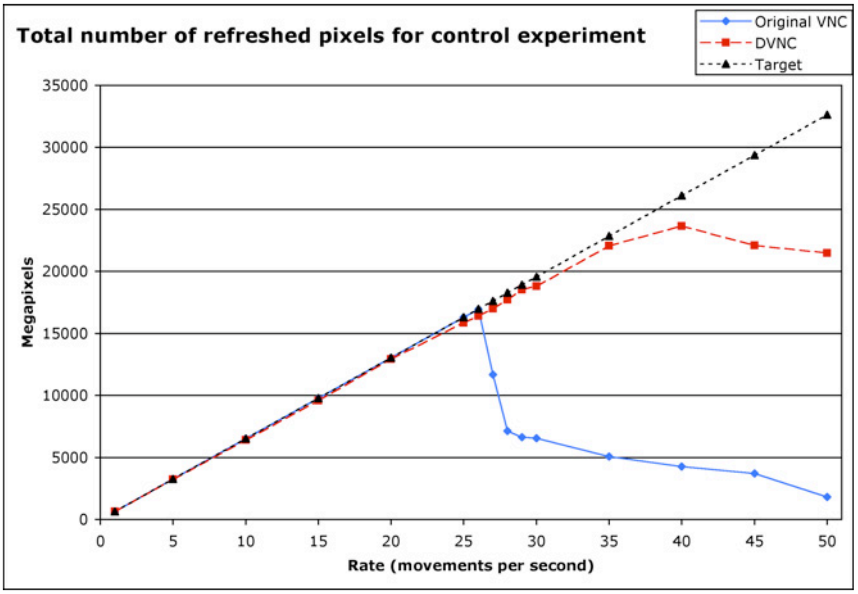


Figure 6.24: Total number of pixels refreshed for the control experiment for the two implementations, as well as the target refresh count. Event generation rates range from 1 to 50.

Figure 6.25 shows the server's CPU load (in percent) at kernel and user level, as well as the total CPU load. The original VNC server incurs a far higher CPU load than the DVNC implementation, until its performance breaks down at an event rate of 26. At this point, the original VNC server is using close to 100% of the available CPU time. When its performance breaks down, there is a marked increase in the kernel level load, at the same time as the user level load goes down. The DVNC server's CPU load behaves more consistently, increasing towards 100% CPU utilization as the event rate approaches 40, before flattening out.

Figure 6.26 shows the total number of bytes transferred from the server to the viewers. The DVNC server sends less data than the original implementation, which is also reflected in the kernel level CPU load. Neither of the two implementations are able to saturate the gigabit Ethernet network, with maximum transfer rates of 37.8 MB/second and 12.5 MB/second for the original and DVNC implementations.

Figure 6.27 shows the cumulative CPU load for the server in the control experiment

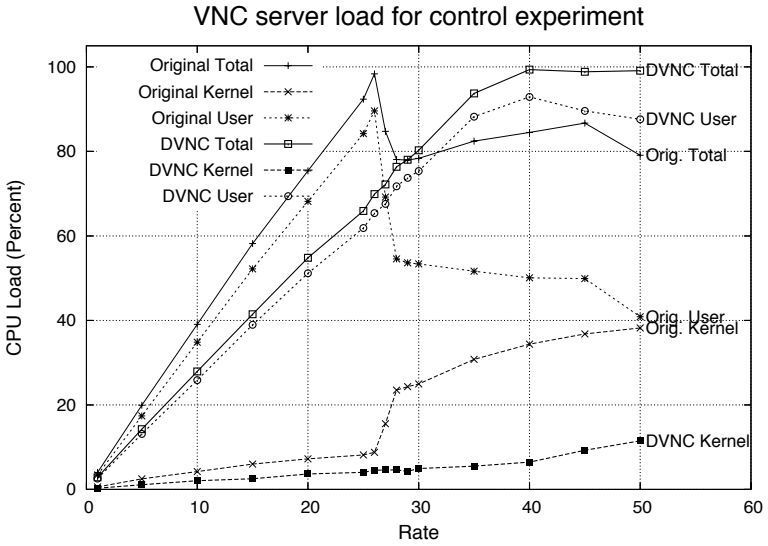


Figure 6.25: CPU load for the VNC server, showing total, kernel, and user level load for both the original and DVNC implementations in the control experiment.

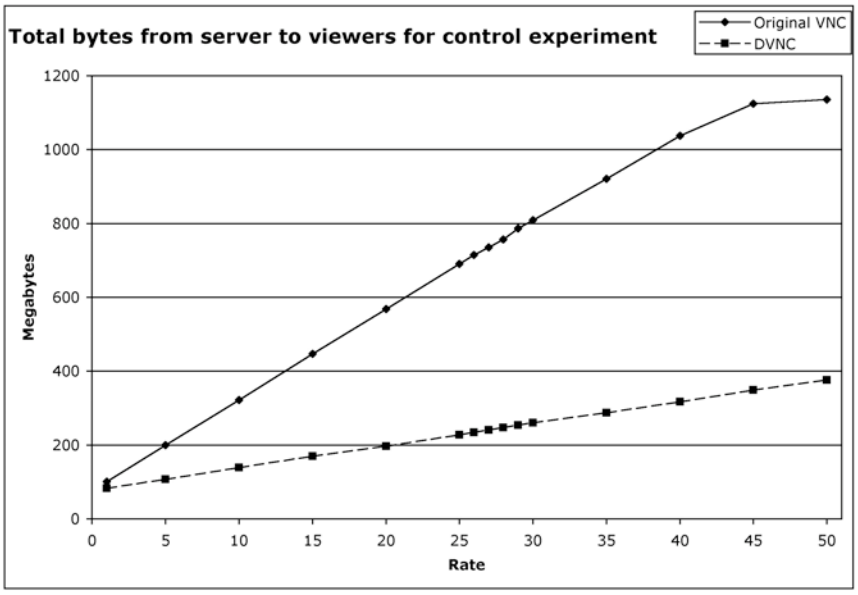


Figure 6.26: Total bytes sent from the servers for the control experiment.

using rate 25. The load increases linearly, which demonstrates that the control experiment successfully incurs an even amount of load throughout the experiment, in

contrast to the “staircase-effect” that can be observed in the load measured during the trace experiments.

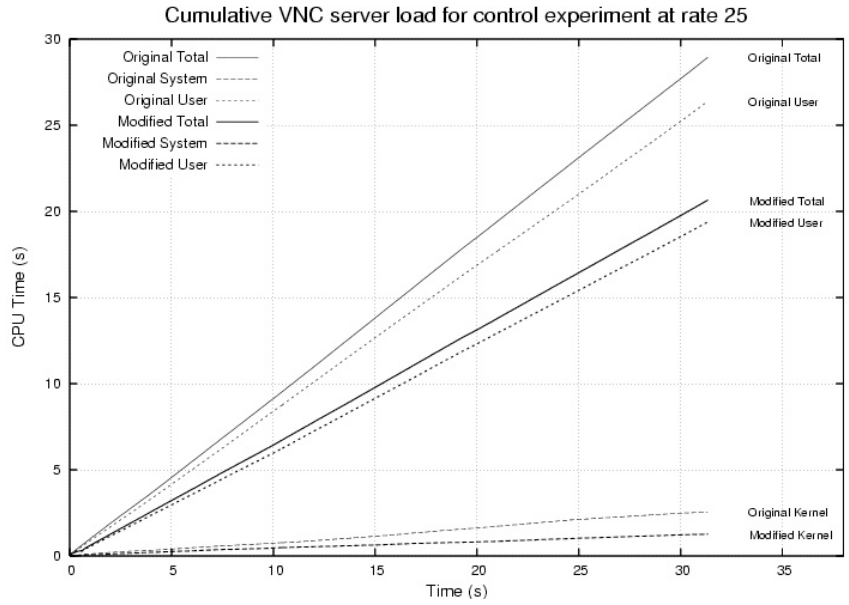


Figure 6.27: The graph shows cumulative VNC server CPU load for one of the control experiments. Total, kernel and user level load is shown for both the original and DVNC implementations. The event rate used was 25.

6.4.4 Discussion

The evaluation documents that DVNC improves the performance of VNC for updating pixels that are moved, but not otherwise changed when using it to drive a tiled display wall. The performance gains can be significant, with gains up to almost a factor of 12 possible as demonstrated by the control experiments. DVNC does not provide any performance gains for pixels that are changed, such as video or other animated content. DVNC does not introduce much additional overhead for such content, with the null-benchmark indicating an overhead of just 0.02%.

The performance improvements are made possible by changing the model at a number of different levels. Instead of having the server do all the work with the viewers remaining passive receivers of pixels, the DVNC approach makes the viewers serve each other in a peer-to-peer style network. The cost is that viewers now need to know about each other, where they before needed no knowledge of other viewers connected to the server. However, the design ensures that each viewer can make independent decisions regarding where to send pixels and from whom they should receive pixels, which avoids deadlocks and keeps the communication between viewers limited to actual pixels, and not coordination messages. Coordination would introduce latency.

The network discovery mechanism works well to connect the viewers to each other. Informal measurements indicate that the start-up time costs incurred by using it are on the order of 1-2 seconds, with viewers appearing "instantly" on the display wall for all practical purposes. An alternative to using the network discovery mechanism for establishing viewer-to-viewer communication is to either: (i) Share a static list of viewers with all the viewers; or (ii) transfer a list of viewers from the VNC server to each viewer as it connects to the server. Both solutions could potentially reduce start-up costs, but with lower flexibility. A static list of hosts would require manual configuration for each display wall environment the system is used in, whereas the discovery mechanism hides these details. Letting the server manage the list has the same downside. In addition, the static list implies embodying the server with knowledge of the display wall's configuration, which is neither necessary nor desirable. The discovery mechanism allows the display wall's configuration to change without having to restart the VNC server. The existing desktop configuration can remain in place as long as the display wall's resolution remains unchanged.

Trace experiments

The trace experiments have demonstrated DVNC's performance in a setting similar to how the display wall desktop environment may be used. The server's kernel level CPU is reduced, leaving more time for the server to work at user level and for other applications on the same computer as the server to execute. The server's user level load is not much reduced, since the server instead spends its time at user level generating more frequent updates to viewers, rather than re-sending already distributed pixels. This is also demonstrated by the reduction in bandwidth usage.

One drawback to using DVNC is the queuing overhead. The queueing overhead is difficult to evaluate: It is easy to measure, but hard to determine what the numbers imply in terms of performance, since there is nothing to compare it to in the original implementation. While the worst-case queueing overhead is about half a second, the original implementation may at this point still be waiting to receive its pixels from the server, due to the potentially large difference in pixel update rate. The evaluation demonstrates that delays of this magnitude are not typical. More than three quarters of the update operations supplied by the VNC server to the viewers are are drawn in less than 0.01 seconds, and more than 90% are drawn in less than 0.03 seconds. Compared to the original VNC implementation, the queueing overhead is deemed insignificant due to the much higher rate at which updates occur. Informally, two years of real-world usage of the system supports this conclusion. The performance gains are also clearly visible in the two DVNC videos that accompany this dissertation.

The staircase effect in the trace CPU load measurements (Figures 6.21 and 6.22) is caused by short periods of user inactivity. During this time, the user typically

repositions the mouse, which results in little activity by the VNC server and thus reduced load. This effect is not apparent in the CPU load measured during the control experiments, as shown by Figure 6.27. The reason is the constant rate of movement in these experiments, which leads to a constant load on the server and a resulting linear increase in cumulative CPU load.

Control experiments

The control experiments were designed to determine the breaking point in terms of performance for the DVNC and original VNC implementations, as well as to determine how much better DVNC is than regular VNC in a situation where the server's ability to use the Copy Rect operation is near maximized. The original implementation breaks down when the server attempts to refresh about $8 * 26 * 7168 = 1490944 = 1.49$ megapixels per second[21]. This is not due to lack of bandwidth, since the amount of data sent continues to increase for higher event rates. Instead, the breakdown comes from a combination of the VNC server having to work harder to keep its framebuffer updated, and having to send more and more redundant data to the VNC viewers.

DVNC performs better in the control experiments than in the trace experiments since movement is constrained to the vertical axis in the control experiments. In contrast, the trace experiments contain diagonal movements, which incur a higher load on the server. The reason is that more pixels must be refreshed (new pixels are introduced not only at the top or bottom, but also to the left or right side of the display wall). Diagonal movements also create more complex dependencies between the VNC viewers when they exchange pixels, although the effect of this compared to the added load on the server has not been quantified.

Server on the Pentium 4 and the Xeon

The trace experiments were conducted on both a Pentium 4 and a Xeon. The Pentium 4 refreshed 1.9 times more pixels than the Xeon for the Image Pan trace, which is surprising given the Pentium 4's older architecture, fewer cores and lower clock speed. To determine why and to further characterize the difference between the Pentium 4 and Xeon, an experiment measuring the two platforms' memory hierarchy performance was conducted. The results from this experiment appear in Appendix D, and demonstrate that the Pentium 4 has a sustained read-modify-write memory bandwidth of 3.78 GB/second, which is 1.75 times more than the Xeon's memory bandwidth at 2.16 GB/second. This correlates well with the difference between the number of pixels refreshed by the two different computers.

[21] Each movement is 8 pixels, and each line is 7168 pixels wide.

The memory bandwidth is thus the bottleneck that prevents DVNC from achieving even higher performance when updating pixels that are moved, but not otherwise updated. This is interesting, given that the update operations moments later have to move across the comparatively slow gigabit Ethernet link. However, since the Copy Rect operation is so small (only 12 bytes), the server can send it much faster than it can repeatedly move memory locally.

6.5 Further improvements

There are further improvements and additional evaluation that could be made to the 22 megapixel laptop and De-centralized VNC. The 22 megapixel laptop's peak performance should be further characterized beyond the follow-up experiment that placed its peak performance at 25 and 18 frames per second for sharing a 32 and 16-bit display, respectively.

At present, BlueTooth is used to determine if a display is nearby or not. This only works if the NAD has BlueTooth available. Of the NADs used in the 22 megapixel laptop implementation, only the Nokia N800 could be detected using BlueTooth. BlueTooth has a surprisingly long range, which makes it possible to detect a NAD as "nearby" even if it is placed across the hall through two or three walls. This could be solved by checking the BlueTooth signal strength, however the current implementation does not do this. Other approaches for determining if a display is close to the user or not could also be employed. Some possibilities include inspecting the network latency and apply a threshold to define the display as nearby, determine whether the display and the computer are using the same wireless access point, or use other means of in-door location, such as WiFi triangulation.

The 22 megapixel laptop is currently limited in that the number and resolution of the virtual displays must be configured at compile-time, rather than at runtime[22]. The kernel extension's sysctl-interface already allows user space applications to change both the number and resolution of the displays. However, due to lacking documentation of the necessary steps needed to make these changes visible to the Mac OS X window server, the changes made internally by the kernel extension are not currently detected by the window server. This in turn necessitates the crude "at compile-time" approach currently taken.

The DSD could be extended to support the Copy Rect operation, which would make window movement, scrolling and other tasks that generate the Copy Rect operation faster when used with NADs in a display wall context. This would further open additional issues that are not handled by the DVNC implementation. Two examples are handling pixel exchange between a collection of NADs that each

[22] It could also be configured through a property list, but a reboot would still be required to have the changes take effect.

have different resolutions, and handling the case where more than one NAD covers the same pixel area.

The DVNC system's performance should be compared to other implementations of VNC. DVNC could then be incorporated into the implementation that performs best. The load on the VNC viewers should be further characterized, and the addition of compression to the viewer-to-viewer communication should be considered. The DVNC system's performance should be characterized for additional display wall configurations. Such an evaluation should include larger or smaller display walls as well as sharing the pixels at different bit-depths.

6.6 Conclusion

This chapter has presented the Pixel Space, the NAD model, the 22 megapixel laptop and De-centralized VNC model and implementation. The Pixel Space represents a near infinite collection of pixel resources. Parts of the pixel space surround users every day, in the form of portable devices with displays, netbooks, laptops, workstations, TVs, projectors and for some, display walls. The NAD model and its associated implementation as the 22 megapixel laptop have shown how such nearby pixel resources may be transparently utilized from a user's personal laptop. The 22 megapixel laptop can drive the Tromsø display wall. Its performance has been evaluated, and shown to be very low when the entire wall is being updated at once, with better frame rates possible as the size of the area being updated goes down. Since the 22 megapixel laptop only updates areas within which pixels have changed, it is possible to utilize the NADs represented by the entire Tromsø display wall at once with performance in the 10-25 frames per second range, as long as the areas that are updated are kept small.

DVNC improves the performance of VNC [17, 26] when it is used to create a desktop environment on tiled display walls. This is done by changing the VNC model to allow VNC viewers to exchange pixels amongst each other. This reduces the VNC server's load by eliminating the transfer of redundant pixels to the VNC viewers when the Copy Rect update operation is used. DVNC only gives performance gains for tiled display walls; no performance is gained when a single viewer is used to access the VNC server's framebuffer.

The main bottleneck both for the 22 megapixel laptop and the DVNC implementation is a scarcity of local resources. For the 22 megapixel laptop, the CPU spends most of its time copying data from one location in memory to another. For DVNC, the memory bandwidth becomes the biggest bottleneck, as long as content is only moved. Neither of the two systems are bandwidth-bound in the experiments conducted.

Chapter 7

Discussion

This chapter presents a discussion of the principles formulated in Section 1.4.1, and a discussion of the systems that have been built to explore the interaction and pixel space concepts in the context of the problem statement given in Section 1.3.

7.1 Interaction Spaces

The Interaction Spaces concept is an abstraction for interaction mechanisms. This dissertation has focused on unencumbered and device-free interaction spaces, where user interaction is detected by the environment, converted to input events and then made available to sequential or parallel applications running on a display wall. However, interaction spaces are also created by and available on devices. The use of an iPod touch to control the MASpace application is one example, where the entire display wall's interaction space is "folded up" and made available on the iPod. The user can control the MASpace application on the display wall, but in a setting limited to and enhanced by the capabilities of the iPod's interaction space. Compared to the Camera-sense system, this means no 3D support, but better accuracy and precision. The iPod version of the MASpace application also taps into the pixel space, acting as a mirror or an extension of the display wall, depending on whether it is used to view the same content as is being shown on the display wall, or to examine an independent display of the data.

In Section 1.4.1, three principles are formulated: (i) Orthogonal interaction mechanism; (ii) "where, not what;" and (iii) pixels as network-available resources. The three principles have been formulated based on the concepts, models and systems developed as part of this dissertation. The first two principles are specific to the interaction space concept and are discussed here; the third principle is discussed in Section 7.2.

The first principle makes the ability to detect interaction into a property of the environment, rather than binding it directly to a given computer. The interaction mechanism is realized independently of the computer or computers it acts on. While the underlying motivation for the different interaction space systems is to enable sound- and gesture-based interaction with display walls, the resulting principle is more general. Each interaction space exists orthogonally to the computers they act on. For the display wall, this is necessary since the display wall's parallel architecture requires that any input mechanism is abstracted away from any one computer in the display wall's cluster. Instead, the interaction mechanism must be able to act on any or all of the applications running on the different computers in the display wall's cluster.

When considering the interaction mechanism as orthogonal to the computers they act on, it becomes possible to extend the interaction spaces beyond just interacting with a display wall. For instance, the input mechanisms to one's personal computer could be augmented by the interaction spaces surrounding the computer at any given time. Bring a laptop into an interaction space, and the laptop's existing interaction mechanisms are augmented with the interaction mechanisms provided by the surrounding interaction space. This however raises the issue of generality. The interaction space systems presented in this dissertation are designed for display walls. The accuracy and precision of the Camera-sense system, for instance, is good enough to enable interaction with a number of applications on a display wall, but it may not be good enough in its current state to enable interaction at the level of accuracy required for smaller devices, except perhaps for some coarse-grained gestures. An iPhone, for instance, has an interaction mechanism whose accuracy is on the order of millimeters. A general interaction space covering "everything" may not be possible, due to the often conflicting and contradicting needs that would arise from trying to support the range from very small to very large displays. However, since different interaction spaces can complement each other, custom interaction space systems for smaller devices could be built that would provide the necessary accuracy and precision, or otherwise cater to special requirements.

The interaction space systems presented in this dissertation are limited in scope to detecting that something is present or that something occurs. None of the systems make any attempt at *interpreting* what they detect. The Camera-sense system only detects an object's location and extent in 3D. The Snap-detect system is limited to locating the origin of snap-like sounds, and the Arm-angle system determines the angle of a user's arm (or other straight object). Interpreting events from the three interaction space systems is left to applications, an approach that is based on the end-to-end principle [160]. The applications are best suited to decide how they want to treat the events from applications, and whether combining events from the different systems makes sense or not.

As an example, the Camera-sense system provides applications with 3D object information. For some applications, information about an object's 2D location may

be all that the application needs in order to provide a regular multi-touch interface. Other applications may go beyond this, and attempt more sophisticated gesture recognition based on the full 3D object data delivered by the system. Where a multi-touch interface only needs 2D points, an interface based on gesture recognition defines a vocabulary of higher level gestures based on the entirety of the underlying object data. Such high level gestures can include pointing, waving and specific finger, hand or arm poses. One design possibility is to have a special-purpose application that refines input events from the three systems into new events representing the higher level gestures, which are then in turn sent to other applications. The 3D object data could also be used in other ways, such as to build a simple "scanner" to further augment the functionality of the Wallboard application.

The "where, not what" principle states that it is sufficient to determine *where* an object is, rather than determine *what* the object is when implementing an interaction space system. For the Camera-sense system, the principle is applied in that users can use any object to interact, including their fingers, hands, other body parts or other objects. Importantly, the system identifies that *something* is present, and then determines the location of that something, rather than spending resources trying to determine *what* that something is. This also reduces the complexity of the problem from doing generalized object recognition and identification, to the simpler case of separating foreground from background. The same principle applies for the Snap-detect system. The system does not try to distinguish between a user snapping his fingers, or using a mechanical clicker to make a similar sound, nor does it try to identify which user made the sound. Instead, the system simply determines where the sound occurred and reports that to applications.

No substantial functionality has been lost in the Camera-sense and Snap-detect systems' respective inability to determine what objects are and what the source of a sound was, although there are drawbacks to the approach. For instance, if the Camera-sense system was able to recognize objects, and not only their presence, one could imagine being able to separate different tools from each other – such as a brush, a pencil and an eraser – and have the different tools act in different ways when used to interact with a display wall. While this would be a useful capability, the problem of identifying and recognizing objects is non-trivial. Further, having to use different objects to perform different tasks, or indeed, having to use any object or *device* at all – is one of the requirements that the Camera-sense and associated systems venture to do away with.

The Arm-angle system also follows the "where, not what" principle. While neither its accuracy nor its precision has been measured through experiments, informal experiences using the system indicate that the quality of its output was to a large extent affected by the content being displayed on the display wall. For instance, any straight line appearing on the display wall resulted in false positives from the system. This is problematic – but in retrospect, should have been expected – since most window systems use rectangular windows, producing a large

number of straight lines that the camera used by the Arm-angle system sees. It is possible that this problem could have been avoided by using a more sophisticated approach to determine the angle of the user's arm, such as isolating the user's silhouette. However, when the display wall does not have any content within the camera's field-of-view, the system worked as intended. Thus, the "where, not what" principle still applies, but the implementation could be improved; some possible approaches are outlined in Section 4.9.

The problem statement presents four requirements that a system for gesture-based interaction with a display wall should meet. The systems should be: (i) Always available; (ii) unencumbered; (iii) multi-user; and (iv) room-wide. To explore potential solutions, three interaction space systems were built: (i) The Camera-sense system; (ii) the Snap-detect system; and (iii) the Arm-angle system.

The Camera-sense system meets three of the four requirements. There is no setup associated with the system; users need merely step up to the wall in order to start interacting with the display wall. The system enables unencumbered interaction, and does not require users to carry devices or wear markers. The system can detect gestures from more than one simultaneous user. The system is only available when the user is close to the wall, and thus does not meet the room-wide requirement. However, since the Camera-sense system is scalable, it could be extended to not only cover a wider display wall, but possibly also a larger part of the room. The cameras could be mounted in the ceiling in a grid-like fashion – rather than along just a single line as in the current implementation – giving room-wide coverage. This is a potential direction for future research.

A further benefit of the Camera-sense system's scalable architecture is that different sizes of display walls can be accommodated by adding additional cameras and computers, subject to the performance of the two centralized components of the system, as discussed in Section 4.7. However, the system's scalability comes at the cost of added complexity. With more cameras, the burden of initially aligning and later re-aligning the cameras whenever they lose their alignment grows. The monetary cost also increases as additional cameras or computers are purchased, and in other equipment costs such as network switches and cables. Managing a large cluster of computers is also non-trivial, and requires a person to keep software up-to-date and otherwise maintain the system. Of the three interaction space systems developed, the system that is most "maintenance free" is the Snap-detect system. It is also the system with the simplest architecture: A single computer handles all signal processing and event generation. If that computer is running, the system works; otherwise, it does not. The microphones generally do not move. When people accidentally bump into them, they eventually return to their expected position – suspended as they are from the ceiling.

The Snap-detect system also meets three of the four requirements. Users can use the system immediately by simply snapping their fingers, without requiring any setup in advance: The system is constantly running and ready to detect input. It is

unencumbered, since users need only snap their fingers or clap their hands in order to interact – no external devices are necessary. However, this does not prevent users from using external devices, such as a mechanical clicker, to interact using the system. The Snap-detect system is available room-wide, given that it can detect sounds emanating from anywhere within the same room as the display wall. Since it does not support the detection of several users snapping their fingers simultaneously, it technically does not meet the multi-user requirement. However, the short duration of a snap – less than a second – makes it possible for several users to quickly take turns using the system.

The Arm-angle system meets two of the four requirements. It is always available, although users need to "call" the interaction space by snapping their fingers first. It is unencumbered, since no markers or devices are necessary to detect the angle of a user's arm. Since the current implementation only uses one camera, the system is not multi-user. Additional users could be supported by incorporating additional, steerable cameras into the system. The current implementation also limits the location of the interaction space to an area about a meter or two from the display wall and along the length of the wall, making it fail the room-wide requirement. However, the movable characteristic of the interaction space could make it possible to extend it for use from "anywhere" within the room. This would require that the system be able to compensate for the camera's viewing angle, and also possibly require a different approach to detecting pointing direction. Another alternative would be to use several cameras mounted at different locations in the room to cover different areas.

By using the systems together, the three systems complement each other to meet all the four requirements set out in the problem statement. Users can interact with the display wall from anywhere in the room, without any advance setup, and without carrying devices or wearing markers.

7.2 Pixel Space

The Pixel Space concept is an abstraction that collects pixel resources from a wide range of displays connected to a wide range of platforms. Any display can be considered a part of the pixel space. Pixels in the pixel space are shared across platforms and devices, and can be utilized by computers to display content.

Three principles were formulated in Section 1.4.1. The two interaction space principles are discussed in Section 7.1. The following discusses the third principle: Pixels as network-available resources.

Pixels are available almost everywhere, on displays of different sizes and resolutions. The displays on handheld devices like iPhones or Nokia Internet Tablets represent a pool of pixel resources that could have been utilized if there was a way

of connecting the display to one's computer. A high-resolution display wall also represents a pool of pixel resources. Utilizing these pixel resources from a single laptop is not physically possible with the technology currently on the market, simply due to the large number of displays involved, and the limit of one to two display ports on a single laptop. By making the pixel resources represented by this range of displays available over a network, any computer can gain the ability to utilize them to extend its own display area. This dissertation demonstrates this through the 22 megapixel laptop, however the principle is more general. Given the cross-platform characteristics of pixels, any device could be given the ability to extend its display area by accessing the network-available pixel resources.

In Chapter 2, it is assumed that gigapixel-scale display walls will be driven by more than one computer. The development of the 22 megapixel laptop and DVNC may seem to contradict this assumption, since they both generate all pixels on a single computer and display them using the pixel resources of a display wall – albeit a display wall with lower resolution[1]. Both systems were motivated in part by the need for having traditional desktop applications available on a display wall. The ease with which this can be accomplished by simply sharing the application's output – its pixels – with a display wall, compared to the insurmountable task of rewriting all existing applications to fit the parallel architecture of a display wall, warrants the performance trade-off.

One drawback to systems sharing pixels over a network, is that their performance is reduced as the number of pixels being shared grows. In some cases, the performance reduction may be so big as to make the system unsuited for displaying certain kinds of content. For instance, games, videos, animations or other types of frequently-changing content typically require high framerates. Depending on the resolution, the system may not be able to deliver the framerates necessary to acceptably display the content. While the performance for sharing such content may be good enough[2] when utilizing the pixel resources from a single display, using an entire display wall significantly increases the load on the computer driving the pixels.

There are three activities where bottlenecks may be introduced in or around a pixel sharing system: (i) Generating the pixels; (ii) reading and possibly compressing the pixels; and (iii) sending the pixels across the network to the pixel resources on which they are displayed. Depending on the number of pixels and the rate at which they should be updated, the bottleneck for sharing the pixels moves. Generating a large number of pixels can put a very high load on the CPU. Depending on the number of pixels being generated, the bandwidth between the CPU and main memory may become exhausted, constraining the system's performance. This was the case

[1] Although the 22 megapixel laptop has been used to create a 100 megapixel display, as shown in Figure 6.13(a) in Section 6.3.5.

[2] "Good enough" for dynamic content is usually between 25-30 frames per second, which corresponds to the frame rate of movies and television.

in the DVNC system, where just moving a large number of pixels (without changing them) was enough to exhaust the server's memory bandwidth. Compressing the pixels and sending them across the network are two inter-related factors. Compression puts additional load on the CPU, when the CPU time might instead have been better spent generating the pixels. However, compression can reduce the amount of data that must be sent, sometimes substantially [149, 155]. The choice of whether to use compression or not depends on the network's available bandwidth, and that the receiving end has the performance and capability to decompress the pixels.

The problem statement presents five requirements for enabling utilization of pixel resources. The systems should be: (i) Cross-platform; (ii) transparent; (iii) dynamic; and they should provide: (iv) structure; and (v) performance. To explore potential solutions, two models were devised, and based on the two models, two pixel space systems were implemented: (i) The 22 megapixel laptop; and (ii) Decentralized VNC.

The 22 megapixel laptop is a system built based on the Network Accessible Display model. It meets four of the five requirements. First, it can utilize pixel resources across platforms, as demonstrated by utilizing resources from a handheld Nokia N800 Internet Tablet, a workstation and the Tromsø display wall. Second, the system is transparent. Applications need not be restarted or modified in any way to utilize additional pixel resources as they become available. The pixel resources are represented as Network Accessible Displays. Since the NADs appear to the underlying window system as if they were regular displays directly connected to the computer via a DVI or VGA cable, applications can utilize them in a transparent manner without requiring that applications are restarted or modified in any way. Third, the 22 megapixel laptop is dynamic, since the collection of NADs is allowed to change at any time. The local display configuration can change on-the-fly to accommodate new or disappearing NADs. The laptop can also filter displays based on their physical proximity to the laptop, by scanning for a display's reported BlueTooth MAC address. Fourth, the 22 megapixel laptop is structure-aware, in that it can group related NADs and arrange its local, virtual displays to match the configuration of the NADs. In particular, this is used to support the grid-structure of typical display walls. Finally, the 22 megapixel laptop does not exploit the parallel architecture of the display wall when it utilizes NADs from the display wall, and thus does not meet the performance requirement. While its performance is acceptable when the content being displayed is mostly static or the area covered by the updates is relatively small (on the order of 1-2 0.7 megapixel NADs), dynamic content that covers the entire 22 megapixel Tromsø display wall can only be updated at approximately one frame per second.

The 22 megapixel laptop does not utilize the parallel architecture of the display wall to improve the performance of pixel sharing. To explore the performance requirement, the De-centralized VNC model was developed, and an implementation of it built. The purpose was to improve the performance of VNC when used to

create a desktop environment for display walls, focusing on the performance requirement, and little focus on the remaining four requirements. The experiments have shown that DVNC improves the performance of regular VNC by a factor of up to 11.88 when pixels are moved, but not otherwise updated. It does this by utilizing the network and computational resources made available by the display wall's cluster. The approach taken by the DVNC model and implementation can potentially be integrated into the 22 megapixel laptop implementation to improve its performance too.

Chapter 8

Conclusion

This dissertation presents the concepts of multiple Interaction Spaces and one Pixel Space. The concepts are explored and documented through the development of ideas, models, architectures, designs and implementations, resulting in a number of artifacts. Three interaction space systems, two pixel space systems and several applications that use the systems are built. The systems are evaluated through experiments, and based on the systems, three principles are formulated.

Two of the principles are formulated in relation to the interaction space systems: (i) Orthogonal interaction mechanism; and (ii) "where, not what." The first principle states that the interaction mechanism is a property of the environment. The mechanism is orthogonal to the devices and computers it acts on, and not bound to any given computer or user. The different systems demonstrate this by acting on a high-resolution, wall-sized, tiled display that is driven by multiple computers, and depending on the system, enabling one or several users to interact simultaneously.

The second principle states that determining where something is, rather than what it is, is sufficient to enable the detection of interaction within the different interaction spaces. The three systems all apply this principle, by focusing only on detecting the presence or occurrence of an object or sound, rather than identifying the object or sound.

The three interaction space systems – Camera-sense, Snap-detect and Arm-angle – each create an interaction space that enable gesture- or sound-based interaction with applications running on a display wall. The different systems complement each other, and can be used alone or in concert. Users do not need to wear markers or carry any devices in order to interact using the systems.

The systems are in use with several applications, including an image viewer and an application to visualize and interact with several genomic microarray datasets on a display wall. The Camera-sense system's latency, accuracy and precision have been evaluated. The system has an end-to-end latency of 113.66 ms. The system is

in use with two display walls, covering areas of 6.0x3.0 m, and 2.7x2.0 m, and is able to locate objects with an accuracy of 1.24 cm, and a precision of 0.72 cm. The system's accuracy ranges from an order of magnitude worse [12] to on par [110] with comparable systems from the state of the art. However, the Camera-sense system exceeds the capabilities of these systems by detecting and locating objects in 3D without the need for markers. Its scalability also helps set it apart from a number of existing systems from the state of the art [12, 105, 56, 112], but at the cost of additional complexity in managing, setting up and calibrating the system. The system's accuracy and precision is sufficient to enable interaction with a range of applications on a display wall. The Camera-sense system is a viable way of enabling interaction with display walls.

The third principle is formulated in relation to the two pixel space systems, and states that pixels should be considered network-available resources. The number of pixel resources available to a computer is determined by the environment the computer is in. For instance, a computer near a display wall should be able to utilize the pixel resources offered by the display wall, and not be limited to its local pixel resources.

