UNIVERSITY OF TROMSØ UIT

FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Interactive Visualization on High-Resolution Tiled Display Walls with Network Accessible Compute- and Display-Resources

Tor-Magne Stien Hagen

A dissertation for the degree of
Philosophiae Doctor

May 2011

Abstract

The vast volume of scientific data produced today requires tools that can enable scientists to explore large amounts of data to extract meaningful information. One such tool is interactive visualization. The amount of data that can be simultaneously visualized on a computer display is proportional to the display's resolution. While computer systems in general have seen a remarkable increase in performance the last decades, display resolution has not evolved at the same rate. Increased resolution can be provided by tiling several displays in a grid. A system comprised of multiple displays tiled in such a grid is referred to as a display wall. Display walls provide orders of magnitude more resolution than typical desktop displays, and can provide insight into problems not possible to visualize on desktop displays. However, their distributed and parallel architecture creates several challenges for designing systems that can support interactive visualization. One challenge is compatibility issues with existing software designed for personal desktop computers. Another set of challenges include identifying characteristics of visualization systems that can: (i) Maintain synchronous state and display-output when executed over multiple display nodes; (ii) scale to multiple display nodes without being limited by shared interconnect bottlenecks; (iii) utilize additional computational resources such as desktop computers, clusters and supercomputers for workload distribution; and (iv) use data from local and remote compute- and data-resources with interactive performance.

This dissertation presents Network Accessible Compute (NAC) resources and Network Accessible Display (NAD) resources for interactive visualization of data on displays ranging from laptops to high-resolution tiled display walls. A NAD is a display having functionality that enables usage over a network connection. A NAC is a computational resource that can produce content for network accessible displays. A system consisting of NACs and NADs is either push-based (NACs provide NADs with content) or pull-based (NADs request content from NACs).

To attack the compatibility challenge, a push-based system was developed. The system enables several simultaneous users to mirror multiple regions from the desktop of their computers (NACs) onto nearby NADs (among others a 22 megapixel display wall) without requiring usage of separate DVI/VGA cables, permanent installation of third party software or opening firewall ports. The system has lower performance than that of a DVI/VGA cable approach, but increases flexibility such as the possibility to share network accessible displays from multiple computers. At a resolution of 800 by 600 pixels, the system can

mirror dynamic content between a NAC and a NAD at 38.6 frames per second (FPS). At 1600x1200 pixels, the refresh rate is 12.85 FPS. The bottleneck of the system is frame buffer capturing and encoding/decoding of pixels. These two functional parts are executed in sequence, limiting the usage of additional CPU cores. By pipelining and executing these parts on separate CPU cores, higher frame rates can be expected and by a factor of two in the best case.

To attack all presented challenges, a pull-based system, WallScope, was developed. WallScope enables interactive visualization of local and remote data sets on high-resolution tiled display walls. The WallScope architecture comprises a compute-side and a display-side. The compute-side comprises a set of static and dynamic NACs. Static NACs are considered permanent to the system once added. This type of NAC typically has strict underlying security and access policies. Examples of such NACs are clusters, grids and supercomputers. Dynamic NACs are compute resources that can register on-the-fly to become compute nodes in the system. Examples of this type of NAC are laptops and desktop computers. The display-side comprises of a set of NADs and a data set containing data customized for the particular application domain of the NADs. NADs are based on a sort-first rendering approach where a visualization client is executed on each display-node. The state of these visualization clients is provided by a separate state server, enabling central control of load and refresh-rate. Based on the state received from the state server, the visualization clients request content from the data set. The data set is live in that it translates these requests into compute messages and forwards them to available NACs. Results of the computations are returned to the NADs for the final rendering. The live data set is close to the NADs, both in terms of bandwidth and latency, to enable interactive visualization. WallScope can visualize the Earth, gigapixel images, and other data available through the live data set.

When visualizing the Earth on a 28-node display wall by combining the Blue Marble data set with the Landsat data set using a set of static NACs, the bottleneck of WallScope is the computation involved in combining the data sets. However, the time used to combine data sets on the NACs decreases by a factor of 23 when going from 1 to 26 compute nodes. The display-side can decode 414.2 megapixels of images per second (19 frames per second) when visualizing the Earth. The decoding process is multi-threaded and higher frame rates are expected using multi-core CPUs. WallScope can rasterize a 350-page PDF document into 550 megapixels of image-tiles and display these image-tiles on a 28-node display wall in 74.66 seconds (PNG) and 20.66 seconds (JPG) using a single quad-core desktop computer as a dynamic NAC. This time is reduced to 4.20 seconds (PNG) and 2.40 seconds (JPG) using 28 quad-core NACs. This shows that the application output from personal desktop computers can be decoupled from the resolution of the local desktop and display for usage on high-resolution tiled display walls. It also shows that the performance can be increased by adding computational resources giving a resulting speedup of 17.77 (PNG) and 8.59 (JPG) using 28 compute nodes.

Three principles are formulated based on the concepts and systems researched and developed: (i) *Establishing the end-to-end principle through customization*, is a principle stating that the setup and interaction between a display-side and a compute-side in a visualization context can be performed by customizing one or both sides; (ii) *Personal Computer (PC) – Personal Compute Resource (PCR) duality* states that a user's computer is both a PC and a PCR, implying that desktop applications can be utilized locally using attached interaction devices and display(s), or remotely by other visualization systems for domain specific production of data based on a user's personal desktop install; and (iii) *domain specific best-effort synchronization* stating that for distributed visualization systems running on tiled display walls, state handling can be performed using a best-effort synchronization approach, where visualization clients eventually will get the correct state after a given period of time.

Compared to state-of-the-art systems presented in the literature, the contributions of this dissertation enable utilization of a broader range of compute resources from a display wall, while at the same time providing better control over where to provide functionality and where to distribute workload between compute-nodes and display-nodes in a visualization context.

Acknowledgements

This dissertation would not have been possible without the help and support of many people, to whom I am very grateful.

I would like to thank my advisor Professor Otto J. Anshus for his guidance and support during my Ph.D. period. Otto has inspired and encouraged me to keep on researching, developing and refining the systems presented in this dissertation. I would also like to thank him for his patience and guidance in helping writing the papers that form the basis for this dissertation.

I would like to thank my co-advisor Associate Professor John Markus Bjørndalen for his support and guidance, and Professor Tore Larsen for all help and support. In addition, I would like to thank Professors Kai Li and Olga Troyanskaya for arranging for me to join them for one year at the Department of Computer Science at Princeton University.

I thank Dr. Daniel Stødle and Associate Professor Lars Ailo Bongo for discussions and support, and for all the great moments we have shared together, both in Tromsø and in Princeton. I would also like to thank Associate Professor Phuong Hoai Ha for discussions and support. I am also grateful for the discussions I have had with Espen Skjelnes Johnsen, Åge Kvalnes, Joakim Simonsson and Elizabeth Jensen. In addition, I would like to thank Eirik Helland Urke for allowing me to use the 13.3 gigapixel image of Tromsø for one of the systems developed as part of this dissertation.

I would like to thank my family and friends for supporting me throughout the work on my Ph.D. I am very grateful for your encouragement and motivation for these years.

I thank the technical and administrative staff at the Department of Computer Science at the University of Tromsø: Jon Ivar Kristiansen, Ken-Arne Jensen, Kai-Even Nilssen, Maria Wulff Hauglann, Svein Tore Jensen and Jan Fuglesteg. You have all made my work easier through the support you have provided.

I am grateful for 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, and (ii) 155550/420, Display Wall with Compute Cluster.

CONTENTS

Introduction 1

1.1 Visualization 5

1.2 Classification of Rendering Models 8

1.2.1 Single Logic Single Rendering (SLSR) 11

1.2.2 Single Logic Multiple Rendering (SLMR) 11

1.2.3 Multiple Logic Single Rendering (MLSR) 12

1.2.4 Multiple Logic Multiple Rendering (MLMR) 12

1.3 The Visualization Distribution Space 13

1.4 Problem Statement 15

1.5 Scientific Contributions 17

1.5.1 Principles 17

1.5.2 Models and Architectures 19

1.5.3 Artifacts 20

1.5.4 Impact 25

1.6 Summary of Papers 26

1.6.1 Background Papers for Network Accessible Compute- and Display-Resources 27

1.6.2 Push-Based Network Accessible Compute- and Display-Resource Papers 28

1.6.3 Pull-Based Network Accessible Compute- and Display-Resource Papers 28

1.7 Organization 29

Display Walls **31**

2.1 Display Wall Hardware 32

2.1.1 Display Technology 32

2.1.2 Computer System Technology 33

2.2 Display Wall Software 34

2.2.1 Virtual Network Computing (VNC) 35

2.2.2 Distributed Multihead X (DMX) 36

2.2.3 Chromium 37

2.2.4 Scalable Adaptive Graphics Environment (SAGE) 37

2.3 The Display Wall at the University of Tromsø 38

2.3.1 Hardware 38

2.3.2 Software 39

Graphics Processing Units **41**

3.1 Introduction to GPUs 41

3.2 The Compute Unified Device Architecture 42

Methodology **47**

4.1 Metrics 49

4.1.1 CPU Load 50

4.1.2 Memory Usage 50

4.1.3 Network Bandwidth Usage 50

4.1.4 Frame Rate 51

4.1.5 Latency 51

4.2 Cluster Wide Experiments 51

Network Accessible Resources **53**

5.1 Network Accessible Display Resources 53

5.2 Network Accessible Compute Resources 54

5.3 NAD - NAC Interaction 54

5.4 Background for Network Accessible Compute- and Display-Resources 56

5.4.1 Gesture-Based Touch-Free Multi-User Gaming on Wall-Sized High-Resolution Tiled Displays 56

5.4.2 Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core 65

5.4.3 Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors 76

Push-Based NADs and NACs 85

6.1 The NAD System 85

6.1.1 Related Work 86

6.1.2 Architecture 88

6.1.3 Design 90

6.1.4 Implementation 91

6.1.5 Experiments 92

6.1.6 Conclusions 99

Pull-Based NADs and NACs 101

7.1 WallScope 101

7.1.1 Related Work 104

7.1.2 Architecture 111

7.1.3 Design 113

7.1.4 Implementation 119

7.1.5 Experiments 122

7.1.6 Conclusions 135

Discussion 137

Conclusions 145

Future Work **149**

References **153**

Papers **173**

A.1 Gesture-Based, Touch-Free Multi-User Gaming on Wall-Sized, High-Resolution Tiled Displays 175

A.2 Liberating the Desktop 191

A.3 Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core 199

A.4 Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors 211

A.5 On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls 223

A.6 Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes 233

A.7 A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls 247

WallScope – Additional Resources **263**

B.1 Interactive Visualization of Data Feeds on High-Resolution Tiled Display Walls 263

CD-ROM **319**

List of Figures

Figure 1.1: The evolution of Intel desktop processors 2

Figure 1.2: A comparison of floating point performance between modern GPUs and CPUs 3

Figure 1.3: LDSView, one of the visualization systems developed as part of this dissertation 5

Figure 1.4: The visualization pipeline 6

Figure 1.5: The interactive visualization pipeline 7

Figure 1.6: The visualization process 8

Figure 1.7: The X11 graphics stack 9

Figure 1.8: The visualization distribution space 14

Figure 1.9: Quake 3 Arena and Homeworld being played on the display wall at the University of Tromsø 21

Figure 1.10: CUDAMandelbrot versus WallCPUMandelbrot 22

Figure 1.11: Three computers using the NAD system to mirror content from the local desktop onto a display wall 23

Figure 1.12: The graphical user interface of the demo client from where users can start and stop demos 24

Figure 1.13: WallGlobe showing a plane after a take-off from Langnes airport, Tromsø, Norway 25

Figure 2.1: Illustration of the display wall lab at the Department of Computer Science, University of Tromsø 31

Figure 2.2: VNC's traditional client-server model 35

Figure 2.3: VNC in a display wall context 36

Figure 3.1: The Compute Unified Device Architecture 43

Figure 4.1: Systems research methodology 47

Figure 4.2: The relation between idea, architecture, design and implementation. 48

Figure 5.1: Architecture of the parallel Quake 3 Arena 58

Figure 5.2: Parallel Homeworld architecture 58

Figure 5.3: The frame rate when running Q3A on 2x2, 3x3 and 7x4 tiles using Chromium, compared to the parallel version's frame rate running on 7x4 tiles ... 62

Figure 5.4: The additional latency introduced in Q3A's parallel version 63

Figure 5.5: The frame rate when running Homeworld on a single display, compared to 2x2, 3x3 and 7x4 tiles 63

Figure 5.6: The total number of frames drawn when running Homeworld on a single display, compared to 2x2, 3x3 and 7x4 tiles 64

Figure 5.7: The assignment of the Mandelbrot set for CPUMandelbrot 68

Figure 5.8: The assignment of the Mandelbrot set for WallCPUMandelbrot (static) 69

Figure 5.9: The assignment of the Mandelbrot set for WallCPUMandelbrot (dynamic) 69

Figure 5.10: The assignment of the Mandelbrot set for WallGPUMandelbrot ... 70

Figure 5.11: The assignment of the Mandelbrot set for CUDAMandelbrot 71

Figure 5.12: The assignment of the Mandelbrot set for CUDAMandelbrot when configured to send the output to a set of display nodes 71

Figure 5.13: Speedup factor of the parallel versions compared to CPUMandelbrot 74

Figure 5.14: The relation between speedup and resolution for the WallCPUMandelbrot versions compared to CUDAMandelbrot configured to send the output of each iteration to the display wall cluster 75

Figure 5.15: The arrangement of threads and warps for coalescing memory access to global memory 77

Figure 5.16: The time used for 30 000 invocations of the RMW object in global memory compared to atomic support in hardware (global memory) 81

Figure 5.17: The software- to hardware-ratio for the 30 000 invocations.....81

Figure 5.18: The time used for 30 000 invocations of the RMW object in shared memory compared to atomic support in hardware (global memory) 83

Figure 5.19: The software- to hardware-ratio for the 30 000 invocations.....83

Figure 6.1: The NAD architecture for single display configurations.....89

Figure 6.2: The NAD architecture for display wall configurations.....89

Figure 6.3: The two phases of the NAD protocol 90

Figure 6.4: The NAD protocol format.....92

Figure 6.5: Frame rate of the different applications at the three resolutions used in the experiments 95

Figure 6.6: Breakdown of average time usage for the main functional units of the NAC 96

Figure 6.7: Breakdown of average time usage for the main functional units of the NAD 96

Figure 6.8: NAC – NAD network bandwidth usage 97

Figure 6.9: CPU usage on the NAC 98

Figure 7.1: WallScope idea 102

Figure 7.2: WallScope architecture 112

Figure 7.3: The main components of the visualization systems.....114

Figure 7.4: Live data set design 116

Figure 7.5: Speedup when going from 1 to 26 compute nodes 128

Figure 7.6: The total number of displayed requests 129

Figure 7.7: The total number of completed requests.....129

Figure 7.8: The cumulative number of requested, completed, and displayed requests with full local caches.....130

Figure 7.9: The number of completed requests for the full local cache configuration 131

Figure 7.10: Time to request and simultaneously display 2432 JPG or PNG encoded image-tiles computed from a 350-page PDF document.....132

Figure 7.11: Speedup factor when requesting and simultaneously displaying 2432 JPG or PNG encoded image-tiles 133

Figure 7.12: Compute node utilization when rasterizing the 350-page PDF document to PNG images 133

Figure 7.13: Compute node utilization when rasterizing the 350-page PDF document to JPG images 134

List of Tables

Table 1-1: Flynn's classification of parallel computers 10

Table 1-2: Classification of rendering models 10

Table 2-1: The hardware specification of the Tromsø display wall 38

Table 5-1: Experiment summary 61

Table 5-2: Hardware- and software-platform 61

Table 5-3: Experiment summary 72

Table 5-4: Hardware- and software-platform 73

Table 5-5: Experiment summary 79

Table 5-6: Hardware- and software-platform 80

Table 6-1: Experiment summary 93

Table 6-2: Hardware- and software-platform 94

Table 7-1: Configurations for experiment series 4 to 11 123

Table 7-2: Experiment summary 124

Table 7-3: Hardware- and software-platform for experiment series 1 to 11 125

Table 7-4: Hardware- and software-platform for experiment series 12 to 15 126

Table 7-5: Time used to request 900 512x512-pixel (236 megapixels) image-tiles (experiment series 1 and 2) 127

Table 7-6: Average latency for a request to complete when using 28 compute nodes 134

Table 7-7: Time to request and simultaneously display 2432 JPG or PNG encoded image-tiles requested from the live data set's cache or from the local cache on each display node 134

List of Abbreviations

NAD	Network Accessible Display
NAR	Network Accessible Resource
NAC	Network Accessible Compute
VFB	Virtual Frame Buffer
VNC	Virtual Network Computing
RFB	Remote Frame Buffer
FPS	Frames Per Second
FOV	Field Of View
CPU	Central Processing Unit
GPU	Graphics Processing Unit
RAM	Random Access Memory
VRAM	Video RAM
VGA	Video Graphics Array
DVI	Digital Visual Interface
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
RGB	Red Green Blue
RGBA	Red Green Blue Alpha

BGR	Blue Green Red
RLE	Run-Length Encoding
API	Application Programming Interface
m	Meter
cm	Centimeter
s	Second
ms	Millisecond
CUDA	Compute Unified Device Architecture
LCD	Liquid Crystal Display
ILP	Instruction Level Parallelism
RAMDAC	Random Access Memory Digital-to-Analog Converter
CMP	Chip Multi-Processor
GPGPU	General-Purpose computation on Graphics Processing Units
GUI	Graphical User Interface
SSH	Secure Shell

CHAPTER 1

INTRODUCTION

Computational science has led to increasingly amounts of data produced by several different sources [1]. This has made possible the fourth paradigm of science[1] [1]. A challenge today is to extract valuable information from these large data volumes [2]. One way of attacking this challenge is to provide a tool that enables users to explore large volumes of data to extract meaningful information. One such tool is interactive visualization. Visualizations are critical to humans' ability to process complex data and an important part of the fourth paradigm for users to understand how data analyses and queries relate to each other [3].

This dissertation presents Network Accessible Compute (NAC) resources and Network Accessible Display (NAD) resources for interactive visualization of data on displays ranging from laptops to high-resolution tiled display walls. A network accessible display is a display having functionality that enables usage over a network connection. Network accessible compute resources produce content for network accessible displays. The workload distribution between NACs and NADs is determined by the hardware technology on both sides, including the interconnects. The separation between displaying and computing is motivated by one of Jim Gray's informal rules for approaching challenges related to large-scale scientific data sets. *"Bring computations to data, rather than data to the computations*" [2]. Thus by moving computations away from displaying and close to the data, while at the same time performing compute-side domain specific production of data for the display-side, future technologies on both sides can be tracked, and shared interconnect bottlenecks can be reduced.

The last decades the computer industry has seen a remarkable increase in computing power (figure 1.1). This increase follows Moore's Law [4], which projects the number of transistors that fits onto a single die to double every 18th to 24th months. Until early 2000, application performance scaled with transistor density because CPU frequency increased with advances in manufacturing technology and because CPU's were superscalar and exploited instruction level parallelism with replicated execution units and deep pipelines [5].

[1] The fourth paradigm of science is also referred to as the data-intensive paradigm.

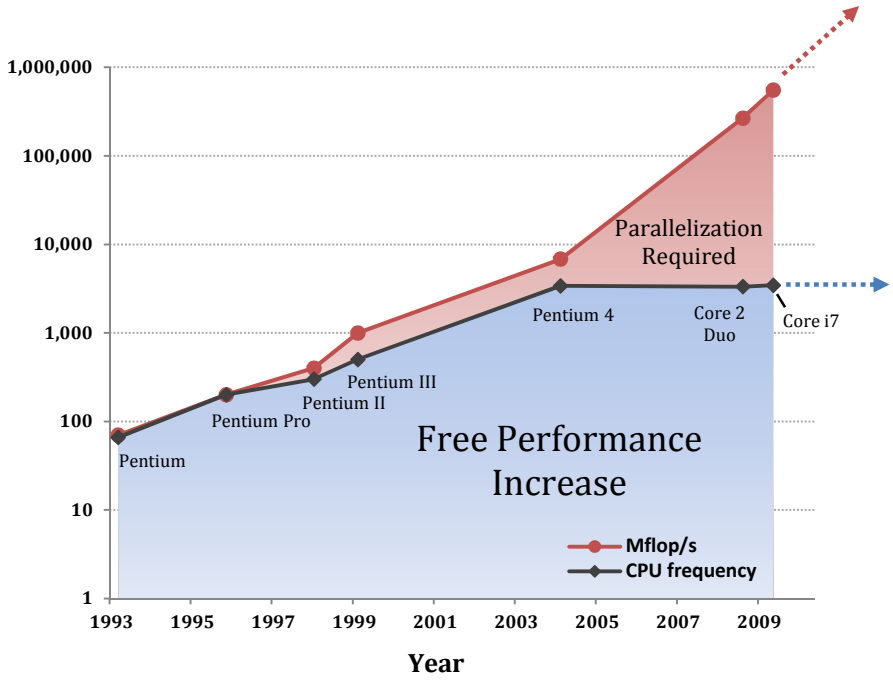

Figure 1.1: The evolution of Intel desktop processors[2]. (Figure inspired by [6]).

However, eventually three walls limited the increase in CPU frequency and instruction level parallelism [5]. These three walls were the power-wall [7] (power dissipation beyond the capacity of inexpensive cooling techniques), ILP-wall [7] (problems finding enough parallelism in instruction streams to utilize the processor), and the memory-wall [7] (gap between processor and memory speed making applications scale only with improvements in memory latencies). This forced CPU vendors into increasing performance by devoting transistors to CPU cores, inter- and intra-chip communication-systems, and cache- and memory-systems rather than increasing the frequency on single cores or creating deep-pipeline superscalar CPUs.

Today modern computers have become both multi- and many-core. State-of-the-art CPUs such as the TILERA TILE-Gx processor family [8] contains up to 100 cores per chip. Contemporary GPUs such as the NVIDIA GeForce GTX 580 [9] contains 512 cores. Compared to current commodity multi-core CPUs, the new generation GPUs is delivering over an order of magnitude the throughput due to transistors being devoted to data processing rather than control logic (figure 1.2). This can in some circumstances speed up data-parallel computations with several

[2] Data-source: http://www.intel.com

orders of magnitude, for instance as presented in chapter 5, section 5.4.2. (Graphics processing units are described more thoroughly in chapter 3).

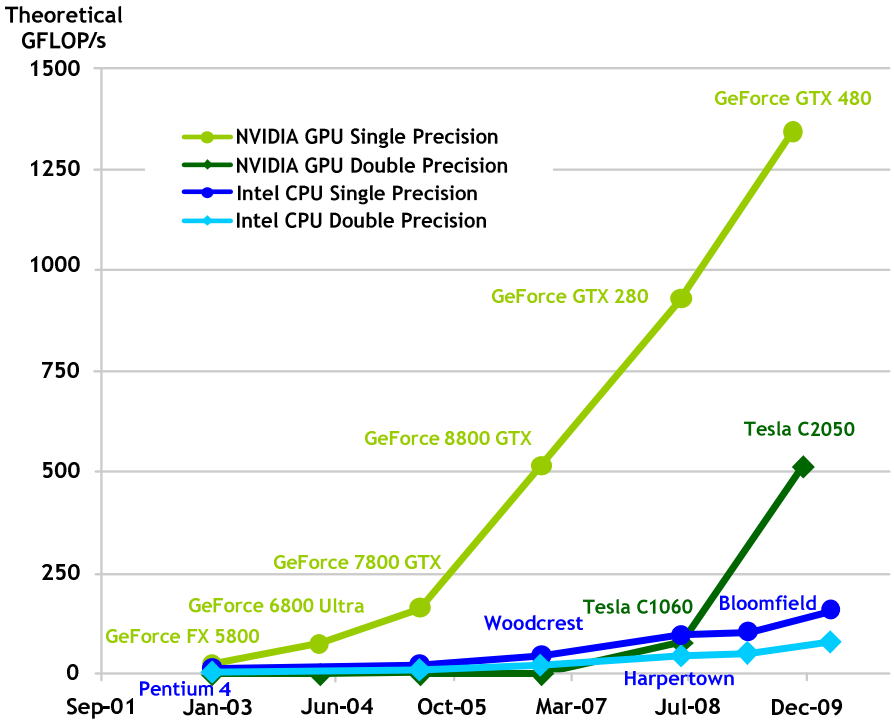

Figure 1.2: A comparison of floating point performance between modern GPUs and CPUs. (Source: NVIDIA CUDA C Programming Guide Version 3.2)[3].

The evolution of transistors has not only increased the processing capacity of multicore chips. Commodity sensors, wireless networks and DNA sequencing machines are just some of the devices that have benefited from the evolution of transistors. These kinds of devices combined with computational resources such as clusters, supercomputers and the widespread use of computers among more and more people around the world, are now producing data with a rate that has made possible a doubling of the total amount of data in the world every year [2]. This rapid increase in data size has led to the fourth paradigm of science.

The doubling of data each year is a challenge. There is a gap between the current data-analysis capabilities and the ability to produce data [2]. Thus, curating, analyzing, and visualizing data is important for keeping track with the increasing amounts of data produced. In addition, data might be located on remote locations.

[3] The figure has been converted from raster- to vector-graphics.

Thus, network bandwidth becomes a limiting factor for the amount of computational power that can be applied before being bottlenecked by network bandwidth and/or latency.

Computer networks are also benefitting improvements in transistor technology with state-of-the-art Ethernet having a theoretical performance of 10 gigabit/s. However, while the latest networking technologies can be utilized in local domains, remote data sets are typically accessed over the Internet where the latency is higher and bandwidth is lower. In addition, latency (and thus interactivity) is hard-limited to both physical location and the speed of light. For example, the round-trip time between the University of Tromsø and NTNU, two Norwegian Universities, is 14.5 milliseconds. This is within the latency required for acquiring data at 60 frames per second (16.6 milliseconds). The round-trip time between University of Tromsø and Princeton University, North-Norway to East-Cost USA, is 125 milliseconds, and not within this limit.

Visualization is an important tool for gaining insight into large amounts of data. The amount of data that can be simultaneously visualized on a computer display is proportional to the display's resolution. While computer systems in general have seen a remarkable increase in performance the last decades, display resolution has not evolved at the same rate. In 1979, the Three Rivers Computer Corporation sold the PERQ, a computer system with a 768x1024 (0.79 megapixel) bit map display [10]. Today, a computer display typically has a resolution ranging from one to four megapixels. However, several visualizations require much more resolution than this to be displayed in full detail. Even a picture taken with a modern digital camera today cannot be shown in full resolution on a modern display. For example, a display with a resolution of 2560x1600 pixels (4 megapixels) such as the Eizo ColorEdge CG303W display [11], can only show 29% of the pixels of an image taken with a Canon Digital IXUS 130 14.1 megapixel consumer camera [12].

To achieve higher resolution, displays can be tiled in a grid to produce a higher resolution image. Several displays tiled in such a grid are referred to as a display wall. Most contemporary graphics cards can drive only a couple of displays. Therefore, display walls are typically driven by several graphics cards connected either to a single computer or by a cluster of computers where each computer in the cluster drives up to a couple of displays. Single computer display walls have an upper limit on the number of displays that can be connected. In addition, such systems have a lower combined bus bandwidth compared to a cluster of computers. However, applications can often be run unmodified since all graphics cards can be presented as one unified resource.

Display walls allow for visualizations with orders of magnitude higher resolution than regular desktop displays. This makes them an interesting environment for visualizing large data sets such as planetary-scale data sets and gigapixel images. Additionally, regular application domains such as spreadsheet, word-processing and presentation-style applications can benefit from the resolution offered by a

display wall, allowing them to display much more content than they normally would (figure 1.3).

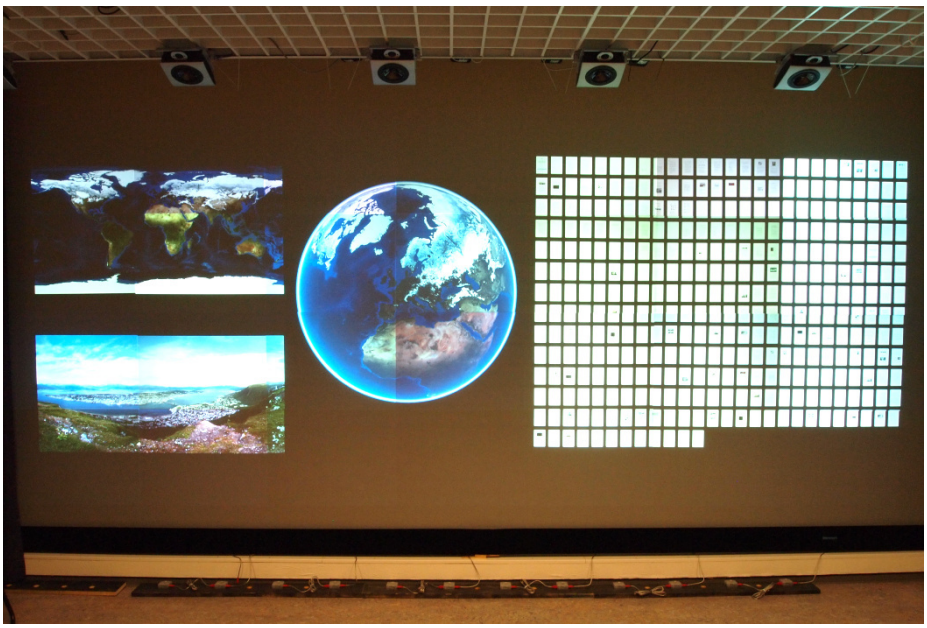

Figure 1.3: LDSView, one of the visualization systems developed as part of this dissertation, showing two gigapixel images, one virtual globe and a 350-page PDF document on a 22-megapixel display wall.