The two pixel space systems – the 22 megapixel laptop and De-centralized VNC (DVNC) – were constructed to utilize the pixel resources of the Pixel Space. The 22 megapixel laptop was built based on the Network Accessible Display (NAD) model. In the NAD model, a display's pixels are made available on the network by augmenting the display with some networking and computational resources. The 22 megapixel laptop is able to transparently *extend* its display area by utilizing pixels from nearby NADs. Applications running on the laptop need not be modified in any way to utilize the additional pixels provided by the NADs. The system is cross-platform, and can utilize displays ranging from a handheld device to a display wall. The system's performance has been measured through experiments. The 22 megapixel laptop can provide pixels at a peak rate of about 27 megapixels/second. As the number of NADs used increases, the main bottleneck in the system is a scarcity of local resources; in particular, memory bandwidth and CPU.

The DVNC system changes the model with which VNC shares pixels, from a fully centralized model to a de-centralized model. The purpose of the change is to improve the performance of VNC when it is used to share pixels with a display wall. Using VNC, a desktop environment for a display wall is created with a resolution matching the display wall's resolution. However, due to the display wall's parallel architecture, the "Copy Rect" operation used by VNC to update remote VNC viewers becomes less efficient, resulting in increased load on the VNC server. The Copy Rect operation is used by the VNC server when a rectangle of pixels should be moved and the pixels inside the rectangle are not otherwise modified. This operation is generated by any action that results in content being moved, including moving windows, scrolling documents or panning images. When this operation is used on a display wall, the server has to send redundant pixels to the viewers. The

DVNC system resolves this issue by enabling VNC viewers to exchange pixels amongst each other. The result is a system that improves the performance of VNC on tiled display walls by a factor of up to 11.88. The main bottleneck in DVNC is the VNC server's CPU to memory bandwidth.

Chapter 9

Future work

This chapter presents some possibilities for future work and research directions for the systems that have been built based on the Interaction Spaces and Pixel Space concepts.

One avenue of future research is to augment portable devices such as iPhones or laptops with the capabilities of the different interaction spaces they enter. This is possible since the interaction spaces exist orthogonally to the devices which they act on and are not bound to any given computer or device.

Manual alignment and calibration of cameras in the Camera-sense system is an issue in its current implementation. One avenue of further research is to remove the need for manual alignment by making camera calibration automatic. At one extreme, one could "sprinkle" cameras on the floor in a random way, and moments later have an interaction space available. To realize this, each camera would need to determine its own location and orientation in space, which the system could use to build an interaction space covering the areas in which sufficient cameras overlap.

The Camera-sense system could also be extended to cover not only wide walls, but entire rooms, for instance by mounting cameras in the ceiling. Additional cameras could also be mounted along the walls to add additional area coverage or potentially improve accuracy and precision. Another avenue of research is to explore the possibilities of building a system with adaptive accuracy. The system could adapt to the size of the display that one wants to enable interaction with, perhaps by physically moving cameras to cover the areas that need additional accuracy in greater detail.

At present, the applications developed as part of this dissertation make little use of the 3D aspect of the Camera-sense system, except for the Wallview, Wallboard and MASpace applications (discussed in Sections 5.1 and 5.5). The Camera-sense system's 3D capabilities represent many opportunities for further research and development, including ways of utilizing the skeleton 3D object representation created

by the system. One idea is to extend the Wallboard application by scanning objects using the Camera-sense interaction space instead of a regular camera. By turning the object slowly inside the Camera-sense interaction space, a rough 3D profile of the object could be constructed. The resulting 3D profile could be textured using image data gathered by the cameras.

The Snap-detect system makes use of microphones in front of the display wall. These microphones are capable of hearing more than just users snapping their fingers. Other approaches to using these resources could provide avenues for future research. Speech recognition that locates the users talking is one possibility. The Arm-angle system at present does not scale to multiple users, while the Camera-sense system demonstrates that a multi-camera approach is viable. It is possible that the Arm-angle system could be extended by adding more cameras, either to increase the accuracy and precision of the system (for instance using a stereo-camera configuration), or by simply enabling the creation of additional, movable interaction spaces.

Topics like access control, encryption and compression have not been handled in the 22 megapixel laptop, with the exception of the very rudimentary RLE compression implemented by the Display Sharing Daemon. To apply the Network Accessible Display model pervasively, a better way of managing display access rights would be required. One could for instance define some displays to be a part of one's own "private" pixel space, with other displays (such as those on a display wall) contributing pixels to a public pixel space which would be open to anyone.

With shared, public pixel resources, one is also faced with the problem of several users wanting to use the same resources at the same time. The implementation of the NADs utilized by the 22 megapixel laptop currently only supports a single client at a time, since it is unclear how to handle pixels from several clients at once. However, earlier work conducted on window sharing [151, 146] suggests one possible way forward. Windows could be placed on the virtual displays, which would give the windows the benefit of the additional resolution provided by the pixel resources. Then, rather than sharing the entire virtual display, individual windows could be shared with the NADs. In this way, windows from different users could overlap on the shared pixel resources, in the same way that windows overlap in existing window systems.

The bottleneck in DVNC is the memory bandwidth on the server. Two possible approaches to handle this issue would be to either: (i) Investigate whether the server's GPU could be used to accelerate the movement operation locally (at present, any GPU resources are not used by the VNC server at all); or (ii) de-centralize VNC further, by having the VNC server not actually perform the Copy Rect operation locally at all. The pixels are already distributed to the viewers, so the operation could in principle be fully delegated to the viewers. The viewers already do the pixel movement themselves, so any changes to the system may only require changes to the server. There are many challenges with such an approach, however. Since

the VNC server in this approach would become inconsistent with what the viewers display, the server must manage the areas which are not up to date in its local framebuffer. It could then either have local applications redraw such areas, or have the pixels sent from the viewers as necessary. The resulting implementation would lessen the need for the server to move as much memory around, and could increase the system's performance.

References

[1] 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):29–37, 2000.

[2] Thomas A. DeFanti, Jason Leigh, Luc Renambot, Byungil Jeong, Alan Verlo, Lance Long, Maxine Brown, Daniel J. Sandin, Venkatram Vishwanath, Qian Liu, Mason J. Katz, Philip Papadopoulos, Joseph P. Keefe, Gregory R. Hidley, Gregory L. Dawe, Ian Kaufman, Bryan Glogowski, Kai-Uwe Doerr, Rajvikram Singh, Javier Girado, Jurgen P. Schulze, Falko Kuester, and Larry Smarr. The OptIPortal, a Scalable Visualization, Storage, and Computing Interface Device for the OptiPuter. *Future Generation Computer Systems*, 25(2):114–123, 2009.

[3] Doug Ramsey. UC San Diego Unveils World's Highest-Resolution Scientific Display System, July 2008. Press Release. Available from http://www.calit2.net/newsroom/release.php?id=1332 (last visited May 27. 2009).

[4] Grant Wallace, Otto J. Anshus, Peng Bi, Han Chen, Yuqun Chen, Douglas Clark, Perry Cook, Adam Finkelstein, Thomas Funkhouser, Anoop Gupta, Matthew Hibbs, Kai Li, Zhiyan Liu, Rudrajit Samanta, Rahul Sukthankar, and Olga Troyanskaya. Tools and applications for large-scale display walls. *IEEE Computer Graphics and Applications*, 25(4):24–33, 2005.

[5] oblong industries, inc. The g-speak spatial operating system. http://oblong.com/ (last visited March 26. 2009).

[6] Thomas Baudel and Michel Beaudouin-Lafon. Charade: remote control of objects using free-hand gestures. *Communications of the ACM*, 36(7):28–35, 1993.

[7] Johnny Chung Lee. Hacking the nintendo wii remote. *IEEE Pervasive Computing*, 7(3):39–45, 2008.

[8] advanced realtime tracking GmbH. Artrack motion tracking cameras and software. http://www.ar-tracking.de/ (last visited March 26. 2009).

[9] Xiang Cao and Ravin Balakrishnan. VisionWand: Interaction techniques for large displays using a passive wand tracked in 3D. In *UIST '03: Proceedings of the 16th annual ACM symposium on User interface software and technology*, pages 173–182, New York, NY, USA, 2003. ACM.

[10] Alexander Bornik, Reinhard Beichel, Ernst Kruijff, Bernhard Reitinger, and Dieter Schmalstieg. A hybrid user interface for manipulation of volumetric medical data. In *3DUI '06: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 29–36, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Pranav Mistry, Pattie Maes, and L. Chang. WUW - Wear Ur World - A Wearable Gestural Interface. In *CHI '09 extended abstracts on Human factors in computing systems*, 2009.

[12] Jefferson Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118, New York, NY, USA, 2005. ACM Press.

[13] Stevie Bathiche and Andy Wilson. Microsoft surface, 2007. http://www.microsoft.com/surface/ (last visited March 6. 2009).

[14] Giora Yahav, Gabi J. Iddan, and D. Mandelbaum. 3d imaging camera for gaming application. In *ICCE '07: Digest of Technical Papers of the International Conference on Consumer Electronics*, pages 1–2, January 2007.

[15] Kyungdahm Yun and Woontack Woo. Tech-note: Spatial interaction using depth camera for miniature ar. In *3DUI '09: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 119–122, March 2009.

[16] Shahzad Malik and Joe Laszlo. Visual touchpad: a two-handed gestural input device. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 289–296, New York, NY, USA, 2004. ACM.

[17] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[18] Kevin E. Martin, David H. Dawes, and Rickard E. Faith. Distributed Multihead X design, July 2003. http://dmx.sourceforge.net/dmx.html (last visited March 6. 2009).

[19] Microsoft Corporation. Microsoft remote desktop; understanding the remote desktop protocol (rdp). http://support.microsoft.com/kb/186607 (last visited May 5. 2009).

[20] Bartels Media GmbH. MaxiVista. http://www.maxivista.com/ (last visited April 25. 2009).

[21] Daniel Stødle, Olga Troyanskaya, Kai Li, and Otto J. Anshus. Tech-note: Device-Free Interaction Spaces. In *3DUI '09: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 39–42, March 2009.

[22] Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus. Lessons learned using a camera cluster to detect and locate objects. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 71–78. IOS Press, 2008.

[23] Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays. *Journal of Virtual Reality and Broadcasting*, 5(10), November 2008.

[24] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. A system for hybrid vision- and sound-based interaction with distal and proximal targets on wall-sized, high-resolution tiled displays. In *Proceedings of the IEEE International Workshop on Human-Computer Interaction 2007*, volume 4796 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2007.

[25] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. The 22 megapixel laptop. In *EDT '07: Proceedings of the 2007 workshop on Emerging displays technologies*, pages 1–4, New York, NY, USA, 2007. ACM.

[26] RealVNC, Ltd. VNC for Unix 4.0, 2006. http://www.realvnc.com/ (last visited April 29. 2008).

[27] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. De-Centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays. In *NIK '07: Norsk Informatikkonferanse*, pages 53–64. tapir akademisk forlag, November 2007.

[28] Lillian Hoddeson and Michael Riordan. *Crystal Fire: The Invention of the Transistor and the Birth of the Information Age*. W. W. Norton and Company, December 1998.

[29] Peter Lyman and Hal R. Varian. How much information?, 2003. http://www.sims.berkeley.edu/how-much-info-2003 (last visited March 23. 2009).

[30] Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Mike Lang, Scott Pakin, and Jose C. Sancho. Entering the petaflop era: the architecture and performance of roadrunner. In *SC '08: Proceedings of the 2008*

ACM/IEEE conference on Supercomputing, pages 1–11, Piscataway, NJ, USA, 2008. IEEE Press.

[31] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), Apr 1965.

[32] Three Rivers Computer Corporation. Three Rivers PERQ, 1980.

[33] Apple Computer, Inc. Apple cinema hd display, June 2004. A 30-inch display with a resolution of 2560x1600 pixels.

[34] Optoma. Optoma Pico Pocket Projector PK101. http://www.optoma.com/ (last visited March 23. 2009).

[35] Marcus Yam. Sony demonstrates flexible, bendable oled, January 2009. http://www.tomshardware.com/news/Sony-OLED-Flex-bend-flexible,6773.html (last visited May 5. 2009).

[36] LG.Philips LCD. LG.Philips LCD Develops World's First Flexible Color A4-Size E-Paper, May 2007. Press Release.

[37] Duncan Graham-Rowe. Flexible screens get touchy-feely. *MIT Technology Review*, February 2009. http://www.technologyreview.com/computing/22232 (last visited May 4. 2009).

[38] Amazon Inc. Amazon kindle dx, May 2009. The third generation ebook reader manufactured by Amazon.

[39] Marek Czernuszenko, Dave Pape, Daniel Sandin, Tom DeFanti, Gregory L. Dawe, and Maxine D. Brown. The immersadesk and infinity wall projection-based virtual reality displays. *SIGGRAPH Computer Graphics*, 31(2):46–49, 1997.

[40] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-performance dynamic graphics streaming for scalable adaptive graphics environment. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 108, New York, NY, USA, 2006. ACM.

[41] Douglas C. Engelbart and William K. English. A research center for augmenting human intellect. In *AFIPS Conference Proceedings of the 1968 Fall Joint Computer Conference*, pages 395–410, December 1968.

[42] Isaac Asimov. *The Foundation Series*. Gnome Press, 1951.

[43] Arthur C. Clarke. *2001: A space odyssey*. New American Library, June 1968. The novelisation of the movie.

[44] Stanley Kubrick and Arthur C. Clarke. 2001: A space odyssey, April 1968. The movie version of the book.

[45] Steven Spielberg, Philip K. Dick, Scott Frank, and Jon Cohen. *Minority Report*. Twentieth Century-Fox Film Corporation, June 2002.

[46] Joel Garreau. Brain wave of the future. *The Washington Post*, April 2009. Thursday April. 23.

[47] Mattel, Inc. Mind flex, January 2009. Demonstrated at the Consumer Electronics Show (CES) 2009.

[48] Nuance Communications, Inc. Dragon NaturallySpeaking. http://www.nuance.com/naturallyspeaking/ (last visited March 24. 2009).

[49] Apple Computer, Inc. Apple first quarterly earnings report 2009, January 2009. http://www.apple.com/pr/library/2009/01/21results.html (last visited March 24. 2009).

[50] Apple Computer, Inc. The newton message pad, August 1993.

[51] Michael Arrington. Microsoft TouchWall can inexpensively turn any flat surface into a multi-touch display, May 2008. http://www.crunchgear.com/2008/05/14/microsoft-touchwall-can-inexpensively-turn-any-flat-surface-into-a-multi-touch-display/ (last visited May 2. 2009).

[52] Chris Flores. The windows team blog: Microsoft demonstrates multi-touch, May 2008. http://windowsteamblog.com/blogs/windowsvista/archive/2008/05/27/microsoft-demonstrates-multi-touch.aspx (last visited January 27. 2009).

[53] Grant Wallace, Peng Bi, Kai Li, and Otto J. Anshus. A MultiCursor X Window Manager Supporting Control Room Collaboration. Technical Report TR-707-04, Princeton University, Computer Science, July 2004.

[54] Peter Hutterer. MPX: The Multi-Pointer X Server. http://wearables.unisa.edu.au/mpx/ (last visited May 4. 2009).

[55] Christian von Hardenberg and François Bérard. Bare-hand human-computer interaction. In *PUI '01: Proceedings of the 2001 workshop on Perceptive user interfaces*, pages 1–8, New York, NY, USA, 2001. ACM.

[56] W. Matthew Vieta and Matthew Bell. Wavescape: a practical robust display with a 3d gesture interface. In *IPT/EDT '08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologies*, pages 1–2, New York, NY, USA, 2008. ACM.

[57] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. Chromium: a stream-processing framework for interactive rendering on clusters. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702, New York, NY, USA, 2002. ACM.

[58] Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[59] Kai-Uwe Doerr and Falko Kuester. CGLX: A Cross-Platform Cluster Graphics Library. http://vis.ucsd.edu/ cglx/ (last visited March 6. 2009).

[60] Han Chen, Kai Li, and Bin Wei. A Parallel Ultra-High Resolution MPEG-2 Video Decoder for PC Cluster Based Tiled Display Systems. In *IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 30, Washington, DC, USA, 2002. IEEE Computer Society.

[61] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. Hybrid vision- and sound-based interaction on display walls. http://www.youtube.com/watch?v=ga8e91KHNuM (last visited March 26. 2009).

[62] Mads Eriksen. *M - De første årene*. Schibsted Forlag, 2007. A Norwegian comic.

[63] Daniel Stødle. Three years of "M" on a display wall. http://www.youtube.com/watch?v=aJelUGgWKKM (last visited March 26. 2009).

[64] Daniel Stødle, Olga Troyanskaya, Kai Li, and Otto J. Anshus. Device-Free Interaction Spaces. http://www.youtube.com/watch?v=4Cp1FbenilY (last visited March 26. 2009).

[65] technabob. The best way to navigate your comic collection, January 2008. http://technabob.com/blog/2008/01/24/the-best-way-to-navigate-your-comic-collection/ (last visited May 5. 2009).

[66] Peter Glaskowsky. Bright ideas – notes from the emerging display technology conference, August 2007. http://news.cnet.com/8301-13512_3-9755113-23.html (last visited May 5. 2009).

[67] Tor-Magne Stien Hagen, Espen Skjelnes Johnsen, Daniel Stødle, John Markus Bjørndalen, and Otto Anshus. Liberating the desktop. In *ACHI '08: Proceedings of the First International Conference on Advances in Computer-Human Interaction*, pages 89–94, Washington, DC, USA, 2008. IEEE Computer Society.

[68] Bård Fjukstad, John Markus Bjørndalen, and Otto J. Anshus. High resolution numerical models on a display wall. In *7th EMS Annual Meeting and 8th European Conference on Applications of Meteorology (ECAM)*, page 2, 2007.

[69] id Software. Quake 3 arena (open source version). http://ioquake3.org/ (last visited April 2. 2009).

[70] Relic Entertainment. Homeworld (open source version). http://www.homeworldsdl.org/ (last visited April 2. 2009).

[71] Myron W. Krueger, Thomas Gionfriddo, and Katrin Hinrichsen. VIDEOPLACE – An artificial reality. In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 35–40, New York, NY, USA, 1985. ACM.

[72] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *SIGGRAPH '93: Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, New York, NY, USA, 1993. ACM.

[73] Carolina Cruz-Neira, Daniel J. Sandin, Thomas A. DeFanti, Robert V. Kenyon, and John C. Hart. The CAVE: audio visual experience automatic virtual environment. *Communications of the ACM*, 35(6):64–72, 1992.

[74] Hiroshi Ishii and Brygg Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241, New York, NY, USA, 1997. ACM.

[75] Brad Johanson, Armando Fox, and Terry Winograd. The interactive workspaces project: Experiences with ubiquitous computing rooms. *IEEE Pervasive Computing*, 1(2):67–74, 2002.

[76] NVIDIA Corporation. NVIDIA GeForce GTX 295 Technical Specifications, January 2009. http://www.nvidia.com/object/product_geforce_gtx_295_us.html (last visited March 23. 2009).

[77] Visbox, Inc. VisWall Datasheet. http://www.visbox.com/wallMain.html (last visited March 26. 2009).

[78] Glenn Bresnahan, Raymond Gasser, Augustinas Abaravichyus, Erik Brisson, and Michael Walterman. Building a large scale, high-resolution, tiled, rear projected, passive stereo display system based on commodity components. In *Stereoscopic Displays and Virtual Reality Systems X: Proceedings of the SPIE Volume 5006*, pages 19–30, 2003.

[79] George Robertson, Mary Czerwinski, Patrick Baudisch, Brian Meyers, Daniel Robbins, Greg Smith, and Desney Tan. The large-display user experience. *IEEE Computer Graphics and Applications*, 25(4):44–51, 2005.

[80] Yuqun Chen, Douglas W. Clark, Adam Finkelstein, Timothy Housel, and Kai Li. Automatic alignment of high-resolution multi-projector displays using an un-calibrated camera. In *VISUALIZATION '00: Proceedings of*

the IEEE Conference on Visualization, Washington, DC, USA, 2000. IEEE Computer Society.

[81] Grant Wallace, Han Chen, and Kai Li. DeskAlign: Automaticlly Aligning a Tiled Windows Desktop. In *PROCAMS '03: Proceedings of IEEE International Workshop on Projector-Camera Systems*, pages 1–7, October 2003.

[82] Ruigang Yang, David Gotz, Justin Hensley, Herman Towles, and Michael S. Brown. Pixelflex: a reconfigurable multi-projector display system. In *VISUALIZATION '01: Proceedings of the IEEE Conference on Visualization*, pages 167–174, Washington, DC, USA, 2001. IEEE Computer Society.

[83] Manuela Waldner, Christian Pirchheim, and Dieter Schmalstieg. Multi projector displays using a 3D compositing window manager. In *IPT/EDT '08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologies*, pages 1–4, New York, NY, USA, 2008. ACM.

[84] Aditi Majumder and Rick Stevens. Color nonuniformity in projection-based displays: Analysis and solutions. *IEEE Transactions on Visualization and Computer Graphics*, 10(2):177–188, 2004.

[85] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. WireGL: a scalable graphics system for clusters. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 129–140, New York, NY, USA, 2001. ACM.

[86] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

[87] Brian Craig Cumberland, Gavin Carius, and Andrew Muir. Microsoft Windows NT Server 4.0, Terminal Server Edition: Technical Reference. August 1999.

[88] Hussein Abdel-Wahab and Mark A. Feit. XTV: A framework for sharing X Window clients in remote synchronous collaboration. In *TRICOMM '91: Proceedings of the IEEE Conference on Communications Software*, pages 159–167, April 1991.

[89] Ethan Solomita, James Kempf, and Dan Duchamp. XMove: A Pseudoserver for X Window Movement. *The X Resource*, 11(1):143–170, 1994.

[90] Tristan Richardson. The RFB Protocol, version 3.8, February 2009.

[91] Christian Pirchheim, Manuela Waldner, and Dieter Schmalstieg. Deskotheque: Improved spatial awareness in multi-display environments. In *Proceedings of VR 2009: The IEEE Virtual Reality Conference 2009*, pages 123–126, March 2009.

[92] The Rocks Cluster Group. Rocks linux cluster distribution, version 4. http://www.rocksclusters.org/ (last visited April 24. 2009).

[93] Unibrain S.A. Unibrain Fire-i Digital Camera Specifications, 2006. http://www.unibrain.com/Products/VisionImg/tSpec_Fire_i_DC.htm (last visited April 6. 2009).

[94] Network Working Group. RFC792 - Internet Control Message Protocol. September 1981. http://www.ietf.org/rfc/rfc792.txt (last visited April 28. 2009).

[95] Daniel Stødle and Otto J. Anshus. Blurring the line between real and digital: pinning objects to wall-sized displays. In *IPT/EDT '08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologies*, pages 1–5, New York, NY, USA, 2008. ACM.

[96] Meredith Ringel, Henry Berg, Yuhui Jin, and Terry Winograd. Barehands: implement-free interaction with a wall-mounted display. In *CHI '01: CHI '01 extended abstracts on Human factors in computing systems*, pages 367–368, New York, NY, USA, 2001. ACM.

[97] Paul Dietz and Darren Leigh. DiamondTouch: a multi-user touch technology. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 219–226, New York, NY, USA, 2001. ACM Press.

[98] Karen Johanne Kortbek and Kaj Grønbæk. Interactive spatial multimedia for communication of art in the physical museum space. In *MM '08: Proceeding of the 16th ACM international conference on Multimedia*, pages 609–618, New York, NY, USA, 2008. ACM.

[99] Bert Bongers and Gerrit C. Veer. Towards a multimodal interaction space: categorisation and applications. *Personal Ubiquitous Comput.*, 11(8):609–619, 2007.

[100] Thomas Riisgaard Hansen, Eva Eriksson, and Andreas Lykke-Olesen. Mixed interaction space: designing for camera based interaction with mobile devices. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1933–1936, New York, NY, USA, 2005. ACM.

[101] Robert G. Kable. Electrographic apparatus, July 1986. US Patent no. 4600807.

[102] Tyco Electronics. Carrolltouch infrared touchscreens. http://www.elotouch.com/Products/Touchscreens/CarrollTouch/ (last visited March 6. 2009).

[103] SK Lee, William Buxton, and Kenneth C. Smith. A multi-touch three dimensional touch-sensitive tablet. In *CHI '85: Proceedings of the SIGCHI con-*

ference on Human factors in computing systems, pages 21–25, New York, NY, USA, 1985. ACM.

[104] Paul Farhi. CNN hits the wall for the election. *The Washington Post*, February 2008. February 5., 2008. Page C01.

[105] Andrew D. Wilson. Touchlight: an imaging touch screen and display for gesture-based interaction. In *ICMI '04: Proceedings of the 6th international conference on Multimodal interfaces*, pages 69–76, New York, NY, USA, 2004. ACM.

[106] Ankur Agarwal, Shahram Izadi, Manmohan Chandraker, and Andrew Blake. High precision multi-touch sensing on surfaces using overhead cameras. *International Workshop on Horizontal Interactive Human-Computer Systems*, 0:197–200, 2007.

[107] Shahram Izadi, Steve Hodges, Stuart Taylor, Dan Rosenfeld, Nicolas Villar, Alex Butler, and Jonathan Westhues. Going beyond the display: a surface technology with an electronically switchable diffuser. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 269–278, New York, NY, USA, 2008. ACM.

[108] RENCI Vis Group. Duke multi-touch wall development - system, November 2008. http://vis.renci.org/multitouch/?p=138 (last visited May 4. 2009).

[109] The Natural User Interface Group. Touchlib, a multi-touch development kit. http://nuigroup.com/touchlib/ (last visited May 4. 2009).

[110] Joseph A. Paradiso, Kai yuh Hsiao, Joshua Strickon, Joshua Lifton, and Ari Adler. Sensor systems for interactive surfaces. *IBM Systems Journal*, 39(3-4):892–914, 2000.

[111] Ismo Rakkolainen and Karri Palovuori. Laser scanning for the interactive walk-through fogscreen. In *VRST '05: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 224–226, New York, NY, USA, 2005. ACM.

[112] SMART Technologies. SMART board interactive whiteboards. http://www.smarttech.com/ (last visited March 6. 2009).

[113] Gerald D. Morrison. A camera-based input device for large interactive displays. *IEEE Computer Graphics and Applications*, 25(4):52–57, 2005.

[114] Nicolai Marquardt, Ricardo Jota, Saul Greenberg, and Joaquim A. Jorge. The Continuous Interaction Space: Integrating Gestures Above a Surface with Direct Touch. Technical Report 2009-925-04, University of Calgary, April 2009.

[115] Vicon Motion Systems. Vicon motion capture. http://www.vicon.com/ (last visited May 5. 2009).

[116] Nobuyuki Matsushita and Jun Rekimoto. Holowall: designing a finger, hand, body, and object sensitive wall. In *UIST '97: Proceedings of the 10th annual ACM symposium on User interface software and technology*, pages 209–210, New York, NY, USA, 1997. ACM.

[117] Evan Hildreth and Francis Macdougall. Multiple camera control system, June 2006. US Patent no. 7058204.

[118] Heinrich-Hertz-Institut. iPoint Presenter, 2008. http://www.iPoint3D.com/ (last visited April 20. 2009).

[119] Takeo Igarashi and John F. Hughes. Voice as sound: using non-verbal voice input for interactive control. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 155–156, New York, NY, USA, 2001. ACM.

[120] Yoshiyuki Mihara, Etsuya Shibayama, and Shin Takahashi. The migratory cursor: accurate speech-based cursor movement by moving multiple ghost cursors using non-verbal vocalizations. In *Assets '05: Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility*, pages 76–83, New York, NY, USA, 2005. ACM.

[121] James Scott and Boris Dragovic. Audio Location: Accurate Low-Cost Location Sensing. In *Proceedings of the 3rd International Conference on Pervasive Computing, PERVASIVE 2005*, volume 3468 of *Lecture Notes in Computer Science*, pages 1–18. Springer Verlag, May 2005.

[122] Joseph A. Paradiso. Tracking contact and free gesture across large interactive surfaces. *Communications of the ACM*, 46(7):62–69, 2003.

[123] Chris Harrison and Scott E. Hudson. Scratch input: creating large, inexpensive, unpowered and mobile finger input surfaces. In *UIST '08: Proceedings of the 21st annual ACM symposium on User interface software and technology*, pages 205–208, New York, NY, USA, 2008. ACM.

[124] Jean-Marc Valin, Francois Michaud, Jean Rouat, and Dominic Letourneau. Robust sound source localization using a microphone array on a mobile robot. In *Proceedings of the International Conference on Intelligent Robots and Systems (IROS)*, volume 2, pages 1228–1233, October 2003.

[125] Daniel Vogel and Ravin Balakrishnan. Distant freehand pointing and clicking on very large, high resolution displays. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 33–42, New York, NY, USA, 2005. ACM.

[126] Xiaojun Bi, Yuanchun Shi, Xiaojie Chen, and Peifeng Xiang. Facilitating interaction with large displays in smart spaces. In *sOc-EUSAI '05: Proceedings of the 2005 joint conference on Smart objects and ambient intelligence*, pages 105–110, New York, NY, USA, 2005. ACM.

[127] Sarah M. Peck, Chris North, and Doug Bowman. A multiscale interaction technique for large, high-resolution displays. In *3DUI '09: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 31–38, March 2009.

[128] Patrick Baudisch, Edward Cutrell, Dan Robbins, Mary Czerwinski, Peter Tandler, Benjamin Bederson, , and Alex Zierlinger. Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch- and Pen-operated Systems. In *Proceedings of Interact 2003*, pages 57–64, August 2003.

[129] Anastasia Bezerianos and Ravin Balakrishnan. The vacuum: facilitating the manipulation of distant objects. In *CHI '05: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 361–370, New York, NY, USA, 2005. ACM.

[130] Azam Khan, George Fitzmaurice, Don Almeida, Nicolas Burtnyk, and Gordon Kurtenbach. A remote control interface for large displays. In *UIST '04: Proceedings of the 17th annual ACM symposium on User interface software and technology*, pages 127–136, New York, NY, USA, 2004. ACM.

[131] George Robertson, Mary Czerwinski, Patrick Baudisch, Brian Meyers, Daniel Robbins, Greg Smith, and Desney Tan. The large-display user experience. *IEEE Computer Graphics and Applications*, 25(4):44–51, 2005.

[132] IEEE Computer Society. IEEE Standard for a High-Performance Serial Bus. *IEEE Standard 1394-2008*, pages 1–906, Oct 2008.

[133] Massimo Piccardi. Background subtraction techniques: a review. In *2004 IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 3099–3104 vol.4, Oct 2004.

[134] Prasanna Kumar Sahoo, S. Soltani, Andrew K. C. Wong, and Ye C. Chen. A survey of thresholding techniques. *Computer Vision, Graphics, and Image Processing*, 41(2):233–260, 1988.

[135] Damien Douxchamps et. al. libdc1394, an open source library for handling FireWire DC 1394 cameras. http://damien.douxchamps.net/ieee1394/libdc1394/ (last visited March 6. 2009).

[136] Charles H. Knapp and G. Clifford Carter. The generalized correlation method for estimation of time delay. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 24:320–327, August 1976.

[137] Ross Bencina and Phil Burk. PortAudio - an Open Source Cross Platform Audio API. In *Proceedings of the International Computer Music Conference*, pages 263–266, 2001.

[138] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[139] Richard O. Duda and Peter E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[140] Deb Agarwal. devserv - open source camera control software, 2005. http://acs.lbl.gov/OldMisc/mbone/devserv/ (last visited April 29. 2009).

[141] I. Scott MacKenzie and Colin Ware. Lag as a determinant of human performance in interactive systems. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 488–493, New York, NY, USA, 1993. ACM Press.

[142] Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays. In *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007*, pages 75–83, June 2007.

[143] Matthew M. Hibbs, Grant Wallace, Maitreya Dunham, Kai Li, and Olga Troyanskaya. Viewing the larger context of genomic data through horizontal integration. *IV '07: Proceedings of the IEEE International Conference on Information Visualization*, pages 326–334, July 2007.

[144] SDL: Simple Directmedia Layer. A cross-platform, open-source library for "low-level access to a video framebuffer, audio output, mouse, keyboard, and joysticks across a wide variety of operating systems." http://www.libsdl.org/ (last visited March 6. 2009).

[145] Espen Skjelnes Johnsen, John Markus Bjørndalen, Tore Larsen, and Otto J. Anshus. Simplifying Applications Use of Wall-Sized Tiled Displays. In *NIK '08: Norsk Informatikkonferanse*, 2008.

[146] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. Support for collaboration, visualization and monitoring of parallel applications using shared windows. In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 228–238. Springer, 2006.

[147] Scott Campbell. Survey: Notebooks surpass desktop sales for first time. *CRN*, August 2005. http://www.crn.com/white-box/169400139 (last visited March 6. 2009).

[148] Tony Lin, Peng-Wei Hao, Chao Xu, and Ju-Fu Feng. Hybrid image coding for real-time computer screen video transmission. *Visual Communications and Image Processing 2004*, 5308-1:946–957, January 2004.

[149] Lars Ailo Bongo, Grant Wallace, Tore Larsen, Kai Li, and Olga Troyanskaya. Systems support for remote visualization of genomics applications over wide area networks. In *GCCB '06: Proceedings of the International Workshop on Distributed, High-Performance and Grid Computing in Computational Biology*, volume 4360 of *Lecture Notes in Computer Science*, pages 157–174. Springer, March 2007.

[150] Grant Wallace and Kai Li. Virtually shared displays and user input devices. In *ATC'07: Proceedings of the 2007 USENIX Annual Technical Conference*, pages 1–6, Berkeley, CA, USA, 2007. USENIX Association.

[151] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. Collaborative sharing of windows between Mac OS X, the X Window System and Windows. In *NIK '04: Norsk Informatikkonferanse*, 2004.

[152] Tim Dreyer. NEC display solutions announces first Windows Vista-compatible projectors and Silicon Optix HQV Processing, June 2007. Press Release. Available from http://www.necus.com/companies/17/NECDisplaySolutions_Announces_First.pdf (last visited May 5. 2009).

[153] Aequitas Technologies. iGala digital photo frame. http://www.i-gala.com/ (last visited May 5. 2009).

[154] Brian K. Schmidt, Monica S. Lam, and J. Duane Northcutt. The interactive performance of slim: a stateless, thin-client architecture. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 32–47, New York, NY, USA, 1999. ACM.

[155] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2005. ACM Press.

[156] The VirtualGL Project. A Study of the Performance of VirtualGL 2.1 and TurboVNC 0.4, 2008. http://www.virtualgl.org (last visited April 29. 2008).

[157] Apple Computer, Inc. *Cocoa Fundamentals Guide*. Apple Computer, Inc., November 2008.

[158] Apple Computer, Inc. *I/O Kit Fundamentals - Hardware and drivers*. Apple Computer, Inc., May 2007.