However, the distributed and parallel architecture of a display wall (described chapter 2) combined with data located on possibly several remote locations, make it non-trivial for interactive visualizations of data from local and remote data- and compute-resources. To explain this, the next section elaborates on visualization.

1.1 VISUALIZATION

This dissertation employs the definition from McCormick et al. to describe visualization [13]: *"Visualization is a method of computing. It transforms the symbolic into the geometric, enabling researchers to observe their simulations and computations. Visualization offers a method for seeing the unseen. It enriches the process of scientific discovery and fosters profound and unexpected insights. In many fields it is already revolutionizing the way scientists do science"*. Based on this definition and inspired by [14] this dissertation defines visualization as *the process of transforming data into a visual representation as pixels*.

The different steps data passes through before it ends up as a visual representation are referred to as the visualization pipeline, shown in figure 1.4. These steps have been identified and refined by Haber and McNabb in [15].

Figure 1.4: The visualization pipeline.

The first stage of the visualization pipeline is data enrichment/enhancement. This step operates on raw simulation data and techniques such as interpolation and smoothing are used to obtain data with desired locations and distribution. The output of the data enrichment/enhancement stage is referred to as derived data. This data is passed to the visualization mapping stage where data is mapped to an Abstract Visualization Object (AVO). The mapping of derived data to AVO properties is performed using transfer functions. A transfer function interprets and translates data values to AVO properties such as color values and vertices. The final part of the visualization pipeline is the transformation of the abstract visualization object into an image. This is performed in the rendering step.

Rendering is the process of rasterizing data (geometry, textures, materials and lights) into pixels. The rendering pipeline has been studied and developed during the last decades. Early approaches did rendering in software on the CPU. However, today more of the rendering functionality has been moved to the graphics cards containing specialized hardware for graphics operations. Operations such as transforming vertices between coordinate systems and eventually to screen space, transferring textures with high throughput, and rasterizing groups of vertices into pixels, utilize the specialized parallel architecture of the graphics card.

The Haber McNabb visualization pipeline does not describe visualization in an interactive context. Figure 1.5 shows a modified version of this pipeline which illustrates interactive visualization as defined by this dissertation. In the interactive pipeline, original data is filtered to get data of interest. This data is processed and mapped to an abstract visualization object. The abstract visualization object is passed to the rendering pipeline, which creates pixels typically shown on a computer display. A user interacts with the output of the visualization using some kind of interaction device. Depending on the user input, up to several stages in the visualization pipeline are triggered and data flows through the pipeline to produce a new image.

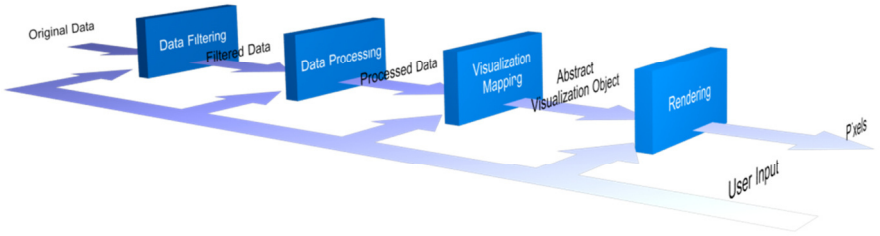

Figure 1.5: The interactive visualization pipeline (figure inspired by [16]).

In this dissertation, the steps before visualization mapping are referred to as the computation. The input to the computation is original data. The output of the computation is processed data, which is mapped to an abstract visualization object and forwarded to the rendering stage. Rendering produces pixels. Normally these pixels are stored in the frame buffer of the graphics card. The frame buffer is a dedicated memory region, usually located in graphic card memory, which is scanned and output as pixels onto the display by the Random Access Memory Digital to Analog Converter (RAMDAC).

The cost of producing data at the different stages of the visualization pipeline varies. For example changing the camera viewing angle can be as simple as rotating the scene, which only requires feedback to the rendering stage by changing the viewing transformation matrix. However, moving to an "unseen" part of the scene could require new data to be processed and thus requested from possibly several remote locations. Depending on the bandwidth and the computational power of the system, this can introduce orders of magnitude the latency compared to a local view change, which can be handled by the local graphics card.

Figure 1.6 shows the visualization process as defined by this dissertation, including the stages data passes through before it ends up as pixels in the frame buffer. Although the visualization pipeline in many cases might be thought of as a process for transforming scientific data, all applications displaying output to a display will follow some of the steps in the visualization pipeline. For example, an XLSX file can be visualized by opening it with a spreadsheet application such as Microsoft Excel [17] or OpenOffice Calc [18]. Excel and Calc contain, in combination with the installed software environment and operating system, the computation and rendering functionality needed for transforming the XLSX file into pixels.

The output of a visualization depends on the computation, visualization mapping, and rendering functionality, which varies between operating systems and software installs. Therefore, sharing pixels is sometimes the only way of making sure visualizations remain the same over different software platforms.

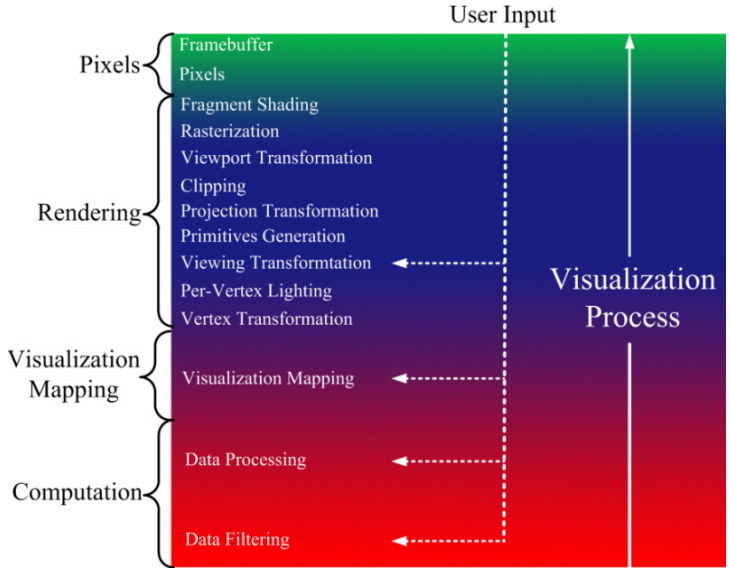

Figure 1.6: The visualization process.

1.2 Classification of Rendering Models

Figure 1.7 shows the design of a modern X11 [19] graphics stack. Applications usually use a graphics library such as GIMP Toolkit (GTK) [20] or Qt [21] for the graphical user interface. GTK uses the GIMP Drawing Kit (GDK) [22] for drawing basic primitives. These primitives are generated using the rendering primitives provided by the X server through Device Independent X (DIX) and Device Dependent X (DDX). The graphics output of the commands from the device driver is a pixmap with an associated texture binding. These textures are composited into the frame buffer by the compositing manager using the GPU.

OpenGL [23] applications in X use GLX [24] for sending OpenGL commands to the graphics card. This can be done in two ways: Indirect or direct. In indirect rendering, GLX encodes the commands and sends them to the X server for rendering. The X server receives the commands and forwards them to the server-side OpenGL implementation (which in many cases is implemented in Mesa [25], a system supporting both hardware (GPU) and software (CPU) supported OpenGL). When direct rendering is used, GLX loads a client-side Direct Rendering Infrastructure (DRI) module that communicates with a kernel Direct Rendering Manager (DRM) module to bypass the X server and instead communicate directly with the graphics card hardware. This gives an application better performance since the extra overhead introduced by the X server is removed. However, the commands cannot be sent over the network to a separate X server.

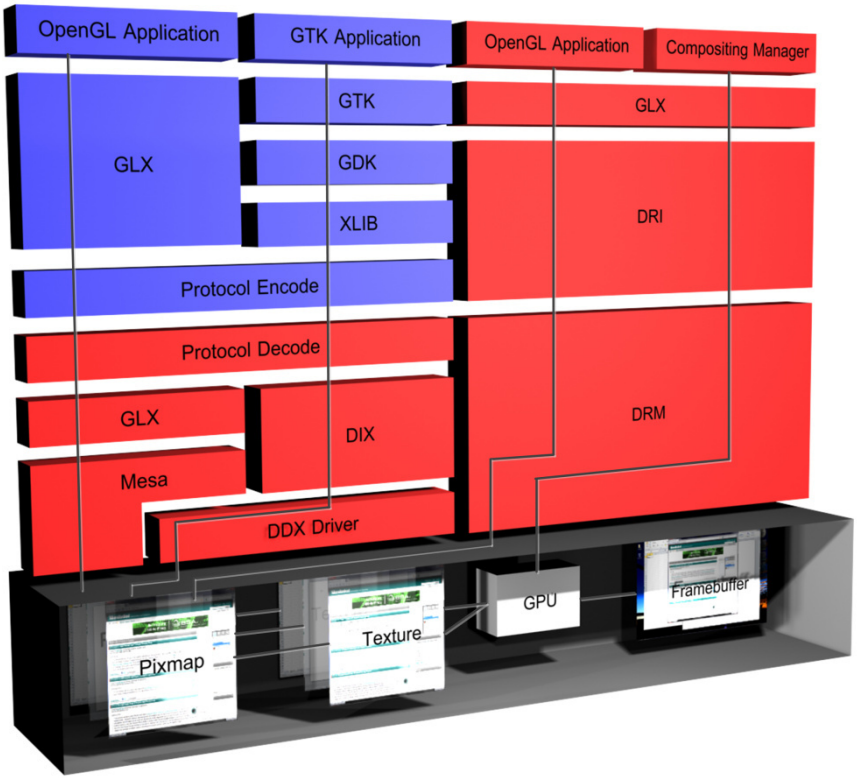

Figure 1.7: The X11 graphics stack. Red represents components running on the same computer as the X server. Blue represents components part of an X client potentially running on a separate computer. The gray box is the graphics card (figure inspired by [26]).

When distributing a visualization from one to multiple computers, several choices can be made on where the different parts of the visualization pipeline and graphics stack are distributed over the available computers. X11 has a networked design and many visualizations not requiring DRI can be remotely rendered by utilizing X11. However, for parallel graphics rendering, the X11 rendering approach can saturate the network since the same display commands must be sent to several computers [27].

The parallelization approach will in many cases dictate the performance of the final system. At each step in the visualization process, data of varying size will pass over interconnects with different bandwidths and latencies. The parallelization involves finding the place in the visualization pipeline where the data size to interconnect bandwidth gives a good tradeoff and does not saturate the computers interconnects.

In 1994, Molnar et al. [28] classified parallel rendering based on where the sort from object space to screen space occurs. Based on where the sorting of primitives are done three classes where identified: (i) Sort-first, (ii) sort-middle, and (iii) sort-last. In sort-first, graphics primitives are distributed early in the rendering pipeline, during geometry processing. The screen is divided into disjoint regions and graphics processors are responsible for all rendering calculations affecting their region. In sort-middle, primitives are redistributed between geometry processing and rasterization. Arbitrary subsets of primitives are partitioned between available geometry processors, and rasterizers are assigned disjoint regions of the screen, as in sort-first. During each frame, primitives are transformed and classified with respect to screen region boundaries by geometry processors, and then sent to the rasterizer responsible for that screen region. In sort-last, the sorting is deferred until all primitives have been rasterized into pixels. Subsets of primitives are divided between graphics processors independent of the screen position. The rasterized pixels are then transferred over a network to compositing processors for pixel visibility resolving.

Because of the strong binding in graphics card geometry processing (where object space to screen space transformations are tightly coupled with rasterization), most distributed rendering models are based on a sort-first or a sort-last method [29].

In 1966, Flynn classified parallel computers into four classes [30]. These were Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD), shown in table 1-1.

Table 1-1: Flynn's classification of parallel computers

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

Inspired by Flynn's classification of parallel computers, this dissertation categorizes rendering models into Single Logic Single Rendering (SLSR), Single Logic Multiple Rendering (SLMR), Multiple Logic Single Rendering (MLSR), and Multiple Logic Multiple Rendering (MLMR), shown in table 1-2.

Table 1-2: Classification of rendering models

	Single Logic	Multiple Logic
Single Rendering	SLSR	MLSR
Multiple Rendering	SLMR	MLMR

Based on the visualization pipeline, the graphics stack and the classification of parallel rendering models, the next sections describe the approaches that can be taken to distribute a visualization over a set of inter-connected computers, with a description of the pros and cons of each selected approach.

1.2.1 Single Logic Single Rendering (SLSR)

This rendering model is used by several applications and operating systems today, with a one-to-one correspondence between application logic and rendering. Often the application logic and rendering is executed on the same computer. However, this is not necessarily the case. The X window system [31] allows the client to be executed on another computer than the X server, which can be seen by the protocol encode/decode in figure 1.7. This allows the application logic to be executed on another computer than the rendering, even though there is a one-to-one correspondence between logic and rendering.

Applications implemented using this model can be distributed over a set of interconnected computers in three ways. The first way is to resize the frame buffer to have a resolution corresponding to the total resolution of the frame buffers of all interconnected computers, and then distribute the rendered content from this frame buffer to the corresponding frame buffers of each computer. This approach can be performed without modifying applications, allowing proprietary systems to be used without requiring any source code modifications. However, in many cases this approach requires a virtual frame buffer hosted in main memory because the total resolution is larger than the size that could be hosted on the graphics card. This implies that the functionality provided by the GPU cannot be utilized and the CPU must do all graphics operations. Using this configuration the network and CPU often become bottlenecks since the CPU must do rendering and compression of pixels, which are then transferred over the network [32].

The second and third approaches involve converting the application to an MLMR model or modifying the underlying graphics libraries to achieve an SLMR model. These two approaches are described in section 1.2.4 and section 1.2.2, respectively.

1.2.2 Single Logic Multiple Rendering (SLMR)

This rendering model involves performing all visualization steps on one computer, intercepting the rendering commands and distributing the commands over the network to the computers responsible for rendering them.

There are several approaches for how this rendering model can be realized:

1. Intercept the rendering commands going to the graphics library. (This approach is used by systems such as Chromium [33], where the user level bindings to the OpenGL functions are replaced by Chromium's

modified library for distributing rendering commands over the network to the responsible rendering nodes).

2. Utilize the design of an already networked windowing system such as X11 and Distributed Multihead X (DMX) [34] for presenting a set of networked X servers as a single X server instance to the clients.

3. Intercept the rendering commands going to the display driver and distribute these to the responsible computers. (Thin-client Internet Computing (THINC) [35] uses this approach for a remote desktop system).

In many cases, these approaches need no modifications to the applications themselves. However, the underlying graphics library or display driver must be modified. This approach might also generate much network traffic, thereby saturating the network [27].

1.2.3 Multiple Logic Single Rendering (MLSR)

Several games can be categorized into this rendering model, where the game logic such as physics, artificial intelligence (AI) and sound processing is distributed over a set of processors, and a separate processor is responsible for performing the main rendering based on the global state of the game logic.

Several X clients rendering to a single X server can also be categorized into the MLSR model. Distributing an application implemented using this model involves the same approaches as distributing an application implemented using an SLSR model.

1.2.4 Multiple Logic Multiple Rendering (MLMR)

The MLMR model involves having both distributed logic and distributed rendering. This model is used by most of the work presented in this dissertation. For applications not originally designed for distributed rendering, modifying them to adhere to this model can involve much work, but will often result in good performance. The reason for this is that the data sent over the network to keep multiple replicas synchronized often is much less than sending pixels or rendering commands over the network. For example, sending camera coordinates (matrix of 16 floats/doubles) and a global clock (1 int/long) requires a maximum of 136 bytes (16x8 + 1x8) to be transferred over the network per frame. For a frame buffer with a resolution of 1024x768 pixels, the worst-case scenario (transferring all pixels) requires 2304 kilobytes ((1024x768x3)/1024) to be transferred over the network per frame.

One way to modify a single rendering application to use this model is to execute a replica of the application on each computer and modify the view frustum of

every computer so that they render only their part of the view, relying on the graphics cards clipping and culling functionality to remove invisible geometry. In many cases, this configuration will result in good performance [36]. However, it requires synchronization of the computation and rendering between all computers and, in addition, can generate much data to the rendering pipeline because every computer computes and passes all data to the graphics card, which then needs to clip away the invisible parts. This can be solved by modifying the application to pass only visible geometry to the graphics card, which can require a formidable amount of modifications if the original application is not designed for it.

1.3 The Visualization Distribution Space

As described in the previous sections there are several approaches for how to distribute a visualization over a set of display nodes. Each approach has different tradeoffs. Some approaches, such as the virtual frame buffer approach, work for proprietary software solutions, but might result in bad performance. Others, such as modifications to the applications themselves, can require more work, but might result in better performance.

The aforementioned approaches to distribute a visualization involves two sides: A compute-side and a display-side. For SLSR and MLSR applications, where a sort-last pixel transfer approach is used, the compute-side is the computer rendering to the (virtual) frame buffer and the display-side is the computers receiving the rendered pixels. For SLMR approaches, the compute-side is the computer where the graphics commands are intercepted and the display-side is the side receiving and rendering these commands. In the MLMR approaches, there is no compute-side since the entire visualization has been distributed to all computers. However, a compute-side can be introduced by moving some of the computational work from the display nodes to back-end compute nodes, such as a cluster or a supercomputer.

Figure 1.8 shows the solution space for distributing a visualization between a compute-side and a display-side (including the communication between these sides). The figure is divided into two parts, compute-side and display-side processing. Red represents computation, the transition from red to blue represents the abstract visualization object mapping, blue represents rendering and green represents rendered pixels (the color representations are the same as in figure 1.6). The layer between compute-side and display-side is communication. The area marked pre-processed (the gray area) describes data that has been computed and stored, usually on disk.

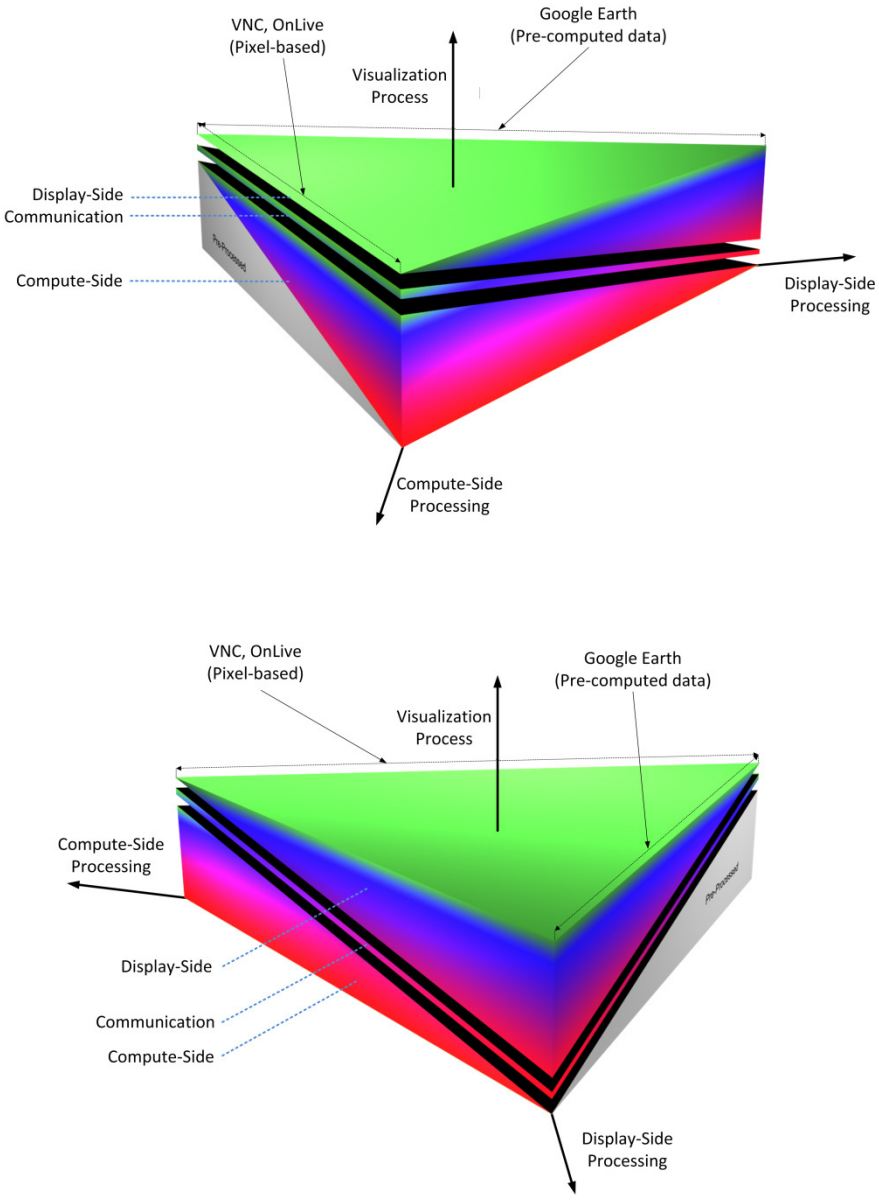

Figure 1.8: The visualization distribution space (shown from two different view-points) describes the possible distributions of a visualization between a compute-side and display-side (gray represents pre-computed data, red represents computation, the transition between red and blue represents visualization mapping, blue represents rendering, and green represents rendered pixels).

Pixel-based systems like VNC [37] and OnLive [38] are marked in the figure. These systems perform all application logic on the compute-side and exchanges pixels and events between the compute-side and display-side. Google Earth (described in chapter 7, section 7.1.1) uses data that has been pre-processed on the server-side. The data ranges from images to elevation data, but common for all is that the compute-side serves this data without processing the data before it is delivered to the display-side.

Workload distribution between the compute- and display-side is in many cases pre-determined for each system. For example, several remote desktop systems, such as VNC, use pixel transfer protocols. The problem with pre-determined workload approaches is the lack of adaption to the compute- and display-side hardware. A thin client with limited computational power, such as limited hardware acceleration for graphics operations, would most likely benefit a remote rendering approach such as VNC where the compute-side performs all rendering. However, for a desktop computer with a multi-core CPU, and a many-core GPU, this approach would most likely not utilize the computational power of the desktop computer. This dissertation addresses this challenge using a data set containing data customized for the particular application domain of the display-side (see chapter 7 for a detailed description of the data set and the systems built to take advantage of this).

1.4 Problem Statement

Performing interactive visualization on high-resolution tiled display walls is a challenge. This challenge is caused by several factors.

Firstly, display walls often have a distributed and parallel architecture [39] [40] [41] [42]. Visualization systems based on distributing graphics output from a central computer allows for running proprietary software, but have scalability problems when the number of display nodes is increased [32] [27]. Distributed visualization systems have better scalability [27] [36], but does not allow for proprietary software. Thus, one of the challenges with display walls today is to support proprietary applications with good interactivity.

Secondly, distributed visualization systems require synchronization of state and display-output in order for a set of display-nodes to appear as one unified resource. Systems performing visualization using lock-step approaches and/or barriers [36] [43] [44] introduce multiple points of failure and, based on the implementation, can have a performance limited to the most heavily loaded node.

Thirdly, supplying visualization systems running on display walls with data from local and remote data sources often requires orders of magnitude the data amounts that a visualization system running on a single node does. This can be a problem when requesting data from remote data sets, where network bandwidth might be low and latency high. In addition, processing this data on the display

nodes implies duplicate data processing since data might overlap between tiles of the display wall.

Fourthly, utilizing both local and remote compute resources for computation of domain specific data for a display wall is a challenge today. Grids and supercomputers have strict security policies such as the lack of opening outgoing connections, which complicates distributed access from a set of visualization clients. Desktop computers are getting more powerful, but there are no simple ways of integrating them with a visualization system running on display walls to provide domain specific data.

Finally, interacting with high-resolution wall-sized displays requires visualization systems that can benefit from several interaction systems. A display wall's combination of resolution and size allows users to stand close to the display wall to look at details. Using a standard keyboard and mouse in these circumstances is impractical, since a mouse usually needs a table surface to work, and wireless keyboards are impractical to carry around [45]. The size of a display wall allows several users to use it at the same time. This approach is not supported using the traditional single cursor approach. Supporting alternative interaction devices is therefore integral to interacting with wall-sized high-resolution displays.

The hardware trend in computer architectures indicates a continuing increase in the number of cores and thereby the computing power. At the same time, more work is being outsourced to remote internet services in the cloud. One example of this is the OnLive service [38] where a user's games are running on remote servers and pixels are transferred to the user's computer. This solution has some positive implications for the user: (i) The user does not need a state-of-the-art computer to be able to play the latest games, since the CPU- and GPU-intensive parts of the visualization are running in the cloud; (ii) the user does not need to own several hardware devices, i.e. Sony Playstation 3 (PS3) and Microsoft Xbox 360 to be able to play the latest games; and (iii) the user does not need to be involved in setting up the environment or keeping the environment updated to the same degree as if the games where running locally (i.e. making sure the latest display driver is installed to get good performance of the GPU). However, the increase in processing power of personal computers and the outsourcing of CPU and GPU intensive tasks to remote internet services does not follow the same track. Instead of moving all computations to the cloud, a more balanced approached could be used. If the client-side of the cloud is a portable device, then using remote rendering might be a good approach. However, for a more powerful workstation the rendering could be handled by the local graphics card, instead of displaying remotely rendered pixels. Another problem with outsourcing work to remote services is that some data might be tied to a certain computer because of compatibility and/or copyright issues. However, processed or selected parts of the data might not be covered by these limitations. Systems for visualization could take advantage of this by offering user selected and/or processed data portions to be shared with the visualization system, while the original data is kept at its main source.

Displaying output from a computer desktop onto remote computer displays is problematic. There exists multiple systems for doing this, but none of them allows for cross-platform sharing of desktop output without making modifications to the local software install, including opening firewall ports. Additionally, remote desktop systems are based on a pull-passed architecture. This complicates using remote screens, especially in display wall contexts, since the connection must be initiated from the remote screen. In addition, some systems only allow the entire desktop to be mirrored, and not only user selected portions of it, leaving no private space for the user to work on the local desktop. Displaying desktop output from a local computer to a display wall is even harder because of its distributed and parallel architecture, and the fact that users are running different operating systems and remote desktop software on their computers.

Sharing projectors and displays in meeting room environments is another problem today. Even a single projector in a meeting room can cause problems for presentations because it may fail to detect input signal from the computer used for the presentation. For mirroring, the resolution of the local display needs to match the resolution of the projector. In many cases, the projector's resolution is lower than the resolution of the local display. This implies that the resolution of the local display must be resized to the projectors lower resolution. For some operating systems such as Windows [46], this will rearrange the desktop icons and thereby modify the local desktop layout, even after the projector is disconnected from the computer. In addition, there are no systems for simple sharing of projectors and displays from multiple computers.

1.5 Scientific Contributions

This section presents the scientific contributions (principles, models and architectures, and artifacts) researched and developed as part of this dissertation.

1.5.1 Principles

This subsection presents the principles formulated based on the research conducted as part of this dissertation.

Establishing the end-to-end Principle through customization

The principle of establishing the end-to-end principle through customization states that the end-to-end principle can be established between a client and a server by customizing one or both sides. In this dissertation, the principle is used for the setup and interaction between a display-side and a compute-side in a visualization context. It involves customizing the compute-side (the producer of data) by the display-side to produce customized data. By following this principle,

display resources and compute resources will always be compatible, since the protocol between them is dictated by the display-side.

Several compute-side resources might use a display-side resource simultaneously. To protect the display-side resource, while at the same time providing a compute-side resource with information about how to produce data, the customization process provides the compute-side with the following information:

1. How to produce data.
2. How the display-side resource is shared.

Based on this information the compute-side produces data suitable and customized for the display-side. The display-side uses this data to produce the final image written to the frame buffer.

The customization of the compute-side is accomplished in two different ways:

1. Physically (directly executing code on the resource).
2. Virtually (using a third party mapper between customized and compute resource behavior).

When a compute-side allows for custom software execution, the customization is done by executing code directly on the compute resource. If not, an intermediate third party handles the actual mapping of customized behavior, thereby presenting a virtual customization of the remote resource, leaving the resource itself untouched.