[159] Video Electronics Standards Association. VESA Enhanced Extended Display Identification Data – Implementation Guide, June 2001. Version 1.0.

[160] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[161] Brad Johanson and Armando Fox. The event heap: A coordination infrastructure for interactive workspaces. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 83, Washington, DC, USA, 2002. IEEE Computer Society.

[162] Greg Eisenhauer, Fabian E. Bustamante, and Karsten Schwan. Event services for high performance computing. In *Proceedings of the Ninth International Symposium on High-Performance Distributed Computing*, pages 113–120, 2000.

[163] Apple Computer, Inc. *Bonjour Overview*. Apple Computer, Inc., May 2006.

[164] Daniel Steinberg and Stuart Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005.

[165] Philip J. Mucci, Kevin S. London, and John Thurman. The cachebench report. Technical report, University of Tennessee, Computer Science, November 1998. Technical report: Available from http://icl.cs.utk.edu/projects/llcbench/index.html (last visited April 16. 2008).

Appendix A

Papers

The papers included in this appendix have all been peer-reviewed prior to publication in their respective venues. They all appear exactly as they were in their final, published state. No modifications have been made. References within the papers refer to the references listed at the end of each paper, and not to the main references of the dissertation.

The papers are listed in chronological order. In cases where a paper has been presented at a conference and later accepted in revised form for a journal or proceedings, the revised paper is included with a reference to the original publication.

A.1 Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays

Citation

Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays. In *Proceedings of the 4th International Symposium on Pervasive Gaming Applications, PerGames 2007*, pages 75–83, June 2007.

Revised:

Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays. *Journal of Virtual Reality and Broadcasting*, 5(10), November 2008.

Abstract

Having to carry input devices can be inconvenient when interacting with wall-sized, high-resolution tiled displays. Such displays are typically driven by a cluster of computers. Running existing games on a cluster is non-trivial, and the performance attained using software solutions like Chromium is not good enough.

This paper presents a touch-free, multi-user, human-computer interface for wall-sized displays that enables completely device-free interaction. The interface is built using 16 cameras and a cluster of computers, and is integrated with the games Quake 3 Arena (Q3A) and Homeworld. The two games were parallelized using two different approaches in order to run on a 7x4 tile, 21 megapixel display wall with good performance.

The touch-free interface enables interaction with a latency of 116 ms, where 81 ms are due to the camera hardware. The rendering performance of the games is compared to their sequential counterparts running on the display wall using Chromium. Parallel Q3A's framerate is an order of magnitude higher compared to using Chromium. The parallel version of Homeworld performed on par with the sequential, which did not run at all using Chromium. Informal use of the touch-free interface indicates that it works better for controlling Q3A than Homeworld.

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays

Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, Otto J. Anshus
Dept. of Computer Science
University of Tromsø
N-9037 Tromsø, Norway
phone, e-mail: +47 77 64 42 22
{daniels, tormsh, jmb, otto}@cs.uit.no

Abstract

Having to carry input devices can be inconvenient when interacting with wall-sized, high-resolution tiled displays. Such displays are typically driven by a cluster of computers. Running existing games on a cluster is non-trivial, and the performance attained using software solutions like Chromium is not good enough.

This paper presents a touch-free, multi-user, human-computer interface for wall-sized displays that enables completely device-free interaction. The interface is built using 16 cameras and a cluster of computers, and is integrated with the games Quake 3 Arena (Q3A) and Homeworld. The two games were parallelized using two different approaches in order to run on a 7x4 tile, 21 megapixel display wall with good performance.

The touch-free interface enables interaction with a latency of 116 ms, where 81 ms are due to the camera hardware. The rendering performance of the games is compared to their sequential counterparts running on the display wall using Chromium. Parallel Q3A's framerate is an order of magnitude higher compared to using Chromium. The parallel version of Homeworld performed on par with the sequential, which did not run at all using Chromium. Informal use of the touch-free interface indicates that it works better for controlling Q3A than Homeworld.

First presented at the 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007. Revised for JVRB.

Keywords: Display wall, multi-touch, device-free, parallelized games

1 Introduction

Wall-sized, high-resolution tiled displays are becoming increasingly common in locations ranging from visualization labs to public spaces. Often, having to carry input devices around in order to interact with applications running on a display wall can be inconvenient. Devices like mice or Nintendo Wiimotes are easily misplaced, and for public installations there is the risk of theft. Asking users to wear optical or electronic markers raises the bar for casual users. Instead, a completely device-free approach to interacting with wall-sized displays is necessary.

Display walls provide high resolution by tiling a set of independent displays in a grid. Each display is usually driven by a computer in a display cluster[LCC+00]. The resolution of a typical desktop display is about 2-3 megapixels, while the resolution of a display wall ranges from 10 to 100 megapixels [LCC+00, SW06] and beyond. The display wall used in this paper is comprised of 28 projectors, each driven by one computer and arranged in a 7x4 grid, for a total resolution of 7168x3072 pixels.

For games, high framerates are important [CCD06]. Maintaining high framerates becomes increasingly difficult as the resolution goes up. Further, the cluster-based architecture of display walls makes running ex-

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

isting games difficult, as very few, if any, games are written to run on a cluster of computers or use more than a few displays. Chromium [HHN+02] can be used to make the display wall appear as a single display to OpenGL-based applications, but at the cost of sub-optimal performance. In addition, not all software works with Chromium.

This paper presents a device- and touch-free multi-user human-computer interface for display walls. Users standing in front of such a display wall can interact with applications directly using hand- and arm-gestures without the need for any devices and without having to wear markers of any kind. The interface has been integrated with two commercial, but now open-source games[1]: Quake 3 Arena (Q3A) [iS08] and Homeworld [Ent08], respectively a first-person shooter (FPS) and a real-time strategy (RTS) game. Games were chosen because they generally require low-latency input to be playable. If the touch-free interface does not provide sufficiently good accuracy or low latency, the games will become unplayable. The two games were parallelized in order to run on the display wall with good performance, and modified to accept position data from the touch-free interface. Figure 1 shows two persons playing Q3A against each other on a display wall. The person in the middle is playing Homeworld.

Figure 1: Two persons playing Q3A and one person playing Homeworld simultaneously on a 7x4 tile display wall. Q3A runs on 2x2 tiles to the left and right, and Homeworld on 3x3 tiles in the middle.

The interface uses 16 cameras and 9 computers to detect objects in front of the display wall, and is able to detect multiple objects simultaneously at a rate of 30 Hz. When three or more cameras see the same object, triangulation can be used to determine the object's position. The interface is referred to as touch-free, as users can interact with the display wall without actually touching its canvas. This is an important advantage over existing solutions that require touch to work [Han05], as the canvas used for our display wall is flexible and thus prone to perturbation when users touch it. The interface's main advantage over other approaches, like the IS-900 tracking system [Int08], is that it is completely device-free. Users need not wear markers to accommodate the interface, but can instead walk directly up to the display wall and start interacting. This is particularly important for public installations, where markers or other input devices might easily get lost or stolen. Even in a lab setting it is easy to misplace input devices, or confuse the different input devices with each other ("Which mouse/Wiimote is the correct one? Where did I leave it?").

Two different approaches were used when parallelizing Q3A and Homeworld. For both Q3A and Homeworld, a copy of the game runs on each tile. Each copy's OpenGL view frustum is modified in accordance with the tile it runs on to create a coherent, multi-tile view. For Q3A, the existing client-server based architecture combined with the concept of spectators was exploited. The server keeps all the clients in sync, and the spectator-concept enables different clients to be configured so as to constantly follow a given player. For Homeworld, a state-synchronizing, master-slave approach was taken. Each copy shares a global clock and random number generator seed. The master distributes all input to the different slaves, with the purpose of having all copies compute the exact same game state for each new frame.

Experiments were conducted to measure the latency of the touch-free interface, as well as the framerate of the two games. The experiments show that the time before an object's position is available to the games averages 116.7 ms, with the majority of this latency incurred by the FireWire-based cameras. Game-side gesture-processing did not incur significant latencies, due to the simple gestures involved. For Q3A, the framerate is shown to be as much as an order of magnitude better than using Chromium. Homeworld's framerate remains high in the parallel configuration, and outperforms the single-display configuration when running on both 2x2 and 3x3 tiles. Homeworld did not work with Chromium at all.

The main contributions of this paper are (i) a distributed, device- and touch-free multi-user interface,

[1] Only the game engines are open source. The data files still require a license.

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

(ii) two approaches to parallelizing games for a display wall environment, demonstrating how different aspects of the two games' existing architectures can be exploited, (iii) a prototype system for gesture-based input to games in the FPS and RTS genres, (iv) an evaluation of the interface's responsiveness when used to interact with two games, and (v) evaluation of three different approaches for making existing games run on display walls.

2 Related Work

The Quake-series of games have been popular targets for modification and extension, both in terms of input devices and display surfaces. Some examples include playing Quake using Nintendo's Wiimote, using eye-tracking to play Quake, or controlling Quake from a PDA[2]. CaveQuake is a limited re-implementation of Quake II and Q3A for use in a CAVE[3], but does not support all the features of the full games, and for the Q3A case does not even support playing. In [KLJ04], the authors present a gesture-based interface to Quake 2. The interface is limited in that only one person may use it at a time, and differs from the touch-free interface presented in this paper by the use of whole upper-body gestures. The touch-free interface only enables hand- and arm-gestures. We are not aware of any work to integrate new input devices or new display surfaces for Homeworld.

In [BBH05], a gaming interface based on a commercially available stool, "The Swopper," is presented. The stool and a light gun is used to produce joystick input events to control an FPS game. By shifting the body weight and rotating on the stool in combination with aiming and firing the gun, the user can navigate and interact with the world. The touch-free interface does not require the use of any external devices, and the large display wall makes it possible to have multiple players playing side-by-side simultaneously. The stool-and-light-gun approach is more expressive compared to the gestures recognized by the touch-free interface.

Gesture VR [SK98] is a video-based, hand-gesture recognition system. The system recognizes three gestures which are used to provide applications with different input events, as demonstrated by controlling Doom, an FPS game developed by id Software. Their solution is centralized, using two synchronized cameras connected to a single computer. The touch-free interface comprises 16 cameras connected to 8 computers, enabling it to cover a larger area at the cost of a more elaborate hardware setup. The touch-free interface only recognizes simple gestures (2D position and radius of detected objects), while Gesture VR allows for detection of 3D position and three different gestures.

In [TGSF06], the authors argue that a digital table is a conductive form factor for general co-located home gaming. By combining speech and hand gestures as input to two commercial games, The Sims and Warcraft III, several persons can interact with the games running on the tabletop. The touch-free interface is based on hand- and arm-gestures alone on wall-sized displays. The physical dimensions of the display wall enables more than a couple of people to play simultaneously, against each other or co-operatively. Further, we have modified the source of the two games, enabling more flexible multi-point interaction. The games used in [TGSF06] are not open source, requiring that custom wrappers are built that translate touch- and speech input to mouse and keyboard events. In [SZP+00], the authors demonstrate a bimodal speech- and gesture-based interface for interacting with a 3D-visualization. Apart from the speech-aspect, this system differs from the touch-free interface in that it supports only one user at a time and has a far more limited area in which interaction can take place.

The authors of [TGSF06] use the Diamondtouch [DL01] tabletop for multi-touch interaction. Other technologies for multi-touch interaction include [Han05], where infrared light is projected into a canvas and internally reflected. The internal reflection of the light is frustrated at points where the user touches the canvas. The escaping light can be detected using a camera mounted behind the canvas. The touch-free interface is based on detecting the presence of objects directly, and does not require the user to actually touch the display wall's canvas. In [Mor05], the author presents a camera-based solution to detecting and positioning objects in front of a whiteboard called the "SmartBoard." The approach is similar to the touch-free interface, except that the touch-free interface utilizes a distributed approach with 16 commodity FireWire cameras connected to a set of computers, whereas the SmartBoard uses custom cameras with on-chip processing to perform object recognition.

[2] http://www.youtube.com/watch?v=n1tsXc2RoeM
http://www.youtube.com/watch?v=3pRWYE2LRhk
http://www.youtube.com/watch?v=tNJXjNBgmLs

[3] http://www.visbox.com/cq3a/

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

Chromium [HHN+02] is a system for distributing streams of rendering commands, allowing many existing OpenGL applications to run on tiled display walls without modifications. Chromium works by conceptually making the individual tiles of a display wall appear as a single, logical display to the application. By making applications use Chromium's OpenGL library, Chromium can intercept rendering commands and forward them to remote rendering nodes. Homeworld did not run using Chromium, and Chromium's rendering performance running Q3A did not scale well beyond 2x2 tiles.

In CaveUT [JH02], a set of modifications to Unreal Tournament is presented that allows it to display in panoramic theaters. The same principle of using spectators to support multi-tile rendering is applied as employed by the parallel version of Q3A. However, no measurements of the resulting performance are presented. This paper presents measurements of the Q3A's framerate and documents the latency incurred by using spectators.

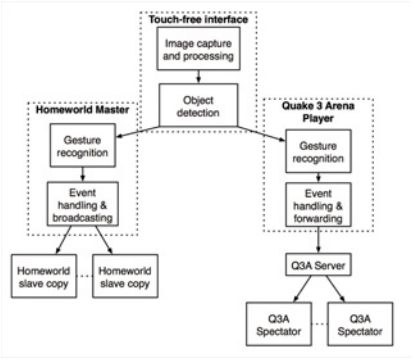


Figure 2: The design of the touch-free interface, and its use with Q3A and Homeworld.

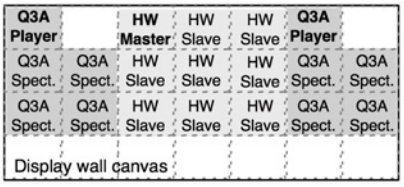


Figure 3: Running Q3A and Homeworld on a 7x4 display wall. To the left and right, two Q3A players control a set of Q3A spectators. In the middle, a single Homeworld master synchronizes the rendering and game simulations of 8 Homeworld slave copies.

3 Design

Quake 3 Arena [iS08], developed by id Software, is an open-source first-person shooter designed for multiplayer gaming. It is based on a client-server architecture where the server maintains the state of the game. At a fixed rate, independent of the connected clients, the server updates its game state, before broadcasting state changes to connected clients. Clients use this to update their view of the game. A client in Q3A is either a player or a spectator. A player is a client that participates in the game. A spectator is a client that instead of participating, follows one of the players around and displays that player's view of the game.

Homeworld [Ent08] is a 3D real-time strategy game developed by Relic Entertainment. In September 2003, the Homeworld engine was made open source. Although the Linux version still lacks some of the features of the complete game, including software rendering, cut-scene playback and networked multiplayer support, the game itself is fully playable in single-player mode. In contrast to Q3A, Homeworld has a monolithic design, with all code running inside a single process.

Figure 2 shows the overall design of the touch-free interface, and its use with Q3A and Homeworld. Images are captured and then analyzed to locate objects in a plane parallel to the display wall's canvas. The positions of these objects are then processed by an object detector that yields the object's 2D position and radius, before the resulting information is sent to the two games. The two games process the data individually, using object positions and radii to detect gestures and handle them in game-specific ways.

The design of the parallelized Q3A uses a modified player that receives input from the touch-free interface. The player uses the positions received to recognize gestures, and converts them to keyboard and mouse events suitable for the game. The player relays its actions to the Q3A server, which then updates all clients with the new game state. This causes the spectators following a given player to update their view.

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

(a) Using a "vertical" hand to control the player's aim.

(b) Controlling aim and firing by making the hand flat.

(c) Moving and aiming simultaneously using both hands.

Figure 4: Gestures for controlling Q3A .

For Homeworld, a single copy is elected as a master. The master becomes responsible for accepting and interpreting input from the touch-free interface. After recognizing gestures, the resulting input is handled and broadcast to the slave copies. Figure 3 shows one configuration of a 7x4 tiled display wall where two users can play Q3A against each other, while a third user simultaneously plays Homeworld. This configuration is identical to the one pictured in Figure 1. Several other configurations are also possible.

3.1 Hand- and arm-gestures

When playing an FPS using a mouse and keyboard, the mouse is used to aim and fire, and the keyboard is used for movement. In addition, the mouse's scroll wheel is often used to switch weapons, and the keyboard to control other actions the player can take (ducking, jumping, etc.). The following gestures, summarized in Table 1, were used for controlling Q3A. When only one hand is detected by the input system, its position is used for controlling the player's aim. When the hand is tilted (making it flat), it will additionally fire the player's weapon. When two hands are detected, the right hand controls aim and firing, and the left hand is used to move the player forwards or backwards. Figure 4 illustrates the gestures.

Action	Gesture
Aim	Move right/only hand
Fire weapon	Flat right/only hand
Move forward	Vertical left hand
Move backward	Flat left hand

Table 1: The gestures in Q3A that the game recognizes and maps to actions.

Homeworld uses a different control scheme. When using a keyboard and mouse, the main controls can all be accessed with the mouse, and the keyboard is mostly used for shortcuts for different menu selections and buttons. When no mouse buttons are pressed, the mouse simply controls an on-screen cursor. Holding down different mouse buttons, the user can pan and zoom the camera, as well as select entities and manipulate them from a contextual menu.

Action	Gesture
Control cursor position	Move right/only hand
Select/click entities	Flat right/only hand
Pan view/contextual menu	Flat left hand
Toggle tactical view	Vertical left hand
Zoom	Flat left and right hand, distance between hands control zoom factor

Table 2: Actions in Homeworld and their corresponding gestures.

Table 2 lists the different actions in Homeworld, and their mapping to gestures. The cursor is controlled using a one-to-one mapping from hand location to the display wall. When the right/only hand is flat (like the fire-gesture in Q3A), the user can select or click items. The user can enter or leave Homeworld's tactical view using a vertical left hand. With a flat left hand, the user can either invoke Homeworld's contextual menu (for moving ships, creating formations, and so on), or panning the camera (by simultaneously moving the right hand). Finally, the user can zoom the camera in and out using a flat left and right hand, varying the distance between them to control the amount of zoom.

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

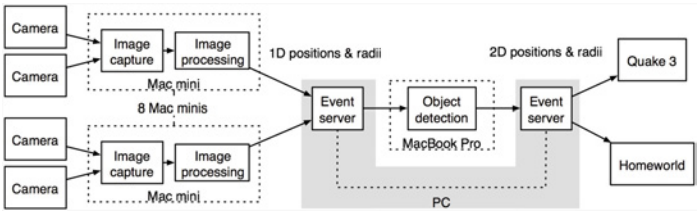


Figure 5: The architecture of the touch-free interface.

4 Implementation

Figure 5 shows the architecture of the touch-free interface. The interface makes use of 16 FireWire cameras, connected in pairs to 8 Mac minis. The cameras are mounted along the floor, enabling the detection of objects in a plane parallel to the display wall's canvas. The cameras have a 42-degree field-of-view. Images are captured at 30 FPS with a resolution of 640x480 pixels in 8-bit grayscale. Each image is processed by subtracting the background, removing noise and thresholding the result to identify objects (which are typically hands or arms). This yields zero or more pairs of 1D position and radius.

Each Mac mini sends its identified positions and radii via an event server to a MacBook Pro that determines the position of each object in 2D space using triangulation (Figure 6). The resulting 2D positions and radii are sent via the event server to either Homeworld or Q3A. The event server's role is to distribute events of different kinds to software used with the display wall. The software for capturing images, detecting and positioning objects was implemented for Mac OS X in Objective-C and C, using libdc1394[4] to communicate with the FireWire cameras; more details on the design and implementation appear in [SHBA08].

Q3A and Homeworld were modified to receive object position events from the touch-free interface, and then interpret them according to the gestures outlined in the previous section. When a gesture is recognized, events corresponding to the action associated with the gesture is injected into the game's input event stream. Depending on the relative amount of movement detected, mouse events can be generated, and the object's radius is used to determine whether it is interpreted as a flat hand or a vertical hand.

[4] http://libdc1394.sourceforge.net/

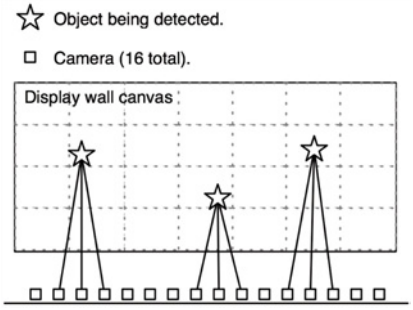


Figure 6: 16 cameras positioned below the display wall's canvas are used to triangulate the position of different objects.

4.1 Parallelizing Q3A and Homeworld

Running Q3A and Homeworld on a tiled display wall requires that each tile displays a part of the total view for each game. To achieve this, the view frustum used by OpenGL for both Q3A and Homeworld must be modified in relation to the tile on which the game runs.

The parallel version of Q3A is controlled by configuring a set of environment variables, and then reading them from within the game. The variables control how the view frustum is configured, as well as whether or not a client is designated as a player or a spectator, and which player a given spectator follows. Due to the client-server architecture of Q3A, this is sufficient to create a parallel version that will run on the display wall. Figure 7 shows a player in the upper-left corner, with four spectators following that player, as it would appear on a tiled display wall.

Homeworld was parallelized by running several

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

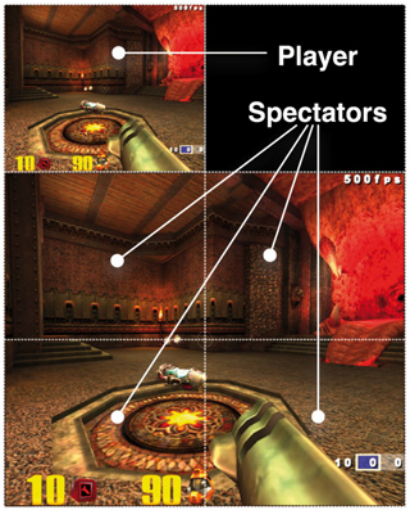


Figure 7: Example Q3A configuration on a display wall. The upper left corner shows the player, while the remaining four clients are spectators following that player, with modified view frustums to match the tiles on which they run.

tightly coupled copies and manually ensuring state consistency between them. Each copy runs on one tile, and the Message Passing Interface (MPI) [BDV94] is used to exchange state information and keep the copies synchronized. One copy is elected as master, and the remaining copies become slaves. For each frame, the master accepts input from the touch-free interface and broadcasts it to the slaves. Before starting a new frame, all the copies synchronize at a barrier. This ensures that each slave receives the same input during the same simulation step in the game, and synchronizes the visual display. To ensure that each copy's game simulation proceeds identically on all nodes, the same value is used to seed each copy's pseudo-random number generator. Finally, a global clock is shared by all the copies and controlled by the master.

5 Experiments

Three experiments were conducted. The first experiment was performed to determine the latency involved in using the touch-free interface, and determine if it is sufficiently low to play games. The next two experiments measured the rendering performance of the two games. For Q3A, the results are compared to Q3A running on the display wall using Chromium; for Homeworld, the results are compared to running Homeworld on a single display.

The hardware used was (i) a display cluster with 28 nodes (Intel Pentium 4 EM64T, 3.2 GHz, 2 GB RAM, HyperThreading enabled, NVIDIA Quadro FX 3400 with 256 MB Video RAM, running the Rocks cluster distribution 4.0) connected to 28 projectors (1024x768, arranged in a 7x4 matrix), (ii) switched, Gigabit Ethernet, (iii) 8 Mac minis (1.66 GHz Intel Core Duo, 512 MB RAM, Mac OS X 10.4.9), (iv) 16 Unibrain Fire-i FireWire cameras, (v) a MacBook Pro (2.33 GHz Intel Core 2 Duo, 3 GB RAM, Mac OS X 10.4.9). Each Mac mini was connected to two cameras. The MacBook Pro was used to run the object detection software.

5.1 Latency Measurements

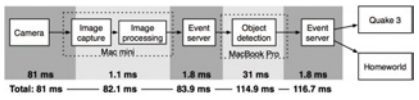
Referring to Figure 5, there are five areas where significant latency may be introduced: (1) The time taken from the camera captures an image, until the image is available to a Mac mini for processing, (2) the time taken by the Mac mini to process the image, (3) the time taken to transfer processed data over the network to the MacBook Pro, (4) time taken by the MacBook Pro to detect objects using information gathered from all the Mac minis, and (5) the time taken to distribute the resulting object positions to the two games.

For Q3A, there is one additional, latency-inducing step. This step is the time from a gesture is recognized, until the action caused by the gesture is shown by the spectators. This latency is caused by the required round-trip from a Q3A player via Q3A's server to the spectators.

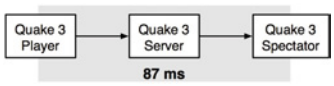
5.1.1 Methodology

The camera-induced latency (1) is measured by pointing a camera at the screen attached to a computer capturing images from the camera. The computer's screen is initially black, before it is turned white. At this point, a timer starts. The timer stops when the images

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10



(a) The latency from when the cameras grab images, until positions of objects are available for processing by either Q3A or Homeworld. Each measurement represents an average measure of the latency.



(b) The additional latency as input events are delivered to a Q3A player, sent to the server and finally made visible by the spectators.

Figure 8: Latency measurements for (a) the touch-free interface and (b) Quake 3 Arena.

captured by the camera show a white screen, with the resulting latency being the elapsed time since the timer was started.

The processing-sensitive latencies (2 and 4) are measured by measuring typical execution times for the code that respectively performs image processing and object detection. The network latencies (3 and 5) are determined by measuring the time taken to send a message from one computer via an event server to the target, and receiving a reply.

To avoid modifying Q3A's server, the added latency in Q3A is determined as follows. When the player fires his weapon, the Q3A engine will cause a weapon-fire sound to be played. The client-side sound-playing code was modified to start a timer when that sound is played. Each spectator reports back to the player when it plays a weapon-fire sound, yielding an estimate of the latency from when something happens at the controlling player, until it is visible to the spectators.

5.1.2 Results

The results from the latency measurements are summarized in Figure 8(a). The additional latency introduced through Q3A's client-server architecture is shown in Figure 8(b). The average latency before an object's position is available to either game is 116.7 ms. The camera-induced latency is the greatest contributor, at 81 ms. Object detection requires 31 ms. For Q3A, the added latency averaged 87 ms with a standard deviation of 59 ms over 1287 samples gathered from 9 spectators.

5.2 Rendering Performance

The metric used to measure the performance of Q3A and Homeworld is frames per second. For both Q3A and Homeworld, input events are recorded over a period of about 30 seconds. The game is started in a known state, and the recorded input events are played back[5]. During playback, the framerate is logged continuously.

5.2.1 Methodology

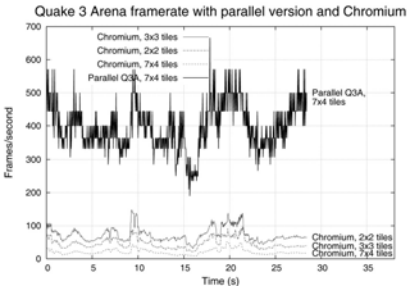
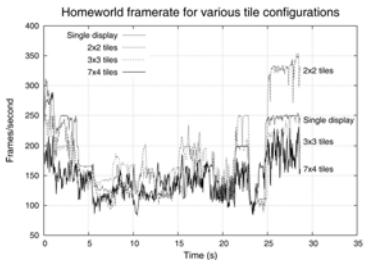


Figure 9: The framerate when running Q3A on 2x2, 3x3 and 7x4 tiles using Chromium, compared to the parallel version's framerate running on 7x4 tiles.

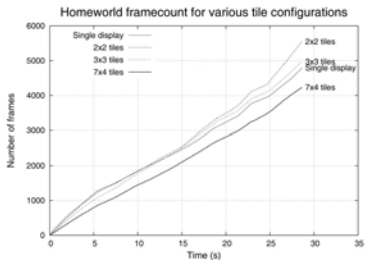
The performance of both Homeworld and Q3A was measured for four different configurations, with 1, 4, 9 and 28 rendering nodes. For Q3A, the framerate was limited to 500, and the performance measured both when using Chromium to distribute the rendering, and when running the parallel version. The Q3A server ran locally on the same network. For Homeworld, which did not work with Chromium, the parallel version's framerate was measured, and compared to running Homeworld on a single display.

[5] This is similar to measuring Quake performance by running a timedemo. The timedemo mechanism already in Quake does not work for the parallel version, as it is designed to run on a single computer only.

urn:nbn:de:0009-6-15001, ISSN 1860-2037



(a) The framerate when running Homeworld on a single display, compared to running it on 2x2, 3x3 and 7x4 tiles.



(b) The total number of frames drawn when running Homeworld on a single display, compared to 2x2, 3x3 and 7x4 tiles.

Figure 10: Homeworld performance measurements.

5.2.2 Results

Figure 9 shows the results from measuring Q3A's rendering performance. The peak performance with Chromium on 4 rendering nodes (2x2 tiles) is 148 FPS, and the average at 73. For 3x3 tiles, the peak FPS is 97 and the average is 47, and for all 7x4 tiles, the peak is 51 and the average 21 FPS. The figure only lists the results from the parallel version running on all 28 tiles, as there were no significant difference in performance when varying the number of rendering nodes for the parallel version. The maximum framerate for the parallel version was 666, and the average framerate was 398.

Figure 10(a) shows the results from measuring Homeworld's framerate, while Figure 10(b) shows the cumulative number of frames drawn by the game during the experiment. The framerate varies much more compared to the Q3A measurements. The maximum framerate for Homeworld running on a single tile, 2x2, 3x3 and 7x4 tiles were respectively 311, 353, 250 and 231. The respective average framerates were 168, 183, 169 and 143. Figure 10(b) shows that running Homeworld on both 2x2 and 3x3 tiles performs better than running it on a single display. The framerate was never lower than 80 for any of the configurations.

6 Discussion

Our expectations prior to implementing touch-free, multi-user support in Q3A and Homeworld were that using gestures to control Q3A would be awkward and difficult, while gestures for controlling Homeworld would be more natural as the pace of the game is slower and the gestures similar to emulating a mouse. Although we haven't conducted any formal user studies, our initial, subjective experiences indicate that the touch-free interface was more natural when controlling Q3A than controlling Homeworld. There are several potential explanations, including the characteristics of the touch-free interface and the intrinsics of the games. For instance, since Homeworld uses a one-to-one mapping between hand position and cursor position, a user might not be able to reach all points on the display wall. Another observation is that as one plays the games for extended periods of time, one's arms become fatigued.

6.1 Latency

In [MW93], the authors investigate the effect of lag (i.e, latency) on human performance in interactive systems. As latency goes up, accuracy deteriorates and time to perform tasks increases. For this reason, it is important for the touch-free interface to provide input with as low latency as possible. In [Arm03], the authors show that Q3A players prefer using Q3A servers where their average ping[6] is no more than 150-180 ms. The touch-free interface has a latency of 116.7 ms, and the average latency from the parallelized Q3A implementation is 87 ms. This gives a total latency of 203.7 ms, 23.7 ms more than the maximum preferred latency. The latency for Q3A fluctuated with a standard deviation of 59 ms, which may be an artifact of the latency measuring experiments, or a result of the Q3A server experiencing varying loads. Even though

[6]The latency from a player takes an action until it becomes observable by other players.

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

the average latency using the touch-free interface is slightly higher than the maximum preferred latency, the touch-free interface can be improved sufficiently to perform below the limit.

The touch-free interface's architecture is currently bound latency-wise by existing camera-technology, which are the biggest contributors to the overall system latency. As camera technology improves, the intrinsic latency of cameras can be reduced, which will directly affect the latency of the touch-free interface. Improvements in the I/O bus and OS will reduce this latency. In earlier work [SHBA07], the latency due to the cameras was found to be 102 ms. More recent experiments puts the latency at 81 ms, as shown in Section 5. We speculate that this reduction in latency is due to an operating system update, as neither the computers or cameras changed in between the experiments. The first set of experiments were conducted using Mac OS X 10.4.8, while the results presented in this paper were obtained on Mac OS X 10.4.9.

The next-biggest contributor to latency is the object detector. The detector waits for all the cameras to provide data before triangulating object positions. This synchronizes the cameras, and ensures that only fresh data from each camera is used for the triangulation. The result is improved accuracy. The cameras all run at 30 FPS, which corresponds well with the 31 ms average latency from the object detector. Improvements in camera technology will also help bring the object detector latency down. As the image capture rate of a camera goes up, the resulting latency incurred by the object detector will go down, as less waiting must be done in order to ensure that fresh data is in use from all cameras. For instance, doubling the camera framerate to 60 FPS, will result in an upper bound on the object detector latency of 16 ms. The architecture of the touch-free interface is scalable, as all image processing is done locally by each computer capturing image data. This reduces the amount of data required to be processed by the object detector by several orders of magnitude.

One problem with the touch-free interface is that its accuracy for positioning objects decreases as the objects move faster. This is caused by the use of many different cameras to capture images. Although each camera operates at the same framerate, they capture images at slightly different points in time. For a moving object, this results in the object appearing at different positions for different cameras. When these positions are used to triangulate an object's 2D position, the result can be inaccurate. These inaccuracies appear as jitter in the object's vertical position. The horizontal position is also affected, although not as much as the vertical position. This problem can be alleviated by using cameras with higher image capture rates, or cameras where the image capture can be synchronized.

6.2 Parallelizing games

Q3A's existing architecture made it possible to rapidly parallelize the game and make it run on the display wall's cluster. In particular, the spectator-concept, which can be viewed as a single data, multiple view model, was useful. This model is absent from Homeworld, making the process of parallelizing Homeworld more laborious. Applications that support this model should be simpler to parallelize for tiled display wall environments. The performance penalty from using spectators in this way is an 87 ms increase in the latency from when a player performs an action until it is visible on the display wall. This latency is independent of the input system used (keyboard/mouse or touch-free interface). Even better results may be achieved by parallelizing the game from scratch, but at the cost of a much greater effort.

Homeworld's architecture made it possible to parallelize it by running synchronized copies on the tiles. However, to determine where to synchronize, the game engine had to be analyzed to identify all places where data is used that could impact the game simulation. At these places the copies must synchronize in order to use identical data. Finding all these synchronization points is difficult, and verifying that all places have been identified requires exercising all possible code-paths of the engine. One way of doing this would be to play the entire game from start to finish; to date only the first level has been completed. Minor bugs and timing issues can also potentially skew the copies out of sync. For these reasons, parallelizing Homeworld required more effort than parallelizing Q3A.

When running Q3A on the entire display wall, the framerate for the parallel version was an order of magnitude higher than the framerate achievable using Chromium. Homeworld outperformed the sequential version when running on 2x2 and 3x3 tiles. This is somewhat unexpected, as the simulation itself was not parallelized. In principle, each copy runs the same code on the same data, with the addition of synchronization overhead for the parallel version. The fact that a higher framerate is still achieved for these tile

urn:nbn:de:0009-6-15001, ISSN 1860-2037

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

configurations, is because the tiles share the rendering workload. For the 7x4 configuration, the framerate is lower than for a single display. We hypothesize that this is due to increased synchronization overhead, mainly from the MPI barriers used.

7 Conclusion

This paper has introduced a touch-free, multi-user interface for controlling applications on wall-sized, high-resolution tiled displays. The interface uses 16 cameras and 9 computers to triangulate the position of objects in a plane parallel to the display wall's canvas. Input from the touch-free interface is converted to hand- and arm gestures, which are then interpreted and injected into Quake 3 Arena and Homeworld as regular mouse and keyboard events. To run on the display wall, the two games were parallelized by exploiting different aspects of the two games' architectures. For Q3A, the spectator-concept was utilized to follow each player on several tiles of the display wall. For Homeworld, a master-slave approach was taken, synchronizing all game state and input.

Players control the games by using one or both hands. Users do not need to use external devices, wear gloves or optical markers in order to interact. In this regard, the interface is not only touch-free, but also completely device-free. This enables the interface to work in a public setting where other input devices might get lost, misplaced or stolen. It also makes interaction more direct, as users no longer must interact through devices like mice or keyboards.

The responsiveness of the touch-free interface was measured by determining its end-to-end latency. The parallel versions of the two games were evaluated by measuring their framerates in both parallel and sequential (unmodified) versions running on the display wall. The touch-free interface's latency was 116.7 ms, with the majority of this latency due to the cameras used. The parallel version of Q3A consistently outperformed the sequential version running on the entire display wall, averaging 398 FPS vs sequential's 21 FPS. The average framerate for Homeworld on a single display was 168 FPS, while running Homeworld on the entire display wall yielded an average framerate of 143 FPS. The high framerates indicate that the parallelized games will scale to more tiles and higher resolutions. The framerates are well beyond what is displayable by a typical LCD panel or projector with a 60 Hz refresh rate.

Acknowledgments

The authors wish to thank Espen S. Johnsen, Tore Larsen and Ken-Arne Jensen for their discussions. Supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

[Arm03] Grenville Armitage. *An experimental estimation of latency sensitivity in multiplayer Quake 3*. In *ICON 2003: Proceedings of the 11th IEEE International Conference on Networks*, pages 137–141. 2003. ISSN 1531-2216.

[BBH05] Steffi Beckhaus, Kristopher J. Blom, and Matthias Haringer. *A new gaming device and interaction method for a First-Person-Shooter*. In *Proceedings of the Computer Science and Magic 2005*. 2005. GC Developer Science Track.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. *LAM: An Open Cluster Environment for MPI*. In *Proceedings of Supercomputing Symposium*, pages 379–386. 1994.

[CCD06] Mark Claypool, Kajal Claypool, and Feissal Damaa. *The Effects of Frame Rate and Resolution on Users Playing First Person Shooter Games*. In *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN)*, volume SPIE-6071, pages 1–11. jan 2006.

[DL01] Paul Dietz and Darren Leigh. *DiamondTouch: a multi-user touch technology*. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 219–226. ACM Press, New York, NY, USA, 2001. ISBN 1-58113-438-X.

[Ent08] Relic Entertainment. *Homeworld*, 2008. www.relic.com/, www.homeworldsdl.org/ and

Journal of Virtual Reality and Broadcasting, Volume 5(2008), no. 10

www.thereisnospork.com/projects/, last visited: April 1st, 2008.

[Han05] Jefferson Y. Han. *Low-cost multi-touch sensing through frustrated total internal reflection*. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-271-2.

[HHN+02] Greg Humphreys, Mike Houston, Ren Ng, Randall Frank, Sean Ahern, Peter D. Kirchner, and James T. Klosowski. *Chromium: a stream-processing framework for interactive rendering on clusters*. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 693–702. ACM Press, New York, NY, USA, 2002. ISBN 1-58113-521-1. ISSN 0730-0301.

[Int08] InterSense. *InterSense IS-900 Systems*, 2008. http://www.isense.com/www.isense.com/products.aspx?id=45, last visited April 1st, 2008.

[iS08] id Software. *Quake 3 Arena*, 2008. www.idsoftware.com/ and ioquake3.org/, last visited: April 1st, 2008.

[JH02] Jeffrey Jacobson and Zimmy Hwang. *Unreal Tournament for Immersive Interactive Theater*. *Commun. ACM*, 45(1):39–42, 2002. ISSN 0001-0782.

[KLJ04] Hyun Kang, Chang Woo Lee, and Keechul Jung. *Recognition-based gesture spotting in video games*. *Pattern Recognition Letters*, 25(15):1701–1714, 2004. ISSN 0167-8655.

[LCC+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 Comput. Graph. Appl.*, 20(4):29–37, 2000. ISSN 0272-1716.

[Mor05] Gerald D. Morrison. *A Camera-Based Input Device for Large Interactive Displays*. *IEEE Computer Graphics and Applications*, 25(4):52–57, 2005. ISSN 0272-1716.

[MW93] I. Scott MacKenzie and Colin Ware. *Lag as a determinant of human performance in interactive systems*. In *CHI '93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 488–493. ACM Press, New York, NY, USA, 1993. ISBN 0-89791-575-5.

[SHBA07] Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto J. Anshus. *Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays*. In *Proceedings of the 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007*, pages 75–83. June 2007.

[SHBA08] Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus. *Lessons Learned using a Camera Cluster to Detect and Locate Objects*. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007.*, volume 15 of *Advances in Parallel Computing*, pages 71–78. IOS Press, 2008. ISBN 978-1-58603-796-3.

[SK98] Jakub Segen and Senthil Kumar. *Gesture VR: Vision-based 3D hand interace for spatial interaction*. In *MULTIMEDIA '98: Proceedings of the sixth ACM international conference on Multimedia*, pages 455–464. ACM Press, New York, NY, USA, 1998. ISBN 0-201-30990-4.

[SW06] Bram Stolk and Paul Wielinga. *Building a 100 Mpixel graphics device for the OptIPuter*. *Future Gener. Comput. Syst.*, 22(8):972–975, 2006. ISSN 0167-739X.

[SZP+00] Rajeev Sharma, Michael Zeller, Vladimir I. Pavlovic, Thomas S. Huang, Zion Lo, Stephen Chu, Yunxin Zhao,

urn:nbn:de:0009-6-15001, ISSN 1860-2037

James C. Phillips, and Klaus Schulten. *Speech/Gesture Interface to a Visual-Computing Environment*. *IEEE Computer Graphics and Applications*, 20(2):29–37, 2000. ISSN 0272-1716.

[TGSF06] Edward Tse, Saul Greenberg, Chia Shen, and Clifton Forlines. *Multimodal Multiplayer Tabletop Gaming*. In *PerGames '06: Proceedings of the 3rd International Workshop on Pervasive Gaming Applications*. 2006.

Citation

Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, Otto J. Anshus, *Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays*, Journal of Virtual Reality and Broadcasting, 5(2008), no. 10, August 2008, urn:nbn:de:0009-6-15001, ISSN 1860-2037.

A.2 The 22 Megapixel Laptop

Citation

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. The 22 megapixel laptop. In *EDT '07: Proceedings of the 2007 workshop on Emerging displays technologies*, pages 1–4, New York, NY, USA, 2007. ACM.

Abstract

Displays are everywhere. To utilize them efficiently, we introduce the notion of the Network Accessible Display (NAD). A user can use displays on nearby computers as if they were physically connected to his computer, including displays on handheld devices and tiled display walls. We present a system adhering to the NAD-model, and demonstrate it by extending a laptop with up to 30 NADs with an area of 22 MPixels connected using both a wireless network and gigabit Ethernet. The system can support one display at 25 Hz and 30 displays at 1 Hz. Even with a refresh rate of only 1 Hz, the system remains useful for displaying relatively static content.

The 22 Megapixel Laptop

Daniel Stødle* John Markus Bjørndalen† Otto J. Anshus‡
Department of Computer Science
University of Tromsø, Norway

Figure 1: *(a) Configuring virtual displays to match a 28-tile display wall. (b) Extending a display to a portable device. (c) Using the 22 megapixel laptop. (d) One laptop extended with both a display wall and a portable device, for a total display area of 22 megapixels.*

Abstract

Displays are everywhere. To utilize them efficiently, we introduce the notion of the Network Accessible Display (NAD). A user can use displays on nearby computers as if they were physically connected to his computer, including displays on handheld devices and tiled display walls. We present a system adhering to the NAD-model, and demonstrate it by extending a laptop with up to 30 NADs with an area of 22 MPixels connected using both a wireless network and gigabit Ethernet. The system can support one display at 25 Hz and 30 displays at 1 Hz. Even with a refresh rate of only 1 Hz, the system remains useful for displaying relativly static content.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Virtual device interfaces;

Keywords: Network Accessible Display, display wall

1 Introduction

The rapid progress in development of computer-related technologies has resulted in a commoditization of computers, storage, displays and other types of hardware. This development has given rise to approaches for building larger systems of cheap components, including hard disk RAIDs, Beowulf/NoW-style computer clusters and tiled displays. As this development continues, displays with processing power can be used as Network Accessible Displays (NADs), offering display services to nearby networked computers.

*e-mail: daniels@cs.uit.no
†e-mail: jmb@cs.uit.no
‡e-mail: otto@cs.uit.no

EDT 2007, San Diego, California, August 04, 2007.
© 2007 ACM 978-1-59593-669-1/07/0008 $5.00

We have built a software system enabling a desktop computer or laptop to utilize tens of displays as if they were directly connected to the computer.

Laptops can typically use both their built-in display and an external display. High-end workstations may be equipped with one or two quad-head graphic cards, capable of supporting up to eight displays in total. Wall-sized, high-resolution, tiled display walls have a pixel area of anywhere between 10 megapixels and 100 megapixels [Li et al. 2000; Stolk and Wielinga 2006], and are built using clusters of computers with displays or projectors. These approaches are lacking in several ways: (i) A laptop can only support one additional display, (ii) a workstation supporting eight displays is expensive, (iii) the number of supported displays is fixed, and (iv) using available, nearby displays from a laptop or workstation is impractical. Finally, a variety of "portable displays" - from watches, to mobile phones, PDAs and tablet computers - are not easily used as extended displays as there is no way of connecting them to computers using regular display cables. An increasing number support networking, however, potentially enabling them to act as NADs.

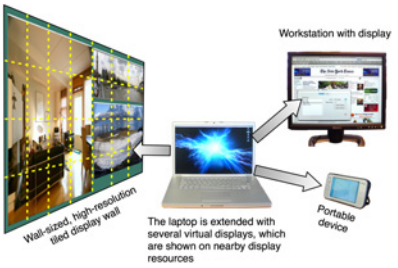


Figure 2: *Example of a laptop extending its local display to utilize the high resolution made available by a tiled display wall, as well as the display resources offered by a workstation and a portable device.*

Software like ZoneScreen, MaxiVista and Screen Recycler[1] lets

[1]http://www.zoneos.com/zonescreen.htm, http://www.maxivista.com

users extend their local desktops to a single other display. These products not only share the user's local display, but *extends it*, essentially making a remote display appear as a secondary local display. MaxiVista can support up to three additional displays, with a resolution up to 4800x2400 pixels. These solutions are lacking in (i) their ability to scale to many displays, (ii) no awareness of the physical arrangement of available display resources, and (iii) limited display resolution.

The above applications use remote desktop software, like Virtual Network Computing (VNC) [Richardson et al. 1998], THINC [Baratto et al. 2005] and Microsoft Remote Desktop, to transfer an extended desktop's pixels to a remote host. VNC shares displays by sending the shared display's pixels to clients, while in THINC better performance is achieved by more efficiently coding the drawing operations used to generate pixels. Another way of sharing display contents is to transmit only drawing operations ("draw string", "fill rectangle", etc.) as used in Microsoft Remote Desktop and the X Window System [Scheifler and Gettys 1986]. Our system makes use of a custom component similar to VNC, but with support for sharing several extended displays.

To support the model of NADs, the system creates virtual displays and shows them on displays ranging from portable displays to tiled display walls. The system extends the local desktop of a laptop running Mac OS X with up to 30 additional, virtual displays of arbitrary resolution[2]. The system then discovers nearby NADs and configures the virtual displays to utilize the available display resources. Our experimental testbed consists of a display wall comprised of 7x4 tiles for a total resolution of 7168x3072 pixels, several workstations and a Nokia N800 "internet tablet" acting as a portable display with a resolution of 800x480. Figure 2 illustrates this setup.

Our main contribution with this paper is the development of the Network Accessible Display model, and in particular: (i) a scalable display sharing model and implementation based on virtual displays, (ii) dynamic mapping of virtual displays to match available display resources, (iii) a system that will fit both the traditional view of displays connected directly to computers, and our vision of the display of the future - the NAD, and (iv) an evaluation of system's performance.

2 Design

The NAD system we developed consists of a number of distinct components: (i) A display service running on computers whose displays we wish to utilize, (ii) a GUI frontend, (iii) a VNC-like display sharing daemon and (iv) a kernel extension to create and maintain a set of virtual displays. Figure 3 illustrates the design.

Any computer wishing to provide its display as a NAD, runs a display service. The display service maintains properties related to the display(s) on the computer it runs[3], and exposes them to clients through a network-based discovery mechanism.

The GUI frontend runs on computers that want to utilize NADs. It discovers nearby display services and queries their properties. Currently available displays and their relative locations are presented to the user, before the user selects the displays he wishes to use. The frontend then configures the virtual displays and tells the daemon to push screen contents to the selected display services.

The display sharing daemon accepts commands from the GUI frontend. It sends the contents of the virtual displays to display services

and http://www.screenrecycler.com.

[2]Limited only by available memory; the system has been tested with resolutions up to 16384x6144.

[3]Bit depth, resolution, location and more.

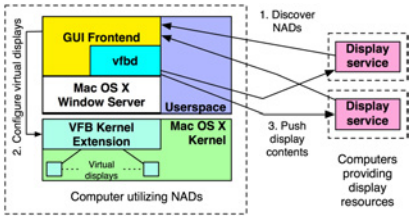


Figure 3: *The system design. Display services running on a set of computers are discovered by the GUI frontend, which proceeds to configure the number, resolution and arrangement of the virtual displays. It then instructs the display sharing daemon (vfbd) to push each display's contents to its associated display service.*

as raw pixel data. The first update consists of all pixels for a given virtual display, while further updates consist of pixels from areas that have changed on the virtual display (incremental updates).

The kernel extension creates a set of virtual displays when the computer boots. The virtual displays appear to the rest of the operating system as real, physically connected displays, but are in reality just a set of memory buffers. The GUI frontend communicates with the kernel extension to configure the number of virtual displays, and uses the window server to configure their resolution, bit depth and arrangement in relation to each other.

3 Implementation

The display service was implemented in C using SDL[4], and BSD sockets for network communication. It currently runs on Linux and Mac OS X. On startup, the display service is configured with the properties for the display resources it should provide. For a regular workstation with a single display, the properties consist of the local display's resolution and bit depth, as well as name and location. For display services running on tiled display walls, the configuration also includes information about the display wall, including the service's location in the grid of display tiles.

The GUI frontend was implemented in Objective-C using Cocoa on Mac OS X. It uses the CGDirectDisplay APIs in Mac OS X to configure virtual displays, including resolution and arrangement. The frontend uses property details from each display service when configuring the resolution and arrangement of virtual displays. Groups of display services that belong together, such as those running on a display wall, are presented together by the frontend, and not mixed with other "free-standing" displays.

The display sharing daemon uses the CGRemoteOperation APIs exported by Mac OS X' window server to access the raw pixels of the virtual displays. These APIs are also used to receive information about areas of the virtual displays where the pixels have changed, supporting incremental updates. The daemon performs run-length encoding of the pixels before sending them to connected display services, in order to reduce bandwidth usage. The daemon receives the network address for a display service from the frontend, then connects to the service and provides it with details about the virtual display. The service then starts accepting pixel data from the daemon.

[4]Simple Direct-Media Layer, a popular cross-platform library often used to develop games; http://www.libsdl.org/

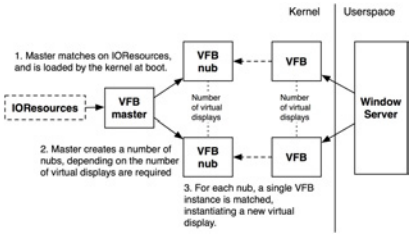


Figure 4: *The kernel extension design. The VFB master class is loaded at boot by the kernel, by matching on the class IOResources. It instantiates a number of VFB nubs. These nubs cause the kernel to start the matching procedure, and instantiate one instance of the VFB class for each nub. The virtual displays are then used by the window server when it starts up.*

The kernel extension, implemented in C++, consists of three classes: VFB (virtual framebuffer) master, VFB nub and VFB, as shown in Figure 4. The master accepts requests from userspace to configure properties of the virtual displays. In particular, it enables the GUI frontend to enable and disable virtual displays, without going through the window server[5]. The purpose of the nub is to provide an endpoint for Mac OS X' IOKit driver system to match and incorporate VFB instances into the kernel. When a nub is instantiated, it registers a service with IOKit. The VFB class matches on this service, making IOKit instantiate one instance of VFB for each nub created by the master.

The VFB class is a subclass of the IOKit class "IOFramebuffer." When it is instantiated, it allocates memory for a framebuffer of some pre-determined resolution (this can vary from instance to instance depending on configuration), before exposing its available resolutions and bit depths to the window server.

4 Evaluation

We document the performance of the system for different numbers of virtual displays. The hardware used was (i) a display cluster with 28 nodes (Intel Pentium 4 EM64T, 3.2 GHz, 2 GB RAM, Hyper-Threading enabled, NVIDIA Quadro FX 3400 with 256 MB Video RAM, running the Rocks Linux cluster distribution 4.0) connected to 28 projectors (1024x768, arranged in a 7x4 matrix), (ii) switched, gigabit Ethernet, and (iii) a MacBook Pro (2.33 GHz Intel Core 2 Duo, 3 GB RAM, Mac OS X 10.4.9).

4.1 Methodology

The MacBook Pro was configured with a number of virtual displays, where each virtual display had a resolution of 1024x768 at 32 bits per pixel. We varied the number of virtual displays between 1, 2, 4, 8, 16, 24 and 28. For each experiment, a window was created that fully covered all the virtual displays (this will be referred to as the "draw" process). The window was completely redrawn 300 times at an attempted rate of 10 Hz[6], after which statistics were gathered. To redraw the window, the draw process copies an image from memory to the window.

For each experiment, we measured the following statistics: (i) The total number of pixels updated by the display services, (ii) total number of bytes used to send pixel data to the display services, (iii) the CPU load both at kernel and user level for the draw process, display sharing daemon (vfbd) and Mac OS X window server.

4.2 Results

Figure 5 shows the target number of Mpixels updated per second compared to the system's actual update rate. With up to four displays, the system tracks the target update rate fairly well. Beyond four displays, the update rate is stable around 24 Mpixels/second, much less than the 30-210 MPixels/second needed to track the target rate. Using 24 virtual displays, the rate is 24.31 Mpixels/second, corresponding to a refresh rate of 1.35 Hz[7].

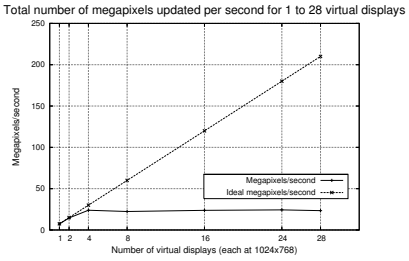


Figure 5: *The graph shows the actual update rate in megapixels/second, and compares it to the target update rate (10 full updates per second).*

Figure 6 shows the measured bandwidth. The bandwidth correlates well with the pixel update rate, with a peak bandwidth of 36.5 megabytes/second with 24 virtual displays. Figure 7 shows the kernel and user level CPU load for the different processes involved in generating and distributing data for the virtual displays. The majority of the CPU is used by the display sharing daemon, followed by the window server and finally the draw process. The combined load peaks at 175% with 24 displays (the MacBook Pro has a dual-core processor).

5 Discussion

The experiments demonstrate that there is a tradeoff between update rate and the size of the area being updated. In the experiments this area equals the combined resolution of the virtual displays. For smaller areas, the update rate can be quite high. As an example, a rate of 24 MPixels/second delivered to a virtual display with resolution 1024x768 corresponds to a refresh rate of 32 Hz. The same rate to a set of virtual displays with a total resolution of 7168x3072 (the size of the display wall used in the experiments) results in 1.14 Hz. Although not shown in the previous section, the best sustained refresh rate for full screen updates at 1024x768 in 16-bit color is 25

[5]The window server does not provide a mechanism to control whether a display is available or not.

[6]The actual rate was lower for most of the configurations, as discussed in the next section.

[7]24 virtual displays in a 6x4 grid results in a total resolution of 6144x3072 pixels; one full update is 18 megapixels, thus the refresh rate is 24.31/18 = 1.35.

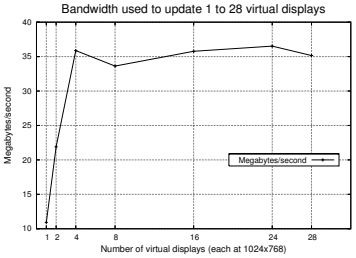


Figure 6: *The graph shows the bandwidth used to update the virtual displays.*

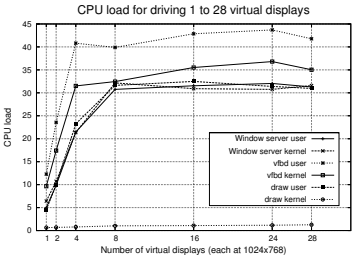


Figure 7: *The graph shows CPU load (in percent) for the display sharing daemon (vfbd), window server and draw process at both kernel and user level.*

Hz, and for 32-bit about 18 Hz. As the resolution increases, the system's performance goes down, but remains usable for mostly static content (images, documents, etc.).

The network is never saturated by the system - a transfer rate of 36 megabytes/second is less than half of the available bandwidth on a gigabit Ethernet. Thus, the network is not the main bottleneck. The CPU load measurements indicate that the main bottleneck is on the laptop. The load correlates well with the total resolution offered by the virtual displays, roughly doubling every time the resolution doubles, until the total CPU load goes beyond what the CPU can deliver at 4 virtual displays.

The draw process incurs little kernel level load, as it only copies pixel data from a buffer to its own window. The window server's CPU load tracks the load of the draw process well. Interestingly, this applies both to the window server's user *and* kernel level load, which indicates that the window server may be doing twice the work necessary to get the pixels to the virtual display (the data appears to be copied twice). The display sharing daemon spends about 55-60% of its time at user level, with the remaining time spent at kernel level. The time spent at user level is due to copying and compressing pixel data, while the time spent at kernel level comes from transferring pixel data over the network. The main bottleneck in the system as the total resolution offered by the virtual displays increases is copying data, and we hypothesize that improved performance can be achieved by eliminating redundant memory copies.

The Mac OS X window server is limited to 32 displays (virtual or not). In practice, the limit is 30, as there usually is a main display attached (a laptop's built-in display, for instance). In addition, the window server has a second, always-available virtual display with a resolution of 1x1 pixel which is always offline. The purpose of this display is unknown to the authors and to the authors' knowledge not documented. Even though the window server detects the presence of additional displays beyond the (practical) limit of 30, they are never used or exposed to clients. While the window server scales well, other parts of Mac OS X are not as scalable. Attempting to configure the virtual displays from System Preferences results in seeing an apparently random selection of at most 10 displays, and the display configuration menu only manages to show 16.

6 Conclusion

We have introduced the Network Accessible Display model, and presented the design and implementation of a system that adheres to the model. A NAD computer runs a display service that communicates with clients wishing to use the NAD. Clients discover NADs using a multicast-based discovery mechanism. We have used the system to extend a laptop with up to 30 virtual displays and map them to nearby physical displays, including a 22 Mpixel wall-sized, high resolution tiled display, and a 0.3 Mpixel portable device.

The bottleneck for increased resolution is copying pixel data locally on the client. When the number of pixels double, the client-side CPU load doubles. At a rate of 24 Mpixels/sec to the NADs, all available CPU is spent. We explain this by (i) load incurred compressing and transferring pixel data, and (ii) copying and compositing pixel data without graphics card hardware acceleration. Despite the low refresh rate for higher resolutions, the system is still useful for displaying static content like images and multiple documents.

Acknowledgements

Thanks to Tor-Magne S. Hagen and Espen S. Johnsen for discussions, and Ståle W. Nilsen for help with the video. Supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

BARATTO, R. A., KIM, L. N., AND NIEH, J. 2005. THINC: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, ACM Press, New York, NY, USA, 277–290.

LI, K., CHEN, H., CHEN, Y., CLARK, D. W., COOK, P., DAMIANAKIS, S., ESSL, G., FINKELSTEIN, A., FUNKHOUSER, T., HOUSEL, T., KLEIN, A., LIU, Z., PRAUN, E., SAMANTA, R., SHEDD, B., SINGH, J. P., TZANETAKIS, G., AND ZHENG, J. 2000. Building and Using A Scalable Display Wall System. *IEEE Comput. Graph. Appl. 20*, 4, 29–37.

RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. 1998. Virtual Network Computing. *IEEE Internet Computing 2*, 1, 33–38.

SCHEIFLER, R. W., AND GETTYS, J. 1986. The X window system. *ACM Trans. Graph. 5*, 2, 79–109.

STOLK, B., AND WIELINGA, P. 2006. Building a 100 Mpixel graphics device for the OptIPuter. *Future Gener. Comput. Syst. 22*, 8, 972–975.

A.3 Lessons learned using a camera cluster to detect and locate objects

Citation

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus. Lessons learned using a camera cluster to detect and locate objects. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007*, volume 15 of *Advances in Parallel Computing*, pages 71–78. IOS Press, 2008.

Abstract

A typical commodity camera rarely supports selecting a region of interest to reduce bandwidth, and depending on the extent of image processing, a single CPU may not be sufficient to process data from the camera. Further, such cameras often lack support for synchronized inter-camera image capture, making it difficult to relate images from different cameras. This paper presents a scalable, dedicated parallel camera system for detecting objects in front of a wall-sized, high-resolution, tiled display. The system determines the positions of detected objects, and uses them to interact with applications. Since a single camera can saturate either the bus or CPU, depending on its characteristics and the image processing complexity, the system supports configuring the number of cameras per computer according to bandwidth and processing needs. To minimize image processing latency, the system focuses only on detecting where objects are, rather than what they are, thus reducing the problem's complexity. To overcome the lack of synchronized cameras, short periods of waiting are used. An experimental study using 16 cameras has shown that the system achieves acceptable latency for applications such as 3D games.

John von Neumann Institute for Computing

Lessons Learned Using a Camera Cluster to Detect and Locate Objects

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, Otto J. Anshus

published in

Parallel Computing: Architectures, Algorithms and Applications ,
C. Bischof, M. Bücker, P. Gibbon, G.R. Joubert, T. Lippert, B. Mohr, F. Peters (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **38**, ISBN 978-3-9810843-4-4, pp. 71-78, 2007.
Reprinted in: Advances in Parallel Computing, Volume **15**,
ISSN 0927-5452, ISBN 978-1-58603-796-3 (IOS Press), 2008.

http://www.fz-juelich.de/nic-series/volume38

Lessons Learned Using a Camera Cluster to Detect and Locate Objects

Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science
Faculty of Science
N-9037 University of Tromsø, Norway
E-mail: {daniels, phuong, jmb, otto}@cs.uit.no

A typical commodity camera rarely supports selecting a region of interest to reduce bandwidth, and depending on the extent of image processing, a single CPU may not be sufficient to process data from the camera. Further, such cameras often lack support for synchronized inter-camera image capture, making it difficult to relate images from different cameras. This paper presents a scalable, dedicated parallel camera system for detecting objects in front of a wall-sized, high-resolution, tiled display. The system determines the positions of detected objects, and uses them to interact with applications. Since a single camera can saturate either the bus or CPU, depending on its characteristics and the image processing complexity, the system supports configuring the number of cameras per computer according to bandwidth and processing needs. To minimize image processing latency, the system focuses only on detecting where objects are, rather than what they are, thus reducing the problem's complexity. To overcome the lack of synchronized cameras, short periods of waiting are used. An experimental study using 16 cameras has shown that the system achieves acceptable latency for applications such as 3D games.

1 Introduction

This paper reports on lessons learned using a cluster of cameras to detect the position of objects in front of a wall-sized, high-resolution, tiled display. The system is used to support multi-user touch-free[a] interaction with applications running on a 220-inch 7x4 tiles, 7168x3072 pixels resolution display wall (Fig. 1). This requires that the system can accurately and with low latency determine the positions of fingers, hands, arms and other objects in front of the wall. To achieve this, a consistent and synchronized set of position data from each camera is needed.

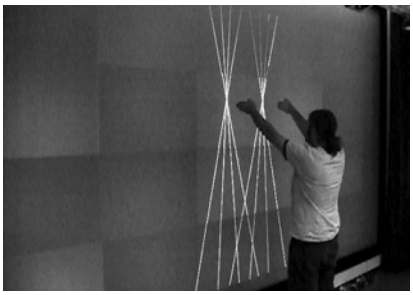


Figure 1. Using the system.

A grayscale camera producing images at a rate of 30 frames per second with a resolution of 640x480 pixels requires a bandwidth of about 8.78 megabytes/second. A FireWire 400 bus can accommodate at most three cameras producing data at this rate; higher-resolution or higher-framerate cameras further decrease this bound. To support more cameras, additional FireWire buses can be used on a single computer. Scalability may now

[a] As the display wall's canvas is not rigid, users must be able to interact with the display wall without actually touching it - thus the term "touch-free."

be limited by the CPU, either due to image processing complexity or deadlines on when results are needed. Finally, most commodity cameras have no support for hardware- or software-based inter-camera synchronization. This limits the accuracy of object positioning, as it reduces the system's ability to relate images captured from different cameras to each other.

This paper presents a parallel system for processing streaming video from several cameras. The system architecture comprises four layers: (i) Camera and image processing, (ii) object-position processing, (iii) event distribution, and (iv) end-application use of position data. The first layer uses 16 cameras connected pairwise to 8 computers. Each computer processes images from two cameras, locating objects and determining their one-dimensional position. When three or more cameras in the first layer see the same object, the second layer can determine the object's 2D position using triangulation. The third layer distributes position data between the other three layers. The fourth layer is comprised of applications using the position data for interaction. An experimental study has shown that the system achieves acceptable latency for common applications like the 3D games Quake 3 Arena and Homeworld (see Section 5 and Ref. 1).

The main contributions of this paper are the lessons learned from building and using the system, including: (i) The flexibility of the system architecture allows configuring available camera and processing resources to accommodate end-applications' needs, (ii) by reducing the complexity of image processing from identifying *what* objects are to identify *where* they are, processing is reduced, and (iii) despite the lack of synchronized cameras, useful results may still be obtained by introducing short periods of waiting.

2 Related Work

There exists much work on multi-camera systems. In Ref. 2, the authors demonstrate how a 100-camera array is used to capture very high-resolution video at 3800x2000 pixels at 30 FPS, or high-speed video with 640x480 pixels at 1560 FPS. Their implementation uses custom circuit boards to communicate with the FireWire cameras and relies on hardware synchronization of cameras, while the system presented in this paper is exclusively based on use of commodity components; cameras without support for synchronization and no use of custom hardware. In Ref. 3, the authors show how displays may be synchronized using an external synchronization source combined with software adjustment of display timings (software genlocking). Their use of a hardware synchronization signal precludes applying their technique to synchronize commodity camera capture.

Other work has used many low-resolution cameras to generate a 3D reconstruction of objects, either for collaborative applications[4] or for creating 3D models. Our system does not attempt to generate high-resolution video or imagery, or reconstruct 3D objects. Instead, the goal is to use a cluster of cameras to determine the 2D position of objects in a plane parallel to the display wall. Previous work reports on different ways of achieving this. In Ref. 5, the author combines internal reflection of infrared light with a camera mounted behind a (rigid) canvas to support multi-touch interaction. Our system differs in that it doesn't require users to actually touch the canvas in order to interact, and in the use of a parallel architecture for capturing and processing images. In Ref. 6, a set of cameras with on-board image processing is mounted in the corners of a large display, and used to detect multiple points of contact. Rather than build custom cameras, our system

uses commodity cameras mounted on the floor in front of the approximately 6 meter wide display wall, and performs all processing on a compute cluster.

3 Design

The system architecture is comprised of four layers, as detailed in the introduction and shown in Fig. 2. The camera and image processing layer captures and processes images from cameras used by the system. To allow for many cameras to be used simultaneously as well as flexibility in image processing complexity, this layer is designed to run in parallel. The layer produces 1D positions and radii for detected objects in each image for each camera. An object's 1D position is defined as the centre of a detected object along the horizontal axis of a captured image (the centner of the finger in Fig. 4), and its radius defined as half the width (in pixels) of the detected object. The object position processing layer combines the position data from each computer in the image processing layer using triangulation, to determine the each object's 2D position.

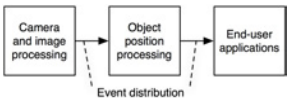


Figure 2. The system architecture.

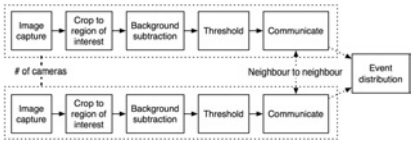


Figure 3. The camera and image processing layer design. The layer can operate in parallel with any number of cameras. Communication happens between each participant and its immediate neighbours.

The camera and image processing layer consists of several steps: (i) Image capture, (ii) cropping, (iii) background subtraction, (iv) thresholding and (v) communication, shown in Fig. 3. When the layer first starts up, it stores the current image from the camera it works with to a buffer. This image will be referred to as the background. As new images are captured, a horizontal region of interest (ROI) is isolated, before the pixel values in the ROI are subtracted from corresponding pixels in the background image. If the absolute difference between a pixel in the current and in the background image is beyond a given threshold, an object is detected at the position of the given pixel in the image. The ROI is determined dynamically when each camera starts capturing images, by identifying the two brightest, horizontal regions in the image[b].

Figure 4 shows an example of how a single image from a single camera is processed. The two horizontal lines (1) indicate two regions of interest in the image. A finger extends from the hand visible in the image, intersecting both ROIs. The background (2) is subtracted from the current image (3), resulting in (4), before the thresholding step is applied, yielding (5). Continuous regions of white indicate where objects have been found in the image.

To account for changes in lighting, the background is updated when too many objects are detected in a single frame from a given camera. An earlier implementation updated the background continuously by merging it with the current image. This did not work well, as users often point at the same location for longer periods of time (on the order of several

[b]The system makes use of a set of "Christmas lights" running along the ceiling, directly above the cameras, in order to create high contrast with intersecting objects.

seconds). The result was "ghost" objects appearing when the user eventually moved his hand.