PC – PCR DUALITY

The PC – PCR duality principle states that a user's computer is both a Personal Computer (PC) and a Personal Compute Resource (PCR). For normal usage, a user is interacting with a computer using attached input devices such as a keyboard and a mouse. The output from the applications is written to the frame buffer, which is scanned out on the attached display(s). In addition to this usage, personal desktop applications can be used as resources available to other clients. The production of data utilizes the fact that the desktop computer has a personal desktop install that can produce customized data from local user-selected data stored on the computer, or from data that sent to the computer. Clients can use the personal compute resource to produce compatible data from data that might be incompatible with the software installed on the client computer. A simple example involves converting from one image format to another image format. This illustrates a conversion between two resolution dependent formats. Several data formats are resolution independent, such as vector graphics. Normally these formats are constrained to the resolution of the personal computer's display(s). However, by decoupling the conversion of vector formats from the local frame

buffer's resolution (for example by redirecting the rendering to off-screen textures or by producing tiled output), the produced data becomes independent of the resolution of the local display. Other examples are conversion between vector formats, such as DOC/DOCX to PDF, or vector format – vector format – pixel format conversions such as DOC/DOCX to PDF to image-tiles.

Domain Specific Best-effort Synchronization

The principle of domain specific best-effort synchronization states that for distributed visualization systems state handling can be performed using a best-effort synchronization approach, where visualization clients eventually will get the correct state after a given time period. The principle applies for state handling when two properties are present in a system:

1. The participants of the state synchronization have established a pre-agreement on their arrangement (the display's placement in the display wall grid).

2. Losing a state synchronization message does not affect the logic.

When these two properties are present in a system, state synchronization can be handled using a centralized push-based heartbeat mechanism. This approach enables central control of refresh-rate by suspending participants when waiting for state heartbeat messages while, at the same time, avoiding multiple points of failure by not requiring feedback from participants to the provider of the state messages. In addition to avoiding multiple points of failure, a visualization system's load on the hardware can be controlled from a single location by adjusting the update rate of the heartbeat state messages.

1.5.2 Models and Architectures

This section presents the models and architectures developed as part of the conduced research.

Network Accessible Display Model

The Network Accessible Display (NAD) model was introduced in [47]. In this dissertation, the model is refined to create a display with the capability to customize a compute resource in order for the compute resource to be able to use the display over a network connection. A two-phase customization protocol integrates the compute resource with the NAD enabling communication between the two parties.

Network Accessible Compute Model

The Network Accessible Compute (NAC) model presents a customized view of compute resources to other clients. Clients only implement a single interface to the NACs and can thereby utilize existing and additional compute resources without understanding their internal behavior. The NAC model defines two types of compute resources: Static and dynamic. Static NACs are compute resources that are considered permanent once installed in the system. Compute resources in this category are grids, clusters and supercomputers. Dynamic NACs are compute resources that are volatile in the sense that they can become a NAC on-the-fly to provide data for a client-side, and then later remove itself. Compute resources in this category range from laptops to hand held devices.

Live Data Set architecture

The live data set architecture is a data space architecture used for separating a display- and compute-side for transparent communication. The live data set contains data customized for the specific application domain of the display-side. The live data set architecture is used to realize the NAC model. The live data set accepts requests from clients, which it translates to compute related messages and forwards to available compute resources (NACs). From the display-side's perspective, the live data set contains multiple data sets, which are a function of the data that the compute resources can produce. The data appears to the display-side as pre-computed. However, several compute requests might be performed to fulfill a client's request.

1.5.3 Artifacts

Several artifacts have been developed to document and demonstrate the research conducted in this dissertation. This section gives a summary of these artifacts.

WallQuake

WallQuake (shown in figure 1.9) is an MLMR parallelization of the First Person Shooter (FPS) game Quake 3. The game was parallelized to run on a display wall and integrated with a touch-free multi-user interface for device-free interaction, with the purpose of documenting the performance of using the touch-free interface with an FPS game. WallQuake was compared with an SLMR approach to document the performance of two different ways of parallelizing a game for a display wall.

WallHomeworld

WallHomeworld (shown in figure 1.9) is an MLMR parallelization of the Real-Time Strategy (RTS) game Homeworld. Similar to WallQuake this game was

integrated with a touch-free multi-user interaction system to document the performance of using the touch-free interface with an RTS game. WallHomeworld was compared to a single display version to demonstrate how the MLMR approach enables the game to scale to several display nodes.

Figure 1.9: Quake 3 Arena and Homeworld being played on the display wall at the University of Tromsø. The persons at the left and right are playing Quake 3 Arena against each other. The person in the middle is playing Homeworld.

WallCPUMandelbrot

WallCPUMandelbrot (shown in figure 1.10) is a parallelization of the embarrassingly parallel Mandelbrot set computation. It comprises a static and a dynamic workload version, and is designed to run on a display wall. These versions were created to compare their parallel speedup with a state-of-the-art GPU.

WallGPUMandelbrot

To speed up the execution of WallCPUMandelbrot, another version was created to utilize previous generation graphics cards. This version maps the Mandelbrot set computation to the OpenGL Shading Language (GLSL) API, and executes the computation in a fragment shader. The purpose of this version is a comparison between the Mandelbrot set computation running on previous generation graphics cards on a display wall with a state-of-the-art GPU version.

CUDAMandelbrot

CUDAMandelbrot (shown in figure 1.10) is a GPU parallelization of the Mandelbrot set computation. It is implemented using the Compute Unified Device Architecture (CUDA) [48]. This version of Mandelbrot is compared to both WallCPUMandelbrot and WallGPUMandelbrot to document the performance of a state-of-the-art GPU.

Figure 1.10: CUDAMandelbrot (displaying on the left screen) versus WallCPUMandelbrot (running on the display wall).

CUDASync

Several state-of-the-art GPUs do not have the ability to communicate safely intra-CMP. To investigate this problem CUDASync was created. It is a library for enabling CUDA GPUs lacking support for communication through global memory to do so in a fault-tolerant way.

The NAD System

To demonstrate the NAD model a system was developed to support mirroring of multiple desktop regions from local computers onto remote displays, among others a 22-megapixel display wall (figure 1.11). By adhering to the NAD model and the principle of establishing the end-to-end principle through customization, this system has the following properties:

- Removes the need to have pre-installed matching remote desktop systems on the client-side and server-side.
- Does not require permanent installation of third party software.
- Removes the need to open firewall ports on computers using the NAD.
- Allows for several user-selected parts of the desktop to be mirrored onto NADs without being limited to mirror only the entire desktop.

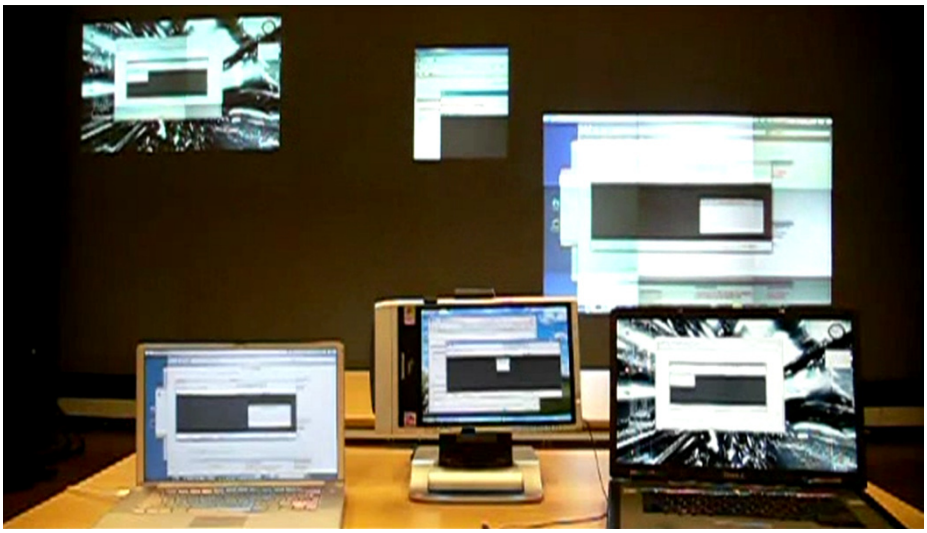

Figure 1.11: Three computers using the NAD system to mirror content from the local desktop onto a display wall. The computer on the left (Mac PowerBook running OS X) mirrors the entire desktop (scaled up two times) onto the right portion of the display wall. The center computer (Fujitsu Stylistic Tablet running Windows XP) mirrors the upper left part of the desktop (scaled up two times) onto the center of the display wall. The computer at the right (Dell Precision M90 running Linux Ubuntu) mirrors the entire desktop (no scaling) onto the upper left part of the display wall.

Demo Framework

The demo framework is a system for demonstrating systems built for the display wall presented in chapter 2, section 2.3, among others several of the systems built as part of this dissertation. It is designed around the principle of establishing the end-to-end principle through customization. A web server and a demo server are running on the display wall's front-end. A user can visit the web server through a web browser to customize the computer, thereby integrating the computer with the demo server. This integration enables the user to start and stop demos on the display wall (see figure 1.12 for an illustration of the client-side Graphical User Interface (GUI)).

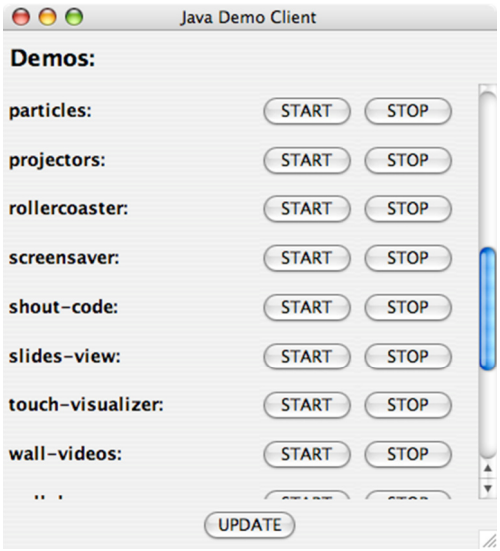

Figure 1.12: The graphical user interface of the demo client from where users can start and stop demos. The demo interface is part of the customization of a user's local computer.

WallScope

WallScope is a system for interactive visualization of local and remote data sets on high-resolution tiled display walls. It comprises a set of components:

- **WallCompute:** WallCompute is the main provider of images in the system. These images range from regular resolution images to high-resolution gigapixel images. In addition, WallCompute can compute maps from vector data, mask satellite images from vector data, and provide elevation data.

- **WallWeather:** A supercomputer executes an on-demand weather forecasting model. Data from this model is processed from a set of other compute resources to produce domain specific weather visualizations used by the display-side's visualization systems.

- **Dynamic Compute Resources:** Laptops and desktop computers can register on-the-fly with the system to become compute nodes for the visualization systems.

- **Live Data Set:** The Live Data Set (LDS) is an implementation of the live data set architecture used as a bridge between NADs and NACs.

- **WallGlobe:** Two virtual Earths, both called WallGlobe, have been created. The first version was implemented in Java using OpenGL for

rendering. Lessons learned from this implementation were used to create a new version in C++ also using OpenGL for rendering (figure 1.13). To support interactive weather forecasting on the display wall, WallGlobe was extended to support weather visualizations that can be overlaid over Earth visualizations. This system utilizes several NACs in WallScope (WallCompute and WallWeather).

- **WallView:** WallView is a visualization system for interactive visualizations of gigapixel images. One of the images is a 13.3 gigapixel image of Tromsø taken from Fløya.

- **LDSView:** LDSView is a visualization system that combines the latest version of WallGlobe with WallView. It queries the live data set at regular intervals to include a visualization of all data that the live data set contains. This data is a combination of all the data the NACs connected to the live data set can produce, which might include additional data from dynamic compute resources that registers on-the-fly with the system.

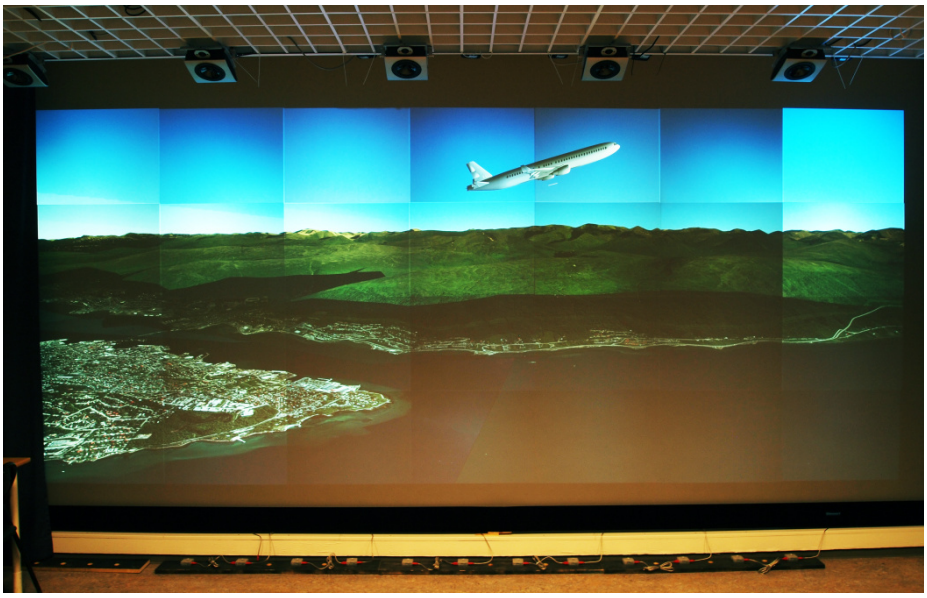

Figure 1.13: WallGlobe showing a plane after a take-off from Langnes airport, Tromsø, Norway.

1.5.4 IMPACT

The work presented in this dissertation has had impact by contributing to state-of-the-art parallel and distributed visualization systems through peer-reviewed

publications. In addition, the systems have been deployed and used on a daily basis at the display wall lab presented in chapter 2, section 2.3, and have been presented on numerous demos and presentations there, among others for high-school classes, Dell, Statoil, the supercomputing-day at the University of Tromsø and the Norwegian branch of the Fulbright Program.

WallScope has been demonstrated to the military at several occasions. WallGlobe has attracted their attention because of its possibilities for combining data from several local and remote locations with visualizations of the Earth.

The NAD system has been in daily use since its first prototype. Users and visitors at the display wall lab that needs to display output from their local computer onto the display wall have been able to do so by simply clicking a link in a web browser. Because of this simple approach, the system has become an important part of the display wall environment. It also enables several proprietary systems to be used on the display wall without requiring any modifications, by simply mirroring their output to the display wall. As of this writing, the system has worked for every user and visitor needing to display output from their local computer onto the display wall.

WallScope including all visualization systems has been presented on numerous demos. WallView was presented in a press conference for enabling interactive visualization of a 13.3 gigapixel image of Tromsø taken from Fløya. This press conference was also covered by NRK [49], a national TV channel, and several newspapers. A recording of this system was uploaded to YouTube[4]. As of this writing the video has a view count of 137 495 views. The system including the YouTube video has been covered on numerous online tech articles among others Engadget [50], SlashGear [51], NRKbeta [52], Hack a Day [53] and Ubergizmo [54]. WallGlobe has been used to demonstrate the display wall and has appeared among others in the Nordlys newspaper [55].

1.6 Summary of Papers

This section includes a summary of the papers forming the basis of this dissertation. The papers are divided into: (i) Background papers, describing some of the motivations and problems leading to some of the principles and models behind network accessible compute- and display-resources; (ii) push-based network accessible compute- and display-resource papers; and (iii) pull-based network accessible compute- and display-resource papers.

[4] http://www.youtube.com/watch?v=8bHWuvzBtJo (thanks to Daniel Stødle for recording, editing and uploading the Gigapix movie to YouTube).

1.6.1 Background Papers for Network Accessible Compute- and Display-Resources

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

This paper presents two different approaches for parallelizing two existing games (Quake 3 Arena and Home World) based on an SLSR model to run on a high-resolution tiled display wall. The purpose of the work is to document and evaluate the performance of the two different parallelization approaches.

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, 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.

Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core

This paper compares the performance of the embarrassingly parallel Mandelbrot set computation on four different systems: (i) A single single-core computer; (ii) a cluster of 28 single-core computers; (iii) a cluster of 28 single-core computers where the main processing is done on older generation multi-core graphics cards on each computer; and (iv) a single single-core computer with a modern multi-core graphics card. The purpose of this paper is a performance evaluation of four different compute systems and the implications of their hardware architectures.

Citation: Tor-Magne Stien Hagen, Oleg Jakobsen, Phuong Hoai Ha and Otto Anshus. Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core. In Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA, 2008.

Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors

This paper presents an implementation of a lock-free synchronization mechanism for graphics cards of CUDA compute capability 1.0 (global and shared memory) and 1.1 (shared memory, since global memory has hardware support for atomic operations) that eliminates lock-related problems like the deadlock and, in addition, can tolerate process crash-failure. This paper addresses the experimental

issues that arise in the implementation of the mechanism and evaluates its performance on commodity GeForce 8800 graphics cards.

Citation: Tor-Magne Stien Hagen, Phuong Hoai Ha and Otto Anshus. Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors. In Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA, 2008.

1.6.2 Push-Based Network Accessible Compute- and Display-Resource Papers

Liberating the Desktop

This paper presents the network accessible display model based on the principle of establishing the end-to-end principle through customization. The purpose of this paper is to demonstrate how a system built on this model and principle enables a simple and flexible way to utilize nearby display resources without requiring permanent installation of new software or opening firewall ports on the NAC.

Citation: Tor-Magne Stien Hagen, Espen Skjelnes Johnsen, Daniel Stødle, John Markus Bjørndalen, and Otto Anshus. Liberating the Desktop. In Proceedings of the First International Conference on Advances in Computer Human Interaction, ACHI, pages 89-94, February 2008.

1.6.3 Pull-Based Network Accessible Compute- and Display-Resource Papers

On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls

This paper presents the live data set architecture, and demonstrates how a system built around this architecture enables seamless communication between NADs and NACs. It also demonstrates how a visualization system can get customized data from local and remote NACs on-demand, enabling visualization systems to get data based on the latest version of available remote data sets.

Citation: Tor-Magne Stien Hagen, Daniel Stødle, and Otto Anshus. On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls. In Proceedings of the International Conference on Information Visualization Theory and Applications, pp. 112–119, May 2010.

Interactive Weather Simulation and Visualization on a display Wall with Many-Core Compute Nodes

This paper presents network accessible compute resources used to produce weather forecasts, and demonstrates how these compute resources can be used with an existing visualization system to provide interactive visualization of user-selected weather forecast regions.

Citation: Bård Fjukstad, Tor-Magne Stien Hagen, Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen and Otto Anshus. Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes. To appear in the Proceedings of the 10th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA, 2011.

A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls

This paper applies the Network Accessible Compute (NAC) model to personal compute resources to utilize desktop applications in a display wall context. The paper defines the two different NAC types, static and dynamic, and demonstrates how a set of dynamic NACs can be used to produce customized data for a visualization system running on a display wall.

Citation: Tor-Magne Stien Hagen, Daniel Stødle, John Markus Bjørndalen and Otto Anshus. A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls. To appear in the Proceedings of the 11th IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS, 2011.

1.7 Organization

The remainder of this dissertation is organized as follows. Chapter 2 presents display walls and details hardware platforms and software systems commonly found in these environments. Chapter 3 gives an introduction to Graphics Processing Units (GPUs) and continues with a description of the Compute Unified Device Architecture (CUDA), NVIDIA's technology for General Purpose computations on Graphics Processing Units (GPGPU). Chapter 4 details the methodology used for the research presented in this dissertation. Chapter 5 presents Network Accessible Resources, in particular, Network Accessible Compute (NAC) resources and Network Accessible Display (NAD) resources, and continues with a description on some of the background leading to some of the principles and models behind NACs and NADs. Chapter 6 details the push-based network accessible compute- and display-resources built as part of this dissertation. Chapter 7 presents WallScope, the system realizing pull-based network accessible compute- and display-resources. Chapter 8 discusses the

research presented in this dissertation, and chapter 9 draws conclusions. Chapter 10 outlines some areas of future research.

Appendix A includes all the papers that this dissertation is based on. Appendix B includes additional work done with the WallScope system. Appendix C gives an overview of the content of the accompanying CD-ROM.

Chapter 2

Display Walls

A display wall is a scalable high-resolution tiled display (figure 2.1). High-resolution implies a resolution that is higher than the resolution found on normal desktop displays, which is typically between one to four megapixels. It is not unusual that the resolution is orders of magnitude higher [56]. Tiled means that the display wall is comprised of multiple displays arranged in a grid. Scalable implies that more resolution can be provided by adding more displays to the grid. In the literature, synonyms for display walls are among others display arrays, tiled displays, display grids and powerwalls.

Figure 2.1: Illustration of the display wall lab at the Department of Computer Science, University of Tromsø.

The key characteristics of a display wall are: (i) Its size enable several people to use it simultaneously; (ii) its resolution allows large amounts of information to be presented, with high fidelity; and (iii) the combination of size and resolution

enable users to get overviews, at the same time being able to walk up close to look at details.

In the following sections, hardware- and software-systems for display walls are presented. Then a description of the display wall used as a basis for much of the research presented in this dissertation is provided.

2.1 Display Wall Hardware

Display walls are typically built by commodity PC components [57] [58]. In the following sections, a listing of the technologies comprising display walls is provided, with a description of the pros and cons of each technology.

2.1.1 Display Technology

There are two common display technologies used to build display walls: (i) LCD displays; and (ii) projectors. There are tradeoffs between the two technologies, listed below.

LCD Displays

Pros:

- Easier to align.
- Better color and brightness correspondence.
- Cheaper per pixel.
- Better DPI/PPI.
- Normally do not need external cooling.
- No separate canvas required.

Cons:

- Borders/mullions between displays. (LCD displays have a border around the visible part of the screen, and thus when tiled, breaks the illusion of a unified display surface). Borders can make text harder to read.

Projectors

Pros:

- No borders/mullions.

- Can be moved backward/forward to increase/decrease the size of the projected area.

Cons:

- Take up space behind the display canvas (projectors are normally rear-projecting on a display canvas to avoid shadows being casted on the display when the user covers that region).
- Needs cooling. This adds maintenance cost to the system.
- Lamps only last for a certain period before they must be replaced, adding additional cost to the system.
- The colors and brightness of the projectors are normally not well matched compared to LCD displays. This problem becomes worse when lamps are replaced, because lamps degrade over time. This creates strong differences in colors when replacing some of the bulbs.

The display technology used for a display wall depends on the requirements for the infrastructure and users' preferences. If borders do not pose a problem for users, LCD displays would probably be the preferred solution. If not, projectors would be a good alternative.

2.1.2 Computer System Technology

A display wall comprises multiple displays, and thus needs a video signal for each display. Contemporary graphics cards are usually dual-headed, which means they can drive up to two displays. However, lately several graphics cards have been introduced with the possibility to drive more than a couple of displays. For example, AMD Eyefinity cards [59] can drive up to 6 displays. Depending on how many displays the display wall comprises the two most widespread configurations for driving display walls are: (i) One computer, which contains one to a handful of graphics card that together drives all the displays; and (ii) multiple computers, where each computer has a single graphics card that drives up to a couple of displays.

Single Computer Systems

Single computer systems for driving display walls combine the fact that certain graphics cards can drive several displays, with the possibility of adding multiple graphics cards within a single system. The result is a single computer having multiple graphics cards connected to a single motherboard. This configuration simplifies application development because the operating systems in many cases can present all graphics cards as one unified resource to the applications running on the system. Additionally, this solution is cheaper since a graphics card is cheaper than a complete computer system.

Although single computer systems maintain the application programming models for developers, there are some drawbacks to these solutions. These systems will only have a single motherboard with one/few (multi-core) CPU(s), and thus the same CPU and I/O processing power as a single desktop computer. Further, the number of displays that can be driven by the system is hard limited to the number of graphics cards that can be plugged into the system (the number of AGP/PCI express slots) multiplied with the number of displays each graphics card can drive. For small to medium display walls, this might be enough, but for ultra-high resolution environments [56] it is not. Single computer systems, strictly speaking, therefore break with the scalable definition of a display wall, since the number of displays that can be driven by a single computer is fixed.

Distributed and Parallel Systems

When a single computer is not an option for driving a display wall, multiple inter-connected computers can be used. Each computer typically drives one to a few displays. Synchronization of the output is achieved using inter-communication. The total computing and I/O processing power of these systems are higher than that of a single computer. However, these systems have some disadvantages. Firstly, this solution requires more hardware and is therefore more expensive. Secondly, the distributed architecture of the display wall introduces one shared interconnect, the network. A common networking technology today is 1 gigabit switched Ethernet. Compared to the interconnects on the north-bridge of a modern computer's mainboard this is an order of magnitude less bandwidth. Finally, most applications today are designed using an SLSR approach and do therefore not natively support the distributed and parallel architecture of a display wall. Depending on their design and source code availability, much work might be required to get them running on the display wall.

2.2 Display Wall Software

This section applies mainly to distributed and parallel systems. For single computer systems, the software approaches are in many cases the same as for single desktop computers, because the graphics cards are presented to the applications as one unified resource.

A challenge for distributed and parallel systems is application compatibility with existing software. For open source applications, the process of changing them to an MLMR approach can be time consuming. For proprietary applications, an SLMR approach can be used. This requires modifications to the underlying libraries. However, when this modification has been carried out, several other applications using the same libraries can be run unmodified on the display wall. Depending on the application type, this configuration can quickly become bottlenecked by the network.

The following sections provide a description of common software that can be used to drive a display wall.

2.2.1 Virtual Network Computing (VNC)

Virtual Network Computing (VNC) [37] is a remote desktop system based on the RFB protocol [60]. The server is the computer hosting the main frame buffer and applications, and the client is the computer showing the remote frame buffer (figure 2.2). The protocol is pull-based, meaning that the client asks the VNC server for updates and provides the server with keyboard and mouse events. These events are inserted into the event queue of the server's window system.

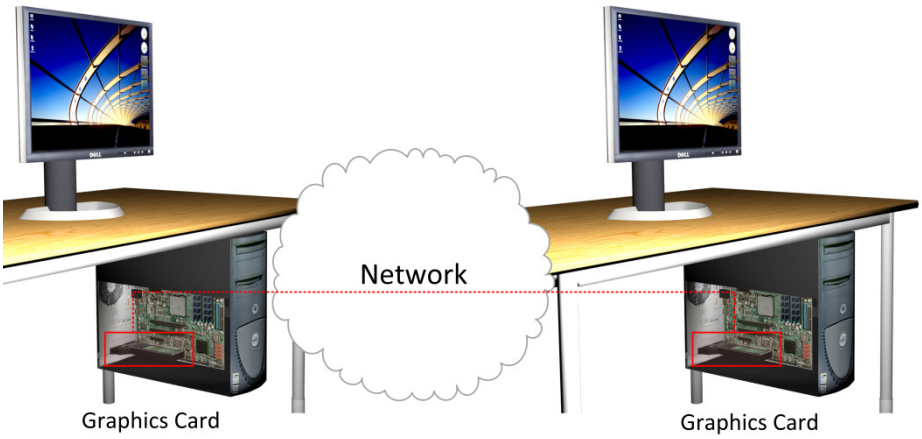

Figure 2.2: VNC's traditional client-server model.

The VNC server keeps track of all the updates performed to the frame buffer since the last client request, encodes these updates using a client-server agreed encoding protocol, and sends them to the client in response to a request for update. The VNC protocol is extendable and therefore supports a wide range of encoding protocols. There are many implementations of VNC, among others RealVNC [61], TightVNC [62], UltraVNC [63] and XVNC [64].

Although the VNC protocol originally was designed for a single client and server, it can be extended to support a display wall. XVNC is a VNC implementation where the VNC server creates an X server with a virtual frame buffer, instead of sharing the frame buffer of an existing X server. This virtual frame buffer is hosted in main memory. This has two implications: (i) The frame buffer does not have the same restriction on the resolution as a frame buffer hosted in GPU memory (this restriction is a hard limit of the texture size of the GPU); and (ii) rendering to the frame buffer is not hardware (GPU) accelerated, and thus the CPU must do all rendering in software. By modifying an existing VNC client and

an XVNC server, multiple clients can be configured to request part of a larger frame buffer. The XVNC server creates a virtual frame buffer equal to the total resolution of the display wall. A client is provided with information about its location in the display wall grid, and based on this information, requests the pixels affecting its part of the frame buffer from the XVNC server (figure 2.3). In addition to performance penalties from software rendering and encoding of pixels, the network between the clients and the server often becomes a bottleneck using this configuration [32].

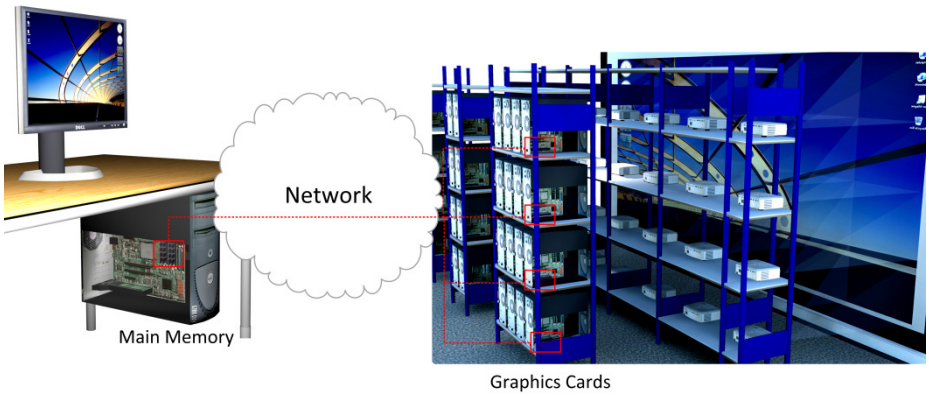

Figure 2.3: VNC in a display wall context.