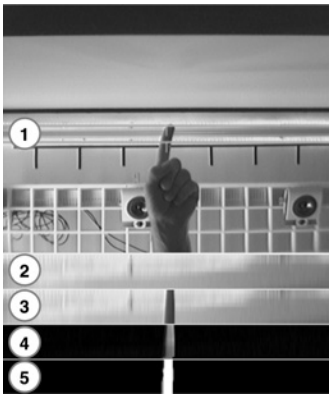


Figure 4. A sample image being processed by the image processing layer. The camera looks directly at the ceiling.

When all objects in the image have been found, the communication step begins. First, each participant sends the number of objects it has detected on the left- and right-hand side of the image to the neighbours on its left and right. The participant receives data from its neighbours, but to avoid introducing additional latency, the participant will use values that are up to 66 ms old[c]. The received values are used to determine if the participant's results coincide with those of its neighbours. If the number of objects it has detected for the left or right side of the image is identical to the number of objects a neighbour has detected for the same side, no further processing is done. However, if the participant has detected fewer objects than its neighbour, it will re-perform the image processing sequence with a lowered threshold, in an attempt at discovering objects lost due to noise in the captured image. Similarly, if it detects more objects than a neighbour, the image processing sequence is re-performed with a raised threshold. Once this is done, the final 1D positions and radii are sent to the object position processing layer using the event distribution layer.

The object position layer receives 1D positions for located objects from the image processing layer, and uses the positions to triangulate their positions. In order to do this successfully, at least three 1D positions from three different cameras are required, as shown in Fig. 5; any less, and false positives occur when multiple objects are visible. The triangulation is performed by computing intersections between lines projecting from the cameras and up, at an angle determined by the 1D positions. An object's position in 2D is successfully identified when two or more points of intersection from different cameras lie sufficiently close to each other. The final 2D position is computed as the average of the X and Y components of the 2D intersection points.

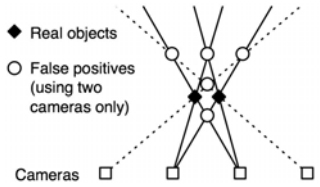


Figure 5. Line segments from each camera and passing through each object are generated. Each line segment is intersected with every other segment. At least three cameras are required to position an object, as using only two cameras results in many false positives.

4 Implementation

The system is comprised of 16 Unibrain Fire-i cameras, connected in pairs to a cluster of 8 Mac minis. In addition, a display cluster of 28 computers, each driving one projector at

[c]This is a tradeoff between system latency, and object detection accuracy. Data that is 66 ms = two frames old may still contain the correct number of (current) objects.

1024x768, is used to provide the graphics capabilities of the display wall, and a MacBook Pro is used to perform object position processing using data from the 8 Mac minis.

The cameras use IEEE-1394 (FireWire) to communicate with the Mac minis, and capture 640x480 grayscale (8-bit) images at 30 frames per second (FPS). They are mounted along the floor and spaced 32 cm apart, as shown in Fig. 6. The cameras do not support external or software-based triggers to synchronize the image capture of multiple cameras, not even when they are on the same FireWire bus. This means that two cameras may capture images spaced in time as far as 33 ms apart (the time between two frames at 30 FPS.).

The Mac minis are interconnected using Gigabit Ethernet. Each Mac mini captures and processes images from the two cameras it is connected to independently of the others, and runs a custom application for performing image capture and processing. This application is written in Objective-C, and uses libdc1394[7] to communicate with the cameras. Each camera is handled by a separate thread within the application, where each thread corresponds to one participant in the camera and image processing layer. Once a frame has been analyzed, the 1D positions and radii of any detected objects are sent to the the object position processing layer.

Figure 6. The image shows 12 of the 16 cameras mounted along the floor and looking at the ceiling.

The object position layer runs a loop operating at the same rate as the cameras, and uses the 1D object positions it receives to triangulate the positions of potential objects. To handle the lack of synchronized cameras, the object positioning software waits for up to 33 ms to receive (possibly empty) sets of object positions from all participating cameras. For the case when a camera has not detected an object, it will notify the object positioning layer of this for the first "no-detect" event only.

To triangulate the positions of objects, the cameras are placed in a coordinate system where cameras are spaced 1 unit apart (1 unit corresponds to 32 cm). For each camera, line segments starting at the camera's position and passing through the centre of each detected object are generated (Fig. 5). The lines are then intersected with all lines from the two cameras to the current camera's left and right. The resulting intersection points are compared, and points that are sufficiently close result in an object being identified. The identified objects' 2D positions and radii are then sent to end-user applications. It is each end-user application's responsibility to interpret the events to allow user interaction.

5 Evaluation

We have evaluated the system by measuring the latency incurred by the system's different layers. In particular, we measure the latency for the following components: (i) Camera capture, (ii) image processing, (iii) event distribution, and (iv) object position processing.

To measure camera capture latency, one camera was connected to a computer and pointed at the computer's display. A custom application fills the computer's display with black, and then starts capturing images from the camera. At one-second intervals, the display is filled with white, and a timer is started. When the difference between average pixel

values from a 20x20 pixel square in the centre of the image in the previous and the current frame exceeds 150 (because the image goes from being black to being white), the timer is stopped, yielding the camera latency. The experiment was conducted on a Mac mini (1.66 GHz Intel Core Duo, 512 MB RAM) running Mac OS X 10.4.9 and a workstation (Intel Pentium 4 3.0 GHz, 2 GB RAM, HyperThreading enabled) running Ubuntu Linux 6.10 to investigate potential differences in latency caused by the operating system or hardware.

The image processing and object position processing latencies were measured by instrumenting the code that performs the two tasks and measure the execution time of 1000 iterations. The event layer's latency was measured using a ping-pong style benchmark, determining the round-trip time for one event sent back and forth. The resulting round-trip time was divided by 2 to find the one-way latency.

	Cam. capture	**Image proc.**	**Event distr.**	**Object pos.**	**Sum**
Samples	923 (852)	1000	1000	1000	-
Average	81 ms (93 ms)	1.16 ms	1.9 ms	31 ms	115 ms
Std. dev.	10 ms (9 ms)	0.11 ms	0.02 ms	10 ms	-
Minimum	58 ms (72 ms)	0.97 ms	1.6 ms	0.008 ms	62.7 ms
Maximum	104 ms (114 ms)	3.3 ms	3.8 ms	139 ms	250.1 ms

Table 1. Results from the latency experiments. Results for camera capture latency in parentheses are from running the experiment on the Linux workstation.

Table 1 shows the results from the experiments. The majority of total system latency of 115 ms is due to the cameras, with about 10 ms separating the measured latency on Mac OS X and Linux. The next biggest contributor to latency is object position processing (object pos.), which incurs an average latency of 31 ms, which is close to the rate at which the cameras deliver data (every 33 ms). Image processing in the system does not incur much latency. Event distribution (only counted once in the table, but generally incurred twice; once for sending events from the image processing layer to the object position layer, and then once more for sending events from the object position layer to end-user applications) incurs a negligible latency.

6 Discussion

The lack of synchronized cameras is the main limiting factor for accuracy in the system. As two cameras can capture images taken as much as 33 ms apart, the accuracy of the triangulation is significantly affected when the object is moving. The effect is further compounded because three cameras are required to accurately position an object. Filtering can reduce the impact of the uncertainty in position, but at the cost of higher latency. The object position processing layer already introduces up to 33 ms of latency to receive updated position data from all cameras. Although latency could be reduced by not waiting for all cameras, this has the effect of reducing the triangulation accuracy and the rate at which object positions are correctly triangulated drops.

Without synchronized cameras, the question of the system's accuracy can be raised. How fast can an object move while still being accurately positioned? Let p and r be the centre of an object O and its radius, respectively. Since the system uses only the horizontal

axis to position objects, position and movement of an object are implicitly assumed to be horizontal[d].

We observe that a position x of the object detected by a camera can be considered accurate as long as x lies within $[p-r, p+r]$. Therefore, the position of a moving object can be detected accurately if there exists a common position $x*$ that satisfies the accuracy requirement for three images taken by three adjacent cameras during the interval $t = 33ms$[e]. Let $p' > p$ be the new horizontal position of the object's centre due to the object movement during the interval t. The common position $x*$ must satisfy $x* \leq p + r$ and $x* \geq p' - r$. Such a common position exists if $p' - r \leq p + r$ or $p' - p \leq 2r$. That means the system can detect an object's position accurately if the object does not move longer than $2r$ - its diameter - during the interval t.

For instance, assume that the object diameter is 1 cm (e.g. the size of the index-finger). In this case, the object's position can be accurately determined if the object moves at a speed less than $\frac{1cm}{33ms} = 0.3m/s$. Higher framerates can increase this bound, since the maximum delay between two cameras capturing an image will decrease. Doubling the framerate makes the maximum delay go down from 33 ms to 16 ms, and also reduce the object position processing latency. Other limiting factors are the number of cameras detecting the same object, the resolution of the cameras, the speed of the objects, the camera shutter speed, and the accuracy of the image processing layer.

The total system latency of 115 ms is sufficiently low to support playing two games (Quake 3 Arena and Homeworld), as we show in Ref. 1. In that paper, the camera latency was measured to be 102 ms, 21 ms more than reported in this paper. We speculate that the difference is due to a newer OS release in between the first set of results and the results presented in this paper[f]. The results from the Linux workstation show that the operating system or hardware architecture has an impact on the latency from the time at which a camera captures an image, until that image can be processed.

7 Conclusion and lessons learned

We have presented a scalable, dedicated parallel system using a camera cluster to detect and locate objects in front of a display wall. The bottlenecks in such a system can range from the bandwidth required by multiple cameras attached to a single bus and CPU requirements to process images, to deadlines on when results from image processing must be available. Due to our system's parallel architecture, the system can scale both in terms of processing and number of cameras. We currently use two cameras per computer, but with either more cameras, higher-resolution cameras or cameras with higher framerates, the system can be scaled by adding more computers.

Processing images can be CPU-intensive. To avoid image processing incurring too much latency, we reduce the complexity of it by focusing on only detecting that an object is present in an image, rather than determining exactly what the object is. This means that the image processing done by our system can be done quickly, resulting in very low image processing latencies (about 1 ms).

[d] Vertical movement translates to slower shifts in the detected, horizontal position of objects.
[e] The maximum delay between two images taken by two different cameras.
[f] The initial results were gathered on Mac OS X 10.4.8, while the new results are from 10.4.9.

Another challenge in systems using cameras to detect and position objects is relating images from different cameras to each other. High-end cameras can resolve this issue by providing support for either software- or hardware-based synchronization. The commodity cameras used by our system supports neither. Our system resolves this by waiting for data from all cameras currently detecting objects, resulting in up to 33 ms of added latency. This still does not solve the problem of cameras capturing images at different points in time - however, it is better than not detecting objects at all because data from related cameras is processed in alternating rounds.

We have used the system for interacting with different applications on the display wall. This includes controlling the two games Quake 3 Arena and Homeworld[1], and control a custom whiteboard-style application with functionality for creating, resizing and moving simple geometric objects as well as drawing free-hand paths. The system works well for tasks that do not require higher levels of accuracy than our system can deliver, despite the intrinsic lack of synchronization between the cameras.

Acknowledgements

The authors thank Ken-Arne Jensen and Tor-Magne Stien Hagen. This work is supported by the Norwegian Research Council, projects 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and 155550/420 - Display Wall with Compute Cluster.

References

1. D. Stødle, T.-M. S. Hagen, J. M. Bjørndalen and O. J. Anshus, *Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays*, in: *Proc. 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007*, pp. 75–83, (2007).
2. B. Wilburn, N. Joshi, V. Vaish, E.-V. Talvala, E. Antunez, A. Barth, A. Adams, M. Horowitz and M. Levoy, *High performance imaging using large camera arrays*, in: *SIGGRAPH '05: ACM SIGGRAPH 2005 Papers*, pp. 765–776, (ACM Press, NY, 2005).
3. D. Cotting, M. Waschbüsch, M. Duller and M. Gross, *WinSGL: synchronizing displays in parallel graphics using cost-effective software genlocking*, Parallel Comput., **33**, 420–437, (2007).
4. J. Mulligan, V. Isler and K. Daniilidis, *Trinocular stereo: A real-time algorithm and its evaluation*, Int. J. Comput. Vision, **47**, 51–61, (2002).
5. J. Y. Han, *Low-cost multi-touch sensing through frustrated total internal reflection*, in: *UIST '05: Proc. 18th Annual ACM symposium on User Interface Software and Technology*, pp. 115–118, (ACM Press, NY, 2005).
6. G. D. Morrison, *A camera-based input device for large interactive displays*, IEEE Computer Graphics and Applications, **25**, 52–57, (2005).
7. D. Douxchamps et. al., *libdc1394, an open source library for handling firewire DC cameras*. http://damien.douxchamps.net/ieee1394/libdc1394/.

A.4 A System for Hybrid Vision- and Sound-Based Interaction with Distal and Proximal Targets on Wall-Sized, High-Resolution Tiled Displays

Citation

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. A system for hybrid vision- and sound-based interaction with distal and proximal targets on wall-sized, high-resolution tiled displays. In *Proceedings of the IEEE International Workshop on Human-Computer Interaction 2007*, volume 4796 of *Lecture Notes in Computer Science*, pages 59–68. Springer, 2007.

Presented at the IEEE Workshop on Human Computer Interaction 2007, held in conjunction with the IEEE International Conference on Computer Vision, Rio de Janeiro, Brazil, October 20. 2007.

Abstract

When interacting with wall-sized, high-resolution tiled displays, users typically stand or move in front of it rather than sit at fixed locations. Using a mouse to interact can be inconvenient in this context, as it must be carried around and often requires a surface to be used. Even for devices that work in mid-air, accuracy when trying to hit small or distal targets becomes an issue. Ideally, the user should not need devices to interact with applications on the display wall. We have developed a hybrid vision- and sound-based system for device-free interaction with software running on a 7x4 tile 220-inch display wall. The system comprises three components that together enable interaction with both distal and proximal targets: (i) A camera determines the direction in which a user is pointing, allowing distal targets to be selected. The direction is determined using edge detection followed by applying the Hough transform. (ii) Using four microphones, a user double-snapping his fingers is detected and located, before the selected target is moved to the location of the snap. This is implemented using correlation and multilateration. (iii) 16 cameras detect objects (fingers, hands) in front of the display wall. The 1D positions of detected objects are then used to triangulate object positions, enabling touch-free multi-point interaction with proximal content. The system is used on the display wall in three contexts to (i) move and interact with windows from a traditional desktop interface, (ii) interact with a whiteboard-style application, and (iii) play two games.

A System for Hybrid Vision- and Sound-Based Interaction with Distal and Proximal Targets on Wall-Sized, High-Resolution Tiled Displays

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus

Dept. of Computer Science, University of Tromsø, N-9037 Tromsø, Norway
{daniels,jmb,otto}@cs.uit.no

Abstract. When interacting with wall-sized, high-resolution tiled displays, users typically stand or move in front of it rather than sit at fixed locations. Using a mouse to interact can be inconvenient in this context, as it must be carried around and often requires a surface to be used. Even for devices that work in mid-air, accuracy when trying to hit small or distal targets becomes an issue. Ideally, the user should not need devices to interact with applications on the display wall. We have developed a hybrid vision- and sound-based system for device-free interaction with software running on a 7x4 tile 220-inch display wall. The system comprises three components that together enable interaction with both distal and proximal targets: (i) A camera determines the direction in which a user is pointing, allowing distal targets to be selected. The direction is determined using edge detection followed by applying the Hough transform. (ii) Using four microphones, a user double-snapping his fingers is detected and located, before the selected target is moved to the location of the snap. This is implemented using correlation and multilateration. (iii) 16 cameras detect objects (fingers, hands) in front of the display wall. The 1D positions of detected objects are then used to triangulate object positions, enabling touch-free multi-point interaction with proximal content. The system is used on the display wall in three contexts to (i) move and interact with windows from a traditional desktop interface, (ii) interact with a whiteboard-style application, and (iii) play two games.

Keywords: Vision- and sound-based interaction, large displays.

1 Introduction

Wall-sized, high-resolution displays are typically built by tiling displays or projectors in a grid. The displays or projectors are driven by a cluster of computers cooperating to produce a combined image covering the display wall, with existing display walls ranging in resolution from 10 to 100 Megapixels [8,16]. Our display wall combines 7x4 projectors back-projecting onto a non-rigid canvas to create a 220-inch display with a resolution of 7168x3072 pixels.

Display walls invite users to stand and move in front of them. To use input devices like mice and keyboards in this context, users must carry them around.

M. Lew et al. (Eds.): HCI 2007, LNCS 4796, pp. 59–68, 2007.

Mice often require flat surfaces to be used, and even for "gyro-mice" and similar devices that don't have this requirement, the accuracy when trying to hit small or distal targets becomes an issue. We believe that ideally, users should not need devices to interact with applications running on the display wall.

We have developed a hybrid vision- and sound-based system that enables device-free interaction with software running on our display wall. The system analyzes and combines input from three main components to enable both distal and proximal interaction. The three components perform the following tasks, respectively: (i) Determine the direction in which a user is pointing his arm, (ii) determine the location at which a user snaps his fingers, and (iii) determine the location of objects (usually fingers or hands) in front of the wall. Figure 1 shows the system in use for playing two games.

Fig. 1. The system in use for playing Quake 3 Arena and Homeworld

Each component contributes to a different part of the device-free workflow on the display wall. The first component lets a user select a distal target by pointing his arm towards it. A camera situated in the ceiling behind the user captures video. Each frame of the video is run through an edge detector, before the Hough transform is applied to determine the angle and position of the prevalent line segments visible in the video. Combining this with knowledge about what is currently visible on the display wall lets the component determine which target the user is attempting to select.

The second component lets the user bring the selected, distal target closer by double-snapping his fingers. Four microphones arranged in a rectangular fashion near the display wall stream audio to a computer. The computer correlates the audio samples with a template audio clip. When the user's snap is detected from at least three microphones, the snap's position can be determined using multilateration.

The third component lets the user interact with proximal targets. 16 cameras arranged in a row on the floor below the display wall's canvas stream data to 8 computers. When an object is visible from at least three cameras as it intersects a plane parallel to the display wall's canvas, its position can be determined using triangulation. Multiple objects can be detected and positioned simultaneously, enabling touch-free multi-point and multi-user interaction. We refer to it as touch-free as the user does not actually have to touch the display wall in order to interact with it. This is important in our context, as the display wall's canvas is flexible, and thus prone to perturbations when users touch it.

The main contribution of this paper is a hybrid vision- and sound-based system for interacting with both distal and proximal targets. We detail the system's implementation and show its use in three different contexts on a display wall:

(i) A traditional desktop interface, (ii) an experimental whiteboard-style application, and (iii) two games.

2 Related Work

Interacting with distal targets on very large displays is an important problem in HCI, and much previous work exists. In [18], the authors present a system that allows distant freehand pointing for interacting with wall-sized displays. Our system does not support pointing from afar, but lets users select distal targets while standing close to the display wall. Our system detects the angle at which a user points his arm, while the authors of [18] use a commercial system that requires the user to wear markers. There are numerous other techniques for reaching distal targets described in the literature; they include drag-and-pop [1], the Vacuum [2] and the Frisbee [7]. These techniques generally work by temporarily bringing distal targets closer. Our approach is complementary in this regard, as distal targets are moved semi-permanently (that is, until the user decides to move them again). The "tablecloth" is an entirely different approach that lets the user scroll the desktop much as he would scroll a window [13].

In [14], untagged audio is used for interacting with a 3D interface connected to a media player. The system recognizes loud sounds above a dynamic threshold, such as snapping fingers, and in turn uses this to determine the position of the sound. Their work focuses on creating 3D interfaces, by creating and placing virtual "buttons" in space. Rather than creating buttons with fixed locations, we utilize the user's actual position to bring distal targets closer to the user. In [6], the authors propose using continuous vocal sounds for controlling an interface, rather than interpreting spoken commands. They do not attempt to determine the user's physical location, while one key capability of the snap-detecting component of our system is that it allows applications to leverage this information. Using vocal sounds to control the position of a cursor is proposed in [9]. This work could conceivably be used to act on distal targets, and indeed one aspect of moving the cursor often includes moving windows. However, no attempt is made at determining the user's location and using this information to improve interaction.

Sound source localization is much used in the field of robotic navigation. In [17], a system is described whereby 8 microphones are used to reliably locate sounds in a 3D environment using cross-correlation and time-delay. The technique used by the snap-detecting component is similar in that it estimates the time-delay between incoming snaps, but differs in that it uses the located sound for interacting with a user interface, rather than interacting with a robot. Our component only attempts to detect snaps, whereas the system in [17] does not discriminate between different sounds. Rather, it is used to locate all "interesting" sound sources, in order to focus the robot's attention towards the sound source.

Much research has been done on systems for supporting multi-touch and multi-point interaction. The Diamondtouch [3] tabletop is one approach, where

the position of touches is detected using electric capacitance. Other technologies include [5], where infrared light is projected into a canvas and internally reflected. The light escapes the canvas at points where the user is touching, which can be detected using a camera. Our system is based on detecting the presence of objects directly using cameras, and does not require the user to actually touch the display wall's canvas. In [10], the author presents a camera-based solution to detecting and positioning objects in front of a whiteboard. They use custom cameras with on-chip processing to perform object recognition, while we take a parallel and commodity-based approach with 16 cheap cameras connected to 8 computers.

3 Design

The hybrid vision- and sound-based system is comprised of three components and a custom event delivery system. The three components are (i) arm-angle, (ii) snap-detect and (iii) object-locator. Each component is responsible for detecting different user actions, with the event system providing communication between the components and end-user applications[1]. Figure 2 illustrates the overall design.

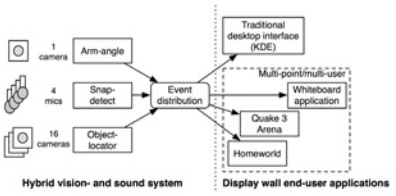


Fig. 2. The system design with three components and event distribution

The purpose of the arm-angle and snap-detect components is to let the user select and access distal targets, while the object-locator enables the user to interact with proximal targets using single- or multi-point interaction. Figure 4 illustrates an example scenario where each component is used in turn to select a distal target, move it closer to the user and then interact with it.

Fig. 3. 16 cameras along the floor enable object detection in front of the display wall

The arm-angle component processes images streamed from a single camera, with the purpose of identifying the angle at which a user is pointing. The camera is mounted in the ceiling, looking at the backs of users interacting with the display wall. The component delivers events to end-user applications, which interpret them according to their needs and current state.

The snap-detect component performs signal processing on audio from four microphones. The microphones are arranged in a rectangle in front of the display wall, with two microphones mounted near the ceiling at op-

[1] The event system is outside the scope of this paper.

posite ends of the canvas, and the other two near the floor. The component's goal is to detect a user snapping his fingers or clapping his hands in at least three of the four audio streams, allowing the signal's origin in 2D space to be determined using multilateration.

Multilateration uses the difference between the time at which snaps are detected by the different microphones to determine possible locations where the user snapped his finger. With a fixed sample rate, it is possible to determine the approximate, relative distance the sound has travelled from the time it was detected by a given microphone until it was detected by another. The resulting points lie on a hyperbolic curve with focus point at the given microphone's position. For many microphones, the intersections of the resulting curves can be used to determine the location of the user in 2D. For each positioned snap, an event containing the snap's 2D location and strength is delivered to end-user applications.

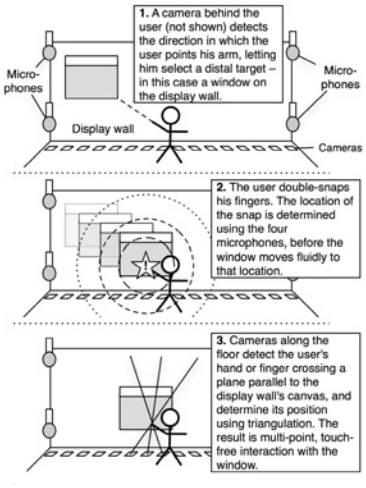


Fig. 4. The user selects a window, double-snaps to bring it closer, and interacts using the touch-free interface

The object-locator component uses 16 cameras to detect objects intersecting a plane parallel to the display wall's canvas, with typical objects being a user's fingers or hands. The cameras are connected pairwise to 8 computers, and mounted in a row along the floor looking at the ceiling (Figure 3). When three or more cameras see the same object, its position can be determined using triangulation. The component can detect several objects simultaneously. If the end-user application supports it, both multi-user and multi-point interaction is possible.

4 Implementation

The arm-angle component, implemented in C, uses a Canon VC-C4R pan-tilt-zoom camera connected to a framegrabber-card on a computer running Linux. The component captures frames at 8-9 frames per second (FPS). Each frame has a resolution of 720x540 pixels and is scaled down to half-size and converted from color to grayscale. Then the Sobel edge detector is used to locate edges in the image, before the equations of lines appearing in the image are determined using the Hough transform [4]. End-user applications receive events from the arm-angle component, which they use to determine the targets that the user is trying to select. For the traditional desktop interface, this is done by determining

the bounding rectangles of currently visible windows, and then intersect each window's bounds with the line indicating the user's pointing direction. Since the Hough-transform typically yields several potential lines for the arm and other straight edges, each line votes for the closest window it intersects with. A red square over the selected window highlights it to the user. The traditional desktop interface on the display wall is created using Xvnc [12,11], with each tile of the display wall showing its respective part of the entire, high-resolution desktop.

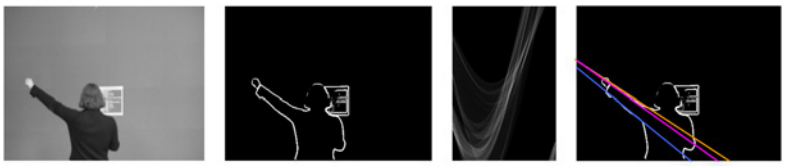


Fig. 5. The arm-angle image processing steps. (a) The input image, (b) the image after edge detection, (c) output from the Hough-transform, (d) the edge-detected image with lines extracted from the Hough-transform super-imposed.

The snap-detect component uses four microphones connected to a mixer. The mixer feeds the audio to a PowerMac G5, which uses a Hammerfall HDSP 9652 sound card to capture 4-channel audio samples at a rate of 48000 Hz. Samples from each channel are correlated with a template sound of a user snapping his fingers. When the correlation for a given channel exceeds an experimentally determined threshold, a snap is detected in that channel. The snap-detect component records a timestamp (essentially a sample counter) for each channel the snap is detected in. When a snap is detected in at least three channels, the resulting timestamps can be used to determine the user's location using the difference in time between when the snap is detected by the first microphone, and when it is detected by the remaining two or three microphones (multilateration). The location is determined by the intersection of conics created from different pairs of microphones, illustrated in Figure 7. For the desktop interface on the display wall, the selected window is moved to the location of the snap over a period of half

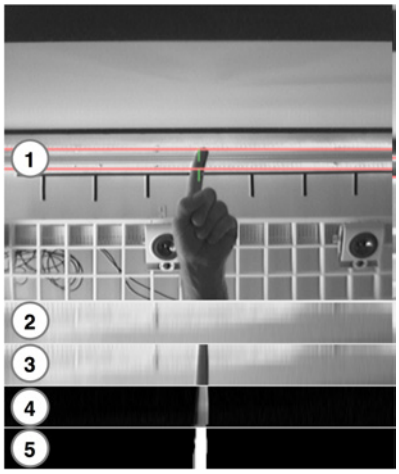


Fig. 6. (1) Image from camera. (2) Current background image. (3) Area of interest from current camera image. (4) The result by subtracting (3) from (2). (5) Result after thresholding the image.

a second. Changes to existing applications are not necessary to allow window movement by snapping fingers.

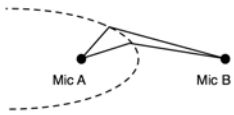


Fig. 7. The possible positions given the time difference between mic A and B are all located on the conic section.

The object-locator component is implemented in Objective-C, and consists of two modules: An image processing module, and a triangulation module. The image processing module runs in parallel on 8 Mac minis. Each Mac mini captures streaming video from two Unibrain Fire-i FireWire cameras in 640x480 grayscale at 30 FPS. Figure 6 illustrates the image processing done for a single camera. When the image processing module on a Mac mini starts up, it sets the first grabbed image to be the background image; the background is subsequently updated dynamically to deal with changing light conditions. For each new image captured, a narrow region of interest is isolated, before the background is subtracted from it. The result is then thresholded, yielding one-dimensional object positions and radii wherever pixel values exceed the dynamic threshold (the radius is set to the number of successive pixels above the threshold divided by two). The one-dimensional positions and radii are gathered by the triangulation module. For each camera, the triangulation module creates line segments that start at the camera's position, and pass through the center of each object detected by that camera. These lines are then intersected with the lines generated for the two cameras to the immediate left and right of the current camera. The resulting intersections are examined, and if two or more intersection points from three different cameras lie sufficiently close, an object is detected, as illustrated in Figure 8. The traditional desktop interface uses the 2D positions reported by the object-locator to control the cursor, and uses the radius to determine if the user wants to click.

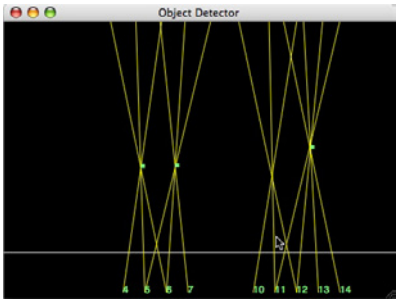


Fig. 8. The object-locator identifying three of four objects and their position

5 Early Deployment

The hybrid vision- and sound-based interface has been deployed in three different contexts: (i) A traditional desktop interface running on the display wall, (ii) a prototype whiteboard-style application supporting multiple users, and (iii) two previously commercial, but now open-source games (Quake 3 Arena and Homeworld). When used with a traditional desktop interface, the system enables

one user to select distal windows on the display wall, and move them closer by double-snapping his fingers. When the window is within reach, the user can interact with it using the touch-free interface. The cursor tracks the position of the user's hand or finger, and a click or drag can be invoked by tilting the hand (making it flat), as illustrated in Figure 9. Its use in the whiteboard-application is similar, where the distal objects drawn on screen can be brought closer by pointing at them and then double-snapping. In addition, the whiteboard brings up a tool palette at the user's location when a user single-snaps, allowing new objects to be added. The touch-free interface is used to support multi-user and multi-point interaction, for instance allowing users to resize objects by varying the distance between their hands.

The final context in which the system has been used is to play two games, Quake 3 Arena and Homeworld, further detailed in [15]. In this context, we only utilize the object-locator component, and to some extent the snap-detect component (people not playing the game can make the players fire their weapons in Quake 3 Arena by snapping their fingers, for instance). The touch-free interface provided by the object-locator enables users to play the two games using gestures only, rather than using traditional mouse/keyboard-based input.

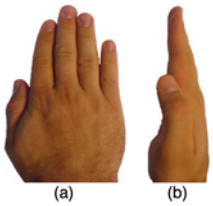


Fig. 9. (a) A flat and (b) vertical hand

6 Discussion

One principle employed throughout the design of the system is the following: Rather than identify what objects are, it is more important to identify where the objects are. The system at present does not distinguish between an arm pointing to give a direction, or a stick being used to do the same; nor does the system distinguish a snap from a clap or snap-like sounds made by mechanical clickers. This principle is also used to realize the touch-free interface - users can interact using their fingers, hands, arms, pens or even by moving their head through the (invisible) plane in front of the display wall.

This principle is not without issues, however, as false positives can occur for all of the components used in the system. For the arm-angle component, one issue is that the content currently on the display wall interferes with the edge-detection and the following Hough-transform to produce results that do not reflect the user's intentions. Another issue is that the resolution offered by the arm-angle component is too coarse to be used for selecting very small targets, such as icons. Techniques like drag-and-pop or the vacuum may be better suited for picking out small targets [1,2]. For the object-locator, the fact that it does not recognize what objects are means that it has no way of distinguishing several users from a single user interacting with several fingers or both hands. Although our ideal is to provide device-free interaction with the display wall for multiple, simultaneous users, the only component of the system that currently supports

multiple simultaneous users is the object-locator. We are currently working on ways to support multiple users with the arm-angle and snap-detect components as well. We are also investigating the accuracy of the three components, to better characterize their performance.

7 Conclusion

This paper has presented a system combining techniques from computer vision and signal processing to support device-free interaction with applications running on high-resolution, wall-sized displays. The system consists of three components utilizing in total 17 cameras and 4 microphones. It enables a user to select distal targets by pointing at them, bring the targets closer by double-snapping his fingers and finally interact with them through the use of a touch-free, multi-point interface. The system does not require the user to wear gloves or use other external devices. Instead, the system has been designed so as not to care exactly *what* a detected object is, but rather *where* the object – whatever it may be – is located. This in turn means that the system does not attempt to tell different users apart, for either of the three components. The interface has been deployed in three different contexts on the display wall: (i) One user at a time can interact with a traditional desktop interface, (ii) several users can interact simultaneously with a custom whiteboard-style application, with the caveat that the distal target selector only works for the user positioned near the center of the display wall, and (iii) up to three persons may play the games Quake 3 Arena and Homeworld simultaneously.