2.2.2 Distributed Multihead X (DMX)

Distributed Multihead X (DMX) [34] is a front-end X proxy server. When used with Xinerama [34], the front-end X proxy server presents a set of back-end X servers as one unified screen to its clients. The back-end X servers can be run on separate physical computers. By using DMX on a display wall existing X clients can utilize the display wall's full resolution, since the display wall is presented as one large display instead of multiple separate tiles. In contrast to the XVNC based system presented in the previous section, DMX uses both commands and pixels[5] (pixels are typically image surfaces generated using XPutImage) to generate the final desktop image. When viewing high-resolution images, this might result in large amounts of raw pixel data to be sent to all back-end X servers. This might saturate the network at the initial loading of the image. Informal experiments with DMX indicated that this would be a performance problem with this system.

[5] The X11 protocol description is available at [19]

2.2.3 Chromium

Chromium [33] (based on WireGL [65]) is an interactive rendering system designed for clusters. Chromium uses streaming processing units (SPUs) to serialize and optionally modify OpenGL stream commands. An SPU takes a stream as input and produces zero or more output streams. The communication between SPUs located on different computers is handled by a point-to-point connection-based networking abstraction. Chromium can be used to redirect application output from OpenGL based applications to display wall tiles. The OpenGL library used by the application is replaced by Chromium's OpenGL library. This library intercepts all OpenGL commands and forwards them to a local SPU. A sort-first tile SPU can be used to sort and forward the OpenGL commands to a set of Chromium servers running on the display wall's tiles. The SPU on each tile forwards the commands to the local graphics card. Several other configurations can be used including a sort-last and a hybrid sort-first sort-last approach. The drawback to using Chromium to drive a display wall by intercepting graphics commands from a central computer is the fact that the PCI-express bus is effectively replaced by a network connection in addition to that the CPU must sort graphics primitives for the display-side, which in some cases will limit its scalability [27] [36]. This depends heavily on the configuration used and the intrinsic of the applications.

2.2.4 Scalable Adaptive Graphics Environment (SAGE)

SAGE [66] is a pixel-based remote rendering system for display walls. In SAGE, rendering is separated from displaying using a dedicated rendering cluster to deliver pixels to a set of display nodes. SAGE incorporates a stream windowing manager allowing streams of pixels to be overlapped, moved and resized on the display wall. A SAGE Application Interface Library (SAIL) is used on the rendering-side to capture the output of the application rendering and stream the output to set of SAGE receivers (one receiver per display node). A FreeSpace manager communicates with SAIL, SAGE receivers and UI clients to control the final composition of the streams on the display wall. The drawbacks to SAGE are: (i) The pixel based protocol which can require high bandwidth links between render clusters and display nodes (SAGE was originally designed for OptiPuter, an infrastructure comprising rendering clusters and display walls connected over optical deterministic high-speed networks); and (ii) the lack of display-side utilization since the display-side only displays pre-rendered images, and therefore does not fully utilize the graphics cards rendering capabilities nor the CPUs computing capabilities.

2.3 The Display Wall at the University of Tromsø

This section describes the display wall at the University of Tromsø (illustrated in figure 2.1). The display wall was built in 2004, and is a used in almost all of the research presented in this dissertation.

2.3.1 Hardware

The display wall comprises a 7x4 grid of projectors driven by 28 computers. Each of the projectors has a resolution of 1024x768 pixels, giving a total resolution of 7168x3072 pixels.

The computers driving each projector has an Intel Pentium 4 3.2 GHz (HyperThreading) CPU, 2 gigabytes of RAM and an NVIDIA Quadro FX3400 Graphics Card with 256 megabytes of Video RAM. All computers are connected using switched gigabit Ethernet. The projectors and computers are located in a separate room. Projectors are rear projecting on a 6x3 meter canvas. This canvas separates the room holding the projectors and computers from the display wall lab.

There are 16 firewire cameras mounted on the floor in front of the display wall canvas. These cameras are part of a touch-free interaction system. Cameras are pair-wise connected to 8 Mac Minis.

Table 2-1 summarizes the display wall's specifications including the touch-free interaction system.

Table 2-1: The hardware specification of the Tromsø display wall

Component	Specification	Quantity	Total
Projectors	1024 x 768	7x4	7168x3072
Display Wall Computers	Dell Precision 370	7x4	28
	Intel 925X Express Chipset		
	Intel Pentium 4, 3.2 GHz CPU w/HyperThreading		
	2 gigabytes RAM		
	NVidia Graphics Card w/256 megabytes VRAM		

Camera System Computers	Mac Mini Intel Core Duo 1.66 GHz CPU 512 megabytes RAM Intel GMA 950 w/64 MB VRAM	8x1	8
Cameras	640x480	16x1	10240x480
Interconnect	Switched gigabit Ethernet		

2.3.2 SOFTWARE

The operating system installed on the display wall nodes is Rocks Linux Cluster Distribution 4.0 32-bit. This distribution is based on Cent OS release 4.2.

The VNC solution presented in section 2.2.1 is used to create a desktop environment on the display wall. A dedicated computer runs an XVNC server with a virtual frame buffer of 7168x3072 pixels. VNC clients on every tile request their part of the frame buffer based on their location in the display wall grid. Users can display output to the display wall from X applications by setting the DISPLAY environment variable to the secondary display of this computer.

Shout, an event system created by Daniel Stødle [45], is used to send events between different providers and listeners running on the display wall. The touch-free interaction device is a provider of touch-events. The microphones and an associated snap click system are providers of sound-location events. The Shout event system has a central server managing the delivery of events between providers and listeners. New providers and listeners can be created using the Shout API (which currently has both C/C++ and Python bindings).

The Network File System (NFS) is used on the display wall cluster to provide unified access to the home folder for every user. The NFS server is hosted by the front-end of the display wall cluster. The NFS file system must be taken into consideration when running experiments that reads/writes files, since file location determines if reads/writes require synchronization over the network to the front-end computer, or can be done locally without intervention from the NFS server.

CHAPTER 3

GRAPHICS PROCESSING UNITS

Graphics Processing Units (GPUs) have emerged as a promising platform for highly-parallel compute-intensive, general-purpose computations. This chapter gives an introduction to GPUs. It continues with a description of the Compute Unified Device Architecture (CUDA), NVIDIA's technology and programming model for General-Purpose computations on Graphics Processing Units (GPGPU).

3.1 INTRODUCTION TO GPUS

A GPU is a processor attached to the graphics card (dedicated) or mainboard (shared). It is used to offload compute intensive tasks such as graphics operations and other more general SIMD operations from the CPU. While a CPU needs transistors for among others control logic and data caching, a GPU devotes more transistors to data processing. This has made the GPU an order of magnitude faster for problems that can be expressed in a data-parallel fashion. GPUs where introduced to perform graphics operations for real-time interactive graphics. Graphics operations are embarrassingly parallel, meaning that graphics primitives can be assigned to processors without any further communication. Early versions of graphics cards performed fixed function operations on the graphics primitives when they flowed through the graphics pipeline. However, as graphics cards evolved, more flexibility was given to the developers by allowing them to control two stages in the graphics pipeline: Vertex- and fragment-processing. Fragment processing is also referred to as pixel processing (for example by Direct3D [67]). The vertex stage is where vertices (positions, normals, colors and texture coordinates) are transformed to world space, colored based on lighting information, and then finally transformed to view space. The fragment stage is where primitives (connected vertices) are rasterized into pixels. Early approaches to GPU computing mapped the general-purpose computation to a problem that could be expressed using the functionality provided in the vertex and/or fragment

stage. For example, by uploading the data for the computation to a texture and then execute the general-purpose program using data from the texture in the fragment shading stage, before reading back the result from a render texture or the frame buffer. While this gave speedup for several types of computations, it had several limitations. The developer needed to learn a graphics API such as Direct3D [67], OpenGL [23] or Cg [68], and had to map the algorithm to vertices- and fragment-programs in one of these APIs. Additionally, GPU memory could not be accessed randomly. A vertex program operates on one vertex at a time, without knowledge about other vertices connecting the primitive. A fragment program operates on one pixel at the time and does not expose information about neighboring pixels. Another problem affecting both graphics processing and GPGPU computations was that depending on the amount of geometry and textures in a scene, the GPU could become either vertex bottlenecked or fragment bottlenecked while the other stage in the pipeline was not fully utilized. This problem was caused by the fact that the GPUs had one set of processors for vertex processing and another set of processors for fragment processing. With the introduction of Direct3D 10.0 this was solved by introducing a new generation of GPUs having unified processors that could do both vertex- and fragment-processing, thereby giving better utilization in both stages. Another feature of the new generation GPUs was the ability to utilize the GPU without expressing the general-purpose computation using a graphics API. NVIDIA called the new architecture of their graphics cards the Compute Unified Device Architecture (CUDA).

3.2 The Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) [69] was introduced by NVIDIA in November 2006 [48]. This architecture uses a new parallel programming model and instruction set architecture compared to previous generation graphics cards. It enables developers to write programs to be executed on a GPU without first mapping them to a graphics API. CUDA improves memory access by giving the programmer full read/write support to the entire device memory with some minor exceptions. From the programmer's perspective, the GPU can be seen as a set of highly parallel multi-core processors. Each processor is capable of running multiple threads in parallel in a SIMD fashion.

The CUDA memory architecture comprises several memory layers (figure 3.1). Each processor has 4 different types of on-chip memory:

- Each core has its own set of on-chip registers.

- Each processor has on-chip shared memory, which is shared by the processor cores. Reads and writes to the shared memory are serialized in case of bank conflicts.
- Each processor has an on-chip read-only constant cache.
- Each processor has an on-chip read-only texture cache.

The processors share device memory, which is divided into global memory, texture memory and constant memory. Reads from texture and constant memory are cached using the on-chip, read-only, constant- and texture-cache. Global device memory is not cached.

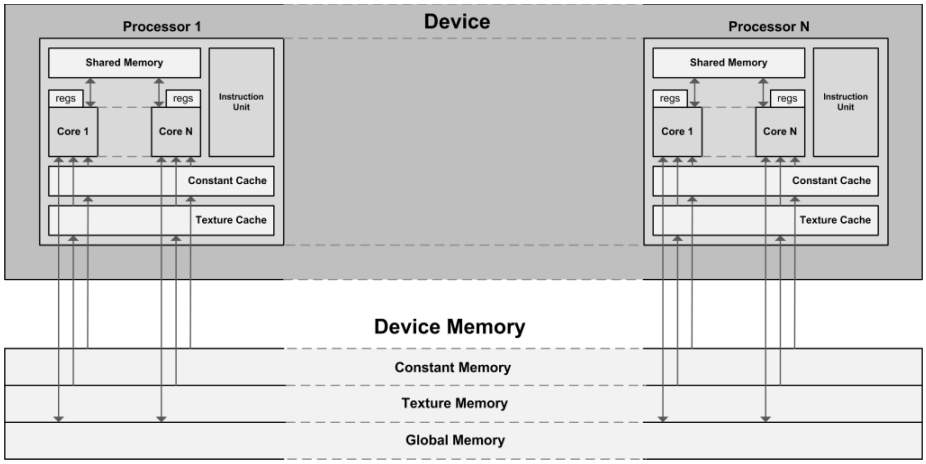

Figure 3.1: The Compute Unified Device Architecture (CUDA).

A CUDA compiled program is referred to as a kernel. The kernel is organized as a grid of thread blocks. These thread blocks are organized into batches and executed on the processors. A block is processed by only one processor to maximize the utilization of the shared memory. A block is split into SIMD groups of threads called warps and each of the threads within the warp is executed on the processor cores in a SIMD fashion. The warps of a running block are time-sliced to maximize the utilization of the processor. The way a block is split into warps is always the same, but the issue order of the different warps is undefined. The time-slicing is hardware scheduled, yielding little overhead for context switches. Threads within the same block can communicate using the processors' on-chip shared memory.

NVIDIA uses the term compute capability to separate the different architectures of their CUDA cards. A card's compute capability is defined by a major revision number and a minor revision number. Devices of the same major revision number have the same core architecture. The minor revision number corresponds to minor

updates to the core architecture. Currently, there exist cards of compute capabilities 1.0, 1.1, 1.2, 1.3, 2.0 and 2.1. Cards of compute capability 1.0 do not have a safe way to communicate through global memory. However, access to global memory has one property that can be used to create synchronization primitives. The device is capable of reading and writing 32-bit, 64-bit and 128-bit words to or from global memory in a memory transaction. This requires the variable type to be a multiple of 4, 8 or 16 bytes, and the read- or write-instructions must be arranged so that the memory accesses can be coalesced into a single contiguous, aligned memory address. This mechanism is used in CUDASync, presented in chapter 5, section 5.4.3.

CUDA's instruction set architecture is called PTX (Parallel Thread Execution). C/C++ code is compiled with NVIDIAs nvcc compiler, which extracts the GPU code from the CPU code and compiles the GPU code into PTX and/or object code. Object code is referred to as cubin objects. The CPU code is left to be compiled by an external tool or by invoking the host complier from nvcc. Code compiled to PTX assembly code is forward compatible with GPUs of future compute capabilities, and can be just-in-time compiled by the driver upon execution. Cubin objects are not portable between GPUs of different architectures.

By splitting a kernel into blocks containing multiple threads, a program will automatically scale to future GPUs with more processors without affecting the underlying program logic. However, this requires that the number of threads created is enough to keep all processors utilized. In addition, multiple optimization strategies exist to increase the utilization of the processors and the high memory bandwidth. Some optimization strategies are (from [48]):

1. **Maximize utilization:**
    a. **Application level:** Maximize parallel execution between the host and the device, and the bus connecting the host to the device (i.e. asynchronous kernel invocations).
    b. **Device level:** Maximize parallel execution between the processors of a device (i.e. use at least as many threads as there are processor cores).
    c. **Processor level:** Maximize parallel execution between the various functional units within a processor (i.e. choose thread block size based on register and shared memory requirements).
2. **Maximize memory throughput**
    a. Minimize data transfer between host and device.
    b. Maximize memory coalescing to global memory.

c. Avoid shared memory bank conflicts.

3. Maximize instruction throughput

a. Sacrifice precision for speed when it does not affect the end-result by using intrinsic function in favor of regular functions.

b. Minimize thread divergence between threads within the same warp.

c. Reduce the number of instructions, for example by optimizing out synchronization points.

CUDAMandelbrot (presented in chapter 5, section 5.4.2) is implemented adhering to the aforementioned optimization strategies to utilize the processing power and high memory bandwidth of the GPU.

CHAPTER 4

METHODOLOGY

The research presented in this dissertation follows a systems approach. This approach involves creating and evaluating computer systems. The first stage in this approach is an idea, which describes the overall goal of the research intended. To evaluate the idea, a system is created to gain insight into its implications on real hardware. The first stage towards a realization of the system is an architecture describing the interaction between the main components of the system.

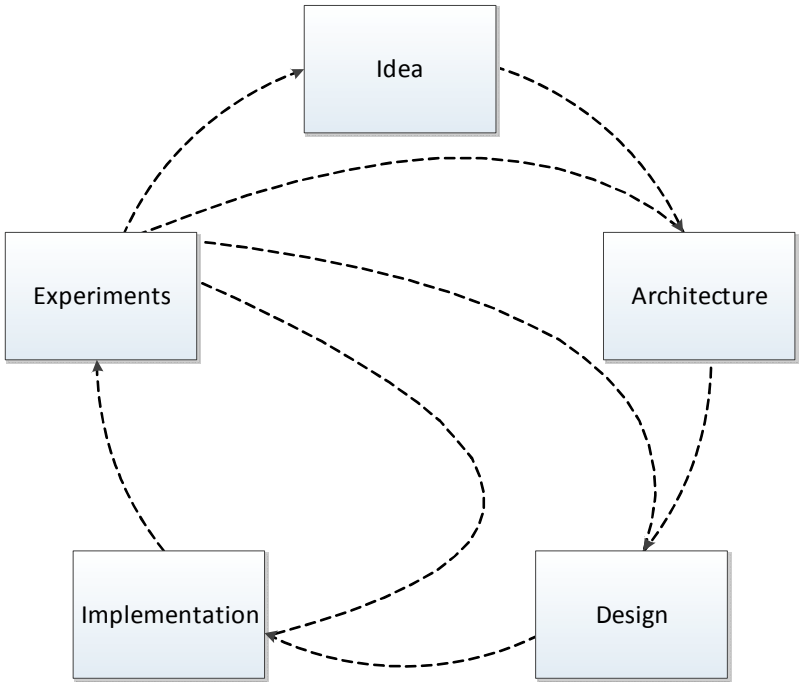

Figure 4.1: Systems research methodology.

Based on the architecture, a specific design is chosen for the system. The design describes the realization of each of the components of the architecture. A specific implementation is developed following the design. The implementation is the actual realization of the system on hardware.

To get performance measurements in a controlled manner, experiments are designed and conducted to evaluate and demonstrate the implications of the overall idea. The results of the experiments are analyzed and conclusions are drawn. Based on the conclusions, the idea, architecture, design or implementation is refined and the cycle continues (figure 4.1).

The mapping between idea and architecture is one-to-many, meaning that several architectures can realize the idea. This also applies to the design and implementation: Several designs can be chosen based on one architecture, and several implementations can be made based on one specific design (figure 4.2).

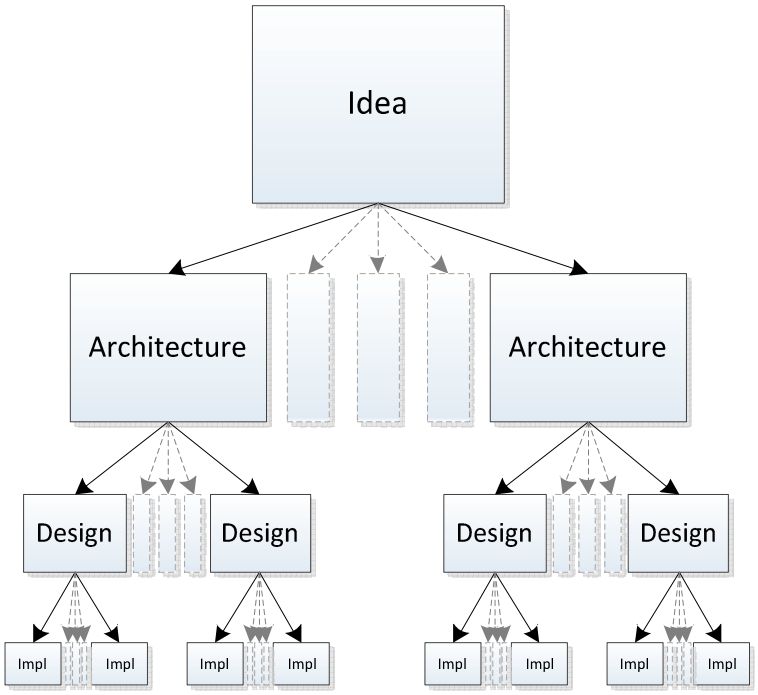

Figure 4.2: The relation between idea, architecture, design and implementation.

The implication of this one-to-many relationship is that even though an experiment does not give the expected results, it might be a result of the path chosen towards the actual implementation of the system, which might require revising the architecture, design or implementation. Nevertheless, the idea, results and the path chosen is often worthwhile to report on in a scientific paper, because

it demonstrates one specific approach to a specific problem and the implications of the choices made.

The reason that an actual implementation is carried out on real hardware is that the interplay between software and hardware is so complex that it is difficult to predict an outcome of a certain idea without creating a system that realizes it. One concrete example on this is the CUDASync research carried out in chapter 5, section 5.4.3. The CUDASync results document that synchronization primitives can be created in global and shared memory of CUDA compute capability 1.0 cards, without the need for synchronization primitive support in hardware, by using the coalesced memory access intrinsic. Although the algorithm for the synchronization had been theoretically proven, and the actual implementation carried out further demonstrated the correctness of the algorithm, a limitation was discovered. The small size of on-chip memory reduced the amount of simultaneous threads that could synchronize through shared memory because the data structures needed for the algorithm exceeded the amount of on-chip memory. This was not discovered before the experiments were conducted, and demonstrates the need to carry out implementations to document the actual performance characteristics and limitations on real hardware.

To demonstrate the advances of a new idea the goal is to compare the proposed system with a similar system in the literature. However, in many cases this can be hard if not impossible. Firstly, there might not exist a comparable system. Secondly, several systems are not open-source and are therefore hard to evaluate. Thirdly, for those systems that do have a documented performance evaluation, the hardware configuration is often not the same, making it hard to compare the systems based on just the performance measurements reported on. Finally, even though a "similar" system is available and open-source, the system does not necessarily have the desired functionality, making it necessary to modify the system to be able to compare them. The modifications made to the system might not reflect its original design. For example distributing a centralized graphics application to compare it with a proposed distributed visualization system might yield a system that is un-optimized because of its centralized design, such as the lack of view-dependent data selection and strict binding between the different stages in the visualization pipeline.

4.1 METRICS

In this dissertation, the following metrics are used for performance measurements: (i) CPU-load; (ii) memory usage; (iii) network bandwidth usage; (iv) frame rate; and (v) latency.

4.1.1 CPU Load

CPU load is the metric describing the time-period a process has been running on the CPU divided by the total time elapsed within the measurement period multiplied by 100. The metric for CPU load is percent. For example, if a process has been allocated the CPU for 500 milliseconds within a period of 1 second, the CPU utilization is 50 percent. Processors that have more than a single core can have a CPU utilization of 100 percent multiplied by the number of cores (this also includes processors with HyperThreading, where the CPU load can reach 200 percent for a single-core CPU with HyperThreading enabled). The CPU load can be divided into user-mode load and kernel-mode load. User mode is the time spent in user space performing non-privileged instructions, while kernel-mode (sometimes referred to as system-mode) is the time spent inside the kernel on behalf of the process. There are several ways to monitor the CPU load of a process. The top utility [70] can be used to get a quick overview of CPU utilization. On Linux top uses the proc file system for querying this information. /proc/pid/ contains numerous files describing the statistics of a process. /proc/pid/stat contains information about the CPU usage. proc is a pseudo filesystem, meaning it is an interface to kernel data structures and updated each time a file is read from. By parsing the /proc/pid/stat file, information about the CPU utilization can be queried. /proc/pid/stat reports CPU utilization in jiffies, the time elapsed between system timer interrupts.

In this dissertation, the proc file system is used to monitor processes externally. For internal instrumentation the getrusage() system call is used in combination with gettimeofday(). The accuracy of the measurements is milliseconds. The overhead of the instrumentation has been informally measured to be negligible.

4.1.2 Memory Usage

Memory usage is the total amount of memory a process has allocated (stack and heap). This information can be obtained by using the top utility or by querying the proc file system directly. /prov/pid/statm contains information about the process memory status in pages. The combination of /proc/pid/stat and /proc/pid/statm is used in this dissertation to externally monitor a process' memory usage.

4.1.3 Network Bandwidth Usage

Network bandwidth usage is the number of sent and received bytes by a process measured within a given time period. It is reported in (mega/kilo) bits per second. This information does not include the headers of the underlying protocols used (such as TCP/IP). The work in this dissertation instruments network bandwidth usage internally in a process by keeping track of the sent and received bytes over the sockets used. This information could also be obtained by using an external

processor monitor combined with a low-level packet interception library such as LIBPCAP [71]. Although, for the experiments conducted in this dissertation internal instrumentation is the preferred way.

4.1.4 Frame Rate

Frame rate is a metric describing the update rate of the displaying for a graphics application. It is measured in Frames Per Second (FPS). A frame is the final image generated by the rendering stage (figure 1.5, chapter 1, section 1.1) of the visualization pipeline. Ideally, the frame rate should match the refresh rate of the attached display(s). However, depending on application intrinsic, lower frame rates can be acceptable.

4.1.5 Latency

Latency is the metric used to describe the delay between two events in a system. For network communication, the latency is often used to describe the delay between a sender and a receiver, which is the time to send an empty message between the two. For interactive visualization, the latency is often used to describe the time taken from the user initiates an event in the system, usually by navigating in the visualization, until the data needed for the visualization has been rendered to the frame buffer. In this dissertation, the events used when measuring latency are described when this metric is used.

4.2 Cluster Wide Experiments

The experiments conducted as part this dissertation involves up to several clusters used simultaneously. On a single computer, a user will often have exclusive access and therefore have complete control over the hardware and software used. On a cluster, several users might be using the resources simultaneously. If compute- or I/O-intensive tasks from other users are running during an experiment, it can, and often will affect the measured results, possibly leading to the wrong conclusions. To guard against this, most of the experiments are run during nights and on weekends. Additionally, experiments are conducted at least two times, including a thorough investigation of the software running on the cluster before the experiments are started. In addition, the hardware is checked, for example to make sure that HyperThreading is enabled for all CPUs.

A process monitor has been developed to measure the performance of a range of processes running on a single computer. The monitor will at regular intervals record among others total- and per-process-CPU load (system and user) and memory usage (stack and heap). This information is recorded along with the process id of each process currently being monitored.

All cluster-wide experiments are designed to start the processes involved in the experiment before the process monitor is started. The processes are designed to listen on a socket for an experiment start-signal. This start-signal is sent to all processes from a central host once the experiment is started, at which point a timestamp is logged on each process using the gettimeofday function. The information recorded by the process monitor is redirected to files and collected post-mortem for analysis.

By comparing the total load on the measured platform with the load generated by each of the processes that was measured, the performance impact of software running concurrently with the experiments can be found. This information is used along with the per-process information when analyzing the results.

CHAPTER 5

NETWORK ACCESSIBLE RESOURCES

This chapter describes network accessible resources, in particular the two network accessible resources developed as part of the work presented in this dissertation: (i) Network Accessible Display (NAD) resources; and (ii) Network Accessible Compute (NAC) resources. This chapter provides an overview of network accessible display- and compute-resources, and the interaction between them. The subsequent chapters describe them in depth. This chapter is based on the following peer-reviewed published papers: [36] [72] [73] [74] [75] [76] [77] [78], in particular papers [36] [72] [74] [75].

A network accessible resource is an abstraction of a specific hardware resource, available over a network connection. Examples of existing network accessible resources are printers and Network Attached Storage (NAS). Network accessible resources have different properties such as one- or two-way communication (reading or writing from/to the resource) and support for single or multiple simultaneous users.

5.1 NETWORK ACCESSIBLE DISPLAY RESOURCES

A Network Accessible Display (NAD) is a display having functionality that enables usage over a network connection. There are two types of approaches for using a network accessible display:

1. **Push-based:** Use a display over a network connection from one or several network accessible compute resources. One example where such usage can be beneficial is meeting room environments where users can share a network accessible projector.

2. **Pull-based:** Connect the display to one or several network accessible compute resources to utilize their collective computing power. An

example of a pull-based usage is remote desktop systems such as VNC [37] and Windows Remote Desktop [79].

A typical PC display does not have the necessary functionality for using it as a network accessible display. However, tethering a normal display from a computer enables network accessibility, where the combination of the computer, the display and the software running on the computer makes up the network accessible display.

For a push-based approach, the display is a server providing one or several displaying services. A push-based approach might be preferred for settings where the user is sitting at a local computer and wants to display remote content on another display.

For a pull-based approach, the display is a client requesting content from one or several network accessible compute resources. Input to the display can be achieved using for example touch interfaces built into the display. If input devices built into a display are not present, more traditional input devices can be used, such as a keyboard and a mouse.

5.2 Network Accessible Compute Resources

A Network Accessible Compute (NAC) resource is a computational resource producing content for a network accessible display. As for network accessible displays, the same two approaches apply for a NAC:

1. **Push-based:** Produce and push content to network accessible displays.
2. **Pull-based:** Produce content based on requests from network accessible displays.

Network accessible compute resources are categorized into static and dynamic compute resources. A static NAC is a compute resource that is long-lived in a system once added. A dynamic NAC is a compute resource that is volatile in the sense that it can register on-the-fly with a system to become a compute resource, and then, at a later point, leave.

5.3 NAD - NAC Interaction

Push-based network accessible compute resources produce data for displays based on a push-based approach. The same applies for pull-based network accessible compute- and display-resources.

The most basic form of service a network accessible display can provide is the ability to receive and display pixels. In this case, the display is a remote frame buffer accessible over a network connection. By sending (writing) pixels to the remote frame buffer, applications running on another computer can show content on the display. This approach is preferable for cross-platform compatibility since pixels are the basic primitives of current display technologies. Display sharing can also happen at the display driver interface, by presenting the display as a display driver to an external computer. Commands sent to this display driver are sent over the network to the external display, thus enabling better utilization of the display resource, since the graphics card on the display-side can render primitives and not only update its local frame buffer with remotely rendered pixels. At the next level in the graphics stack is the X11 network approach, where graphical primitives are sent over a network graphics protocol. This approach is well suited when modifications to the operating system is not an option. Instead, applications use a library providing the necessary functionality for forwarding graphics commands to the remote display.

A common approach for the interaction between NACs and NADs is to use a pre-agreed protocol between them [37] [79] [31]. Thus, a NAC and a NAD implementing the same protocol can communicate.

One example of the problems with pre-agreed protocols is remote desktop systems. Both sides need compatible remote desktop systems to be able to communicate. A VNC client can talk to a VNC server, but not necessarily to a RDP server. Thus, a pre-agreed protocol approach complicates usage of network accessible resources if both parties do not have pre-installed compatible systems. In addition, updating the protocol requires support for backward compatibility or alternatively an updated protocol for both parties.

The work presented in this dissertation employs an on-demand customization mechanism activated at the time of usage of the NAD. Instead of pre-installing compatible software on both sides, the display-side customizes the compute-side according to the display-side preferences.

For a push-based approach, the NAC initiates contact with the NAD. The NAD responds with code that physically customizes the NAC, thereby enabling the NAC to produce content for the NAD.

For a pull-based approach, the customization phase is either physical or virtual. For dynamic NACs, the customization phase is physical, as for the push-based approach. For static NACs the customization is virtual, meaning there is a third party involved in the customization phase. This third party handles the translation between NAD custom requests to NAC-side behavior.

5.4 Background for Network Accessible Compute- and Display-Resources

This section presents motivations and problems leading to some of the principles and models behind network accessible compute- and display-resources.

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

Games are demanding visualizations because of their requirements for ever-increasing realism, interactivity and high framerates [80]. This makes them interesting for gaining insight into characteristics and performance of interactive visualizations on high-resolution display walls. Several games are designed to run in a single computer environment, Single Logic Single Rendering / Multiple Logic Single Rendering (SLSR/MLSR)[6] [36]. Moving from a single computer environment to a distributed parallel environment is challenging because internal interconnects, such as the PCI express bus, are replaced with network connections with less bandwidth and higher latencies. Consequently, maintaining high frame rates becomes increasingly difficult when increasing resolution by introducing several displays driven by individual computers.

To investigate interactive visualization on high-resolution tiled display walls, two existing games, Quake 3 Arena (Q3A) [81] and Homeworld [82], were ported to run on a display wall. In addition, the games were modified to accept input from a touch-free multi-user interaction system.

Quake 3 Arena is a First-Person Shooter (FPS) multiplayer game developed by id Software [83]. Homeworld is a popular 3D Real-Time Strategy (RTS) game developed by Relic Entertainment [84]. Both these games were proprietary when released. However, later on, the game-engines were open-sourced (although the game-engines are open-source, the data files needed for the gameplay requires a license).

In the following sections the architecture, design, implementation and evaluation of the parallelization and input modifications to these games are presented.

Related Work

The open sourcing of the Quake-series has made them popular targets for modifications and extensions. Some examples include using the Nintendo's Wiimote [85], using eye tracking to play Quake, or controlling quake from a

[6] Described in section 1.2.

PDA[7]. Quake II and Q3A have also been modified to be used in a CAVE[8]. However, these versions do not support all the features of the full games, and for the Q3A case does not even support playing.

It is not only the Quake series that have been ported to large displays. In CaveUT [86], Unreal Tournament is modified to run in panoramic theaters. The system uses the spectator functionality of the game to run it on several computers. A spectator is a virtual camera that follows the same view as a player of the game. However, no measurements are provided of the resulting performance.

There are multiple other systems that can be used for parallel rendering on display walls. These are presented in chapter 7, section 7.1.1.

ARCHITECTURE

Q3A has 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 state based on information from the clients, and broadcasts this state to them. A Q3A client is either a player or a spectator. A player is an entity participating in a game, while a spectator follows the view of a selected player. Homeworld has a monolithic architecture, and therefore does not have a clear division of responsibility between a client-side and a server-side.

The parallelization of the games follows two different approaches. Q3A already has a client-server architecture. By exploiting this architecture and Q3A's spectator functionality, the parallelization is carried out by assigning several spectators to follow a single player (client-server approach), shown in figure 5.1. The players send updates to the server, which broadcasts this state to all connected clients.

Homeworld is parallelized using another approach, where several instances are run in parallel and the state for these instances are synchronized (master-slave approach), shown in figure 5.2. A master process is responsible for sending information to the slaves to keep all instances synchronized. In contrast to the Quake 3 approach, the parallelization of Homworld requires all clients to receive the state from the master for each iteration of the main-loop. This is because the state received from the master contains information for updating the game logic at a specific frame. The slaves use this information in combination with the state from the previous frame to calculate the new frame.

---

[7] http://www.youtube.com/watch?v=n1tsXc2RoeM
http://www.youtube.com/watch?v=3pRWYE2LRhk
http://www.youtube.com/watch?v=tNJXjNBgmLs

[8] http://www.visbox.com/cq3a/

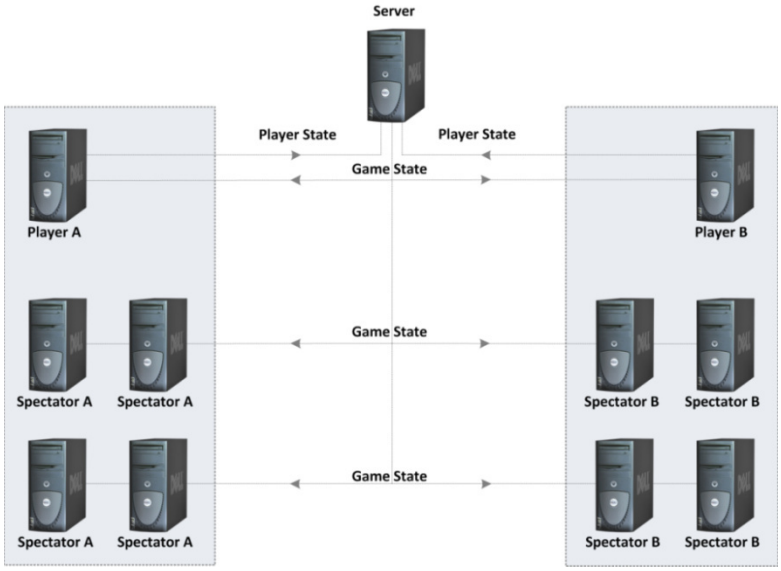

Figure 5.1: Architecture of the parallel Quake 3 Arena. Players send their state to a server which maintain the entire game state. The server sends updated state to all connected clients.

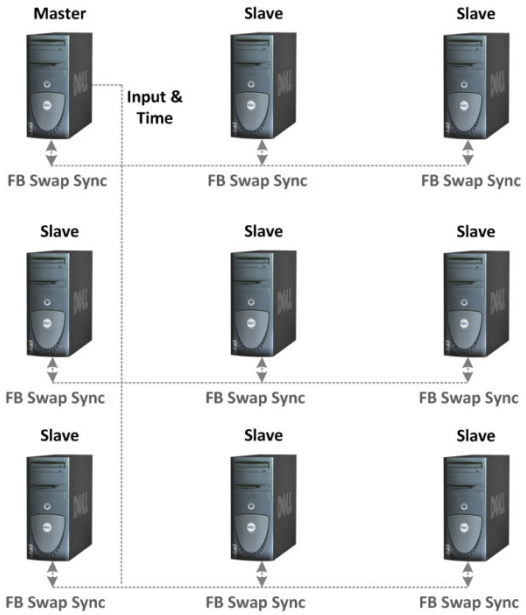

Figure 5.2: Parallel Homeworld architecture. The master gets input, which it broadcasts to every slave at the start of each iteration of the game's main-loop. Before the frame buffer is swapped (at the end of the main-loop) all slaves synchronize through a barrier.

Design

Q3A is modified to use a single player for every user, where each player is connected to the state server. A player is assigned a set of spectators, which receive the same updates as the player, but are modified to show their view of the scene based on the position in the display wall grid. A user interacts with the spectator part of Q3A, and the input from the user is directed to the player part.

Homeworld is modified to have one process of the game running on each display wall tile. Every process shows the scene from a view calculated from their position in the display wall grid. The game logic for each client is synchronized. The synchronization is done by using a global clock mechanism in addition to providing all random generators with the same seed. One of the processes is selected as master. The master is responsible for handling input and propagating this input to all slaves in addition to the global time.

The input to the games is provided by the multi-user touch-free interaction system. The input from the interaction system is sent to the player (Q3A) or master (Homeworld), which in turn detects gestures from the input, and translates them into mouse and keyboard events. For Homeworld, the master broadcasts the corresponding events to the slaves and waits at a barrier at the end of the game loop. The broadcast is implicit in Q3A since clients periodically sends and receives state from the state server.

Implementation

The modifications to the Q3A game affects two parts of the code: The input handling in the player code and the view frustum settings used by the spectators.

Q3A is modified to use a set of environment variables, which are read from within the game. The variables control how the spectator view frustum used by OpenGL 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, these modifications are sufficient for the parallel version. The player receives object positions and gestures from the touch-free input system and interprets them (appendix A, section A.1 gives an explanation of the gestures used). When a gesture is detected, a corresponding action associated with the event is inserted into the game's input stream.

Homeworld is parallelized by running several tightly coupled processes in a lockstep fashion. Each process runs on one tile, and the Message Passing Interface (MPI) [87] is used to exchange state information to keep processes synchronized. One process is elected as master, and the remaining ones become slaves. For each frame, the master accepts input from the touch-free system and broadcasts it to the slaves. Before starting a new frame, all processes synchronize at a barrier. This ensures that each slave receives the same input during the same simulation step in the game, and synchronizes the frame buffer swap between all

processes. In addition to running all processes in lockstep, the same value is used to seed each process' pseudo-random number generator to make sure the game logic stays in sync. Finally, all processes share a global clock controlled by the master. Using a similar approach to Q3A, the master receives input from the touch-free system, maps the input to gestures with associated events, broadcasts these to all slaves, and finally inserts them into the game's input stream.

The main drawback to the parallelization approach taken by Homeworld is that it requires great familiarity with the source code in order to guarantee that all the game simulations end up running identically. If a single branch statement makes processes diverge, the simulations might get out of sync, resulting in non-coherent displaying output.

The touch-free interaction device comprises sixteen cameras, eight MAC minis and a MacBook Pro used for object detection. The cameras detect 1D position of objects at 30 frames per second. These positions are sent to the MacBook Pro, which performs object detection and triangulation of the corresponding 2D positions. These 2D positions along with their corresponding radii are sent to the games for gesture detection.

EXPERIMENTS

To evaluate the performance of the games, four experiment series were designed and conducted.

The purpose of the first experiment series was to measure the rendering performance of Q3A when using Chromium to distribute the rendering primitives.

For the second experiment series, the purpose was to measure the rendering performance of Q3A when using the parallel version.

The purpose of the third experiment series was to measure the extra latency introduced as a result of the Q3A player-server-spectator approach used.

For the fourth experiment series, the purpose was to measure the rendering performance of the parallelized Homeworld.

METHODOLOGY

The metric used to measure the performance of the parallelized games was frames per seconds. To assure repeatability over the different configurations used, input events were recorded over a period of 30 seconds. These events were sent to the games, which were started in a known state. During this playback, the frame rate was logged every frame.

Chromium was used to compare the performance with the parallelized version for Q3A. For the parallelized version of Q3A, the frame rate was limited to 500

(using the built in frame rate limitation already supported by the game). Homeworld did not run using Chromium.

For Q3A, one additional latency-inducing step is introduced because of the parallelization approach. This is the time taken from a gesture is recognized in the player part, until all spectators updates their view from updates received from the game state server. To be able to measure the additional latency introduced by the player-server-spectator setup (figure 5.1) the player and spectator code was modified. When the weapon is fired in the player code, a weapon-fire event is generated. The modifications involved starting a timer once this event was detected (by monitoring the event queue). Each spectator reports back to the player once the same event is detected. This yields a rough estimate of the latency between when something happens in the player part, until it is reflected in the spectator part.

A summary of the experiment series, metrics and factors is listed in table 5-1. The hardware used in the experiment is listed in table 5-2.

Table 5-1: Experiment summary

Experiment Series	Metric	Factor
1	Frame rate (Q3A Chromium)	1 (1x1), 4 (2x2), 9 (3x3) and 28 (7x4) display nodes
2	Frame rate (Q3A parallel)	1 (1x1), 4 (2x2), 9 (3x3) and 28 (7x4) display nodes
3	Latency (Q3A parallel)	-
4	Frame rate (Homeworld)	1 (1x1), 4 (2x2), 9 (3x3) and 28 (7x4) display nodes

Table 5-2: Hardware- and software-platform

	Display-Side	Touch-Free Interaction Device		
Type	Display Cluster	Unibrain Fire-I FireWire Cameras	Mac Mini	MacBook Pro
Number of Nodes	28	16	8	1

CPU	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading	-	Intel Core Duo 1.66 GHz	Intel Core 2 Duo 2.33 GHz
RAM	2 GB	-	512 MB	3 GB
Graphics Card	NVidia Quadro FX3400 w/256 MB VRAM	-	Intel GMA 950 w/64 MB	ATI X1600 w/256 MB VRAM
Operating System	Rocks Linux Cluster Distribution 4.0	-	Max OS X 10.4.8	Max OS X 10.4.8
Interconnect	Switched gigabit Ethernet			

Results and Discussion

Figure 5.3 shows the results of experiment series 1 and 2 (Q3A using Chromium and parallelized).

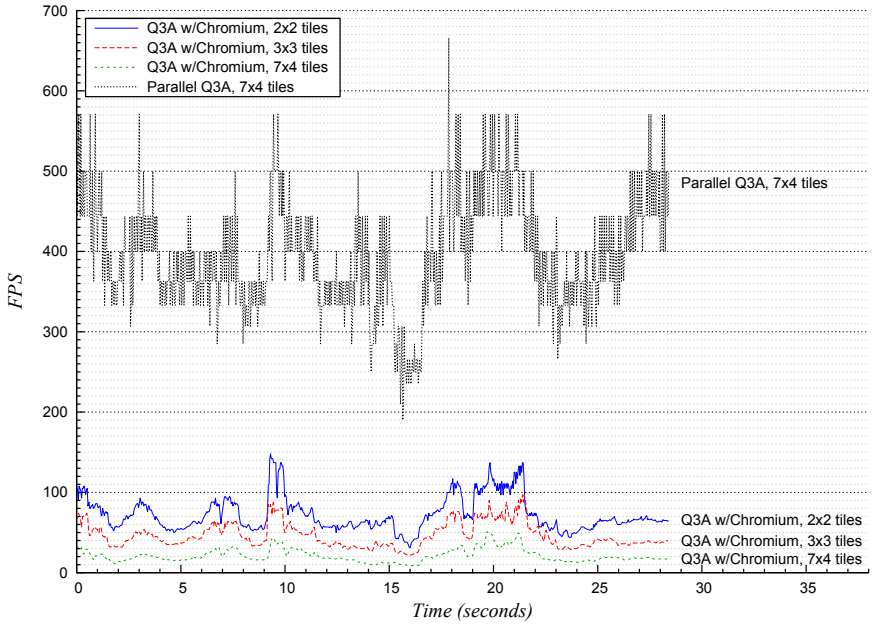

Figure 5.3: The frame rate when running Q3A on 2x2, 3x3 and 7x4 tiles using Chromium, compared to the parallel version's frame rate running on 7x4 tiles.

When using four rendering nodes (2x2 tiles) the peak FPS using Chromium is 148 frames per second, and the average is 73. For nine rendering nodes (3x3 tiles), the peak FPS is 97 and the average 47. For the 28-node configuration (7x4 tiles) the peak frame rate per second is 51 FPS and the average 21. For the parallel version, only the 28-node (7x4 tiles) configuration is listed because there were no significant differences in frame rate when varying the number of display nodes. For this version, the maximum frame rate is 666, and the average frame rate 398.

The additional latency introduced by the spectator approach used for Q3A is shown in figure 5.4. This latency was measured to be 87 ms.

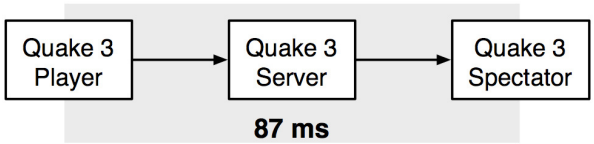

Figure 5.4: The additional latency introduced in Q3A's parallel version.

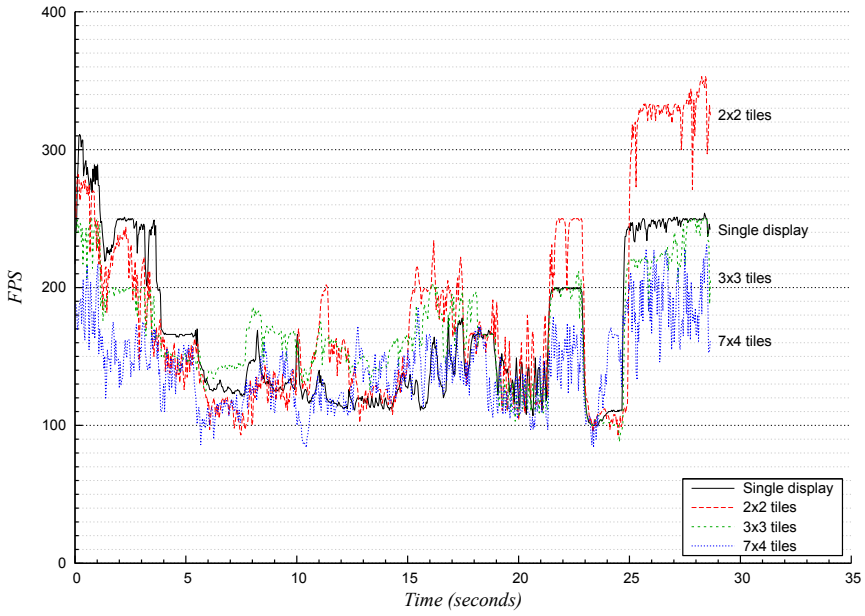

Figure 5.5: The frame rate when running Homeworld on a single display, compared to 2x2, 3x3 and 7x4 tiles.

Figure 5.5 shows the results from experiment series 4 (Homeworld). The frame rate for Homeworld was not capped during the experiment, which is why the frame rate is fluctuating much more compared to the Q3A measurements. To get

a better view of the frame rate, figure 5.6 shows the total number of frames drawn during the experiment.

The maximum frame rate for Homeworld running on a single tile is 311. For the 2x2, 3x3 and 7x4 tiles, the maximum frame rate is 353, 250 and 231. Respective average frame rates are 168, 183, 169 and 143. Figure 5.6 shows that the frame rate for the 2x2 and 3x3 configurations are higher than for a single computer. For all configurations, the performance is never lower than 80 FPS.

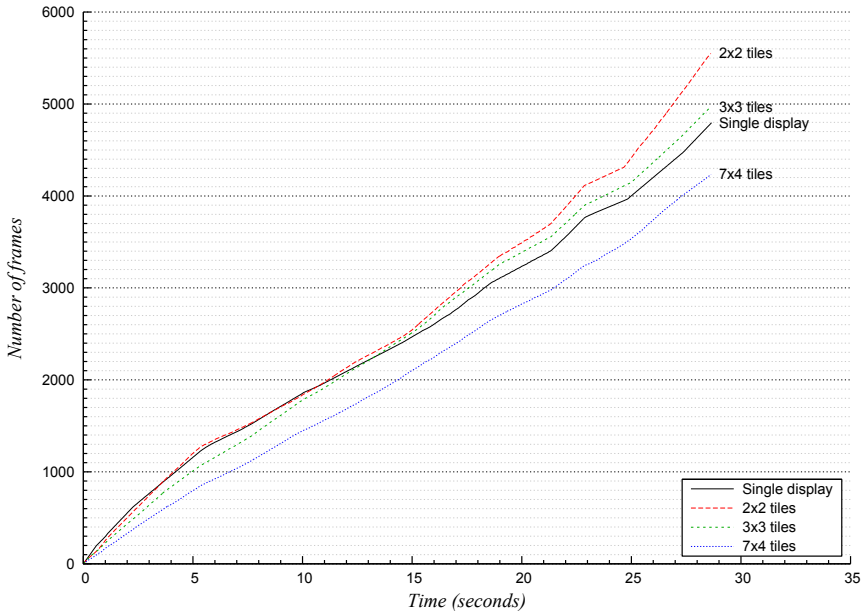

Figure 5.6: The total number of frames drawn when running Homeworld on a single display, compared to 2x2, 3x3 and 7x4 tiles.

Both parallelization approaches used are able to drive the games with frame rates higher than the refresh rate of the projectors. This indicates that the selected approaches are both promising ways to design games for parallel rendering on a display wall. The fluctuation of the frame rates of both games is a result of different complexity in the scenes, determined by the view of the camera.

Chromium has performance problems when increasing the resolution. For the 2x2 configuration, the average frame rate is higher than the refresh rate of the projectors. However, for the 3x3 tile configuration, the average frame rate is slightly lower than the refresh rate of the projectors, and for the 7x4 tile configuration it is only 21 FPS. This shows that Chromium used in this configuration has performance problems when increasing the resolution. The

reason for this is the cost of processing and sending graphics commands over the network, also documented in [27] and [36].

Conclusions

The work presented in this section shows three different approaches for making games designed using an SLSR approach run on a display wall. Each approach has different advantages, disadvantages and tradeoffs. Using Chromium is a quick approach to get output from an application onto the display wall. In addition, Chromium does not need any application modifications. However, as demonstrated in the presented results and documented in [27] and [36], Chromium used in this fashion has performance problems when more rendering nodes are introduced. The reason for this is that rendering commands needs to be sorted on the client-side before they are sent over a network connection, which has an order of magnitude less bandwidth than the PCI Express bus [88] on the north-bridge [89].

The two parallelization approaches used for transforming the versions into parallel versions, show that the two games can run with frame rates higher than the refresh rate between the computers and the projectors. The parallelization approaches have different tradeoffs. The client-server architecture of Q3A provides a convenient way of parallelizing the game. However, as measured in the experiments, this client-server approach leads to a higher latency in the system versus the sequential version. Homeworld does not have this latency as the input is immediately broadcasted to every copy in the display cluster. The main problem with the parallelization approach for Homeworld is synchronization of game state, learned through several difficulties in the implementation of the parallelized version. If a single branch in the code makes processes diverge, the game state will slowly but surely get out of sync. This problem is hard to address since it requires potentially every branch of the game logic to be executed to make sure the game stays synchronized. Q3A does not get out of sync if a message is lost, since the next message will provide the game state necessary to stay synchronized. The simplicity and robustness of the state server led to the principle of domain specific best-effort synchronization, used for the work presented later in this dissertation (pull-based network accessible display- and compute-resources, chapter 7).

5.4.2 Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core

The GPU has traditionally been used to offload the CPU from graphics computations to accelerate the rendering of interactive visualizations such as real-time graphics used in games. With the introduction of general-purpose computing environments for GPUs such as CUDA, the data-parallel processing power can now be utilized for general-purpose computations without having deep knowledge of graphics APIs and GPUs intrinsic for graphics processing. To

investigate the GPU as a general-purpose computing platform and gain insight into the implications of using GPUs as accelerators for NACs in display wall contexts, the following work presents a performance comparison of the embarrassingly parallel Mandelbrot set computation on a CUDA capable GPU, with the available computational resources of a display wall.

Related Work

The Mandelbrot set computation has been used for several parallel and multi-core performance studies [90] [91] [92] [93] [94] because of its simplicity and its embarrassingly parallel nature. There have also been conducted several studies comparing the performance between GPUs and CPUs, among others in [95] [96] [97] [98] [99]. However, contrary to most of related work, which has used a graphics API and vertex and/or fragment shaders for running the application on one or several graphics cards, the work presented in this section removes much of the limitations introduced by previous generation GPUs, such as the need for using a graphics API for executing the code on the GPU. The combination of CUDA and the Mandelbrot set computation allows the same code to be used for both CPU and GPU versions with minor changes to specify the mapping of the computation to the GPU cores. This allows for a more direct comparison between GPU and CPU implementations. Further, CUDA provides full scatter and gather memory operations, which is not possible on older generation GPUs. Finally, some applications running on previous generation cards, have been bottlenecked by the DRAM memory bandwidth, thereby underutilizing the GPU's computational power [100]. While there have been conducted performance comparison studies between CUDA capable graphics cards and clusters [101], none have used the embarrassingly parallel Mandelbrot set computation, which has a very low communication to computation ratio. This allows for focusing on computational power of different compute architectures while limiting the use of shared interconnects that might create bottlenecks. In addition, a display wall can be treated as a large frame buffer where the results can be viewed on each tile, avoiding transmission of data to a central computer each iteration for displaying. While some related work have focused on auto-tuning and auto-mapping between computations and compute cores [91] [102] [103], the work presented in this section focuses on manual tuning between computations and compute cores.

Architecture

The architectures of the computational platforms used as part of this work are the Compute Unified Device Architecture (CUDA) and a display wall with associated display cluster. The architecture of CUDA is described in chapter 3. Display walls are described in chapter 2. A description of the hardware comprising both the CUDA capable graphics card and the display wall is provided in the methodology section.

DESIGN

The requirement for the benchmark used in the experiments is a computation that requires no communication between compute cores. For this reason, the Mandelbrot set computation was chosen for the experiments.

The Mandelbrot set is a set of points in a complex plane that are quasi-stable when computed by iterating a function, usually $z_{k+1} = z_k^2 + c$ (where $z = a + bi$ and $i = \sqrt{-1}$) [104]. The iterations are continued until the magnitude of z is greater than 2, or the number of iterations has reached an arbitrary limit. Each point in the plane can be computed without any knowledge of the surrounding points. This property makes the Mandelbrot set particularly convenient to parallelize, as each point can be separately assigned to compute cores without any further communication between the cores. While the number of points in the complex plane is known in advance, the number of iterations needed to compute the result of each point is unknown. Therefore, statically dividing the Mandelbrot set between compute cores would not result in the same workload-balance. Dynamic partitioning has been documented to give better performance [92] due to a more balanced utilization of each compute core. The challenge in using dynamic partitioning is to find a task size that gives the right balance between compute core utilization and the overhead introduced by handing out tasks and collecting results.

Five different versions of the Mandelbrot set computation are developed to evaluate the different hardware platforms. These versions are divided into CPU- and GPU-based:

1. **CPU-based:**
    a. A single-process multi-threaded CPU version (CPUMandelbrot).
    b. A statically workload-partitioned multi-process multi-threaded CPU version (WallCPUMandelbrot (static)).
    c. A dynamically workload-partitioned multi-process multi-threaded CPU version (WallCPUMandelbrot (dynamic)).
2. **GPU-based:**
    a. A statically workload-partitioned multi-process multi-threaded GPU version (WallGPUMandelbrot).
    b. A dynamically workload-partitioned single-process multi-threaded GPU version (CUDAMandelbrot).

All versions are designed to start at a pre-determined starting-point and, through several iterations, zoom into a pre-determined end-point.

Implementation

The complex plane of the Mandelbrot set is stored as an array of 32-bit integers. Each entry in the array holds a 32-bit value for the corresponding part of the Mandelbrot set. This value is a color calculated from the number of iterations used to compute the result for that point in the complex plane. Each iteration can be calculated independently of previous iterations by specifying a zoom-box into the Mandelbrot set.

All the versions developed use the same algorithm for computing the Mandelbrot set. OpenGL is used for the rendering. For the CPU versions, displaying the result from iterations require data to be copied from main memory to GPU memory. For the GPU versions, this copy can be done internally in graphics card memory.

Single-process multi-threaded CPU version (CPUMandelbrot)

The single-process multi-threaded CPU version (figure 5.7) is the baseline for comparison. The Mandelbrot set is divided into horizontal lines, and each thread computes points for lines that are multiples of their assigned ids (ids are assigned to threads incremental, starting at 0). Assigning threads to different parts of the Mandelbrot set allows for utilization of multi-core architectures. The number of threads actually used is based on the number of CPU cores.

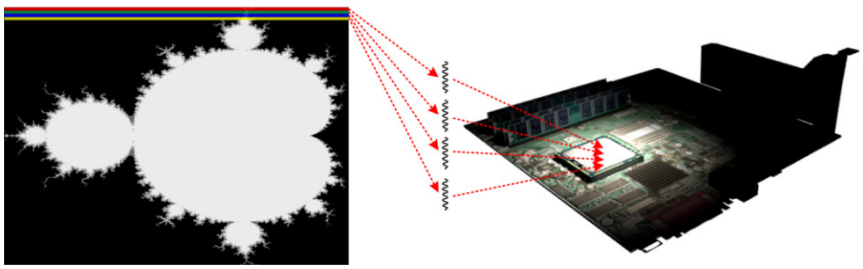

Figure 5.7: The assignment of the Mandelbrot set for CPUMandelbrot.

Statically Workload-partitioned multi-process multi-threaded CPU version (WallCPUMandelbrot (static))

This version splits the Mandelbrot set into multiple parts (1 per display node), as shown in figure 5.8. The parts are divided with respect to the display nodes placement in the display grid, i.e. the upper left part of the Mandelbrot set is computed by the upper left display node. The computation of the result uses the same approach as the single-process multi-threaded version, where threads compute points from lines that are multiples of their assigned ids.

At the end of each iteration, all nodes synchronize through a barrier, at which point the result is displayed. The barrier is performed using MPI_Barrier

(LAM/MPI). The communication overhead of this version is low. However, the time to complete each iteration is limited to the time spent by the most loaded node.