Acknowledgements

The authors thank Espen S. Johnsen and Tor-Magne Stien Hagen for their discussions, as well as the technical staff at the CS department at the University of Tromsø. This work has been supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

1. Baudisch, P., Cutrell, E., Robbins, D., Czerwinski, M., Tandler, P., Bederson, B., Zierlinger, A.: Drag-and-Pop and Drag-and-Pick: Techniques for Accessing Remote Screen Content on Touch- and Pen-operated Systems. In: Proceedings of Interact 2003, pp. 57–64 (2003)
2. Bezerianos, A., Balakrishnan, R.: The vacuum: facilitating the manipulation of distant objects. In: CHI 2005. Proceedings of the SIGCHI conference on Human factors in computing systems, pp. 361–370. ACM Press, New York (2005)
3. Dietz, P., Leigh, D.: DiamondTouch: a multi-user touch technology. In: UIST 2001. Proceedings of the 14th annual ACM symposium on User interface software and technology, pp. 219–226. ACM Press, New York (2001)

4. Duda, R.O., Hart, P.E.: Use of the hough transformation to detect lines and curves in pictures. Commun. ACM 15(1), 11–15 (1972)
5. Han, J.Y.: Low-cost multi-touch sensing through frustrated total internal reflection. In: UIST 2005. Proceedings of the 18th annual ACM symposium on User interface software and technology, pp. 115–118. ACM Press, New York (2005)
6. Igarashi, T., Hughes, J.F.: Voice as sound: using non-verbal voice input for interactive control. In: UIST 2001. Proceedings of the 14th annual ACM symposium on User interface software and technology, pp. 155–156. ACM Press, New York (2001)
7. Khan, A., Fitzmaurice, G., Almeida, D., Burtnyk, N., Kurtenbach, G.: A remote control interface for large displays. In: UIST 2004. Proceedings of the 17th annual ACM symposium on User interface software and technology, pp. 127–136. ACM Press, New York (2004)
8. Li, K., Chen, H., Chen, Y., Clark, D.W., Cook, P., Damianakis, S., Essl, G., Finkelstein, A., Funkhouser, T., Housel, T., Klein, A., Liu, Z., Praun, E., Samanta, R., Shedd, B., Singh, J.P., Tzanetakis, G., Zheng, J.: Building and Using A Scalable Display Wall System. IEEE Comput. Graph. Appl. 20(4), 29–37 (2000)
9. Mihara, Y., Shibayama, E., Takahashi, S.: The migratory cursor: accurate speech-based cursor movement by moving multiple ghost cursors using non-verbal vocalizations. In: Assets 2005. Proceedings of the 7th international ACM SIGACCESS conference on Computers and accessibility, pp. 76–83. ACM Press, New York (2005)
10. Gerald, D.: A camera-based input device for large interactive displays. IEEE Computer Graphics and Applications 25(4), 52–57 (2005)
11. RealVNC, Ltd. VNC for Unix 4.0. `http://www.realvnc.com/`
12. Richardson, T., Stafford-Fraser, Q., Wood, K.R., Hopper, A.: Virtual Network Computing. IEEE Internet Computing 2(1), 33–38 (1998)
13. Robertson, G., Czerwinski, M., Baudisch, P., Meyers, B., Robbins, D., Smith, G., Tan, D.: The large-display user experience. IEEE Comput. Graph. Appl. 25(4), 44–51 (2005)
14. Scott, J., Dragovic, B.: Audio Location: Accurate Low-Cost Location Sensing. In: Gellersen, H.-W., Want, R., Schmidt, A. (eds.) PERVASIVE 2005. LNCS, vol. 3468, pp. 1–18. Springer, Heidelberg (2005)
15. Stødle, D., Hagen, T.-M.S., Bjørndalen, J.M., Anshus, O.J.: Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays. In: Proceedings of the 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007, pp. 75–83 (June 2007)
16. Stolk, B., Wielinga, P.: Building a 100 Mpixel graphics device for the OptIPuter. Future Gener. Comput. Syst. 22(8), 972–975 (2006)
17. Valin, J.-M., Michaud, F., Rouat, J., Letourneau, D.: Robust sound source localization using a microphone array on a mobile robot. In: Proceedings of Interation Conference on Intelligent Robots and Systems (IROS), vol. 2, pp. 1228–1233 (October 2003)
18. Vogel, D., Balakrishnan, R.: Distant freehand pointing and clicking on very large, high resolution displays. In: UIST 2005. Proceedings of the 18th annual ACM symposium on User interface software and technology, pp. 33–42. ACM Press, New York (2005)

A.5 De-centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays

Citation

Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus. De-Centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays. In *NIK '07: Norsk Informatikkonferanse*, pages 53–64. tapir akademisk forlag, November 2007.

Abstract

This paper presents changes to the Virtual Network Computing (VNC) model to improve performance on wall-sized, high-resolution tiled displays. VNC does not fully utilize data distributed to the tiles, noticeably reducing interactive performance when panning images and moving windows. By de-centralizing the VNC model, the VNC viewers can exchange pixels amongst each other, improving performance. The VNC server changes from individually servicing viewer requests, to servicing the viewers once everyone has requested an update. The model is implemented, and its performance documented through experiments. When panning images, the number of pixels refreshed is increased three times or more, while reducing the server's bandwidth by 74% and CPU load by 35%. When moving windows, the number of pixels refreshed is increased by a factor of 1.8, while reducing the server's bandwidth by 68% and CPU load by 19.7%. The paper demonstrates how conceptually simple changes to VNC, while complex to realize, can yield significant performance improvements.

De-centralizing the VNC Model for Improved Performance on Wall-Sized, High-Resolution Tiled Displays

Daniel Stødle daniels@cs.uit.no
John Markus Bjørndalen jmb@cs.uit.no
Otto J. Anshus otto@cs.uit.no

Abstract

This paper presents changes to the Virtual Network Computing (VNC) model to improve performance on wall-sized, high-resolution tiled displays. VNC does not fully utilize data distributed to the tiles, noticeably reducing interactive performance when panning images and moving windows. By de-centralizing the VNC model, the VNC viewers can exchange pixels amongst each other, improving performance. The VNC server changes from individually servicing viewer requests, to servicing the viewers once everyone has requested an update. The model is implemented, and its performance documented through experiments. When panning images, the number of pixels refreshed is increased three times or more, while reducing the server's bandwidth by 74% and CPU load by 35%. When moving windows, the number of pixels refreshed is increased by a factor of 1.8, while reducing the server's bandwidth by 68% and CPU load by 19.7%. The paper demonstrates how conceptually simple changes to VNC, while complex to realize, can yield significant performance improvements.

1 Introduction

Using high-resolution, tiled display walls for visualization and collaboration is becoming increasingly popular. The high resolution and large physical size of display walls make them useful for visualizing data from many domains. Users often need to run applications written for standard desktop environments on the display wall, such as the weather forecasting application shown in Figure 1. Virtual Network Computing (VNC) [1], a remote desktop solution, is one way of achieving this. It is usually used to share regular-sized desktops, but for display walls, VNC can also be used to create a very high-resolution desktop. In the latter case, the desktop is maintained by a VNC server, which transmits

Figure 1: Weather forecasting on a tiled display wall with 28 projectors behind the canvas, using VNC to provide the desktop environment.

This paper was presented at the NIK-2007 conference; see http://www.nik.no/.

tiles of the desktop to corresponding clients (VNC viewers) running on a display cluster. Due to VNC's centralized approach to rendering and distributing pixels, it does not scale well to large display walls. With typical display walls ranging in resolution from 10 to 100 megapixels [2, 3] and beyond, a single complete refresh requires sending between 38 MB to 380 MB in total to the viewers.

This paper presents De-centralized VNC (DVNC). DVNC modifies the VNC model, allowing viewers to exchange pixels when screen content moves, but is not otherwise modified. Cases where this happens include panning large images, moving windows on the desktop or scrolling in windows. Work is delegated to the viewers, letting the server focus on sending new pixels rather than resending already transmitted pixels to the viewers. When the viewers receive pixels from both the server and each other, the correct ordering of display updates becomes important in order to preserve consistency of the display.

DVNC was implemented by modifying an open-source version of VNC [4], and its performance measured by comparing it to the original on a tiled display wall with a total resolution of 7168x3072 pixels. As a result, we found interactive performance to be significantly better when panning images and moving windows. The main contribution is the modified VNC model, where viewers go from being passive receivers of pixels, to become active participants in distributing pixels.

2 Related work

There has been much work on improving the performance and utility of VNC [1], including new compression techniques [5, 6] and support for 3D acceleration [7]. This paper is not focused on these aspects of VNC. DVNC instead aims at improving performance when using VNC to create a desktop on tiled display walls, by delegating work to viewers. In THINC [8], performance is improved compared to VNC and other remote desktop solutions by efficiently encoding and transferring raw graphics operations generated by applications. THINC is focused on thin-client usage, and is currently not suitable for use in creating desktop environments for display walls as it can only export desktops with the same resolution as the computer it is running on has.

Microsoft Remote Desktop and the X Window System [9] (X11) are two other ways of accessing or creating desktops over the network. Both approaches use drawing operations ("draw line", "draw string", and so on) to achieve good performance. The former is limited to a maximum resolution of 4096x2048, and does not allow different regions to be displayed by different viewers. Xdmx [10] can be used to enable X11 application to run on a tiled display wall. Xdmx acts as a proxy to a set of X servers running on the display cluster. This differs from DVNC in that no data is exchanged between the different X servers on the display cluster to improve performance. DVNC uses a single X server to render into a virtual framebuffer, which is then distributed to the tiles using the VNC protocol.

SAGE [11] is a system for streaming high-resolution graphics from rendering or storage clusters to one or several display walls. Pixel data is received by "SAGE Receivers" and then displayed. While this can be used to display multiple VNC desktops at once, no pixel data is exchanged between the different SAGE Receivers.

3 Model and design

When VNC is used on standard displays, a single viewer typically has access to the pixels for the server's entire desktop. On a tiled display wall, each viewer runs on its own computer showing a small region of the server's desktop, as shown in Figure 2 (a) and (b). In the original VNC model, the server does all the work. The viewers do nothing except receive and display pixels. In DVNC, this model is modified by letting the viewers exchange pixels amongst each other for a certain class of update operations. The purpose of this is to reduce the server's load and improve end-user performance. The viewers go from being passive receivers to being active slaves in a master-slave relationship to the server. Figure 2 (c) illustrates this change.

VNC uses the Remote Framebuffer (RFB) protocol [12] to send display updates from the server to the viewers. Viewers request the area they are interested in from the server, which responds with update operations for that area. The RFB protocol uses three operations to update a region of the display: Image Rect, Fill Rect and Copy Rect. The Image Rect operation contains a rectangular set of pixels which is drawn by the viewer at the location indicated by the rectangle. The Fill Rect operation is used to fill a rectangle with a given color. The Image and Fill Rect operations offer no obvious ways for distributing network load.

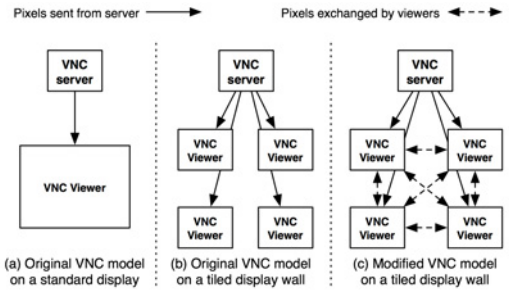


Figure 2: The original VNC model for (a) a standard display, (b) a tiled, 2x2 display wall. In the modified model (c), the viewers exchange pixels with each other in addition to receiving pixels from the server.

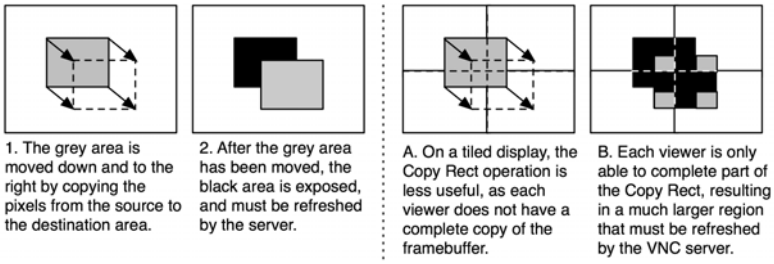


Figure 3: The Copy Rect operation as it is used for a single viewer for the entire remote desktop, and its behaviour when used with multiple viewers each showing a region of the server's desktop on a tiled, high-resolution display.

The Copy Rect operation is used whenever an area of the screen is moved, but the pixels inside the area remain unchanged, shown in Figure 3 (1) and (2). This is common when moving windows, scrolling in documents or panning images. Since a Copy Rect only takes 12 bytes to send regardless of the size of the area being updated, the Copy Rect

operation is important for reducing the server's bandwidth usage. To make the best use of it, the viewer must have access to all of the pixels being moved for the entire desktop. On a tiled display, this is not the case, as each viewer only has the pixels covering its own area of the display. Copy Rect operations that span the areas of more than one viewer force the server to split the operation, resulting in a larger set of exposed areas, shown in Figure 3 (A) and (B). This incurs additional load on the server, which DVNC alleviates by letting the viewers themselves exchange the necessary data. Figure 4 illustrates this.

In the de-centralized VNC model, the viewers receive updates not only from the server, but also from each other. This creates consistency issues, both for the viewer's own display, and for pixels sent by the viewer to other viewers. For instance, a viewer receiving one update from the server and a second update from a different viewer, needs to know which of the two updates to apply first in order to ensure a consistent display. In DVNC, the consistency issues are solved by imposing a total ordering on all updates sent by the server to the viewers, and by ensuring that all the viewers see the same Copy Rect operations.

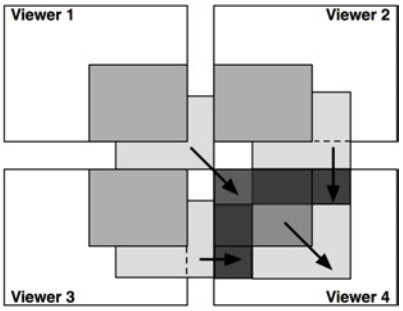


Figure 4: A Copy Rect operation spanning four viewers. The darkened, grey area moves down and to the right. Viewers 1, 2 and 3 transfer some of their pixels to viewer 4. (Other pixel transfers are not shown, such as from viewer 1 to viewer 2.) The arrows indicate direction of movement.

The viewers make independent decisions about where to send pixel data based on the Copy Rect operations they receive. Pixel data is always pushed to other viewers. To avoid circular dependencies between different viewers, the Copy Rect operation is split into two phases: A Copy Rect pre-phase, and a Copy Rect post-phase. During the pre-phase, the viewer determines which viewers it should send pixels to, copying and sending data as necessary. During the post-phase, a viewer applies updates from other viewers in the correct order.

4 Implementation

DVNC was implemented by modifying RealVNC's free VNC distribution (available under the GPL license), version 4. Both the VNC server and the VNC viewer required modifications. There are many implementations of VNC, including TightVNC, UltraVNC, and others. Since the modifications involve changing the model, they could also have been implemented by modifying a different VNC implementation.

VNC server modifications

The VNC server was modified to ensure that all connected viewers see the same Copy Rect operations in the same order. Instead of accumulating updates for each viewer, the server accumulates the same set of updates for all viewers, sending them once all the viewers have requested an update. The drawback to this approach is that the server can provide updates no faster than the slowest viewer. When updates are sent, the Copy Rect operation's rectangle is no longer clipped to the area which the viewer requests from the server, but sent regardless of whether the Copy Rect actually intersects with the viewer's

area. The server continues to clip Fill and Image Rect operations to the area requested by the viewer. A 4-byte, logical timestamp was added to the RFB protocol's framebuffer start message. The logical timestamp is incremented once for each group of update operations, and is used by the viewers to match updates received from other viewers to the correct Copy Rect operation. Finally, the server was modified to measure its load during the various experiments.

VNC viewer modifications

The VNC viewer was modified to receive framebuffer updates from other viewers. A separate thread is responsible for sending and receiving pixel data to and from other viewers. This thread also handles the logic necessary to determine which pixels should be sent and received, as well as the order in which updates are applied. Also, both the original and modified viewers were changed to record various statistics used for the experiments.

To better overlap communication with computation, incoming update operations from the server are queued. If the operation to be queued is a Copy Rect, its pre-phase is executed before queueing it (in some cases, execution of the pre-phase may be delayed to ensure consistency). The pre-phase copies data and sends it to other viewers, increasing the chance that other viewers will have the data they need when they begin executing the Copy Rect's post-phase.

When the server signals that it is done sending updates, all the queued operations are applied by the viewer. Applying a Copy Rect operation is done by executing its post-phase. During the post-phase, the viewer scans its list of updates received from other viewers, matching them to the current Copy Rect using the VNC server timestamp and other data contained by the operation. If the viewer hasn't received all the necessary data from other viewers, it will block waiting for the remaining data to arrive.

The queueing strategy introduces a queueing overhead not present in the original implementation. To minimize this overhead, the modified viewer avoids queueing when possible. If the rectangle covered by the incoming operation does not overlap with the rectangles of any queued operations, the operation can be applied immediately.

Determining where a viewer sends its pixels for a given Copy Rect operation is done by examining the data given by each Copy Rect operation. A Copy Rect operation consists of a source rectangle R=(x, y, width, height) and a delta point (dx, dy). The delta point indicates where the pixels identified by the source rectangle should be moved, yielding a destination rectangle. The viewer intersects the source rectangle with its own area. If the intersection is non-empty, the destination rectangle is computed by offsetting the clipped source rectangle by the operation's delta point and intersecting the result with the viewer's area. If the source and destination rectangles have different sizes (indicating that part of the destination rectangle falls outside the viewer's area), the viewer will transmit some of its data to other viewers.

When the viewer starts up, it is given the area of the VNC desktop that it should display as part of its arguments. The viewer then connects to the server and to the all other viewers by means of a multicast discovery mechanism. When a connection to another viewer is established, the viewers exchange a handshake, before they can exchange framebuffer updates. The handshake consists of five long integers: A magic number followed by the area the viewer covers. All future messages consist of a four-byte field containing the length of the message, followed by the actual message itself. These messages consist of the VNC server timestamp, the rectangle and delta point from the Copy Rect operation, followed by the pixels for the update. The pixels are currently not compressed.

5 Experiments

The performance of the original VNC and modified DVNC implementations is measured using three metrics: Total number of pixels refreshed, total number of bytes sent from the server to the viewers, and the server's CPU load. A high pixel refresh count is better than a low refresh count, as more pixels updated means better interactive performance. The DVNC implementation is also expected to reduce bandwidth used by the server, and reduce the server's CPU load. This is because the modified model is based on distributing load from the server to the viewers.

Hardware and software setup

The hardware used was (i) a display cluster with 28 nodes (Intel Pentium 4 EM64T, 3.2 GHz, 2 GB RAM, HyperThreading enabled, running the Rocks cluster distribution 4.0) connected to 28 projectors (1024x768, arranged in a 7x4 matrix), (ii) switched, gigabit Ethernet, (iii) a dual Intel Xeon 3.8 GHz with 8 GB RAM, and (iv) another Pentium 4 (same hardware as the nodes in the display cluster). The Xeon and the last Pentium 4 were used to run the server, and ran RedHat Enterprise Linux 4. The image viewer used was "xloadimage" by Jim Frost. The event generator for the control experiments used the XTestExtension to post input events to the server, and was custom-made for these experiments. The VNC distribution was RealVNC version 4 [4], exporting a 16-bit desktop.

Server and viewer instrumentation

The original and modified servers were instrumented to record their CPU load over the duration of an experiment, recording both time spent at user level, and time spent on behalf of the servers at kernel level. The servers recorded 10 samples per second, sending performance data to a second computer on the same local network. The additional network traffic generated by sending performance data is negligible at less than 500 bytes per second.

The original and modified viewers were instrumented to record the statistics outlined in Table 1. Each viewer makes its own measurements. At the end of each experiment, the number of pixels refreshed and number of bytes exchanged is summed. The queueing overhead's global maximum and minimum values are determined, and the global queueing overhead average is calculated by averaging the averages from each viewer. The total number of bytes sent between the viewers was also recorded, but these data have not been used to characterize performance in this paper.

Name	Description
Total Pixels	Total number of pixels refreshed by this viewer.
Server Bytes	Total number of bytes received by this viewer from the server.
Viewer-to-viewer Bytes	Total number of bytes received by this viewer from other viewers.
Queuing Overhead	Minimum, maximum and average overhead caused by queueing incoming operations.

Table 1: Statistics gathered from the viewers.

Experiments and methodology

Two sets of trace experiments and a set of control experiments were conducted. The trace experiments aim at measuring the performance for a user interacting with the desktop. In particular, the answers to the following four questions were of interest: (i) How many

more pixels can the DVNC implementation refresh compared to VNC? (ii) How much bandwidth does DVNC save? (iii) How does the DVNC changes affect the server's load? (iv) How big is the queuing overhead? The trace experiments play back two recorded user traces, where a user either pans an image or moves a window (see Table 2). In the first set of trace experiments, the server ran on the Xeon, and in the second set, the server ran on the Pentium 4.

Trace	Description
Image pan	A user pans an image sized at 9372x9372 pixels. The visible portion of the image covers almost the entire display wall, the rest of which is covered by the image viewer's window decorations. The trace lasts for 255 seconds.
Window move	A window sized at 2592x1944 pixels is moved around on screen. The trace lasts for 145 seconds.

Table 2: The traces used for measuring performance.

The control experiments have two purposes: (i) Get an objective view of the system's performance, and (ii) measure the maximum performance gain in a situation where the server's possibility for using Copy Rect operations is near maximized. The image from the Image pan trace is moved vertically up and down in a controlled manner. The rate at which movement occurs is varied for each experiment, ranging from one to fifty times per second, with each movement scrolling the picture 8 pixels up or down. An event generator is used to move the image at the constant rate defined by each experiment, with each experiment lasting 30 seconds.

Where the trace experiments measure the system's performance in a setting similar to real-world use, the control experiments allow for external repeatability. Before running either trace or control experiments, the server was restarted, and its desktop configured to match the experiment's starting point (open windows and window positions on the desktop). Then the viewers were restarted, and the experiment was conducted, before performance data was gathered.

A null-benchmark measured the overhead incurred by the changes to the VNC protocol. The server displayed a static image, and the number of bytes required to refresh the viewers was measured. The original sent a total of 85688.97 KB, while the modified sent 85707.69 KB - an overhead of 0.02%.

Trace results

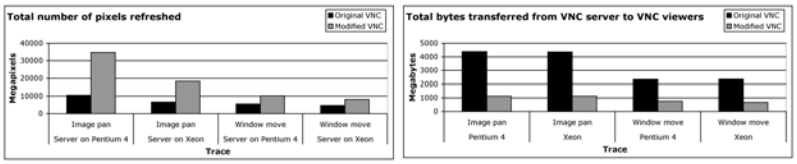


Figure 5: Left: Total number of pixels refreshed for each trace by the original and modified VNC viewers. Right: Total number of bytes sent by the server to the viewers.

Figure 5 shows the total number of pixels refreshed by the original and modified viewers as well as bytes sent from the server to the viewers for each trace. With the server on the Pentium 4, the modified implementation refreshes 34.6 gigapixels (GPx) for the Image pan trace, 3.29 times more than the original's 10.5 GPx. For the Window move

trace, the modified implementation refreshes 1.83 as many pixels. The number of bytes sent is reduced by 74% for the Image pan trace, and by 68% for the Window move trace. On the Xeon, number of pixels refreshed increases from 6.5 to 18.2 GPx and 4.6 to 8.1 GPx for the two traces respectively. Interestingly, the Pentium 4 is able to refresh almost twice as many pixels as the Xeon for the Image pan trace. The amount of data transferred is approximately the same regardless of where the server runs.

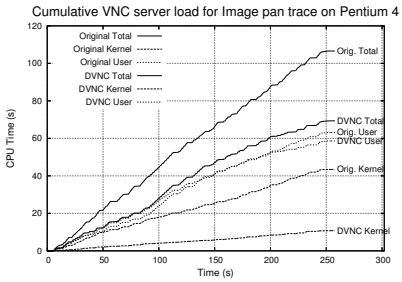


Figure 6: Cumulative server CPU load for the Image pan trace on Pentium 4, with total, user and kernel level load for both implementations.

Figure 6 shows the original and modified servers' cumulative CPU load, measured in seconds, when running on the Pentium 4 for the Image pan trace. The X axis shows the running time of the trace, and the Y axis shows the CPU time consumed by the server. The DVNC server's load is reduced by 35% compared to the original VNC server (from 106.6 to 69.3 CPU seconds). The biggest reduction happens at kernel level, where the load is reduced by 75%, while the difference in user level load is only 4%. The reduction in kernel level load correlates well with the reduction in bandwidth used by the DVNC server. For the window move trace, the reduction in CPU load is 19.7% (from 64.8 to 52.0 CPU seconds). The server load on the Xeon has similar characteristics.

Table 3 shows the modified implementation's queuing overhead. The maximum queuing overhead is 0.56 seconds, which means that an update operation received by one of the viewers was queued for a little over half a second before it was actually drawn. The average queuing overhead is between 0.008 and 0.011 seconds, and the minimum overhead is 0.000 seconds.

Trace	**Min**	**Avg**	**Max**
Image pan *(P4)*	0.000 s	0.009 s	0.546 s
Window move *(P4)*	0.000 s	0.008 s	0.467 s
Image pan *(Xeon)*	0.000 s	0.011 s	0.582 s
Window move *(Xeon)*	0.000 s	0.011 s	0.504 s

Table 3: The queueing overhead, measured in seconds, for the traces on the Pentium 4 and the Xeon.

Control experiment results

Figure 7 shows the number of pixels refreshed by the viewers for the control experiment. The measured values are compared to a target number of pixels that should have been refreshed if sufficient resources to avoid all bottlenecks were available. The target value is calculated by measuring the number of pixels refreshed when scrolling the image vertically by 8 pixels, and multiplying that number with the duration of each experiment and rate at which the image is moved.

The number of pixels refreshed increases linearly with the event generation rate. At first, both implementations closely follow the target refresh count. The original implementation reaches its maximum at an event rate of 26, while the modified implementation keeps tracking the target up to 40 events per second. The original's performance goes down by 57.8% when the event generation rate is increased from 26 to 28. The DVNC performance goes down by only 6.6% when increasing the event generation rate from 40 to 45. At an event rate of 50, DVNC refreshes 11.9 times as

many pixels as the original.

Figure 8 shows the server's total, kernel and user level load in percent for both implementations. Initially, the CPU load increases linearly for the implementations, with the original's load increasing almost twice as fast as the modified's load. At the peak in load, close to 100%, the event rate for the original and modified server is respectively 26 and 40. This is also the rate at which the two implementations peak in number of pixels refreshed.

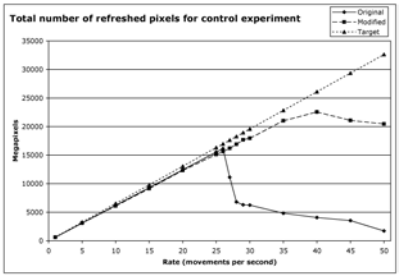


Figure 7: Total number of pixels refreshed for the control experiment for the two implementations, as well as the target refresh count. Event generation rates range from 1 to 50.

Figure 9 shows the total number of bytes transferred from the server to the viewers. The number of bytes transferred increases linearly with the event rate, with a slower growth for the modified implementation. The original implementation peaks at 1135 MB, while the modified implementation peaks at 375 MB. This corresponds to a bandwidth use of 37.8 MB/s and 12.5 MB/s, respectively, neither of which is close to the maximum transfer rate of gigabit Ethernet at about 90 MB/s. Interestingly, the bandwidth used by the original implementation continues to climb even after having peaked both in CPU load and number of pixels refreshed.

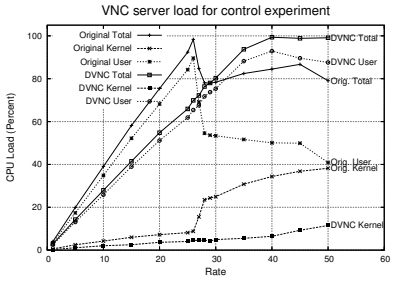


Figure 8: CPU load for the VNC server, showing total, kernel, and user level load for both the original and modified implementations in the control experiment.

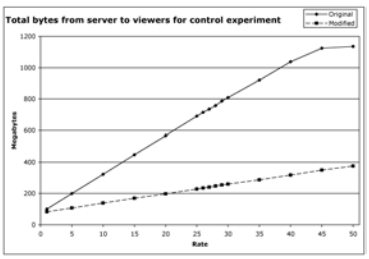


Figure 9: Total bytes sent from the servers for the control experiment.

6 Discussion

The results from the trace experiments show that the DVNC implementation can refresh more than three times as many pixels compared to the original. The control experiment documents that DVNC can outperform the original by a factor of up to 11.9. For the case where the server sends no Copy Rect operations - and hence no gain can be expected from delegating work to the viewers - DVNC adds very little overhead; only 0.02% in the null-benchmark. DVNC provides no performance benefit for content that is updated using

operations other than Copy Rect - typically video, animated or otherwise “fresh” content. DVNC provides a significant performance increase for certain operations that the server is able to translate into Copy Rect operations.

Trace experiments

The server spends less CPU time at kernel level since it sends less data, leaving more resources for the server and other applications. The server’s user level load is not reduced as much, since the server provides viewers with more frequent updates, while sending less data.

The maximum queueing overhead was half a second, and can be observed as occasional stutters during playback of the traces. Even though there is some queueing overhead associated with keeping each viewer consistent, overall performance is still much better than the original implementation.

The staircase effect in Figure 6 is caused by periods of lower user activity. When the user is not moving the image or the window, fewer updates take place and the server experiences low load. Typically, this occurs when the user repositions the cursor to drag the image or move the window. This effect is not present in graphs depicting the CPU load for the control experiments (not included in this paper).

Control experiments

DVNC’s performance compares even more favorably to the original in the control experiments, than it did in the trace experiments. The reason for this is that the pay-off from each Copy Rect generated in the control experiments is greater than it is in the trace experiments. In the trace experiments, many Copy Rects move diagonally. Diagonal movements cause the areas covered to be smaller, meaning that larger areas must be refreshed by the server. New pixels must be sent by the server to refresh not only the top or bottom edge, but also the right or left edge of a given area. Diagonal movements also cause more complicated dependencies between the different viewers when they exchange pixels. A vertical or horizontal Copy Rect operation only requires that a viewer sends its pixels to one other viewer, while a diagonal Copy Rect can require a viewer to send pixels to three different viewers.

The original implementation’s sudden drop in pixel refresh count (Figure 7) is not caused by lack of network bandwidth, as the bandwidth used continues to increase even after the drop in refreshed pixels (Figure 9). The drop is caused by the server having to work harder to keep its own framebuffer updated, which delays updates to the viewers. The delay makes each viewer accumulate a larger dirty region, requiring more bytes to refresh. This behaviour also explains why the original starts spending more time at kernel level when the drop in performance occurs, as the kernel is heavily involved in the communication.

Server on Pentium 4 and Xeon

The performance measured by the trace experiments on the Pentium 4 and on the Xeon were not as expected. The Pentium 4, with its older CPU architecture, performed better than the newer Xeon. The server implements Copy Rect by moving memory from one location to a different location in the server’s framebuffer. To investigate whether memory bus speeds were the issue, the two computers’ processor-memory bandwidth was measured using CacheBench [13]. The sustained read/modify/write bandwidth to

memory for the Pentium 4 was 3.78 GB/s, while the Xeon only managed 2.16 GB/s. This is a factor of 1.75, which correlates well with the difference in refreshed pixels, 34 GPx vs. 18 GPx, a factor of 1.88.

Lessons learned

The performance improvements achieved by DVNC is made possible by changing the model at a number of different levels. From a model where the server does all work and the viewers are passive receivers, the new model makes the viewers partially serve each other, off-loading the server. The viewers, which previously needed no knowledge about other viewers, now need to know about every other viewer in order to exchange pixels with them. Each viewer makes its own decisions about where to send pixels, as opposed to having the server handle this task.

Discovering that the server's memory bandwidth is a bottleneck was surprising, given that the pixels moments later must be moved over a "slow" gigabit Ethernet. This is becase the server may have to move up to 80 MB of data before sending a Copy Rect operation describing the movement, while the A Copy Rect operation itself only requires 12 bytes to transfer.

7 Conclusion

This paper has presented a modification to the VNC model that improves performance when VNC is used to create the desktop environment for tiled display walls. The Decentralized VNC (DVNC) system increases performance for tasks like navigating large images and moving windows on the desktop. The main principle employed is to let the VNC viewers exchange data amongst each other, freeing the VNC server from re-sending already distributed pixel data.

The DVNC model has been implemented by modifying an open-source VNC implementation, and its performance evaluated. A tiled 7x4 display wall with a total resolution of 7168x3072 pixels was used for the experiments. The system's performance was measured through several user trace and control experiments, and compared to VNC without modifications. The results show that end-user performance was significantly improved. For panning large images, DVNC could refresh three to twelve times more pixels on the display wall compared to the original implementation. These improvements are expected to carry over to other cases where screen content moves, but otherwise remains unchanged, such as scrolling in documents.

The performance improvements are a result of distributing work between the server and viewers. This lets server-side processing overlap with viewer-side pixel distribution. In addition, the bandwidth required for the server to keep the viewers updated is reduced. Consequently, the server can spend more cycles keeping its framebuffer updated, as well as leaving more cycles for other applications.

8 Acknowledgements

The authors wish Lars A. Bongo and Espen S. Johnsen. This work has been supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

[1] Tristan Richardson, Quentin Stafford-Fraser, Kenneth R. Wood, and Andy Hopper. Virtual Network Computing. *IEEE Internet Computing*, 2(1):33–38, 1998.

[2] 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 Comput. Graph. Appl.*, 20(4):29–37, 2000.

[3] Bram Stolk and Paul Wielinga. Building a 100 Mpixel graphics device for the OptIPuter. *Future Gener. Comput. Syst.*, 22(8):972–975, 2006.

[4] RealVNC, Ltd. VNC for Unix 4.0. http://www.realvnc.com/.

[5] Lars Ailo Bongo, Grant Wallace, Tore Larsen, Kai Li, and Olga Troyanskaya. Systems support for remote visualization of genomics applications over wide area networks. In *Proc. of GCCB'06. LNBI 4360*, 2006.

[6] Tony Lin, Pengwei Hao, Chao Xu, and Ju-Fu Feng. Hybrid image coding for real-time computer screen video transmission. Januar 2004. Visual Communications and Image Processing (VCIP) 2004, part of the IS&T/SPIE Symposium on Electronic Imaging 2004.

[7] dcommander. VirtualGL. http://www.virtualgl.org.

[8] Ricardo A. Baratto, Leonard N. Kim, and Jason Nieh. THINC: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2005. ACM Press.

[9] Robert W. Scheifler and Jim Gettys. The X window system. *ACM Trans. Graph.*, 5(2):79–109, 1986.

[10] R. E. Faith and K. E. Martin. Xdmx: Distributed, multi-head X. http://dmx.sourceforge.net/.

[11] Byungil Jeong, Luc Renambot, Ratko Jagodic, Rajvikram Singh, Julieta Aguilera, Andrew Johnson, and Jason Leigh. High-Performance Dynamic Graphics Streaming for Scalable Adaptive Graphics Environment. SuperComputing 2006, 11.-17. November 2006.

[12] Tristan Richardson. The RFB Protocol, version 3.8.

[13] Philip J. Mucci. Low-level characterization benchmarks. Available from http://icl.cs.utk.edu/projects/llcbench/index.html.

A.6 Blurring the line between real and digital: Pinning objects to wall-sized displays

Citation

Daniel Stødle and Otto J. Anshus. Blurring the line between real and digital: pinning objects to wall-sized displays. In *IPT/EDT '08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologies*, pages 1–5, New York, NY, USA, 2008. ACM.

Abstract

Billboards are everywhere, enabling users to interact by leaving documents, images, ads or clippings for others to see. There is currently no simple and transparent way to replicate this interaction pattern in a wall-sized display context. Users must first employ devices like scanners or digital cameras to digitize the content they wish to share. Then the digitized content must be manually transferred to some computer, before the user can display and arrange it on the desktop. This paper presents a system that supports the classic billboard interaction pattern in a display wall context. The user briefly holds the content to digitize anywhere in front of the display wall, and an image of it appears at the same location. The system comprises a 6x3 m high-resolution wall-sized display, a gesture-based human-computer interface and a ceiling-mounted steerable camera, which together enable transparent and low latency object imaging.

Blurring the line between real and digital: Pinning objects to wall-sized displays

Daniel Stødle*
Department of Computer Science
University of Tromsø, Norway

Otto J. Anshus†
Department of Computer Science
University of Tromsø, Norway

Figure 1: (a) A user pinning a document to a combined white- and billboard. (b) A user pinning a document to the Wallboard. The document is held at the location where the user wants it to appear. (c) The content appears on the display wall. (d) The user removes the physical content, leaving the digitized version behind.

Abstract

Billboards are everywhere, enabling users to interact by leaving documents, images, ads or clippings for others to see. There is currently no simple and transparent way to replicate this interaction pattern in a wall-sized display context. Users must first employ devices like scanners or digital cameras to digitize the content they wish to share. Then the digitized content must be manually transferred to some computer, before the user can display and arrange it on the desktop. This paper presents a system that supports the classic billboard interaction pattern in a display wall context. The user briefly holds the content to digitize anywhere in front of the display wall, and an image of it appears at the same location. The system comprises a 6x3 m high-resolution wall-sized display, a gesture-based human-computer interface and a ceiling-mounted steerable camera, which together enable transparent and low latency object imaging.

1 Introduction

Wall-sized displays are becoming ever more common, with resolutions ranging from 10 to 100 megapixels and beyond [Li et al. 2000; Stolk and Wielinga 2006]. Display walls are typically built using a cluster of computers driving a set of tiled displays or projectors. Our display wall is built using 28 projectors and computers arranged in a 7x4 grid, forming a 22 megapixel, 7168x3072 display covering an area of 6x3 m.

There has been much work on moving whiteboard-style interaction to the realm of wall-sized displays, with commercial products such as the SMART Board [SMART Technologies] available. However, one fundamental issue that has yet to be addressed is making billboard-style interaction possible. On a billboard, users are less concerned about drawing or writing, and care more about leaving content of some kind behind for other users to see. This is typically done by simply fixing a document, news clipping, picture, advertisement or similar to the wall using pins, staples or magnets, as shown in Figure 1 (a).

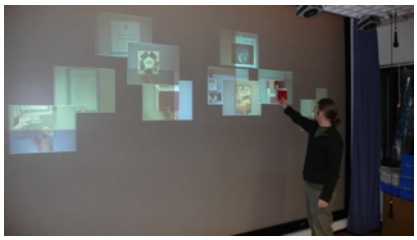


Figure 2: The entire display wall being used in a billboard-like fashion.

The equivalent steps in current systems reduce to first digitizing the relevant content, either using a scanner or a digital camera. Then the content must be transferred in some way, with multimedia MMS messages, e-mail or BlueTooth file transfer among the many ways of doing this. Once transferred, the content must be brought up on the display wall, and manually placed at a location determined by the user. The entire process is time-consuming and requires a knowledgeable user.

*e-mail: daniels@cs.uit.no
†e-mail: otto@cs.uit.no

IPT/EDT 2008, Los Angeles, California, August 9–10, 2008.
© 2008 ACM 978-1-60558-211-5/08/0008 $5.00

This paper presents the design and implementation of Wallboard, a system that replicates the billboard-style interaction pattern. To achieve this, there are three important requirements that must be satisfied. (i) A user should not need to employ any devices, wear special gloves or be fitted with markers in order to "pin" content to the display wall, as the interaction should be as direct as it would be on a regular billboard. (ii) The content should appear on the display wall where the user is holding it, in order to match the behaviour of pinning content to a billboard. The user should be able to pin content anywhere on the display wall, and not be restricted to some designated region. (iii) As users expect to pin content to a billboard instantaneously, the time required to pin content to the display wall should also appear instantaneous to the user. Figure 1 (b)-(d) shows a user pinning a document to the Wallboard, and Figure 2 shows a large part of the display wall in use for imaged objects. Users are free to move and scale imaged objects once they appear on the display wall.

The main contribution of this paper is Wallboard, a scalable system for transparently imaging objects on wall-sized displays. The system is not limited to imaging objects, but also demonstrates how user content can be augmented with other kinds of data, including voice annotations and sensor measurements describing the content, captured in the moments preceding the action of pinning content to the display wall. The system demonstrates how the act of knowing *where* something is can sometimes be far more powerful than being able to identify exactly *what* that something is, while at the same time being a less complex and computationally expensive problem to solve.

2 Related work

There has been much work on creating digital whiteboards. In general, most of it has focused on ways of augmenting whiteboards with already-existing digital content, sharing content between different whiteboards (virtual or real), interaction styles or ways of supporting content creation. There has been little to no focus on making whiteboards act more like billboards, and in particular the act of pinning objects to the board.

The Xerox Liveboard system [Elrod et al. 1992] was one of the first digital whiteboards, upon which applications like Tivoli [Pedersen et al. 1993] were built. The Xerox Liveboard work identified aspects like image resolution as important to users, but also mentioned the need to "add [a] scanner." The Xerox Liveboard differs from our system in that it does not incorporate content from the environment into its applications.

In their work on Tangible Bits [Ishii and Ullmer 1997], the authors introduce the transBOARD. The transBOARD is a regular whiteboard augmented with sharing and storage capabilities through the use of a stroke recorder to store whiteboard contents. Physical content can be incorporated through the use of "phicons," barcode-tagged objects which represent real or virtual objects. This differs from Wallboard in that physical objects must be "attached" to such phicons before they can be used, and even then, do not actually appear on the transBOARD, but rather on a digital replica on a display nearby. Wallboard allows users to image any object, without manually having done so prior to pinning it to the display wall.

Mynatt et.al created Flatland [Mynatt et al. 1999], an augmented whiteboard intended for use in offices. The Immersive Whiteboard [Shae et al. 2001] is an attempt at bridging a physical whiteboard with a virtual counterpart. A video camera is used to create an avatar of the user, but can not be used to share other physical content, like documents or images.

There are many examples of surfaces that are active in the sense that they enable the inclusion of physical objects. The AMLCD panel [Abileah and Green 2007] enables the screen itself to scan documents, but is limited to capturing grayscale images of objects very close to the display. Microsoft's Surface [Bathiche and Wilson 2007] is a multi-touch enabled table that can sense devices like mobile phones and transfer images from the devices. Apart from not aiming to be a billboard, Microsoft Surface differs from Wallboard in that it is not possible to image arbitrary objects and have them appear on the surface. The EnhancedDesk [Koike et al. 2001] can recognize tagged documents placed on its surface, and augment the documents with interactive content. Recognition is done using a camera that looks for a matrix-code printed on the content to scan. No attempt is made at directly incorporating the imaged content; instead content must be tagged and recognized by the system. In [Klemmer et al. 2000], the authors demonstrate a desk that can digitize post-it notes. Their implementation only works with post-it notes, and requires the user to actually write the note on the digital desk. Our system can accomodate any content on a surface that is much larger than the desk demonstrated in [Klemmer et al. 2000].

Multi-touch and multi-point interaction has been an active field of research for several years, with commercial products like the Apple iPhone and Microsoft Surface available. There are many approaches to implementing multi-touch interfaces, including the use of electric capacitance in the Diamondtouch tabletop [Dietz and Leigh 2001], use of total internal reflection of infrared light [Han 2005], and the optical approach taken in [Morrison 2005]. The first two approaches require users to touch the canvas or screen. The device-free input system built for the Wallboard uses a touch-free approach, making for a cleaner surface and enabling the use of a flexible canvas. The approach taken by the SMART Board [Morrison 2005; SMART Technologies] is the one most similar to ours, but differs in its use of fewer and custom cameras with on-chip processing to perform object detection.

3 Design and implementation

Figure 3 illustrates the overall system architecture. The system comprises five major components: (i) a device-free input system, (ii) input analysis, (iii) camera and sensor control, (iv) the Wallboard application and (v) the Shout event system. Figure 4 illustrates how the various components are deployed.

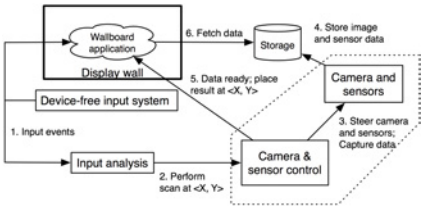


Figure 3: The overall system architecture and design.

3.1 Device-free input system

The device-free input system is used to enable multi-point, multi-user interaction with different applications running on the display wall, including Wallboard. The input system is built using 16 cameras mounted along the floor in front of the display wall's canvas. Images from each camera are analyzed in order to determine the location of objects intersecting two planes parallel to the display wall's canvas - the input system's "region of interest." Typical objects include hands, fingers or arms, although the interface does not distinguish between the different objects other than reporting different object radii. The intersection with each plane is found using

triangulation, as shown in Figure 5.

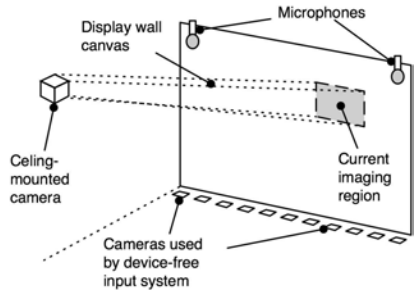


Figure 4: Schematic of the system deployment. Cameras are mounted along the floor to enable device-free interaction with the display wall. The camera used for imaging content is mounted in the ceiling at the back of the room. Microphones are deployed in front of the display wall canvas.

The 16 Unibrain Fire-i firewire cameras are connected in pairs to 8 Mac minis. Images from each camera are processed by applying two common techniques from computer vision: Background subtraction and thresholding. The result of processing a single image is a set of 1D positions (visible as the black dots inside the rectangles in Figure 5, and also shown in Figure 6), which when combined with data from the remaining cameras enable the triangulation of 2D object positions. The design and implementation of the device-free input system is based on the system presented by [Stødle et al. 2008].

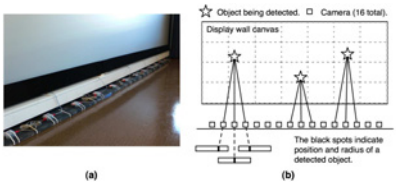


Figure 5: (a) The cameras for the device-free input system as they are mounted along the floor. (b) Triangulating object positions using image data gathered from each camera (triangulation shown for a single plane only).

The device-free input system is used for interacting with the Wallboard, as well as determining where to point the camera in order to image objects. This highlights an important principle employed throughout the design of Wallboard: Instead of determining *what* an object is, it is more fundamental to determine *where* it is. This principle is applied successfully in the design of the input system, and for Wallboard it makes it possible to easily determine where the content to image is located. In contrast, one could design a computer vision-based system where the area in front of the display wall is scanned by a camera to first identify the user, and then determine his location. Once the user's location has been found, the immediate surrounding area could be analyzed to find objects, before the camera can finally be accurately pointed at it and zoomed in. The latter approach would be more computationally demanding, time consuming and also very difficult to make reliable - if at all possible. It might also introduce assumptions about the content to image that are not ideal; for instance, assuming that content is always white and rectangular in shape.

3.2 Input analysis

The second component is the input analysis component. It is responsible for interpreting input events from the input system, and determining if a user is attempting to capture content for the Wallboard. For objects inside the device-free input system's region of interest, attributes like the object's movement and radius are used to determine whether to image the content or not. If the component determines that the content should be imaged, it sends an event to the camera and sensor control component, which will steer the camera and capture the targeted content.

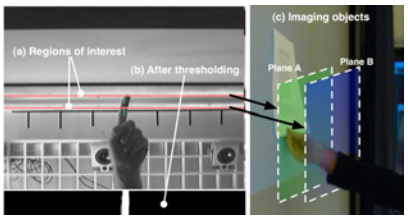


Figure 6: (a) The input image, showing the two planes in front of the display wall with a finger intersecting both. (b) The result after background subtraction and thresholding of one of the two planes. (c) The hand must intersect planes A and B, which are both parallel to the display wall surface, in order to target content for capture. In addition, the hand must remain stationary for one second and have a detected radius above an experimentally determined threshold.

The input analysis component is an important part, as it will essentially make or break the user's impression of the system. If it over-eagerly begins imaging content, spurious images will appear on the display wall. On the other hand, if it requires too much effort to invoke, users will end up frustrated with the system's behaviour. The input analysis component uses the following three factors, all supplied by the input system, to make a decision on whether or not to image an object: (i) The 2D position of an object (usually a finger, hand or arm) intersecting two planes in front of the wall (Figure 6). When a set of 2D positions is sufficiently close to each other, the remaining two factors are considered for that set. (ii) The width of the objects in the set. If the width is above an experimentally determined threshold, the object is tracked. (iii) If an object is tracked for more than 1 second and remains stationary, it will be interpreted as if a user wants to image the content held at the given location.

3.3 Camera and sensor control

The camera sensor and control component manages the camera and microphones in use by the system. When instructed to do so by the input analysis component, it will capture data from the camera. It will then notify the Wallboard application that it should fetch the newly captured data and position it on the display wall at the location where the user originally held the content to be captured. It

also continuously records audio from the environment, with audio from the 15 preceding seconds being associated with the imaged content.

The camera used is a Canon VC-C4R with pan-tilt-zoom functionality and capable of generating images with a resolution of 720x540 in interlaced mode. It is mounted in the ceiling at the back of the room (see Figure 4), pointing towards the display wall. The camera is moved in response to a "scan" event from the input analysis component. To steer the camera, a mapping between the camera's pan and tilt coordinates to areas covered on the display wall is used. This mapping is created by determining the extreme values for pan and tilt at maximum zoom levels when aiming at the corners of the display wall. Linear interpolation is then used to map coordinates from the device-free input system to the camera's pan- and tilt-values, before the camera can be steered to the correct location.

One problem discovered in an earlier implementation of the system was that captured images were often affected by motion blur. Motion blur is introduced either by users being unable to hold the content to image stationary, lingering camera movement, or both. The problem is exacerbated by the fact that the camera in use is only capable of producing interlaced images. To handle this problem, the control component continuously captures images from the camera. Each new image captured is subtracted from the previous image, and used to calculate the average pixel intensity change, as well as the pixel intensity change's standard deviation. Whenever the standard deviation is below an experimentally determined threshold[1], that image will be eligible for being pinned to the display wall.

3.4 Wallboard

The fourth major component is the Wallboard application itself, whose main responsibility is to provide the graphical output on the display wall, as well as allow users to interact with imaged content. It accepts input events directly from the device-free input system, enabling multiple users to interact with it simultaneously using one or both hands. Since currently only one camera is in use, different users can not overlap imaging of content, but must interleave their use of the imaging feature. Wallboard also receives events from the camera sensor and control component, informing it when newly imaged content is available and where it should be placed on the display wall.

The Wallboard application is written in C using an in-development cluster-based backend to the Cairo [Worth and Packard 2003] rendering library. It responds to input events from the device-free input system, enabling users to not only image content, but also scale and move the resulting objects around on the display wall afterwards. When instructed to position an imaged object on screen, it will load the image representing it and place it at the coordinates given in the "fetch data" event. Using the event system it can also trigger playback of the audio associated with the imaged content.

3.5 The Shout event system

The fifth and final component is a network event system called Shout. Shout provides the other four components with the ability to send and receive events, and thus acts as an "event substrate" in between the components. It is designed to be both extendable and enable efficient event delivery. Shout is implemented in C, using a centralized event server to receive and distribute events from different clients. For efficiency and reduced bandwidth consumption, a binary format is used. The content and types of events is not pre-defined by the event system, but instead defined by the applications using the system. By default, a client receives all events, but event filters can be configured in order to limit the events received to specific types (such as the "fetch data" and "scan" events). To aid system efficiency, clients may also tell the server about which event types they intend to provide to the server. TCP is used for client-server communication.

[1] The threshold used is affected mainly by camera noise and lighting factors.

4 Initial results

The latency for capturing content has been measured. The interval measured ranges from when the system determines that a user wants to capture content, until the content appears on the display wall – that is, not including the initial one-second delay used by the system to determine user intent. The methodology to measure this latency was as follows. An object was imaged by one of the authors 30 times at different locations on the display wall. Every time an event instructing Wallboard to image an object was received, a timer was started. That timer was stopped when a corresponding "fetch data" event was received, at which point the imaged content would appear on the display wall.

The results from this experiment yielded an average latency of 1.08 seconds, with a standard deviation of 0.26 seconds. The maximum observed latency was 1.73 seconds, and the minimum latency was 0.65 seconds.

5 Discussion

The latency for imaging objects is stems from the following factors. First, the camera requires some time to capture an image. Frames from the camera are captured using a frame grabber card, which provides new frames at a rate of 12.5 frames per second. At this rate, the time between an event prompting the control component to image content arrives, until the camera is ready with a new frame, can be up to 1.0 s / 12.5 frames/s = 0.08 s. With the actual latency about one order of magnitude higher, the camera frame rate is not the issue. The majority of the latency is due to two factors: (i) The camera is steered to target the content before an image is captured, and (ii) due to camera movement and stabilization, the technique used to avoid motion blur will prevent a number of the initial images from being recognized as valid images. The event system's latency has been measured at 1.9 milliseconds [Stødle et al. 2008], and thus contributes very little to the overall latency.

The use of a steerable camera to image content in the Wallboard system has some drawbacks. First, the pan- and tilt coordinates used to control where the camera is pointing, are far more coarse-grained than the coordinates provided by the device-free input system. This manifests itself as slight inaccuracies when imaging content, such as missing the top or one of the sides of a document. Second, it is quite common for parts of the fingers or hands to appear as part of the image representing the content. However, any other approach would require either (i) tagging all content in advance, which is impractical and violates the device-free aspects of the system design, or (ii) applying sophisticated computer vision techniques in an effort to recognize either the object or the fingers. The unwanted parts of the image could then be masked out. Finally, with the camera mounted in the ceiling, it is possible for the user to obscure the content to image when holding it at approximately waist-height or below.

The current implementation brings up a black box on the display wall, behind the content which is being imaged. This serves two purposes: First, it removes clutter from the resulting image by temporarily hiding other content at the location being captured. More importantly, however, it prevents light from the projectors leaking through the content being scanned. This is especially visible when imaging single sheets of paper. This effect could also be exploited for positive gain by the system, by allowing the color of the background to be changed. This could allow better capture of content like transparencies. It could also enable the system to respond to changes in the room's current light levels, enabling better camera

exposure.

The Wallboard system is scalable along many different axes. It can be extended with additional cameras to enable several users to image objects simultaneously, and the device-free input system already easily accomodates more than one user - three persons may play Quake 3 Arena against each other at the same time on the display wall [Støddle et al. 2007]. The resolution of the images captured can be increased by using more expensive cameras without changing other parts of the system, and additional sensors may be added beyond the camera and microphones currently in use.

6 Conclusion

This paper has introduced Wallboard, a system that enables content to be moved from the real world to a display wall in a way that mimics the interaction pattern used to share information on a billboard. Without requiring the use of any devices, users briefly hold documents, images or other content in front of the display wall at the location where they want it to appear. A device-free input system determines where the object to capture is located, before a camera and associated sensors capture the object.

The system requires in total about two seconds to capture an image of the content a user wishes to place on the Wallboard. The first second is used to determine user intention ("does the user really want to capture this content and place it on the display wall?"), while the rest is caused by system latency incurred by camera movement and avoiding motion blur. An important design principle has been identified that greatly simplifies the implementation of both the device-free input system, and the process of determining where the content to capture is located: Instead of determining *what* an object is, it is more fundamental to determine *where* it is. Applying this principle enables Wallboard to avoid some very hard problems in computer vision (object recognition and pose estimation), while still resulting in a system that achieves the design requirements set out in the introduction: It mimics a billboard, it does not require the user to wear or use any devices, and the time required to pin content to the display wall is on the order of a few seconds.

Acknowledgements

The authors thank Espen S. Johnsen for creating the cluster-based backend to Cairo, John Markus Bjørndalen, Tor-Magne Stien Hagen and the technical staff at the CS department at the University of Tromsø. This work is supported by the Norwegian Research Council, projects No. 159936/V30 and No. 155550/420.

References

ABILEAH, A., AND GREEN, P. 2007. Optical sensors embedded within amlcd panel: design and applications. In *EDT '07: Proceedings of the 2007 workshop on Emerging displays technologies*, ACM, New York, NY, USA, 7.

BATHICHE, S., AND WILSON, A., 2007. Microsoft surface. http://www.microsoft.com/surface/.

DIETZ, P., AND LEIGH, D. 2001. DiamondTouch: a multi-user touch technology. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, ACM Press, New York, NY, USA, 219–226.

ELROD, S., BRUCE, R., GOLD, R., GOLDBERG, D., HALASZ, F., JANSSEN, W., LEE, D., MCCALL, K., PEDERSEN, E., PIER, K., TANG, J., AND WELCH, B. 1992. Liveboard: a large interactive display supporting group meetings, presentations, and remote collaboration. In *CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, New York, NY, USA, 599–607.

HAN, J. Y. 2005. Low-cost multi-touch sensing through frustrated total internal reflection. In *UIST '05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, ACM Press, New York, NY, USA, 115–118.

ISHII, H., AND ULLMER, B. 1997. Tangible bits: towards seamless interfaces between people, bits and atoms. In *CHI '97: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, New York, NY, USA, 234–241.

KLEMMER, S., NEWMAN, M. W., AND SAPIEN, R. 2000. The designer's outpost: a task-centered tangible interface for web site information design. In *CHI '00: CHI '00 extended abstracts on Human factors in computing systems*, ACM, New York, NY, USA, 333–334.

KOIKE, H., SATO, Y., AND KOBAYASHI, Y. 2001. Integrating paper and digital information on enhanceddesk: a method for realtime finger tracking on an augmented desk system. *ACM Trans. Comput.-Hum. Interact. 8*, 4, 307–322.

LI, K., CHEN, H., CHEN, Y., CLARK, D. W., COOK, P., DAMIANAKIS, S., ESSL, G., FINKELSTEIN, A., FUNKHOUSER, T., HOUSEL, T., KLEIN, A., LIU, Z., PRAUN, E., SAMANTA, R., SHEDD, B., SINGH, J. P., TZANETAKIS, G., AND ZHENG, J. 2000. Building and Using A Scalable Display Wall System. *IEEE Comput. Graph. Appl. 20*, 4, 29–37.

MORRISON, G. D. 2005. A camera-based input device for large interactive displays. *IEEE Computer Graphics and Applications 25*, 4, 52–57.

MYNATT, E. D., IGARASHI, T., EDWARDS, W. K., AND LAMARCA, A. 1999. Flatland: new dimensions in office whiteboards. In *CHI '99: Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, New York, NY, USA, 346–353.

PEDERSEN, E. R., MCCALL, K., MORAN, T. P., AND HALASZ, F. G. 1993. Tivoli: an electronic whiteboard for informal workgroup meetings. In *CHI '93: Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, ACM, New York, NY, USA, 391–398.

SHAE, Z.-Y., TSENG, B., AND LEUNG, W. H. 2001. Immersive whiteboard collaborative system. *Ann. Softw. Eng. 12*, 1, 193–212.

SMART TECHNOLOGIES. SMART board interactive whiteboards. http://www.smarttech.com/.

STØDLE, D., HAGEN, T.-M. S., BJØRNDALEN, J. M., AND ANSHUS, O. J. 2007. Gesture-based, touch-free multi-user gaming on wall-sized, high-resolution tiled displays. In *Proceedings of the 4th Intl. Symposium on Pervasive Gaming Applications, PerGames 2007*, 75–83.

STØDLE, D., HA, P. H., BJØRNDALEN, J. M., AND ANSHUS, O. J. 2008. Lessons learned using a camera cluster to detect and locate objects. In *Parallel Computing: Architectures, Algorithms and Applications. Proceedings of the International Conference ParCo 2007.*, IOS Press, vol. 15 of *Advances in Parallel Computing*, 71–78.

STOLK, B., AND WIELINGA, P. 2006. Building a 100 Mpixel graphics device for the OptIPuter. *Future Gener. Comput. Syst. 22*, 8, 972–975.

WORTH, C. D., AND PACKARD, K. 2003. Cairo: Cross-device rendering for vector graphics. In *Proceedings of the 2003 Linux Symposium*. http://cairographics.org/.

A.7 Tech-note: Device-Free Interaction Spaces

Citation

Daniel Stødle, Olga Troyanskaya, Kai Li, and Otto J. Anshus. Tech-note: Device-Free Interaction Spaces. In *3DUI '09: Proceedings of the IEEE Symposium on 3D User Interfaces*, pages 39–42, March 2009.

Abstract

Existing approaches to 3D input on wall-sized displays include tracking users with markers, using stereo- or depth-cameras or have users carry devices like the Nintendo Wiimote. Markers makes ad hoc usage difficult, and in public settings devices may easily get lost or stolen. Further, most camera-based approaches limit the area where users can interact.

This paper presents Interaction Spaces – a distributed, optical sensor system for 3D input that lets users interact without needing markers or hand-held devices. An Interaction Space is created that covers the display wall. Inside it, objects like hands or fingers are tracked in 3D. This enables actions like moving or zooming a view on the wall. The added depth dimension allows images to be zoomed using a single hand instead of the two-hand "pinch" gesture used in other systems. The system's distributed aspect enables simple scaling to cover smaller or larger areas.

The system is built using four computers and eight web cameras mounted along the floor. Each camera image is divided into vertical slices. Each slice is processed to detect 1D object positions, before 2D positions are determined using triangulation. The 3D position of an object can be inferred from its corresponding 2D positions in each slice. The system is currently being used to control a microarray visualization on a 2x2 display wall. The system's accuracy has been evaluated, and is shown to be about 1 cm.

Tech-note: Device-Free Interaction Spaces

Daniel Stødle* University of Tromsø, Norway
Olga Troyanskaya† Princeton University, USA
Kai Li‡ Princeton University, USA
Otto J. Anshus§ University of Tromsø, Norway

ABSTRACT

Existing approaches to 3D input on wall-sized displays include tracking users with markers, using stereo- or depth-cameras or have users carry devices like the Nintendo Wiimote. Markers makes ad hoc usage difficult, and in public settings devices may easily get lost or stolen. Further, most camera-based approaches limit the area where users can interact.

This paper presents Interaction Spaces – a distributed, optical sensor system for 3D input that lets users interact without needing markers or hand-held devices. An Interaction Space is created that covers the display wall. Inside it, objects like hands or fingers are tracked in 3D. This enables actions like moving or zooming a view on the wall. The added depth dimension allows images to be zoomed using a single hand instead of the two-hand "pinch" gesture used in other systems. The system's distributed aspect enables simple scaling to cover smaller or larger areas.

The system is built using four computers and eight web cameras mounted along the floor. Each camera image is divided into vertical slices. Each slice is processed to detect 1D object positions, before 2D positions are determined using triangulation. The 3D position of an object can be inferred from its corresponding 2D positions in each slice. The system is currently being used to control a microarray visualization on a 2x2 display wall. The system's accuracy has been evaluated, and is shown to be about 1 cm.

Index Terms: I.3.1 [Computer Graphics]: Hardware Architecture—Input devices

1 INTRODUCTION

There are many approaches to provide 3D input to applications running on wall-sized displays. A user's hand- or body movement can be tracked using cameras that identify and position a set of passive markers mounted on the user. Users can also carry devices like a 3D mouse or the Nintendo Wiimote, or her location can be determined without markers using stereo- or depth- cameras. These approaches are limited in different ways. The use of markers makes ad hoc usage difficult. Users must spend time mounting the markers to their body, or wear special clothes with embedded markers for full-body tracking. In public settings, a 3D mouse or Wiimote may easily get lost or stolen, and most camera-based approaches limit the area in which interaction can take place.

This paper presents a distributed optical sensor system for 3D multi-point input. The system removes the need for markers and hand-held input devices, enabling the user to interact freely along a wall-sized, high resolution tiled display. The system creates a 3D Interaction Space that is as long and as tall as the display wall itself, and up to about 35 cm deep. The width of the Interaction Space is mainly limited by the number of optical sensors used. Inside the Interaction Space, objects - like a user's hands - are discovered and their 3D position determined. Each object's position is sent to applications in events which can be used for various purposes like moving, zooming, and rotating a view on the display wall.

Using the depth dimension, it is possible to zoom images using a single hand instead of the two-hand "pinch" gesture used in other systems like the Apple iPhone. The system is not limited to detecting hands and using them for input; it can detect any object that gives sufficient contrast to the mostly static background, including for instance the user's elbows, head or other body parts. Since the system avoids using markers or special hand-held devices, ad hoc usage is possible with no preparation on the user's part. The system's intrinsic distributed aspect makes it easily scalable by adding additional cameras and computers, creating larger Interaction Spaces.

The system is currently being used with different applications on two wall-sized displays, including a parallel multi-image viewer and a genomics-application to explore relationships between different microarrays, shown in Figure 1(a). To evaluate the system, experiments measuring the accuracy on a per-slice level have been performed, demonstrating that the system has an average accuracy of about 1 cm for the slice closest to the display wall. The main contribution of this paper is a 3D input system that makes scalable interaction in 2D and 3D possible using commodity components, while still maintaining reasonably good accuracy. Users do not need special devices or markers to interact with the system.

2 RELATED WORK

There has been much work on input devices for display walls. The VisionWand [2] provides input by optically tracking a wand-like object in 3D using two cameras, but is limited in that it requires markers and a known object (the wand) to operate. Further, the area in which interaction can take place is limited. When Nintendo introduced the Wii console, they also made the first "mass-market" 3D input device in the "Wiimote." The Wiimote combines accelerometer data with tracking of up to four infrared dots to provide input in 3D, and its potential for "hackability" has enabled the creation of cheap DIY multi-point input devices [7]. The VisionWand and the Wiimote are examples of systems that use markers to provide 3D input.

Another class of 3D input systems employ image recognition to determine the pose of hands and fingers without using markers directly. The Visual Touchpad is an example of this, where a user's hand can be positioned over a specially designed touchpad [9]. Other approaches include using stereo cameras combined with infrared illumination [12], and depth-cameras that capture both color and depth for each pixel in an image [14]. Both systems can provide device-free interaction, but suffer from a lack of large-area coverage. This is a commonality for most camera-based systems: The user has to be inside the camera's field of view, which introduces scalability problems as the size of the area one wishes to interact with goes up. Further, it is not clear how existing systems could be extended to increase the area of interaction. The Interaction Spaces system is designed to be scalable, and is currently used with two different display walls measuring 6x3 and 2.7x2 meters.

The Interaction Spaces system is similar to a number of other multi-touch systems currently available, like the Microsoft Surface [1] and TouchWall, and the Diamondtouch tabletop [3]. Jeff Han pioneered multi-touch sensing using frustrated total internal reflection

*daniels@cs.uit.no
†ogt@genomics.princeton.edu
‡li@cs.princeton.edu
§otto@cs.uit.no

IEEE Symposium on 3D User Interfaces 2009
14-15 March, Lafayette, Louisiana, USA
978-1-4244-3812-9/09/$25.00 ©2009 IEEE

of infrared light [4], which has been commercialized by Perceptive Pixel as a "collaboration wall" and used extensively by CNN to cover the 2008 US presidential election. In [13], the authors describe a system that detects hands and fingers interacting with a whiteboard using a single camera. It is similar to our system in its use of simple image differencing to segment the foreground and background. In [10], a few custom cameras are used to triangulate the position of objects on the SMART Board. This approach differs from ours in its use of custom cameras with on-chip image processing to do object detection. None of these systems provide input in 3D. GestureTek is a company that offers both 2D and 3D camera-based, device-free input solutions [6]. The scalability of their products is unclear, as it appears that all the cameras cover the same (limited) region of interest, but from different angles. In the Interaction Spaces system, different sensors cover different but overlapping regions, cooperating to create a larger space in which interaction can take place. Further, the Interaction Spaces system does not require high-end synchronized cameras, but can operate using commodity web cameras.

3 Design and Implementation

The design of the Interaction Spaces system is based on the following permeating principle: It is more important to determine *where* an object is, as opposed to *what* the object is. This approach is fundamentally different from other camera-based systems. Instead of trying to determine what different objects are – a hand, a finger, a pen, and so on – the system only seeks to discover that an object has entered the Interaction Space, and determine where that object is. The system is based on earlier work that only provided object positions in 2D [11], and uses a set of optical sensors to detect the presence of objects in each optical sensor's field of view. By using information about the relative position of these objects in each optical sensor's view, the object's location can be determined.

To extend this design to 3D, each sensor divides its field of view into a number of distinct slices. Within each slice, the sensor locates foreground objects. As a result, each object's 1D position and extent in a slice is determined for each sensor. A coordinator can then determine an object's 2D position by collecting 1D positions from all the sensors. By treating each 1D position in a slice as a beam from the sensor's position and up, the 2D positions of possible objects can be found using triangulation at the intersections of beams from different sensors, as illustrated in Figure 1(b). To support multi-point interaction and avoid false positives, an object must be initially tracked by at least three sensors. The sensors do not attempt to associate objects from different slices with each other. Instead, the coordinator uses the object extent and calculated 2D position to associate objects from different slices with each other.

The Interaction Spaces system has been implemented using eight commodity web cameras (Unibrain Fire-i @ 640x480 pixels in grayscale) and four computers (Mac mini @ 1.83 GHz). The system is used to interact with applications running on two display walls: One 2.7x2 meter 4-projector, 2048x1536 pixel wall, and one 6x3 meter, 28-projector 7168x3072 pixel wall. The latter is extended with 4 additional Mac minis and 8 additional cameras to cover the entire width of the wall. The software consists an image processing component and an analysis component.

The image processing component runs on each Mac mini to capture images from the cameras, and processes them to detect the presence of foreground objects. Each image is divided into 25 independent vertical slices, as shown in Figure 1(c). Foreground objects are detected using image differencing, thresholding and a dynamically updated background image. This results in clusters of white pixels where objects have been detected. The center position and extent of each cluster is then transmitted using a network event system to the analysis component, along with the slice index in which the object was detected.

The analysis component receives data for each slice from each sensor, and uses it to triangulate object positions in 2D per slice. Since many objects can be detected in a given slice, each individual object must initially be detected by at least three cameras for the triangulation to be successful. This is necessary to avoid false positives caused by the presence of other objects, which would create a number of "ghost objects" where imprints created by one object intersect the imprints from other objects. Such potential false positives are highlighted in Figure 1(b).

To detect an object's 3D position, the analysis component first gathers the information it has about 2D object positions in all available slices. It then begins at the outer-most slice (farthest from the display wall), and assigns 2D objects to new or existing 3D objects. A new 3D object is created any time a 2D object from one of the outer-most slices appear that are considered too far from any already existing 3D objects, or if there are more 2D objects in a given slice than there are existing 3D objects. At present, the system is limited to associating a single 2D object from each slice to a given 3D object; this means that an arm that extends into the Interaction Space and then divides into an open hand with spread fingers will only use one of the finger positions, instead of incorporating all of them into the same 3D object.