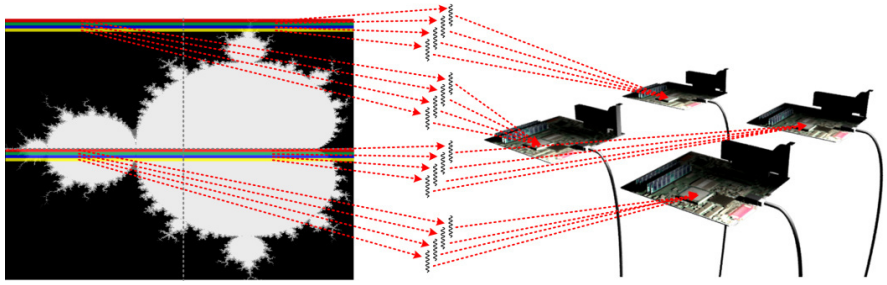

Figure 5.8: The assignment of the Mandelbrot set for WallCPUMandelbrot (static). The example illustrates how the Mandelbrot set would have been divided if 4 compute nodes were used.

Dynamically Workload-partitioned multi-process multi-threaded CPU version (WallCPUMandelbrot (dynamic))

To better the CPU utilization from the statically workload-partitioned version, this version uses dynamic partitioning of work tasks to solve the Mandelbrot set (figure 5.9). A task is a subset of the Mandelbrot set for one node. A separate dispatcher node keeps track of all tasks. Every node starts out with an initial task, which is half the height of the node's part of the Mandelbrot set.

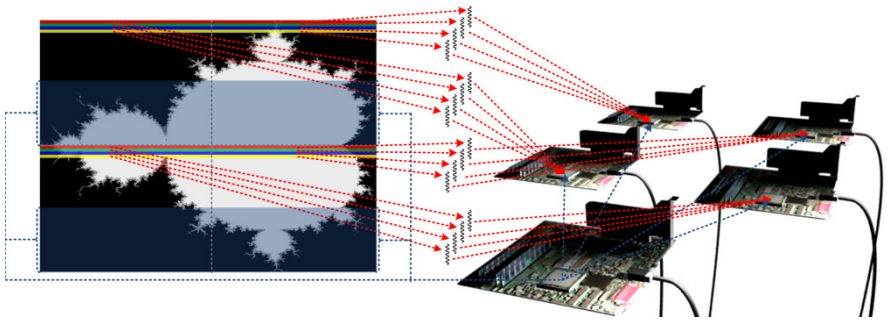

Figure 5.9: The assignment of the Mandelbrot set for WallCPUMandelbrot (dynamic). The example illustrates how the Mandelbrot set would have been divided if 4 compute nodes were used, including a separate node used as dispatcher.

The dispatcher keeps track of the remaining tasks for each node. If there is none left, it selects a task from the node with the most remaining tasks and assigns it to the requesting node. This node transfers the result to the node responsible for the

requested area when finished. When all nodes have completed all assigned tasks, the dispatcher sends a NOOP-task to all nodes, resets the remaining tasks counter for all nodes, and increments the zoom-counter. The program terminates when the zoom-level has been reached.

Statically Workload-partitioned multi-process multi-threaded GPU version (WallGPUMandelbrot)

This version statically divides the Mandelbrot set between all nodes, and utilizes the GPU for the computation (figure 5.10). For each iteration, the CPU computes a new zoom box into the Mandelbrot set. However, instead of computing the points in the zoom box on the CPU, the parameters of the zoom box are transferred to the GPU and the result is computed in a GLSL fragment shader. The parameters are transferred to the fragment shader using uniform variables. The result of the computation is rendered to an off screen texture located in GPU memory and scaled to the back-buffer before swapping screen buffers. Dynamic load balancing is not used for this version, and thus the time for each iteration is constrained to the most heavily loaded node.

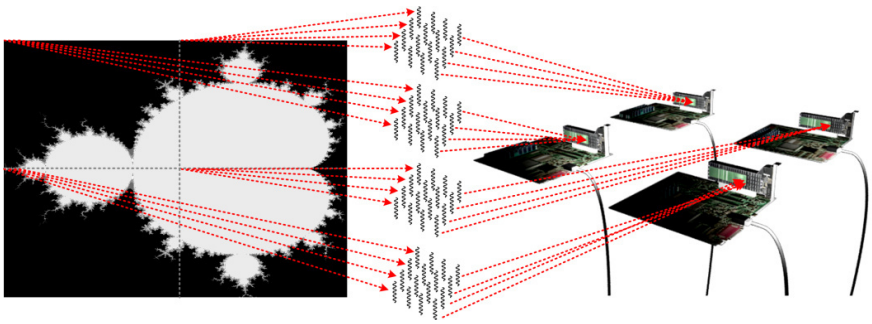

Figure 5.10: The assignment of the Mandelbrot set for WallGPUMandelbrot. The example illustrates how the Mandelbrot set would have been divided if 4 compute nodes were used.

Dynamically Workload-Partitioned Single-process multi-threaded GPU version (CUDAMandelbrot)

This version uses CUDA to solve the Mandelbrot set (figure 5.11). The Mandelbrot set is divided into multiple tasks, and each task is processed by one thread. The processors can process 8 SIMD threads in parallel, and additionally a hardware thread scheduler periodically switches between warps of threads to maximize the utilization of the processor's cores and hide global memory latencies. To accommodate the massively parallel architecture of the graphics card and the low overhead of creating new threads, the Mandelbrot set is divided into a minimum of 10 000 thread tasks, depending on the Mandelbrot set resolution. Thereby, the automatic load balancing provided by the CUDA

architecture through the use of over-decomposition in combination with hardware thread scheduling is utilized.

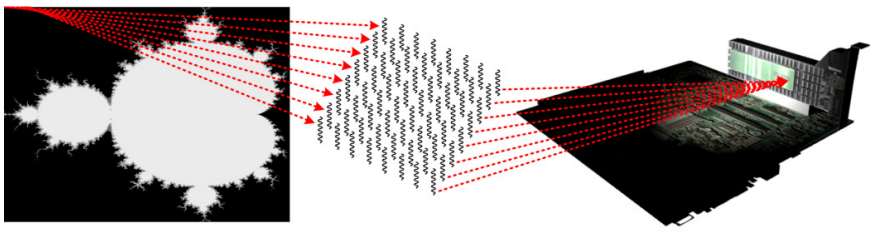

Figure 5.11: The assignment of the Mandelbrot set for CUDAMandelbrot.

Since each of the points in the Mandelbrot set can be computed individually, there is no need for communication between processor cores within each iteration. Each thread writes the result of their task to global device memory, which has a latency of 400 – 600 clock cycles. Since each task only requires one global memory operation, the hardware thread scheduler should hide the memory latency to a certain extent. After each iteration, the CPU calculates a new zoom box into the Mandelbrot set, which it passes to the graphics card for a new iteration. CUDAMandelbrot can also be configured to divide and send the result of each iteration to a number of display clients (figure 5.12). If this configuration is enabled, the result of the computation is copied from device memory (VRAM) to host memory (main-memory) and sent to each corresponding display client. No compression is used.

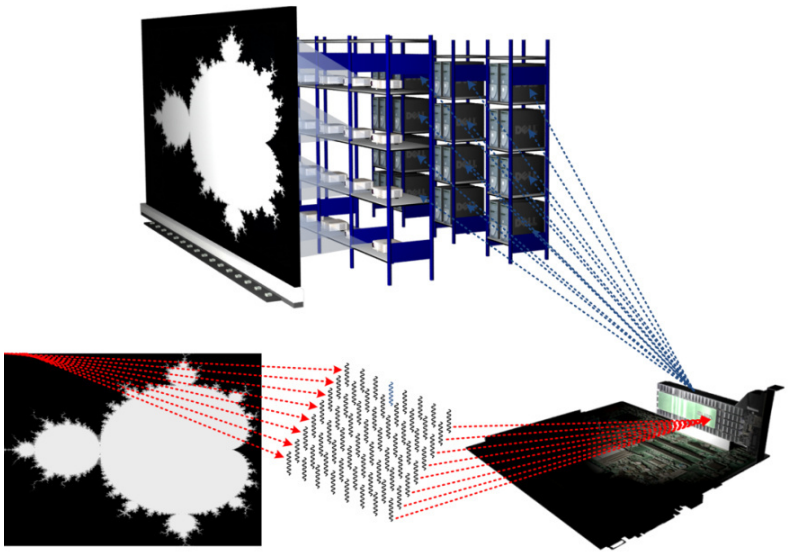

Figure 5.12: The assignment of the Mandelbrot set for CUDAMandelbrot when configured to send the output to a set of display nodes.

EXPERIMENTS

To evaluate the performance of the different implementations, 15 experiment series were conducted.

METHODOLOGY

The factor for each experiment series was the resolution of the Mandelbrot set. This factor had the following values:

1. 1024x1024
2. 2048x2048
3. 4096x4096

The performance (time for a predefined number of iterations to complete) of the single-core CPU version was used as the baseline of the experiments. This version used 4 threads (found to be the best number of threads through experimentation) for computing the Mandelbrot set. For the multi-process versions 4, 9, 16 and 28 compute nodes were used. These also used four threads per process for computing the Mandelbrot set. For the single-process multi-threaded CUDA version, the Mandelbrot set was computed on a single node using all the compute cores of the graphics card. For all experiments, the maximum number of iterations for each point of the Mandelbrot set was set to 100.

A summary of the experiments is listed in table 5-3. The hardware used in the experiment is listed in table 5-4.

Table 5-3: Experiment summary

Experiment Series	Description	Factor (Resolution)
1	CPUMandelbrot	1024x1024, 2048x2048 and 4096x4096
WallCPUMandelbrot (static)		
2	4 nodes	1024x1024, 2048x2048 and 4096x4096
3	9 nodes	
4	16 nodes	
5	28 nodes	

WallCPUMandelbrot (dynamic)		
6	4 nodes	1024x1024, 2048x2048 and 4096x4096
7	9 nodes	
8	16 nodes	
9	28 nodes	
WallGPUMandelbrot		
10	4 nodes	1024x1024, 2048x2048 and 4096x4096
11	9 nodes	
12	16 nodes	
13	28 nodes	
CUDAMandelbrot		
14	Standard configuration	1024x1024, 2048x2048 and 4096x4096
15	Configured to send the resulting output from each iteration to the display wall	

Table 5-4: Hardware- and software-platform

	Compute Nodes	CUDA Capable GPU
Type	Display Cluster	Desktop Computer
Number of Nodes	28	1
CPU	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading
RAM	2 GB	2 GB
Graphics Card	NVidia Quadro FX3400 w/256 MB VRAM	Bliss GeForce 8800GT PCX w/1024 MB VRAM

Operating System	Rocks Linux Cluster Distribution 4.0	Linux Ubuntu 7.04
GCC version	3.4.4	4.1.2
Compiler options	-O2	-O2
Interconnect	Switched gigabit Ethernet	

RESULTS AND DISCUSSION

Figure 5.13 shows the speedup factor of the parallel versions compared to (divided by) CPUMandelbrot for a resolution of 4096x4096. The x-axis is the number of nodes and the y-axis is the speedup factor. Different resolutions did not have any significant impact on the performance and therefore only the speedup graph for the highest resolution is shown.

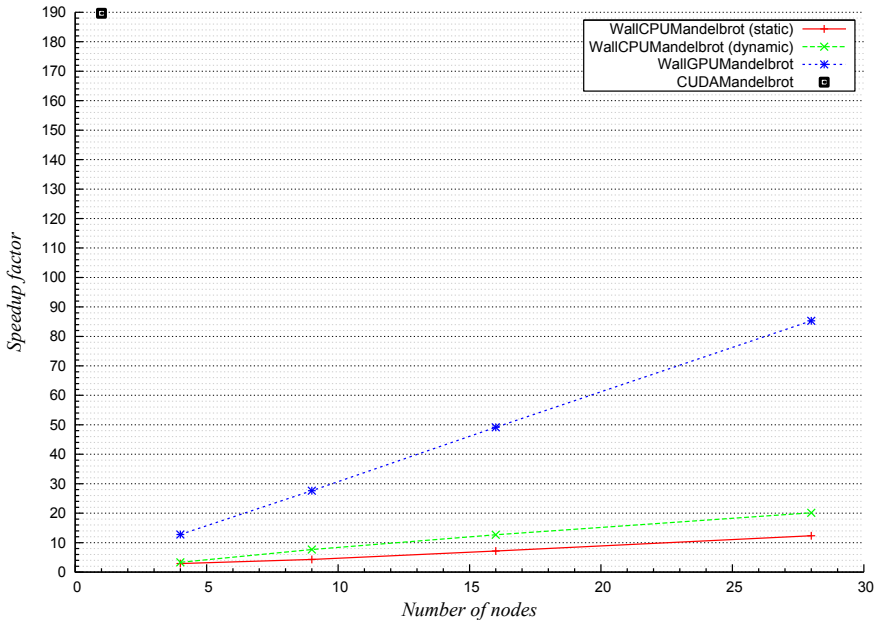

Figure 5.13: Speedup factor of the parallel versions compared to CPUMandelbrot.

The figure indicates a linear increase in speedup when increasing the number of display nodes. CPUMandelbrot (dynamic) is faster than the statically partitioned version. WallGPUMandelbrot is the fastest of the cluster versions with a performance increase of 85.3 using 28 nodes. The performance increase using the

GeForce 8800GT PCX card is higher than all versions. Compared to the single-process multi-threaded version (CPUMandelbrot) it is almost 190 times faster. Compared to the parallel versions computing on the CPU, the 8800GT PCX is almost an order of magnitude faster. Compared to the multi-process multi-threaded GPU version computing on previous generation graphics cards (WallGPUMandelbrot), the 8800GT PCX is over twice as fast.

Figure 5.14 shows: (i) The resulting performance when sending the output of each iteration from the GeForce 8800GT PCX card to the display wall cluster; (ii) the performance of WallCPUMandelbrot (static); and (iii) the performance of WallCPUMandelbrot (dynamic). The x-axis is the resolution for the Mandelbrot set in megapixels, and the y-axis is the speedup factor. As the figure shows, it is faster to compute the Mandelbrot set on the 8800GT PCX card, and send the output of each iteration to the display cluster, than to compute the result locally on each cluster node, even for the load-balanced version.

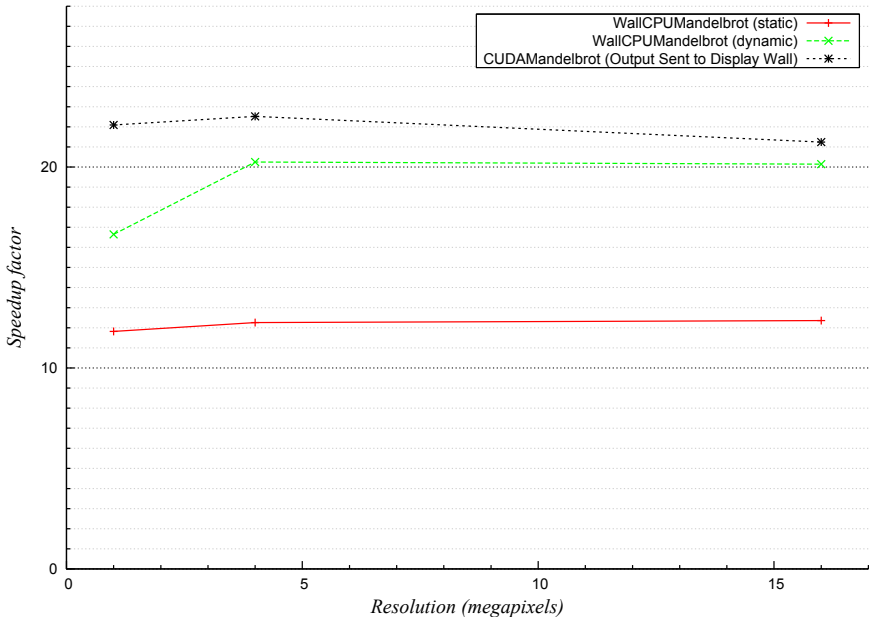

Figure 5.14: The relation between speedup and resolution for the WallCPUMandelbrot versions compared to CUDAMandelbrot configured to send the output of each iteration to the display wall cluster (speedup is calculated based on CPUMandelbrot).

The high performance increase using the 8800GT PCX card can be explained by the high data-parallelism, extreme thread-level parallelism and the high memory bandwidth of the graphics card. Combining small task sizes with a large number of threads utilizes zero-overhead thread creation [48], limits thread divergence and utilizes the hardware thread scheduler, which again results in increased

utilization of the high GPU global memory bandwidth. This occurs despite the fact that the result needs to be copied from GPU memory to main memory and sent over the network to each cluster computer for each of the iterations.

Conclusions

The new generations of GPUs promise new possibilities in scientific computing. The combination of unified access to the GPU compute cores and memory, and the fact that the GPU can be programmed using simple C extensions open new possibilities in high performance computing. Experiments show that a single graphics card can outperform an entire cluster of computers by almost an order of magnitude, due to the GPU's high level of data-parallelism and extreme level of thread-parallelism. In fact, for a cluster of 28 nodes (Pentium 4 3.2 GHz CPUs) interconnected with gigabit Ethernet, it is faster to compute the Mandelbrot set on a GeForce 8800GT PCX graphics card and send the result to the cluster nodes, than to divide and compute the Mandelbrot set locally on each node.

The work presented in this section has shown that for certain types of visualizations and computational resources, it can be beneficial to move compute intensive tasks to nearby network accessible compute resources. This is the concept of NADs and NACs and the visualization distribution space introduced in chapter 1, section 1.3. Depending on the technologies available, it can be beneficial for a visualization system to allow parts of the visualization process to be moved to available compute resources, depending on the computational power of the compute resources, the local display resources and the network connecting these.

5.4.3 Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors

Graphics processors are, as documented in the previous section, a promising platform for SIMD computations. However, many existing CUDA capable GPUs, including some of the Tesla [105] cards (compute capability 1.0) aimed at high-performance computing, do not support any strong synchronization primitives like test-and-set and compare-and-swap, which are usually used in constructing fault-tolerant synchronization primitives [106]. The lack of synchronization primitives for global and shared memory is a limitation for using a CUDA capable GPU as a NAC, because most parallel applications need some synchronization mechanism to synchronize their concurrent processes [106].

The following work presents a study of fault-tolerant synchronization mechanisms for CUDA capable GPUs. A wait-free synchronization mechanism [107] is implemented that eliminates lock-related problems like deadlock and, in addition, can tolerate process-crash failure.

ARCHITECTURE

The computed unified device architecture is described in chapter 3, section 3.2. The feature of this architecture, used to create synchronization primitives, is the ability to read and write multiple 32-bit, 64-bit and 128-bit words to and from global memory in one memory transaction. This requires the variable type to be a multiple of 4, 8 or 16 bytes, and the read or write instructions must be arranged so that the memory accesses can be coalesced into a single contiguous aligned memory address [48], shown in figure 5.15.

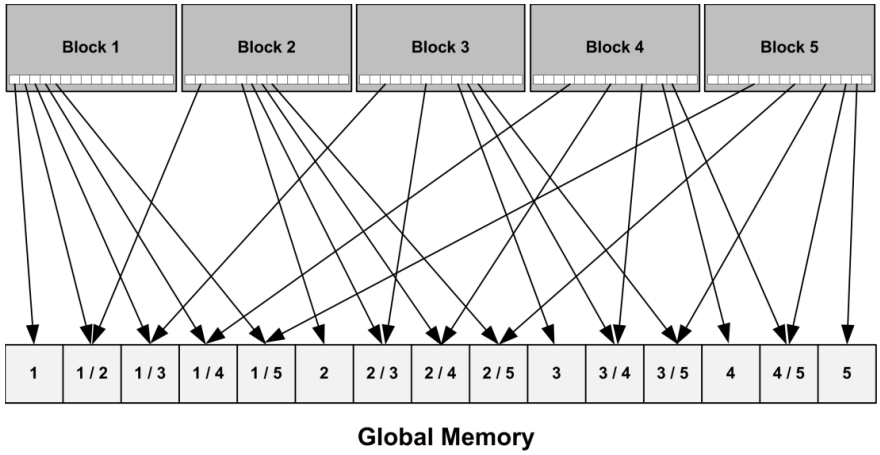

Figure 5.15: The arrangement of threads and warps for coalescing memory access to global memory.

DESIGN

The synchronization primitives are based on an algorithm developed in [106]. The algorithm is based on a long-lived consensus [106], which is used to achieve an agreement between concurrent processes. From the properties of the long-lived consensus [106] a Read-Modify-Write (RMW) object [106] is constructed that encapsulates a memory region through which participants can communicate.

The operation RMW(X, f), where X is a shared variable and f is a mapping, is defined to be equivalent to the indivisible execution of the following function [108]*:*

```
function RMW(X,f)
        begin
                temp ← X;
                X ← f(X);
                return temp;
        end
```

The *RMW* object is realized by combining the *LongLivedConsensus* presented in [106] with a round numbering scheme. A participant that invokes an operation on

the *RMW* object is assigned to a round. If more than a single participant invokes the *RMW* object within the same round, the *RMW* algorithm [106] ensures that all participants will agree upon a common sequence of accesses, and thereby ensure the integrity of the *RMW* object. Each of the participants belonging to the same round suggests an order of accesses in that round. The *LongLivedConsensus* algorithm is used to achieve an agreement on the order to use.

Implementation

For global memory, the property of coalesced memory access is utilized to establish an agreement between warps (potentially running in different blocks). By arranging warp writes as described in the *LongLivedConsensus* (shown in figure 5.15), an agreement on a common winner can be established among warps. For shared memory, the writes do not need to fulfill the requirement of coalesced memory access, but memory writes do need to be arranged according to the *LongLivedConsensus*.

The *RMW* object is implemented as a CUDA function encapsulating a shared memory region. The function and arguments of each warp that invokes the *RMW* object is written to the warp's part of a shared memory location. This memory location is readable from all warps that invoke the *RMW* object. Each warp reads the function and parameters from all other warps that participate in the same round, calculates a value of the *RMW* object based on its own sequence of the functions, and then writes the result to a known memory location that can be read by every participating warp. The warp then invokes the *LongLivedConsensus* using the memory location of its proposal. For global memory, five threads of a half-warp in each block write to global memory in one coalesced memory operation. For shared memory, the first sixteen threads of a warp (the first half-warp) write to shared memory. After the writes, each memory entry is compared to the others in order to find the warp that wrote first. This is done using an *Ordering* algorithm [106] in combination with a *RoundCheck* algorithm [106]. After the execution of the *Ordering* algorithm, the warp that wrote first will be known for all warps participating in the same round. Since each of the warps executes one function on the *RMW* object at a time, functions are ordered according to both the round they participate in and the order agreed upon by warps in the same round.

The current implementation of the algorithm supports wait-free synchronization between five warps using global memory (limited by the memory segment size for coalesced memory access), and fifteen warps using shared memory (limited by the algorithm (*2M-2* half-warps for *M=16* banks)), and in addition supports cards of any compute capability. However, the actual number of warps synchronized through shared memory is limited to five, because the data structures that encapsulate the *RMW* object consume too much on-chip memory (over 16 kb) when the number of warps exceeds this limit. This will be optimized for future implementations. The algorithm is designed for an asynchronous

memory model. For CUDA, the access speed to shared memory is the same as for registers if no bank conflicts occur [100]. For this reason, the shared memory version contains several duplicates that can be removed in future optimizations of the implementation. However, the duplicates are kept for the current implementation to achieve a literal implementation of the algorithm.

The *RMW* object supports any read-modify-write operation such as the atomic operations in graphics cards with compute capability 1.1 and up. That is: *ADD*, *SUB*, *EXCH*, *MIN*, *MAX*, *INC*, *DEC* and *CAS*. In addition, the *RMW* object supports atomic operations on floating point numbers.

EXPERIMENTS

To evaluate the implementations of the *RMW* object for both global and shared memory, two experiment series were conducted.

METHODOLOGY

The atomic operation *atomicAdd* was used to evaluate both hardware and software support. For both experiment series this operation was invoked 30 000 times. The performance of the operation was measured by recording a timestamp before the CUDA kernel was started, and another timestamp when the kernel was finished.

The first series of experiments was conducted to determine the overhead of software synchronization in global memory. For these series of experiments, the *RMW* object was invoked 30 000 times. The factor was the number of blocks, which was increased from 1 to 5, each block having 16 threads belonging to the same warp (the first half-warp). Each experiment was repeated 10 times.

The second series of experiments was conducted to compare software support in shared memory to hardware support in global memory. The *RMW* object was invoked 30 000 times and the time to complete all invocations was recorded. The factor was the number of warps used, which was increased from 1 to 5 warps. Each of the experiments was repeated 10 times. The experiment series are summarized in table 5-5. The hardware used in the experiments is listed in table 5-6.

Table 5-5: Experiment summary

Experiment Series	Description	Factor
One	Software synchronization in global memory versus hardware support for the same operation in global memory	1-5 blocks

Two	Software synchronization in shared memory versus hardware support for the same operation in global memory	1-5 warps

Table 5-6: Hardware- and software-platform

	CUDA 1.0 Card	CUDA 1.1 Card
Type	Dell Precision 370	Dell Precision 370
Number of Nodes	1	1
CPU	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading
RAM	2 GB	2 GB
Graphics Card	Bliss GeForce 8800GT PCX w/1024 MB VRAM	Bliss GeForce 8800GTS w/640 MB VRAM
Operating System	Linux Ubuntu 7.04	Linux Ubuntu 7.04
GCC Version	4.1.2	4.1.2
Compiler Options	-O2	-O2
Interconnect	Switched gigabit Ethernet	

The GeForce 8800GT PCX card was used as a performance baseline for the *atomicAdd* in hardware. Both graphics cards were used in the performance measurements of the global and shared software implementations.

Results and Discussion

Figure 5.16 shows the time used to invoke the RMW object in global memory compared to hardware support for the same operation.

Figure 5.17 shows the ratio between software and hardware support in global memory. The values are calculated by dividing the time for software synchronization by the time used for hardware support.

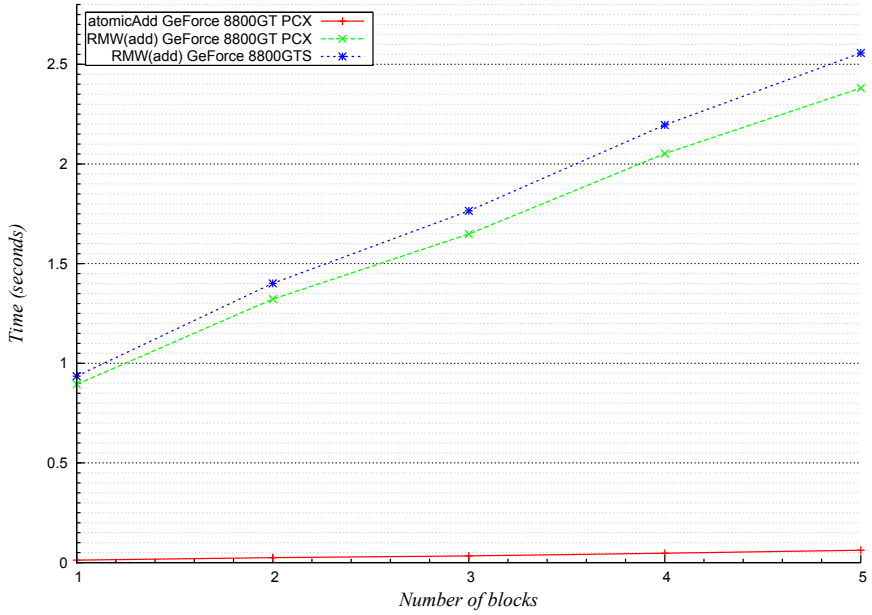

Figure 5.16: The time used for 30 000 invocations of the RMW object in global memory compared to atomic support in hardware (global memory).

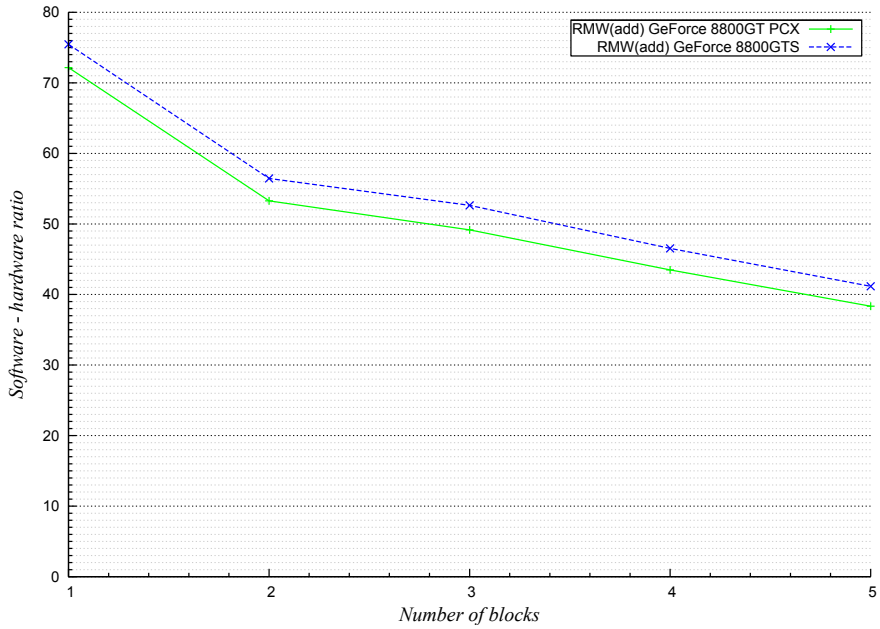

Figure 5.17: The software- to hardware-ratio for the 30 000 invocations (software support in global memory divided by hardware support in global memory).

As figure 5.16 illustrates, hardware support is an order of magnitude faster than software support. For 1 block, the atomic operation in hardware uses 0.0124 seconds to complete. For the GeForce 8800GT PCX card, the time to invoke the RMW objects is 0.89 seconds and the GeForce 8800GTS card uses 0.935 seconds. For 5 blocks, atomic operation in hardware takes 0.062 seconds for 30 000 iterations. For software support, the time is 2.38 seconds for the 8800GT PCX card and 2.56 for the 8800GTS card. The figure indicates a linear increase in time for both software and hardware support as the number of blocks increases.

For 1 block, hardware support is 72.16 times faster than software support for the 8800GT PCX card and 75.45 seconds faster for the 8800GTS card (figure 5.17) However, for 5 blocks this factor has decreased to 38.33 for the 8800GT PCX card and 41.17 for the 8800GTS card. The graph shows that the software- to hardware-ratio decreases as the number of blocks increases.

Figure 5.18 shows the time used to invoke the RMW object in shared memory compared to the time to invoke the same operation using hardware support in global memory.

Figure 5.19 shows the ratio between software support using shared memory and hardware support in global memory. The values are calculated by dividing the time used for software support in shared memory by the time used for hardware support in global memory.

Figure 5.18 shows that synchronization using hardware support is faster in global memory than software support in shared memory. For 1 warp, the hardware supported atomic operation uses 0.01239 seconds to complete compared to 0.303 seconds for the 88800GT PCX card and 0.396 seconds for the 8800GTS card. For 5 warps, the atomic operation in hardware uses 0.0621 seconds and the shared memory version uses 1.07 seconds for the 8800GT PCX card and 1.471 seconds for the 8800GTS card. The time to do synchronization in shared memory is greater than hardware support in global memory for all warp configurations. This indicates that the computational steps of the RMW object is the factor limiting the speed of the software synchronization, as accessing shared memory is two orders of magnitude faster than global memory.

For 1 warp, the factor between hardware support in global memory and software support in shared memory is 24.46 for the 8800GT PCX card and 31.95 for the 8800GTS card (figure 5.19). For 5 warps, this factor has decreased to 17.22 for the 8800GT PCX card and 23.67 for the 8800GTS card. As opposed to software synchronization in global memory, software synchronization in shared memory seems to flatten out between four and five warps.

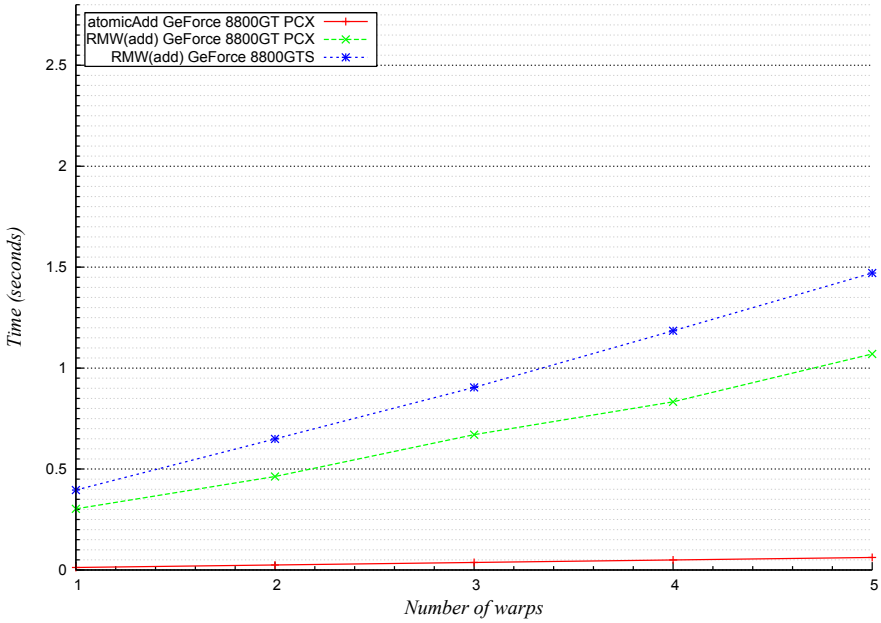

Figure 5.18: The time used for 30 000 invocations of the RMW object in shared memory compared to atomic support in hardware (global memory).

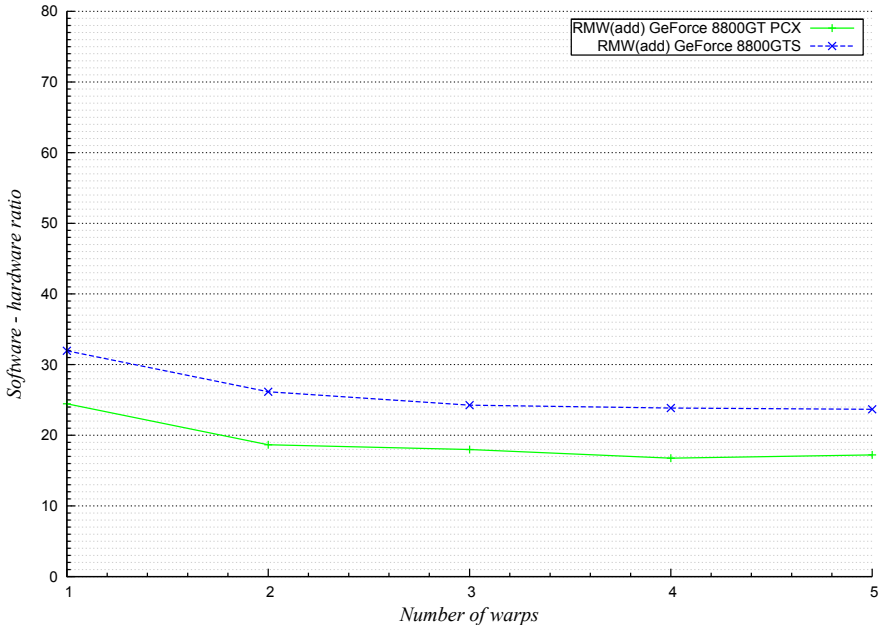

Figure 5.19: The software- to hardware-ratio for the 30 000 invocations (software support in shared memory divided by hardware support in global memory).

Conclusions

This section has presented an experimental evaluation of a Read-Modify-Write (RMW) object for CUDA capable graphics cards. The RMW object implemented enables CUDA cards of any compute capability (as defined by NVIDIA in [48]) to communicate through global and shared memory. The current version of the RMW object implemented in global memory supports five warps running in different blocks, and the synchronization is wait-free for up to four failing warps. For the shared memory version, the synchronization primitive has a limit of fifteen warps. However, the amount of on-chip shared memory reduces the actual number to five.

The experiments conducted indicate that the performance bottleneck of the RMW object is the computational steps needed to ensure consensus between the participating warps. The hardware implementation is an order of magnitude faster than the software implementation. However, as the number of warps increases from one to five, the performance gap between software and hardware is reduced from 72.16 times to 38.33 times for global memory, and 24.46 times to 17.22 times for shared memory.

The work presented in this section has shown that CUDA GPUs of compute capability 1.0 can support atomic operations through global and shared memory by utilizing the memory intrinsic of the graphics cards. This shows that future NACs based on cards of this compute capability, such as some of the Tesla cards, can support parallel applications that require synchronization through both global and shared memory.

CHAPTER 6

PUSH-BASED NADS AND NACS

Network accessible compute- and display-resources are categorized into push-based and pull-based. This chapter describes the push-based network accessible compute- and display-resources built as part of the work presented in this dissertation. The chapter is based on the following peer-review published paper: [73].

6.1 THE NAD SYSTEM

A number of problems are introduced when connecting a display or a projector directly to a computer by a DVI/VGA cable:

1. Detecting the video signal between a computer and a projector might fail.
2. The resolution of the desktop display is often inconveniently reduced to the resolution of the external display.
3. The entire mirrored or extended desktop is displayed, instead of giving the user a finer control of what to display.
4. There is limited support for using multiple projectors and displays from a single computer.
5. When connecting a computer to a projector the projector may be reconfigured to a sub-optimal configuration.
6. Multiple users cannot simultaneously share the same projectors and displays.

For presentations, an in-room projector-computer rig running all programs and presentations can be used. However, this approach may cause problems because of missing applications or application being incompatible with the data formats used.

Another approach is to use an in-room projector-computer rig running a remote desktop system, from where the user can bring the displaying output of the local computer. However, this requires compatible remote desktop systems on the user's computer and the projector-computer rig. If this is not present, compatible remote desktop systems must be installed. This often requires the firewall on the user's computer to be opened, since contact is initiated from a remote computer to a server running on the user's computer. In addition, there are trust issues, since several remote desktop systems require the username and password to be submitted in order to access the computer.

The idea behind the NAD system is to customize a NAC in order for the NAC to use a NAD. This solution follows the principle of establishing the end-to-end principle through customization. The purpose of this idea is a simple and flexible way to use nearby display resources without requiring permanent installation of software or opening firewall ports on the NAC.

To overcome the aforementioned problems a system is built to realize the presented idea. The system adheres to the NAD model, which enables displays to be used by customized NACs as if they were physically connected.

To following requirements are integral to the presented system:

1. The NAC software and the NAD software should be cross-platform and available for Windows, Linux and Mac OS X.

2. The customization of the NAC should be transparent to the user.

The following sections describe the NAD system, as well as the experiments designed and conducted.

6.1.1 Related Work

Some of the functionality provided by the system presented in this section can be achieved using a remote desktop system. A remote desktop system is a system that enables a user to view and interact with the desktop of one computer from another computer. One such system is VNC [37]. The protocol used by VNC is extendable and has been adopted by several systems. Some of these are RealVNC [61], TightVNC [62] and UltraVNC [63]. Windows Remote Desktop [79] is another popular remote desktop system shipped with a selection of the Microsoft Windows operating systems. In addition to VNC and Window Remote Desktop, there are multiple other remote desktop systems. All share a common goal: Bring

the content from the desktop of a remote computer to a computer display where the user is located.

Remote desktop systems have a number of drawbacks when users want to share information on a common display/projector to demonstrate programs on-the-fly (some already mentioned in the previous section). Some of the systems are platform dependent. This is inconvenient in meeting room environments because attendees need to have matching remote desktop systems. Another problem with remote desktop systems is that the users in some cases must install third party software that permanently alters the local install, for example by requiring users to open firewall ports or by installing third party libraries in system folders of the operating system.

The main problem with remote desktop systems when used to give presentations on remote display/projectors and in meeting room contexts is their pull-based architecture, which implies that users must initiate the contact from a remote computer. If several people want to display output on a common display, this requires all users to share a common keyboard and mouse. In addition only one user can control the appearance of the window of his/her remote desktop.

There are systems that enable a push-based remote desktop approach. One such system is the TightProjector [109]. TightProjector enables a user to multicast the desktop of a Microsoft Windows computer to other computers in the same local area network. However, the system is only available for Windows, and users still needs to manually download and start a binary. There are other systems that enable a push-based remote desktop approach. Some of these are The 22 Megapixel Laptop [47] (the work that inspired the creation of the NAD system), MaxiVista [110], ZoneScreen [111] and ScreenRecycler [112]. However, these systems require installation of third party software, and some alter the software at the kernel level [47]. In addition, for these systems, the focus is on extending the desktop to the remote screens, where in the NAD system, the focus is to enable mirroring of content from the local desktop.

A Windows Network Projector [113] is a network accessible projector running Windows Remote Desktop (RDP) software. The software is only available for selected windows installs, and requires the user to open up the local firewall for incoming connections.

The X Window System [31] enables an X client to send the display commands over a network to a remote X server. X can therefore be used to forward windows from the local desktop onto remote displays. However, X is not available on every operating system natively. For example, Microsoft Windows does not natively support the X Window System. There are other problems concerning the X Window System in this context [37], but the main problem is that X generally is not available on all operating systems.

The inSpace Projector [114] is a system that enables users to mirror their desktop to multiple displays over a network connection. Although one user can use several displays, several users cannot share the same display. In addition, there is no support for user-selectable regions, the user cannot interact with the desktop from the remote display, and the system's performance is not documented.

Virtually shared displays and input devices [115] are abstractions sharing characteristics with the NAD system. The idea is to use remote displays as extensions to the local displays over a network connection. However, while virtually shared displays allows for display sharing at the window level, the sharing is implemented using a VNC-based pixel-transfer system that needs to be installed on the computers in advance. In contrast, the NAD system uses customization to seamlessly integrate NACs and NADs. Consequently, for meeting room and presentation style contexts, the NAD systems can be used immediately by participants by clicking on a link in a browser, instead of manually obtaining and installing the necessary software to be able to use the external display(s).

The customization applied by the NAD system shares common characteristics with Universal Plug and Play (UPnP) [116]. UPnP is a collection of network protocols that enable different types of devices to communicate over a network (typically a user's home network). UPnP contains functionality for enabling devices to join networks, obtain IP address, convey capabilities, and learn about the present and capabilities of other devices [116]. While UPnP enable devices to share media content such as audio and video, it has to the author's knowledge not been used for sharing display content between computers and display devices. UPnP could potentially be used by future versions of the NAD system to enable seamless discovery of display devices from a NAC, instead of the current approach used where a user must know the address of the device in advance. However, customization still needs to be applied to integrate the NAC with the NAD.

6.1.2 ARCHITECTURE

The architecture of the system is based on a client-server model, where the client is the network accessible compute resource and the server the network accessible display (figure 6.1). Display walls are treated as one coherent display surface and accessible as a single network accessible display with a resolution equal to the total resolution of all display nodes comprising the display wall (figure 6.2). The choice of using a single front-end for NADs comprising multiple displays is twofold: (i) The front-end has central control over the shared resource, and can such easily control among others the z-ordering of the windows (the order on which remote windows are overlapping on the display wall); and (ii) from the NACs perspective the architecture remains the same independent of whether it is using a single display or a distributed display wall.

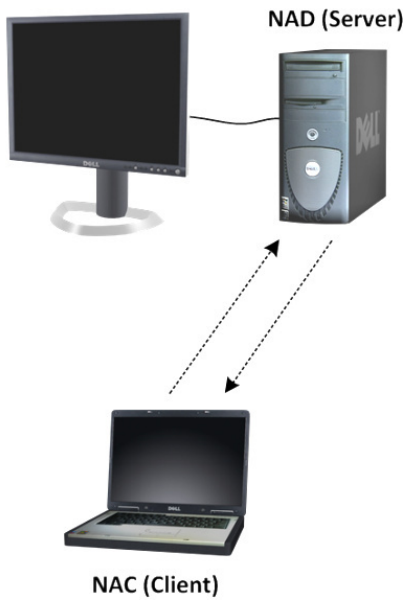

Figure 6.1: The NAD architecture for single display configurations.

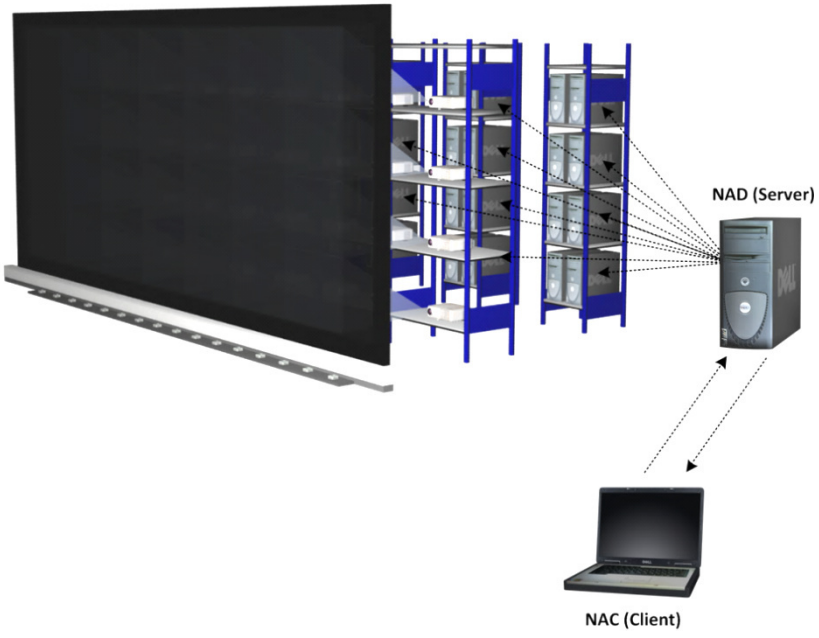

Figure 6.2: The NAD architecture for display wall configurations.

6.1.3 DESIGN

A NAC is customized by code downloaded from a NAD. This customization code needs to be transferred to the NAC once it contacts the NAD. To accomplish the integration of the NAC with the NAD, a two-phase protocol is used. In phase one, the NAC is customized by code downloaded from the NAD. In phase two, the integration phase, the whole display or user-selected regions of it are mirrored onto the NAD. The two phases are illustrated in figure 6.3.

The NAD runs a web server, which is the initial entry point for a NAC. A user wishing to use a NAD contacts it by accessing the web server through a web browser. A page is presented containing information about the NAD such as resolution and location, in addition to a button to launch the NAC software. When the user pushes the button, the software is downloaded to a temporary directory on the user's computer and is then launched.

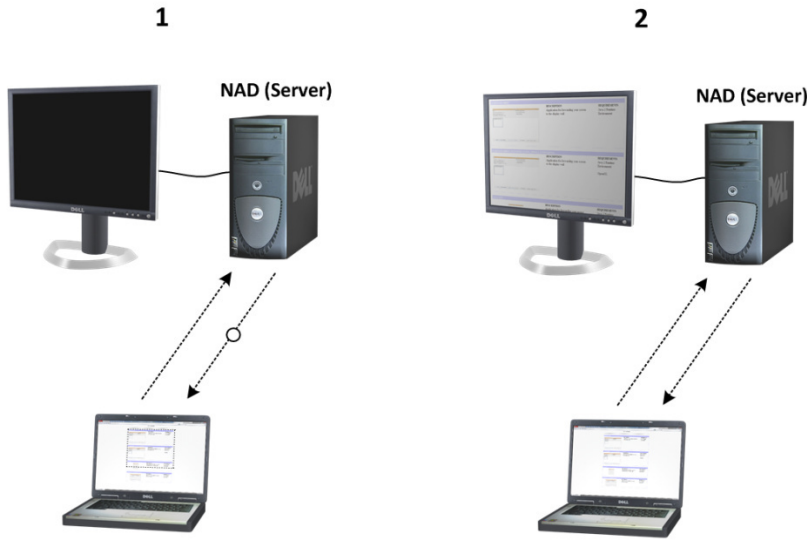

Figure 6.3: The two phases of the NAD protocol; (i) The NAC is customized by code downloaded from the NAD; and (ii) the customization enables mirroring of user-selectable regions onto the NAD.

The NAC software enables the user to select multiple regions (using the mouse cursor) of the desktop to be mirrored onto the NAD. These mirrored regions are read from the frame buffer of the local computer, encoded and transferred over to the NAD using a custom pixel transfer protocol. This protocol also allows for events to be transferred from the NAD back to the NAC for remote control.

A graphical user interface enables the user to scale and position the mirrored regions from the desktop computer. The GUI also contains a checkbox to accept remote events from the NAD. By selecting this checkbox, the user can enable

interaction with the mirrored regions from the NAD through connected interaction devices. These events are encoded and sent from the NAD to the NAC and inserted into the NAC's event queue. For display walls, the VNC system presented in chapter 2, section 2.2.1 is used for transferring the content from the front-end running the NAD system to the computers of the display wall.

6.1.4 Implementation

The core components of the NAD system are implemented in Java 2 Standard Edition [117]. Java was chosen because of the initial requirements to the system (cross-platform and transparent two-phase protocol). Java is cross-platform and contains a mechanism for launching applications from a web browser using the Java Web Start technology [118]. These properties were important for supporting the flexibility required for the system. The two main arguments against using Java are performance cost against machine code [119] (although for some platforms this gap is small [119]), and the fact that users must have the Java Runtime Environment (JRE) installed. Based on informal experience from using the NAD software on the display wall lab presented in chapter 2, section 2.3, every visiting user has had the JRE installed on their computer.

The integration phase is based on a protocol for transferring pixels from the NAC to the NAD and, optionally, for bringing events back to the NAC from user interaction at the NAD. The NAC software reads the frame buffer at regular intervals, encodes pixels, and sends them to the NAD. The protocol combines run-length encoding [120] with caching using an in memory copy of the previous frame, in order to reduce network traffic. The pixels have a 24-bit color-depth requiring 3 bytes per pixel. The NAC software iterates over the pixels captured from the frame buffer and encodes them to an encoding buffer using the following algorithm:

1. If one or more succeeding pixels exist in the cache, a SKIP header is added to the encoding buffer, describing the number of pixels to skip.

2. If two or more consecutive pixels have the same value, an RLE header is appended to the encoding buffer, describing the number of pixels followed by the color value (3 bytes).

3. If pixels do not satisfy the two preceding conditions, a RAW header is appended to the encoding buffer, describing the number of raw pixels followed by the color value of each proceeding pixel (3 bytes each).

Headers are 1 byte, where the two most significant bits are used to describe the type (SKIP, RLE, RAW or EXT). The remaining 6 bits are used as a pixel counter (figure 6.4). The EXT type is used to extend a header if the pixel count cannot be encoded using 6 bits by prepending it before the main header. Additionally, several EXT headers can be prepended to encode as much pixel data as needed.

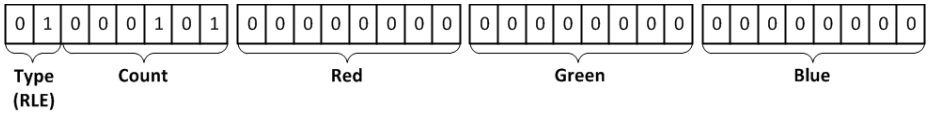

Figure 6.4: The NAD protocol format. The above example shows an RLE message containing 5 consecutive black pixels.

The encoded pixels along with other properties (mouse coordinates, scaling and position of the remote image) are sent over the network to the NAD. The NAD decodes the pixels and updates its mirror of the desktop. If the user has allowed remote control, the NAD propagates input events back to the NAC when events occur over the mirrored regions. The NAC receives the events, and inserts them into the local event queue of the window system.

Initial measurements on the NAC indicated that frame buffer capturing as well as performing the encoding protocol is CPU intensive. Since the NAC might be a battery-powered laptop, the frequency of these two operations needs to be kept to a minimum. To accommodate this situation the customization code of the NAC employs a frame-rate limitation mechanism. If the number of updated pixels in a frame is under a user-configurable threshold, the frame buffer capturing and change detection frequency is reduced. Otherwise, it is increased. This mechanism avoids unnecessary updates when the content rendered to the NAC frame buffer is mostly unchanged, for example when showing a slide show presentation. For regular usage, the lower frame rate of this mechanism is set to 5 FPS. The upper limit is set to 25 FPS by default. The upper and lower limit settings can be controlled from the graphical user interface.

6.1.5 Experiments

The developed system is evaluated by conducting 10 series of experiments. The frequency and size of graphical updates produced by an application affects the performance of the system. The content of the updated pixels affects the compression ratio. For these reasons, the experiment series were designed using four different applications with different graphical output characteristics.

Methodology

The four different types of applications used to evaluate the system were:

1. A slideshow presentation changing slides every fifth second.
2. Scrolling through a PDF document with a constant speed.
3. A video.

4. An application developed to produce predetermined output patterns to the frame buffer to give best- and worst-case update scenarios.

The best-case scenario is when updates are small, and the content of the updates are regular. The worst-case scenario is when the updates are big and the content of the updates are irregular. For each experiment the average frame rate, average bandwidth usage, CPU load, and time spent in the main parts of the system were measured. The factor for the experiments was the resolution of the mirrored region, which had the following values:

1. 800x600
2. 1024x768
3. 1600x1200

Table 6-1 summarizes all experiment series. The hardware and software used in the experiments are listed in table 6-2.

Table 6-1: Experiment summary

Experiment Series	Description	Factor (Resolution)
Frame rate limitation off		
1	Best-case application	800x600 (0.48 megapixels) 1024x768 (0.78 megapixels) 1600x1200 (1.92 megapixels)
2	Slideshow presentation	
3	PDF Scroll	
4	Video	
5	Worst-case application	
Frame rate limitation on		
6	Best-case application	800x600 (0.48 megapixels) 1024x768 (0.78 megapixels) 1600x1200 (1.92 megapixels)
7	Slideshow presentation	
8	PDF Scroll	
9	Video	
10	Worst-case application	

Table 6-2: Hardware- and software-platform

	NAC Side	NAD Side
Type	Laptop	Desktop
Number of Nodes	1	1
CPU	Intel Centrino Duo 2.16 GHz	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading
RAM	1 GB	2GB
Graphics Card	NVidia Quadro FX2500M w/512 MB VRAM	NVidia Quadro FX3400 w/256 MB VRAM
Operating System	Linux Ubuntu 7.04	Rocks Linux Cluster Distribution 4.0
Interconnect	Both computers interconnected over switched gigabit Ethernet	

Results and Discussion

Figure 6.5 shows the frame rate achieved for the first five experiment series (frame rate limitation off). The y-axis is the achieved frame rate and the x-axis is the number of megapixels that are mirrored.

As shown in figure 6.5 the frame rate is affected by the size and content of the mirrored region. The best-case application has a frame rate of almost 160 FPS for a region size of 0.48 megapixels, close to 100 FPS for 0.78 megapixels, and 50 FPS for the largest region size of 1.92 megapixels. For the worst-case application, the frame rate is 27.2 FPS for the lowest regions size of 0.48 megapixels, 16.77 FPS for the region size of 0.78 megapixels and 6.82 FPS for the largest region size of 1.92 megapixels. For the video, the resulting frame rates are 38.6 FPS, 27.9 FPS and 12.85 FPS. If a target frame rate of 25 FPS is acceptable, all data points except for the worst-case application's two largest region sizes and the video's largest region size fall within this refresh limit. The graph indicates an inversely proportional relationship between the resolution and the frame rate.

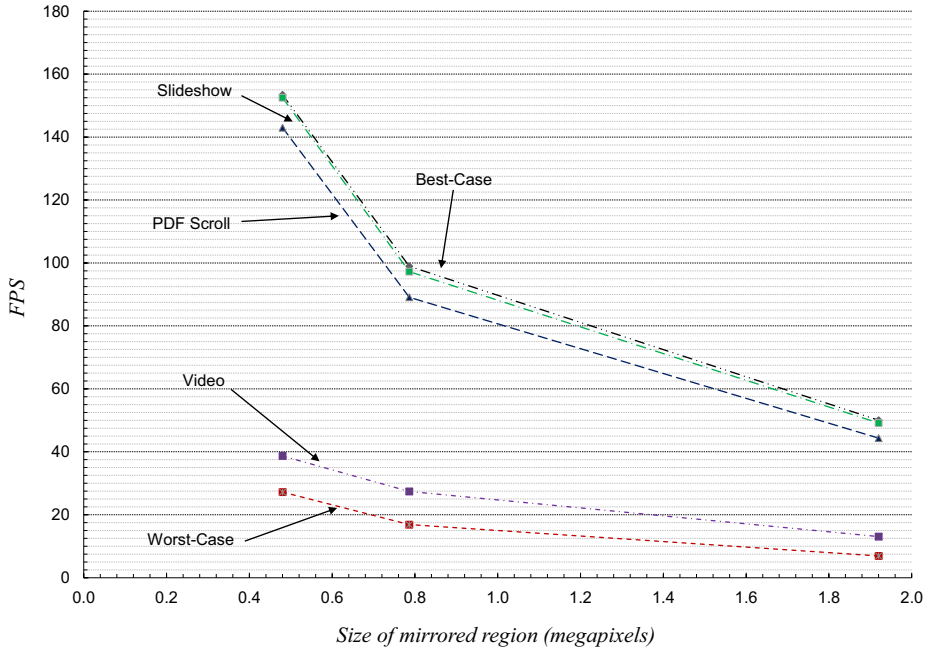

Figure 6.5: Frame rate of the different applications at the three resolutions used in the experiments (frame rate limitation off).

Figure 6.6 shows the time spent in the main components of the NAC software. The y-axis is the average number of milliseconds spent in the main functional tasks (communication, encoding and frame buffer capturing) and the x-axis shows the applications and resolutions used.

Figure 6.7 shows the average time spent in the main functional units of the NAD-side software (communication, decoding and rendering). The y-axis is the time in milliseconds and the x-axis shows the applications and resolutions used.