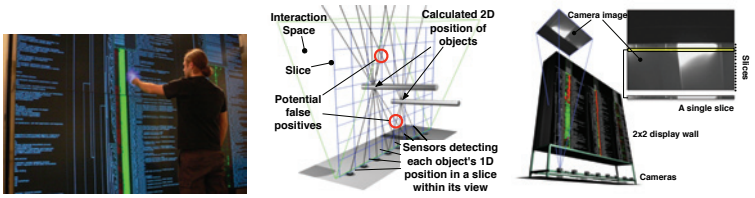
Once a 3D object has been detected, an event is created containing the location of the object's tip in 2D, and its depth position, as well as events for the raw 2D locations for each object in each slice. These events are used by applications to enable interaction. Currently, the depth position is a direct translation from the slice index, with a value of 0.0 corresponding to touching the wall, and a value of 1.0 being the outer-most slice that is recognized; in the future this will correspond to the actual depth in real world units. The 2D position is reported in centimeters relative to the left side of the display wall and the floor.

To provide accurate output, the system is calibrated using a limited camera model with the following parameters: position, field of view, left-right rotation and distortion (further refinement is planned). To calibrate the cameras, an operator touches 18 target points with known real world coordinates on the display wall. Each camera records the position of the object it detects, if any. When all the targets have been touched, each camera will have a set of detected objects and their associated real world coordinates. Each camera is then automatically adjusted by iteratively changing different parameters with the goal of minimizing the error between known target location, and where that object would be placed given the current camera parameters.

4 Applications

Interaction Spaces is currently being used to interact with a system for visualization of genomic microarray data, shown in Figure 1(a). The viewer a custom display wall version of HIDRA [5]. In HIDRA, users explore different datasets by selecting genes in one dataset, and observing where they are located in other datasets. Genes in close proximity to each other in the microarray indicates that they might be correlated.

To use the viewer, the system must support navigation and selection of genes. In Interaction Spaces, these actions are mapped to moving ones hand in the space in front of the display wall. The depth dimension is used to control what action is performed. If the user touches the display, the genes under the user's finger will be selected. Otherwise, the view is panned according to the user's hand movements. To avoid accidental panning when the user intends to select genes, the system prevents panning if the object is seen to move closer to the wall. The viewer can also be controlled using an iPhone. The depth dimension is also used to control zoom in an image viewer application, where the view zooms closer as the user's hand approaches the wall, and zooms back out when the hand is moved away.



(a) (b) (c)

Figure 1: (a) Interacting with a microarray visualization. The bright spot under the user's finger is in reality a set of animated particles that tells the user that he is giving input to the system. (b) An object's 2D position in the center slice is found by triangulating the 1D positions detected by the different sensors. The two circles indicate potential false positives if only two sensors were to be used in determining the position of the object. (c) A sample image from one of the cameras, and its relation to the world.

5 Evaluation

There are two important technical performance metrics when evaluating input devices: Latency and accuracy. Latency is important to help users create a connection between the actions they make, to what they see happen on screen [8]. Good accuracy is important for selecting small targets or do other tasks that require precise input. However, a system could in principle be used even in the face of great inaccuracies if the applications were designed to expect noisy and inaccurate input. Previous work [11] has shown that the latency of the Interaction Spaces system is about 115 ms. This evaluation will focus on accuracy. In the Interaction Spaces system, there are many variables that together determine the total system accuracy. The distributed nature of the system means that sensors covering different areas may combine to produce very different accuracy levels for the areas they cover.

It is difficult to design an experiment that objectively and empirically measures the accuracy of an input system such as the one presented in this paper, while enabling other researchers to reproduce the results in a consistent manner. Without designing a mechanical arm or similar that can be made to consistently produce the exact same movements, the only option left is to have one or several users test the accuracy of the system. However, such tests are very hard to reproduce, and will inevitably be affected by the different characteristics of each user. Thus, such tests do not help in methodically exploring how changes to the system affect its accuracy. For instance, to quantify the effect of varying lighting conditions or changing the foreground/background segmentation algorithm, it is essential that the experiment be repeatable and identical to earlier trials. For this case, users are not useful, as they will be hard-pressed to conduct the exact same movements time and time again.

In spite of these concerns, the system's accuracy in positioning an object at the innermost slice was measured by having a user interact "mechanically" with the system. The user touched 100 target points on the display wall in turn. For each target, the system recorded the currently detected object's position 30 times. Each target was shown alone on the display wall as a white square on a black background. To provide the user with feedback about what the system detects, a fountain of particles appear at the location where the system thinks the object is. Once the system has gathered enough samples for the target, the screen briefly flashes to indicate its readiness to sample the next target, and the next target appears on the display wall.

The results are shown in Figure 2. The accuracy of the system is measured in centimeters. The X axis indicates the offset from the left side of the display wall, which measures 272 cm in total. The Y axis shows the offset from the bottom of the display wall (not

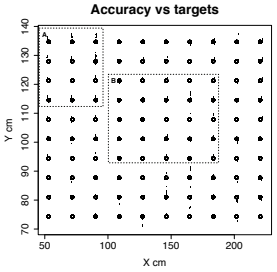


Figure 2: 100 targets (circles) and the positions detected by the system for each target (dots). The X and Y axis show the horizontal and vertical location on the 2.7x2 m display wall. Boxes A and B highlight areas where the system exhibits low and high accuracy.

the floor) to the top, which measures 202 cm. All the targets were located within an interior rectangle that measured 167x60cm. The size of this area was chosen based on the typical area in which the system is used for interaction; anything much above is usually too far up to reach without effort, and the system does not support initial touches to the far left and far right (which is what the experiment tests), as these objects are only seen by two cameras. This is one less than the three that are required to get a positive lock on the object (note that once the system has acquired an object, it can be tracked even if only seen by two cameras).

The plot indicates that the system is more accurate along the horizontal axis than the vertical axis, with the mean horizontal delta (dX) between target and observed location being -0.21 cm, and the mean vertical delta (dY) -0.47 cm. The mean distance from observations to actual targets is 1.1 cm, with a 0.72 cm standard deviation. Further, 90% of the targets had a vertical standard deviation less than 0.5 cm, and 93% of the targets had a horizontal standard deviation less than 0.1 cm.

Two boxes are highlighted in Figure 2. Box A shows an area where the system exhibits low accuracy. The detected object's location flickers up and down (as indicated by the vertical spread of the dots), while the object's position along the X axis remains fairly

constant. Inside box B, the system appears to be more accurate: Apart from a few problem spots, most samples are very close to their targets. Why does the system exhibit such differing accuracy behaviour? Some answers to this question will be given in the discussion.

6 Discussion

The evaluation has shown that the system exhibits differing accuracy-levels depending on where the user interacts. To analyze why this is the case, it is necessary to know which factors impact the system accuracy. The factors that govern accuracy in the Interaction Spaces system are: (i) Timing of data from the sensors, (ii) object speed, (iii) lighting, (iv) precision of the foreground/background segmentation, (v) object extent, (vi) physical placement and alignment of the sensors, and (vii) system calibration.

Timing of sensor data is important when the objects being tracked are moving. Since the system relies on unsynchronized web cameras, one step of the triangulation may rely on data from different cameras separated by up to 33 ms. The experiment was designed to eliminate both timing and object speed as factors, by keeping the object to track stationary.

The precision of the foreground/background segmentation is one factor that helps explain some of the observations made in the evaluation. Typically, when the position of a detected object exhibits much vertical jitter, a single camera "flickers" between detecting and not detecting the object within that particular slice, or detecting it at two slightly offset 1D positions due to random noise or lighting effects. This causes the position to jitter up and down.

As an object's extent grows wider, the extent of the lines projected from the cameras grow. If segmentation was perfect and the cameras perfectly synchronized, the object extent would not play a role in the system's accuracy. While not entirely eliminated in the evaluation (the extent of a user's finger may vary slightly depending on the exact finger pose), it is not a major factor. As part of the evaluation, the object extent was recorded for each sample, varying between 2.34 and 6.49 pixels.

The cameras' physical alignment is important to give a best possible starting point for determining object positions. Since perfect alignment is difficult to achieve, the system requires calibration. In the experiment, most samples had a standard deviation less than 0.1 cm horizontally and 0.5 cm vertically. Thus, while the system's accuracy varies from target to target, it is consistent in where it positions objects relative to the targets, pointing to calibration as the cause of the varying accuracy, since different cameras may have been calibrated more or less accurately.

Some ways to improve the system accuracy would include using infrared illumination coupled with visible-light filters on the cameras, or using cameras with higher framerates or lower noise. Lighting also affects the quality of the segmentation. The results presented in the previous section were gathered under fairly ad hoc lighting conditions, with two lamps mounted in the ceiling giving good backlight to parts of the scene, but not covering it completely, leaving large dark regions (visible in Figure 1(c)). Despite the varying light levels, the system yields an accuracy of about 1 cm.

The evaluation has only touched on the accuracy of positioning an object within a slice, and not on how this relates to the depth dimension. Since every slice is processed in the same way, it is reasonable to assume that the resulting accuracy would be roughly the same. However, conducting an experiment to prove this is very difficult, as it is hard for users to accurately point at targets shown on a display wall as much as 35 cm away. For this reason, an evaluation of the slice accuracy further away from the display wall is left as future work. Support for recognition of 3D object pose (such as pointing direction) and a more refined camera model is also planned, as well as an evaluation of the system's accuracy when tracking moving targets.

7 Conclusion

This paper has presented the Interaction Spaces system for providing 3D input to applications running on wall-sized displays. The system allows one or several users to interact with applications simultaneously using one or both hands, or any other object or body part they might see fit to use. Unlike most existing systems, the Interaction Spaces system can easily be extended to provide interaction along larger areas. It is non-intrusive, in that it does not require users to wear special markers or carry devices to interact. The system is in use with several different applications, including an application for microarray visualization used to study relationships between genes. In this application, the depth dimension can be used to differentiate between navigating the visualization and selecting genes. The experiments have demonstrated that the system has an average accuracy of about 1 cm for the innermost slice.

Acknowledgements

Supported by the Norwegian Research Council (159936, 155550), NSF (DBI-0546275), NIH (R01 GM071966, T32 HG003284), NIGMS (P50 GM071508). Thanks to Matthew Hibbs, Tor-Magne S. Hagen and Espen S. Johnsen.

References

[1] S. Bathiche and A. Wilson. Microsoft Surface, 2007. http://www.microsoft.com/surface/.

[2] X. Cao and R. Balakrishnan. VisionWand: interaction techniques for large displays using a passive wand tracked in 3D. In *UIST'03: Proceedings of the 16th annual ACM Symposium on User Interface Software and Technology*, pages 173–182, 2003.

[3] P. Dietz and D. Leigh. DiamondTouch: a multi-user touch technology. In *UIST'01: Proceedings of the 14th annual ACM Symposium on User interface Software and Technology*, pages 219–226, 2001.

[4] J. Y. Han. Low-cost multi-touch sensing through frustrated total internal reflection. In *UIST'05: Proceedings of the 18th annual ACM symposium on User interface software and technology*, pages 115–118, 2005.

[5] M. Hibbs, G. Wallace, M. Dunham, K. Li, and O. Troyanskaya. Viewing the Larger Context of Genomic Data through Horizontal Integration. *IV '07: Information Visualization*, pages 326–334, July 2007.

[6] E. Hildreth and F. Macdougall. Multiple camera control system, June 2006. US Patent no. 7058204.

[7] J. C. Lee. Hacking the Nintendo Wii Remote. *IEEE Pervasive Computing*, 7(3):39–45, 2008.

[8] I. S. MacKenzie and C. Ware. Lag as a determinant of human performance in interactive systems. In *CHI'93: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 488–493, 1993.

[9] S. Malik and J. Laszlo. Visual touchpad: a two-handed gestural input device. In *ICMI'04: Proceedings of the 6th Int'l Conference on Multimodal Interfaces*, pages 289–296, 2004.

[10] G. D. Morrison. A Camera-Based Input Device for Large Interactive Displays. *IEEE Computer Graphics and Applications*, 25(4):52–57, 2005.

[11] D. Stødle, P. H. Ha, J. M. Bjørndalen, and O. J. Anshus. Lessons Learned using a Camera Cluster to Detect and Locate Objects. In *ParCo'07: Proceedings of Parallel Computing: Architectures, Algorithms and Applications.*, volume 15 of *Advances in Parallel Computing*, pages 71–78. IOS Press, 2008.

[12] W. M. Vieta and M. Bell. WaveScape: a practical robust display with a 3D gesture interface. In *IPT/EDT'08: Proceedings of the 2008 workshop on Immersive projection technologies/Emerging display technologiges*, pages 1–2, 2008.

[13] C. von Hardenberg and F. Bérard. Bare-hand human-computer interaction. In *PUI '01: Proceedings of the 2001 workshop on Perceptive user interfaces*, pages 1–8, 2001.

[14] G. Yahav, G. Iddan, and D. Mandelboum. 3D Imaging Camera for Gaming Application. *ICCE'07: Int'l Conference on Consumer Electronics.*, pages 1–2, Jan. 2007.

Appendix B

The Shout event system

This appendix details the Shout event system. Shout is a network event system based on the idea that applications need to exchange events of different kinds, including input events, performance data and debug information. Since the applications run on different computers in the (parallel) display wall environment, the system must allow applications to share events across machine boundaries.

B.1 Related work

The Shout event system is an important enabling component in several of the systems and applications presented in this dissertation. However, neither its design nor the idea behind it is novel. There are a number of already existing network event systems. The systems include: (i) the X Window System [86], where clients may run on computers separate from the X server and send events to each other using the `XSendEvent()` call; (ii) The Event Heap [161], which provides a tuple-space based event infrastructure for the Interactive Workspaces project [75]; and (iii) ECho [162], an event delivery system positioned towards parallel and grid applications.

B.2 Model and architecture

In Shout, clients send and receive events to and from each other through a centralized server. An event can be any kind of occurrence that a client wants to make known to other clients. Each event consists of an event type and a number of standard fields shown in Figure B.1. The rest of the event format is type-specific and defined by the clients that use the given event types.

Clients configure a type filter when they connect to the server. The filter controls which events the client receives. If the filter is empty, the client receives events of all types. All events are coded in a binary format, using four bytes per field. A binary format was chosen to keep the event size down and to simplify server-side processing, compared to using a textual format like XML. The minimum event size is 40 bytes.

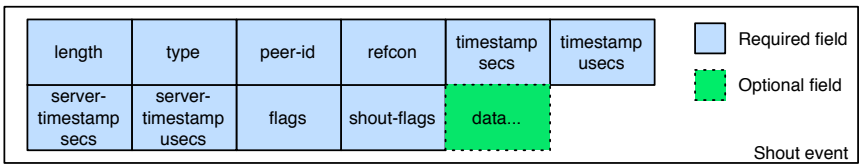


Figure B.1: The event format used by Shout. Each field is 4 bytes long, and encoded in network byte order.

The standard fields of a Shout event are: (i) Length: The event's length in bytes (including the length-field itself). The length is always a multiple of four; (ii) type: The event's type; (iii) peer-id: A value that identifies the client. The value is not persistent, and changes each time a client connects to the Shout server; (iv) refcon: A value that is defined by the event type; (v-vi) timestamp: A local timestamp taken by the client in seconds and microseconds when the event is sent; (vii-viii) server timestamp: A timestamp, in seconds and microseconds, recorded by the Shout server when the event is received; (ix) flags: Event-type specific flags; (x) Shout-flags: Flags internal to the event system, which are used to support bytestream events and loopback functionality. The flags are set by setting different bits of the field.

The rest of the event is defined by the event type, and is assumed to be in four-byte fields unless the bytestream Shout-flag is set. All fields are swapped to network byte order before they are sent, unless the event is a bytestream event. A bytestream event contains the length of the bytestream in the first data-field[1], followed by the actual bytes. The bytes are not byte-swapped. The last field of a bytestream event is padded to make the length of the event an even multiple of four.

The server receives events from clients, checks their type and then forwards each event to all clients whose type filter matches the event's type, or whose type filter is empty.

B.3 Design

The Shout event system is designed around a centralized server with clients running on different computers and devices on the same local-area network as the server.

[1]This is necessary since the number of bytes sent in an event is always a multiple of four.

The clients connect to the server using a reliable communication channel. The server design comprises three threads, as shown in Figure B.2: An input thread, a processing thread and an output thread. The input thread receives data from clients. It inspects the length field, and uses it to separate different events from each other. It also modifies the server timestamp fields of the event with the current server timestamp, and changes the event's peer-id field to the client's actual peer ID. The events are queued for processing as they arrive.

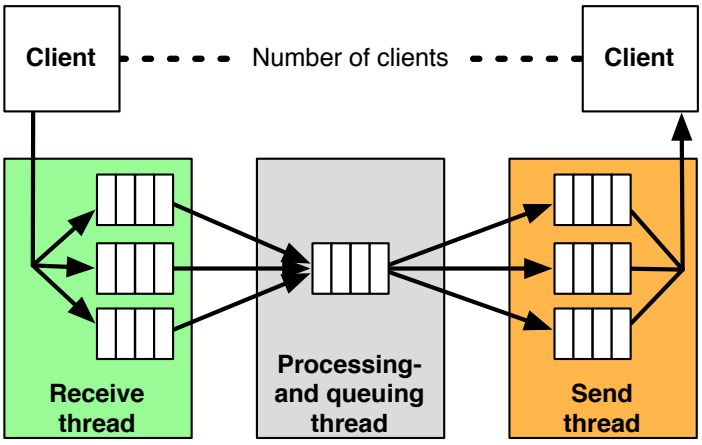


Figure B.2: The server's threaded design. The processing thread dequeues events from each client's receive queue, processes them, then re-enqueues them on the appropriate send queues.

The processing thread examines the event type of any event on the queue. If it is a Shout-specific event type (shown in Table B.1), it is handled directly by the server. Otherwise, the event is queued on the outgoing event queues for each client whose type filter matches the event type or is empty. The output thread sends data to a given client as soon as data is available on the client's output queue.

Type	Description
Set filter	Sent by a client to set its type filter. Each data field contains an event type that the client wants to receive. The number of event types is derived from the event's length field ($\frac{length-40}{4}$).
Set name	Sent by a client to set its human-readable name on the server. The bytestream Shout flag is set for the event. The length of the name stored in the bytestream length field, and ASCII characters follow in the rest of the data part of the event.

Table B.1: The two Shout specific event types.

A client sends two Shout-specific configuration events to set its type filter and a human-readable name. Each item in the filter consists of a four-byte event type; the number of event types is derived from the event's length field by subtracting the minimum event size of 40 bytes and dividing the resulting number by 4. The

human-readable name is used for debugging and development purposes, and is only visible to the server. It is set by sending a bytestream event with the "set name" event type.

B.4 Implementation

The Shout event system has been implemented in C, using POSIX threads and BSD sockets. The system runs on platforms including ARM, x86 and PowerPC, with either Linux or Mac OS X. The system consists of two components: The Shout server, and the Shout client library.

The server opens a TCP socket on port 3737[2] on which it accepts connections from clients. As clients connect, they are added to a list of clients which the server manages. The server associates two event queues with each client: One for receiving events from the client, and one on which events from other clients are queued to be sent to the client. The queues are implemented as a linked list of reference-counted events, and are unbounded. Events are delivered to clients in the same order as they are received. However, no guarantees are made by the server that events from two different sending clients are delivered in the same order to two different receiving clients; however events from a single sending client are always delivered in the same order as they are received. Events are reference-counted instead of making a copy of an event for each client that should receive the event. This reduces load due to memory copies, keeps memory usage down.

The input and output threads both use the `select()` call, respectively to wait for available data on the different client sockets, and to check if it is possible to send more data on the socket associated with a client. The output-thread further makes use of a condition variable to synchronize with the processing thread. The condition is signaled by the processing thread whenever a new event is queued for sending to any client. The processing thread uses a condition variable to wait for new events on the input thread.

Clients send and receive events using a library that is based on the server implementation. The library handles creating new events and memory management of the events. It also handles queueing incoming and outgoing events, as well as communicating with the Shout server. In cases where the connection to the server is closed, the library automatically tries to reconnect to the server at regular intervals. The interval ranges from 1, 2, 4, 8 seconds, up to a maximum interval of 60 seconds. Clients may poll, block or use the `select()` call to wait for incoming events. Listing B.1 shows how a simple Shout client is implemented using the Shout client library, using the blocking method of waiting for events.

[2]The port can be changed if necessary; the choice of port was arbitrary and based on available ports on the designated event server.

```
#include <shout.h>

// An example client that responds to PING events by sending a
// PONG event in return with the PING event sender's timestamp.
int main(int argc, char *argv[]) {
  shout_t       *connection = 0;
  shout_event_t *evt = 0;
  uint32_t      filter[] = { 'PING' }, // the type filter
                timestamp[2];

  connection = shout_connect("hostname", 3737 /* port */, "Demo client");
  shout_set_event_filter(connection, sizeof(filter)/4, filter);
  while (1) {
    evt = shout_wait_next_event(connection);
    if (evt != 0) {
      // Record the remote timestamp from the event
      timestamp[0] = evt->data[0];
      timestamp[1] = evt->data[1];
      // Release our reference to the event:
      shout_free_event(evt);
      // Send an event in response
      evt = shout_create_event('PONG', 0 /* shout-flags */, 0 /* event flags */,
                               0 /* ref-con */, 2 /* num data elems */,
                               timestamp /* pointer to data elems */);
      // Queue the event
      shout_queue_event(connection, evt);
      // Release our reference to the event
      shout_free_event(evt);
    }
  }
  return 0;
}
```

Listing B.1: Sample Shout client code.

B.5 Evaluation

The Shout event system has been evaluated by measuring the roundtrip event delivery latency for a variable number of clients and events in flight. The approach taken is similar to that taken by the standard "ping" utility, and is illustrated in Figure B.3. A client, referred to as the "ping master," sends a ping-event. The event contains a timestamp stored in two data items, in seconds and microseconds as reported by the `gettimeofday()` system call. The timestamp is taken just before queuing the event.

The master then waits for one or more pong-events in return. Pong-events are sent by "pong slaves," one for each ping-event they receive. The slaves copy the ping-event timestamp into the pong-event, which enables the master to calculate the roundtrip event delivery latency by comparing the timestamp taken when the event was originally sent, to the current time.

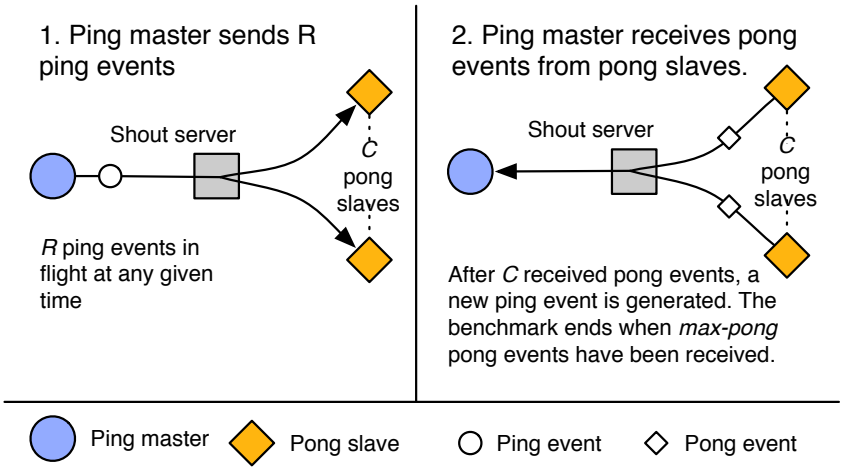


Figure B.3: Illustration of the roundtrip latency experiment.

The experiments are conducted by varying two parameters: (i) Rate (R): The number of ping-events sent by the master before it waits for corresponding pong-events to arrive; (ii) Pong-slaves (C): The number of slaves that respond to each ping-event. The rate was varied between the powers of two from 1 to 10 (i.e., 2^0 to 2^10). The number of pong-slaves was varied between 1 and 64. The experiment was conducted on the Tromsø display wall cluster. The event server ran on the cluster's front-end. The ping master and pong slaves were all evenly distributed on each of the 28 cluster nodes. No other clients were using the Shout server when the experiment was conducted. For each configuration, 100000 ping-events were sent.

B.5.1 Results

Figure B.4 shows a plot of the results from the experiment, and the numbers in Table B.2. As the number of events in flight grows, either as a result of increasing the rate (R) parameter, or increasing the number of clients, the latency goes up. The lowest roundtrip latency was 0.26 ms, which grew to 1.65 ms with 64 clients. In comparison, the "ping" utility reports a mean roundtrip latency between two computers in the Tromsø display cluster at 0.095 ms, after sending one million ICMP ping packets from the front-end to one of the cluster computers in flood-ping mode with up to 5 ping packets in flight[3].

[3]The command and options used were: `sudo ping -c 1000000 -f -q -l 5 hostname`

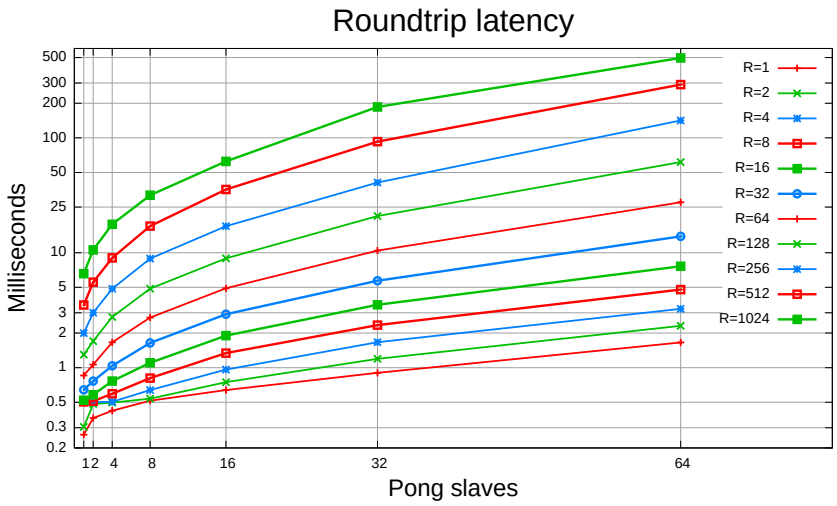


Figure B.4: Results from the latency experiment. The Y axis shows the latency in milliseconds. The X axis shows the number of pong slaves. The Y axis is logarithmically scaled.

B.6 Discussion

The latency of the system is sufficiently low to support interactive applications, compared to the baseline latency of 8.3 ms which the authors of [141] term as "negligible." The roundtrip latency stays below 10 ms with up to 16 events in flight simultaneously for up to 64 clients, at which point the server is processing about 133000 events per second.

When sending one ping event at a time, Shout has a roundtrip latency of 0.26 ms, which is about 2.7 times more than the network latency reported by the "ping" tool. This discrepancy can be explained by the following three factors: (i) The ping utility measures the time taken for a packet to be sent between two computers, while the experiment measured the time for a packet to be sent from one computer, via the event server, to a third computer, and back the same way; (ii) the "ping" utility sends packets as raw ICMP IP-packets, while Shout communicates over TCP connections[4]; (iii) the Shout server incurs some overhead in queueing, processing and re-sending events. However, the difference is not very big.

It is possible that further refinements to the server implementation could reduce the latency of the Shout system further. Profiling the server has revealed that allocating and releasing memory is responsible for a large part of the server's CPU consumption. Using a different memory allocator or having pools of "common" event sizes at the server's disposal, may reduce the load incurred by frequent memory allocations. At present, however, the Shout server's performance is deemed

[4]The connections have the Nagle algorithm disabled, by setting the `TCP_NODELAY` flag on the sockets used for communication.

Slaves	R=1	R=2	R=4	R=8	R=16	R=32
1	0.26	0.30	0.50	0.50	0.52	0.64
2	0.36	0.48	0.50	0.51	0.58	0.76
4	0.42	0.49	0.50	0.59	0.76	1.03
8	0.51	0.53	0.63	0.81	1.10	1.64
16	0.63	0.74	0.96	1.34	1.89	2.91
32	0.90	1.19	1.66	2.34	3.52	5.71
64	1.65	2.31	3.24	4.78	7.63	13.87

Slaves	R=64	R=128	R=256	R=512	R=1024
1	0.85	1.29	1.99	3.50	6.57
2	1.06	1.70	3.01	5.55	10.59
4	1.65	2.76	4.85	9.01	17.68
8	2.72	4.87	8.89	17.06	31.70
16	4.90	8.92	17.03	35.52	62.38
32	10.44	20.85	40.80	92.65	185.62
64	27.51	61.50	141.71	291.07	496.25

Table B.2: Mean roundtrip latency for sending 100000 ping events, and receiving R*100000 pong replies. All times are in milliseconds.

satisfactory.

Earlier results reported that the roundtrip latency of Shout were an order of magnitude higher than reported here. This discrepancy was due to a calculation error when converting values from seconds to milliseconds. The latency measurement experiments for Shout were re-conducted, with results as presented here.

B.7 Conclusion

This appendix has documented the Shout event system and its latency for delivering events. The Shout event system provides a mechanism for distributing different kinds of events between applications running on different computers. It is instrumental in enabling several of the systems and applications presented in this dissertation.

Appendix C

Network discovery mechanism

This appendix gives a very brief overview of the network discovery mechanism used by the 22 megapixel laptop and De-Centralized VNC implementations (Sections 6.3 and 6.4). It was implemented at the time due to lacking cross-platform implementations and APIs for Bonjour [163]. Bonjour is Apple's implementation of Zeroconf networking [164], where services are advertised on the local network for potential users of the service to discover.

The network discovery mechanism is based on providers of a service periodically multicasting advertisements on the local network. Clients of the service listen for UDP multicast packets on a pre-determined port. Clients can also send a solicitation which prompts all running providers to advertise their presence. This speeds up the discovery process.

Providers send advertisements at regular intervals ranging from 1 to 3600 seconds, beginning at 1 second, and then gradually increasing the delay between advertisements until the maximum of 3600 seconds is reached. The delay is reset whenever a solicitation is received from a client by a provider.

An advertisement contains a magic number and the TCP port number that the provider is listening on. The magic number is used by clients to check that the packet actually contains an advertisement, and is not just random data from a different multicast application that happens to use the same multicast address and port combination as the network discovery mechanism. The port number is used by the client to connect to the provider. For the 22 megapixel laptop, this is the port the NAD component listens for connections from the laptop component; for De-Centralized VNC, this is the port each VNC viewer listens on to receive connections from other VNC viewers. The De-Centralized VNC implementation further augments each advertisement with the area covered by the advertising viewer.

The network discovery mechanism is implemented in C using standard BSD sockets for communication and pthreads for threading. When the mechanism is ini-

tialized by the client, a multicast socket is created and a separate thread started to handle listening and sending packets on the multicast socket. Whenever an advertisement arrives, the advertisement is parsed and a callback function is called in the provider or client's code.

Appendix D

Pentium 4 and Xeon memory bandwidth

This appendix documents the memory bandwidth of the Pentium 4 and Xeon used to run the VNC server in the experiments measuring the performance of the Decentralized VNC and original VNC implementations.

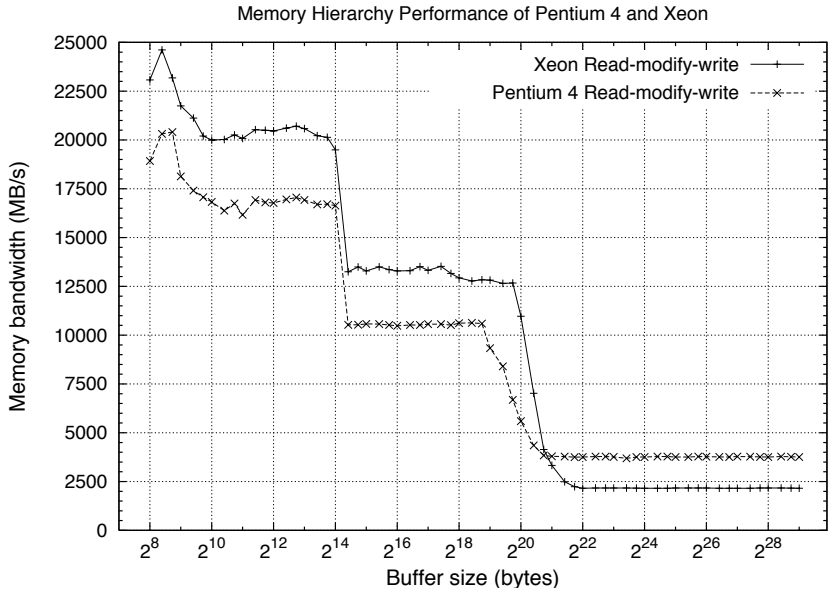


Figure D.1: Memory hierarchy performance of the Xeon and Pentium 4.

CacheBench, a part of the Low-Level Characterization benchmark suite [165], was run on both the Pentium 4 and Xeon. The benchmark measures performance in three ways: (i) Read; (ii) write; and (iii) read-modify-write. Of these three, the read-modify-write approach is most similar to the actions taken by the VNC server

when it executes a Copy Rect operation. Vectors of different lengths are read, modified and written back to memory, resulting in number describing how the different caches in the system interplay with the system's overall memory bus speeds. This is similar to a Copy Rect operation in that a block of memory is read, then moved to a different area in memory.

The results from running CacheBench on the Pentium 4 and Xeon are shown in Figure D.1. For small buffer lengths, the L1 and L2 caches are able to keep the apparent memory bandwidth high[1]. As the buffer lengths grow beyond the L2 cache capacity, the sustained CPU-to-memory bandwidth becomes evident. For the Pentium 4, the sustained memory bus bandwidth was 3.78 GB/second, while the Xeon sustains 2.16 GB/second. The Xeon has a higher memory bandwidth for small buffer sizes, but performs worse than the Pentium 4 for sustained read-modify-writes above 2 MB.

[1] Both the Pentium 4 and the Xeon have a 16 KB L1 cache. The Pentium 4's L2 cache is 1 MB, while the Xeon has a 2 MB L2 cache.

Appendix E

CD-ROM

A CD-ROM accompanies this dissertation. The contents of the CD-ROM are:

- A PDF copy of the dissertation.
- Individual PDF copies of the papers on which this dissertation is founded.
- The Hybrid vision- and sound poster.
- All the videos produced in conjunction with the research presented in this dissertation. The videos are further detailed in Chapter 1.
  - Hybrid vision- and sound-based interaction on display walls
  - Three years of the comic “M”
  - Device-free interaction spaces
  - Microarray visualization
  - The 22 megapixel laptop
  - De-centralized VNC: Two videos, one of the original VNC implementation, and one of the DVNC implementation.

The CD-ROM is also available as a disk image from:

`http://www.cs.uit.no/~daniels/`