As figures 6.6 and 6.7 illustrate the iteration time (the time used to perform frame buffer capturing, encoding and communication for the NAC, and communication, decoding and rendering for NAD) for both the NAC and the NAD increases with the resolution, frequency and size of the graphical updates produced by the applications. All functional parts are executed sequentially and the total iteration time is therefore the sum of all these parts added together.

From both figures 6.6 and 6.7, it can be seen that the time spent on communication is not equal on both sides, which is an effect of the synchronous communication protocol and the measurement methodology. If the NAD is waiting for updates from the NAC, the waiting time is reflected in the communication part of the iteration time. Therefore, where there is a difference in communication time between the NAD and NAC, the real time spent sending data over the link is the lowest one.

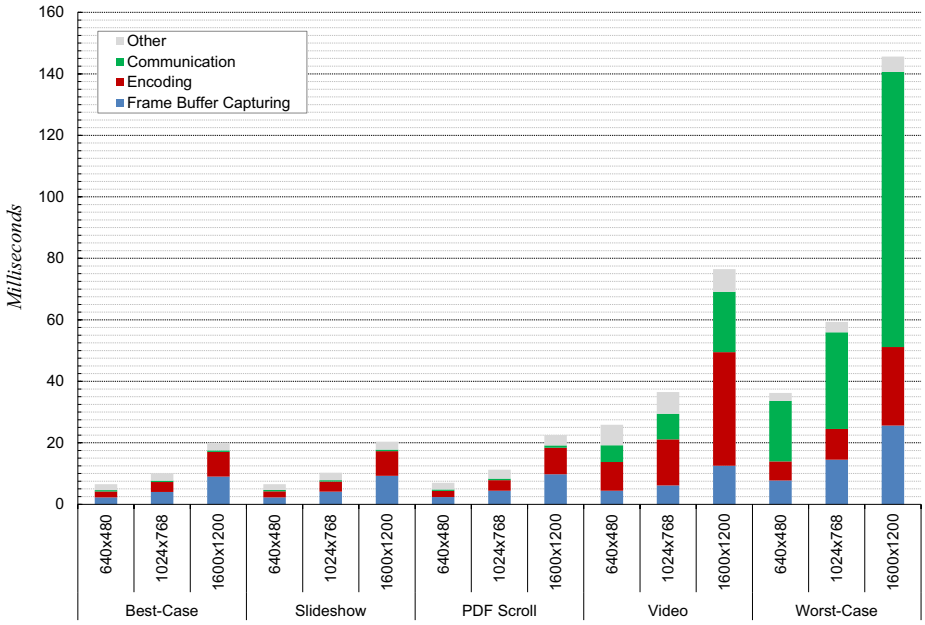

Figure 6.6: Breakdown of average time usage for the main functional units of the NAC (frame rate limitation off).

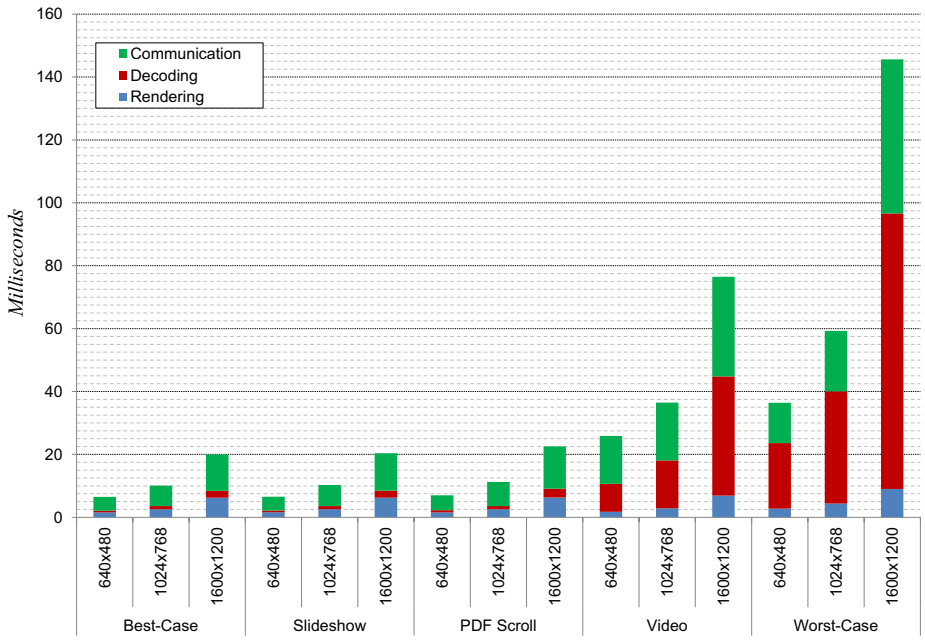

Figure 6.7: Breakdown of average time usage for the main functional units of the NAD.

Figure 6.8 shows the average bandwidth usage of each application. The y-axis is the number of megabits per second and the x-axis is the number of megapixels mirrored.

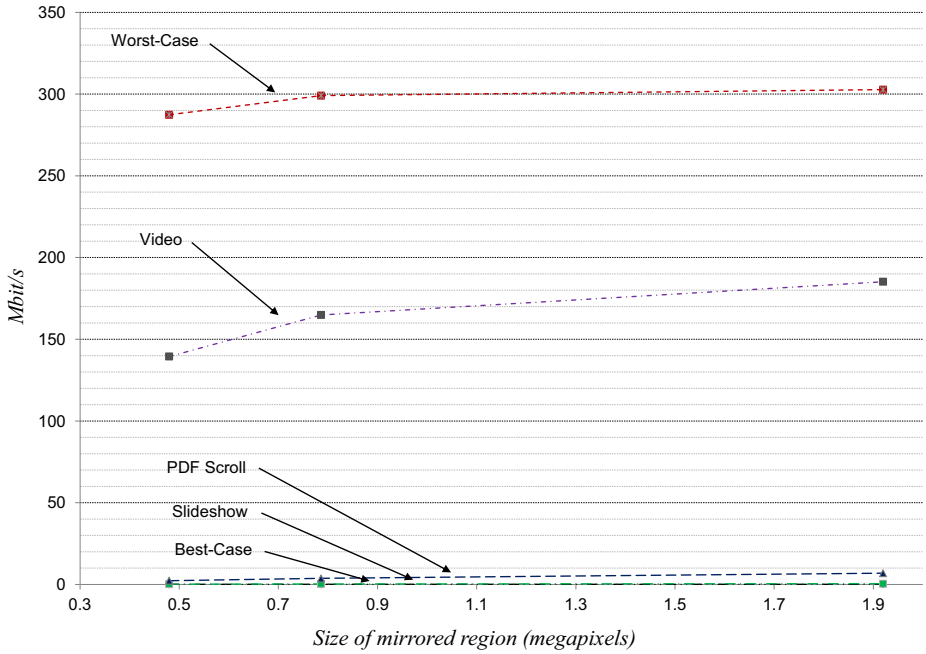

Figure 6.8: NAC – NAD network bandwidth usage.

Reducing the iteration time to increase the frame rate requires the time of at least one of the functional parts to be reduced. The frame buffer copying is hardware accelerated and uses DMA for memory transfer. The time of this part is mostly determined by the pixel read-back speed, which is constrained to the hardware used.

For pixel encoding/decoding and networking there is a tradeoff between the time used to encode/decode data and the network bandwidth available. The chosen run-length encoding/decoding algorithm requires only one pass over the captured frame buffer, which reduces CPU usage at the expense of network bandwidth usage. Using other compression algorithms might yield better compression ratios. However, for the chosen experiment the network is not a bottleneck, which can be seen by figure 6.8. The worst-case application uses one third of the available bandwidth, which shows that the network is not a bottleneck for increasing the frame rate using the current compression algorithm.

Since the functional units of the code utilize different hardware components (graphics card, CPU and network card), pipelining them could give increased

frame rate. A pipeline approach would potentially give an iteration time equal to the largest iteration time of the functional units. Generally, the longest part is the encoding/decoding part. For the video application for example, this part is half of the iteration time, which would yield a potential speedup of two when mirroring the output from this application. However, for the worst-case application the largest part is decoding at the NAD-side, which constitutes a much larger part, and would therefore not give the same improvement.

Figure 6.9 shows the effect of the frame rate limiter on CPU load. The y-axis is the CPU load in percent and the x-axis shows the applications and resolutions used.

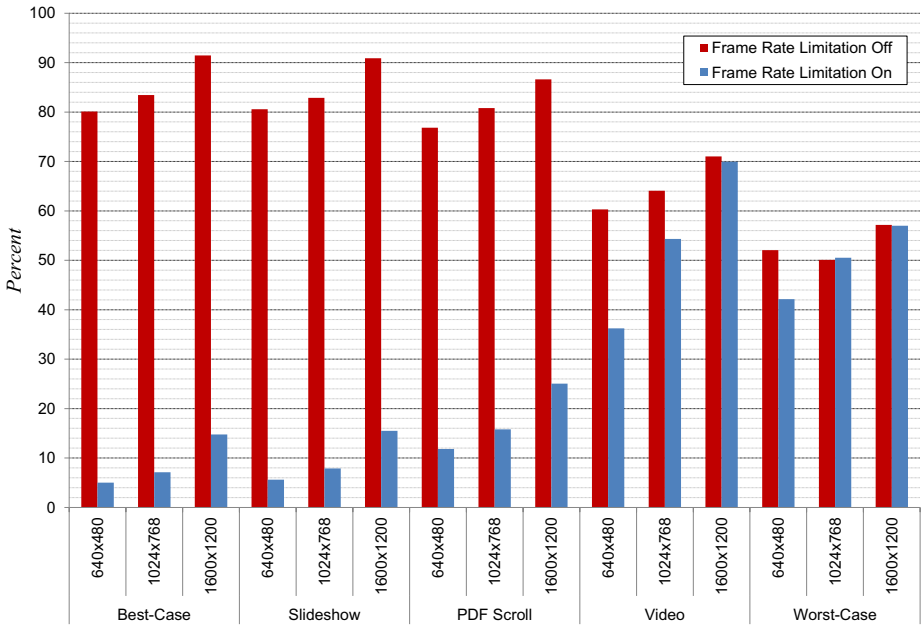

Figure 6.9: CPU usage on the NAC.

For several of the applications the frame rate limiter reduces the CPU load considerably. The applications that benefits this the most, is the applications that are not producing frequent updates (best-case, slideshow, and PDF scroll). These applications do not require frequent change detections, as the content shown is more or less static. For the video and the worst-case application, the frame rate limiter detects the frequent changes and therefore does not cap the rate of the frame buffer capturing and change detection. This also increases the CPU load on the NAC.

6.1.6 CONCLUSIONS

This chapter has presented the NAD system, a system for cross-platform desktop mirroring of user-selectable regions from one or several compute resources onto nearby network accessible projectors and displays. The system is based on a push-based NAC - NAD approach and the principle of establishing the end-to-end principle through customization. This enables a flexible use of nearby network accessible display resources, without requiring: (i) Separate usage of a DVI/VGA cable; (ii) permanent installation of third party software; and (iii) opening firewall ports on the local computer.

A user wishing to use a NAD can contact the NAD through a web browser, where he/she is presented with a button to launch the NAC software. By clicking on the button, a two-phase customization protocol is started. In phase one, the NAC is customized by non-intrusive software downloaded from the NAD. In phase two, the integration phase, the whole display or user-selected regions of it, is mirrored onto the NAD. Phase two supports propagation of events from the NAD back to the NAC, enabling remote control of mirrored regions.

At a resolution of 800 by 600 pixels, the system can mirror dynamic content (video) between a NAC and a NAD at 38.6 FPS. At 1600x1200 pixels, the refresh rate is 12.85 FPS. For static content such as images and slideshow presentations, the system's bandwidth usage is within the capacity of an 11 Mbit/s wireless network. For dynamic content such as movies and games, the system requires at least a 100 Mbit/s connection. The bottleneck of the system is frame buffer capturing and encoding/decoding of pixels. Pipelining the main parts of the NAC-side and NAD-side software would most likely give a frame rate increase in all cases, and by a factor of two in the best case.

CHAPTER 7

PULL-BASED NADS AND NACS

This chapter describes the pull-based NADs and NACs that have been created as part of the work presented in this dissertation. In contrast to push-based resources, where the NAC initiates and provides content to the NAD, pull-based NACs waits for requests from NADs, which are responsible for initiating the request. This chapter is based on the following peer-reviewed published papers: [76] [77] [78].

7.1 WALLSCOPE

WallScope is a system developed to document pull-based NADs and NACs, and several of the principles and models presented in this dissertation. It comprises several components that collectively enable high-performance interactive visualization of data on high-resolution tiled display walls.

WallScope adheres to:

1. **Principles:**
    a. Establishing the end-to-end principle through customization.
        i. Physical and virtual customization.
    b. PC – PCR duality.
    c. Domain specific best-effort synchronization.
2. **Models:**
    a. Network accessible compute model
3. **Architectures:**
    a. Live data set architecture

The overall idea of WallScope is to separate display resources from compute resources using a data set containing data customized for the particular application domain of the display resource's visualization systems. An illustration of the idea is shown in figure 7.1.

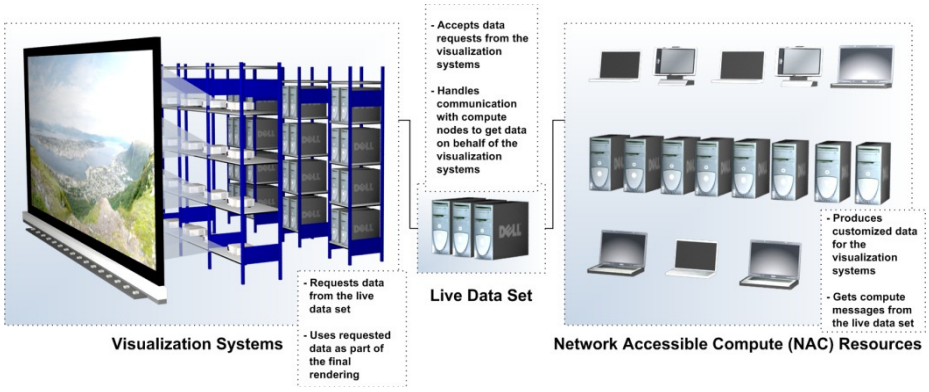

Figure 7.1: WallScope idea.

The implication of the separation of displaying and computing is twofold: (i) The computational power of the system is not constrained by the number of display nodes; and (ii) the display-side can access data processed in a customized format allowing a tradeoff between data size, display-side functionality and display-side utilization.

The visualization systems request data from the data set. The data set is live in that it translates these requests into compute messages and forwards them to available network accessible compute resources. The data produced by the compute resources is returned to the visualization systems for the final rendering.

Compute resources are categorized into static and dynamic. A static compute resource is a compute resource that is considered permanent to the system once added. Static compute resources are virtually customized by the live data set, meaning that the compute resources are unmodified and the live data set translates between the requests received from the visualization systems to the specific protocol used by the compute resources. Static compute resources are computational resources such as clusters and supercomputers. These compute resources typically have strict underlying security and access policies (software running on a supercomputer is often prohibited to make outgoing connections).

A dynamic compute resource is a compute resource that is volatile in the sense that it can register with the system on-the-fly to become a compute node in the system, and then at a later point leave. A dynamic compute node is customized by the live data set on-the-fly to produce data for the system. Examples of dynamic compute resources are laptops and desktop computers. A computer can become a

dynamic compute resource in the system by registering with the live data set to become customized. The customization enables the compute resource to provide information about the type of requests that it can process and what data it will share with the system. Once this information has been provided, the compute node waits for requests from the live data set.

The main components of WallScope are:

1. **Display-side components:**
    a. WallGlobe
        i. Visualization system supporting virtual globes.
    b. WallView
        i. Visualization system supporting high-resolution images.
    c. LDSView
        i. Visualization system combining the functionality of WallGlobe and WallView.
    d. The live data set
        i. The coordinator between the display-side and the compute-side.
2. **Compute-side components:**
    a. WallCompute
        i. System for computing images, elevation data and maps for the display-side.
    b. WallWeather
        i. System for computing weather forecasts and high-resolution weather forecast images for the display-side.
    c. Dynamic compute resources
        i. System for transforming personal computers into personal compute resources for the display-side.

7.1.1 Related Work

WallScope shares characteristics with several visualization and data processing systems.

Two such visualization systems are Google Maps and Google Earth. Google Maps [121] is a web mapping service application accessed through a web browser. Users can view street-maps, satellite images, traffic and much more. Other objects may be retrieved from various sources and rendered by the client as overlays. An extensive API [122] is available for third parties to create custom applications using the mapping services provided by Google.

Google Earth [123] is a stand-alone client for exploring the Earth in 3D. It enables a user to navigate anywhere in the world to explore satellite images of varying resolution mapped onto a virtual globe. Later versions include the ability to explore the universe and go underwater. There are three different versions of Google Earth: (i) A free version allowing users to "fly" anywhere on the Earth to view images, maps, etc.; (ii) a pro version for commercial use extending the free version with more functionality and better performance; and (iii) an enterprise version which allows corporations and government agencies to view their data on Google's globe database, or alternatively, host Google Earth internally on their own servers and data sets.

The server-side of Google Maps and Google Earth uses Google's BigTable [124] to store data. BigTable is a distributed storage system for managing structured data, described by the authors as a sparse, distributed, persistent multi-dimensional sorted map. BigTable uses the Google File System (GFS) [125] to store log and data-files. For Google Maps and Google Earth, one table is used to pre-process data, and a different set of tables are used for serving client data. The pre-processing pipeline uses MapReduce [126] over BigTable to transform data. The client service system uses one table to index data stored in GFS. This table requires low latencies, and is therefore hosted across hundreds of servers.

WallScope's main difference from Google Map and Google Earth is the use of on-demand computation of data. An implication of this on-demand computation model is that clients can get customized data from the compute-side. For example, compute servers can remove objects from images on-the-fly (remove oceans, show only rivers, show only mountains, etc.), or do re-projection of data fetched from external data sets not natively supported by the display-side's visualization clients. Google has pre-processed the data and can therefore not do this easily. In some cases, this is a drawback, as the server-side can have available processing power. Since WallScope clients can offload parts of the visualization process to the server-side, the architecture allows for load-balancing between clients and servers. Another advantage of the architecture is that clients may instruct the server to do speculative pre-computation of data by specifying a request script, thereby giving some of the same performance benefits as Google's pre-computed data sets.

In [127] is a blog showing images of Google Earth running on a tiled display wall. The author is using Chromium [33] to distribute the rendering primitives from one central computer running Google Earth, to a set of rendering nodes running on each computer connected to the display wall. The author does not document any performance numbers of the resulting system. The advantage of this solution is the fact that very small changes are needed to run OpenGL programs with Chromium. However, as mentioned in chapter 2, section 2.2.3 and documented in [27] [36], Chromium has scalability issues as the network can become a bottleneck.

Another project using Google Earth is HiPerWall [128]. In lack of the source code, their solution uses Google Earth's API to control one or more instances of Google Earth running on each display node. A controller node accepts input from the user and propagates this input to a set of controller nodes running on each display node. This solution differs from WallScope in all the aforementioned differences to Google Earth.

Liquid Galaxy [129] is a Google 20 percent project, which has evolved into becoming an official part of Google Earth. This project extends Google Earth to be set up for multiple displays. However, the system still lacks the on-demand computation model used by WallScope. In addition, the system only supports interaction methods predefined by the application, limiting the use of external input from alternative input devices such as the touch-free interaction system used by all of WallScope's visualization systems.

The Bing Maps Platform [130] is the next evolution of the Microsoft Virtual Earth platform. It consists of set of APIs that allows easy integration of map functionality into users' applications. The 3D part of Bing Maps [131], which allows the user to navigate the Earth to view satellite images mapped onto a virtual globe, requires installation of additional software, which is only available for Microsoft Windows based operating systems. There are to the author's knowledge not any projects using Bing Maps on tiled display walls.

World Wind [132] is an open source virtual globe originally developed by NASA. The original World Wind was implemented in C# [133] using Direct3D [67] for the rendering. Another version [134] has been implemented in Java [117] using OpenGL for the rendering. There are also other forks and clones of the original World Wind project [135] [136]. World Wind uses the Blue Marble [137] data sets for the lowest resolutions. Additional high-resolution data sets (among others Landsat7 [138]) are loaded as the user zooms in. There are to the author's knowledge not any projects using World Wind on tiled display walls.

ArcGis Engine [139] is a collection of components that can be used to create custom GIS solutions. In [140] is a description of a system using ArcGIS Engine to do parallel map rendering on a tiled display wall. This system uses a master node and six rendering nodes, all running ArcGIS Engine and connected to a back-end GIS database. The master shows the full scene of the data set and takes

control input from the user. The rendering nodes retrieve layer data from the back-end and viewing information form the master. However, this solution has no separate compute resources. The architecture of WallScope enables load-balancing between display nodes and compute nodes. Therefore, the processing power of WallScope is not limited to the number of display nodes. In addition, the ArcGIS components are only available for Windows.

In [43] a real-time terrain rendering system for tiled displays is presented. The system uses several techniques similar to WallGlobe. These are sort-first retained-mode parallel rendering, a Level-Of-Detail (LOD) algorithm for tiled rendering, view-frustum culling, and out-of-core data management. However, the data is pre-processed and replicated over the visualization nodes. Therefore, the system does not support on-demand computation of data from external data sources, and the computational power of the system is limited by the number of display nodes.

Active Data Repository (ADR) [141] is an object-oriented framework providing support for applications employing user-defined mapping and aggregation operations on large-scale multi-dimensional data sets. The ADR back-end comprises customized compute resources that communicate directly with the clients. This limits the system's computational power to compute resources that allows for custom code execution. Further, data sets are distributed over the local disks on the back-end compute cluster, which implies manual synchronization between remote data sets and back-end disks. Finally, the system does not store processed data, and the experiments only document performance results for output to 512x512 pixel images, which are orders of magnitude smaller than the resolution of a display wall.

DataCutter [142] is a framework designed to enable exploration and analysis of scientific data sets in distributed and heterogeneous environments. DataCutter uses a filter-stream programming model. Components of applications are decomposed into filters (potentially executed on different computers) that communicate via uni-directional stream pipes. The filters require an application binary that implements the filter specification on every host, in addition to an application daemon to start the filtering service. This limits the execution of filters to compute resources that can run custom code. Further, filters communicate through uni-directional pipes, which require the compute resource to open outgoing socket connections for bi-directional communication, an operation that is not allowed by some computational resources such as supercomputers. Finally, filters do not cache results, which could be inconvenient in virtual globes and mapping applications where queries can be repetitive.

In [143] the authors investigate multi-query workload optimization using active semantic caching. The active semantic cache comprises a transformation framework with operators that expose how a data product was generated and how it relates to other data products. This information allows for active transformation in case of partial reuse opportunities. WallScope does not use active semantic

caching because clients follow a strict tiling pattern, combined with the fact that navigating visualizations of the Earth often involves requesting data with increasing level of detail, yielding limited use of transformation operations.

Scalable Parallel Visual Networking (SPVN) [144] is an application framework for visualization of large data sets. The architecture comprises application servers that receive data from a database manager. Data is rendered to pixels and sent to display servers running on the computers driving the displays. In contrast to SPVN, WallScope uses both local and remote compute resources to provide display clients with domain specific data, which can be rendered to pixels using the clients' graphics hardware. In SPVN, this must be done by placing the application servers on the same machine as the display servers, which limits the computational power of the resulting system to the number of display nodes. Placing the application servers on dedicated clusters require sending pixels over the network, which for lossless encoding can introduce bottlenecks [32].

ParVox [145] is a parallel volume rendering system for supercomputers. It can render structured and unstructured grids into images, which are sent to a user's workstation. ParVox assumes that the user has a limited function workstation, that the user has access to a supercomputer, and that the data is located on the supercomputer's disk. This is not a requirement for WallScope. In contrast, the WallScope architecture comprises both local and remote compute- and data-resource that can be transparently utilized by the display-side. Further, WallScope caches data to avoid unnecessary re-computation. Finally, the domain specific data is not restricted to pixels (compressed or uncompressed), but can be any type of data customized for the display-side.

In [146] a grid-based visualization framework for spatial data sets is presented. The system assumes an active storage model where data sets are local to the CPUs. This is in contrast to WallScope that can utilize both local and remote data- and compute-resources. Further, the compute resources communicate directly with the clients. Finally, in contrast to the system presented, WallScope uses caching to avoid re-computation of processed data.

The Digital Light Table (DLT) [147] is a tool for visualizing large data sets. DLT comprises multiple graphics engines that read tiled images and digital elevation files from disk, and renders them as terrain. DLT is run on the Silicon Graphics (SGI) Reality Engine with multiple graphics pipes. A later version Multi-Surface Light Table (MSLT) [147] has been developed for commodity PCs. Both of these rendering systems use pre-computed images loaded from disk and have no separate data processing system to provide the visualization system with on-demand domain specific data.

The Remote Interactive Visualization and Analysis system (RIVA) [148] is a system for interactive data exploration and visualization of large data sets using a supercomputer. A low-resolution copy of the data set is loaded in a RIVA data navigator residing on an SGI workstation, where the user can navigate and select

desired views. The viewpoint is transmitted to the supercomputer via a network interface program, where an image is rendered using the full data set. In contrast to RIVA, WallScope renders the final image on the display-side, thereby utilizing both the compute- and graphics-capabilities of the display nodes. Further, the domain specific data is not rasterized from the current camera viewpoint, yielding better possibilities for tiling and caching. In addition, the domain specific data output is not restricted to be pixels. Finally, WallScope does not need a lower resolution sample of the original data set for out-of-core rendering.

The Scalable Adaptive Graphics Environment (SAGE) [66], based on TeraVision [149] and TeraScope [150], is a middleware for streaming pixels in real-time from remote rendering and storage clusters to high-resolution tiled display walls. SAGE requires each rendering application to be linked with a client-side library. This limits the computational power of the system to compute resources that can run custom code. Further, the middleware is based on streaming pixels over the network, which can require high-bandwidth links between rendering- and display-nodes. In addition, the display nodes receive and decompress pre-rendered pixel streams, thereby not utilizing their compute- and rendering-capabilities.

Tellurion [151] is a planetary visualizer with real-time data-manipulation capabilities. Using a GPU centric approach, the system is able to combine disparate planetary-scale data sets to produce a uniform composite visualization of them. The visualizer has been ported to several display walls. Although the visualizer has real-time capabilities for blending data sets with different projections, the data sets have to be downloaded to disk in advance. WallScope uses on-demand downloading and revalidation of data. The implication of this is that WallScope can download data from remote data sets once they are updated, for example to include a visualization of the latest images available from real-time image feeds. In addition, WallScope supports blending image data sets using layers with transparency, and can also choose where to blend images: At the display-side or at the compute-side.

OptiStore [152], based on TeraScope [150], is and on-demand data processing middleware for extremely large-scale visualization applications in the context of the OptiPuter [153]. OptiPuter is a research project where distributed storage, computational resources and visualization resources are connected using optical deterministic high-speed networks. The OptiStore framework uses several techniques and mechanisms to achieve overall utilization. These are load-balancing data partition and organization, multi-resolution analysis, view-dependent data selection, runtime data pre-processing and dedicated parallel data filtering. OptiStore is similar to WallScope in that both are using on-demand processing of data and separate data sets, compute resources and display rendering. However, there are some important differences between these two systems. OptiStore is based on an architecture where data is separated from compute resources and visualization systems initiates contact directly with the compute nodes. In addition, compute nodes are required to run the OptiStore

software, and visualization clients must be compiled and linked with a client-side library. WallScope, on the other hand, separates visualization systems from compute systems using a live data set containing data customized for the particular application domain of the visualization systems. WallScope does not put any architectural limits on the compute-side of the system nor requires any custom code to be running on the compute nodes. Thus, WallScope support both computational resources that can be installed code on, and computational resources that are available for data processing, but do not allow for execution of custom code. Additionally, the live data set allows applications to perform their own speculative pre-fetching, in contrast to OptiStore where pre-fetching is done by the servers from view-information passed from the clients.

In [154] the authors present OptiPuter middleware technologies supporting real-time collaborative visualizations of 3D multi-gigabyte Earth science data sets. However, the system does not have computational resource for separate data processing before data is sent to the visualization nodes, thereby reducing the computational power of the system to the number of visualization nodes.

The dynamic compute nodes in WallScope shares common characteristics with public (global) computing, where the idea is to use the world's computing power and disk space to create virtual supercomputers capable of solving problems and conduct research previously infeasible. There are a number of projects focusing on public computing, among others SETI@home [155], Predictor@home [156], Folding@home [157], and Climateprediction.net [158]. These projects use the BOINC (Berkley Open Infrastructure for Network Computing) [159] platform. The overall goal of BOINC is to make it easy for scientists to create and operate public-resource computing projects. A user wanting to participate in the BOINC project downloads and installs a BOINC client, which is used to communicate with the server-side. While there are similarities between the dynamic NACs and BOINC, there are some important differences. Firstly, in BOINC the focus is to make it easy to utilize available computational resources. For the dynamic compute nodes, the focus is to utilize desktop applications for domain specific computation of data for a set of visualization clients. Secondly, in BOINC, each client requests jobs from the servers that host the projects. This is in contrast to WallScope where the live data set sends compute messages to compute nodes on-demand. BOINC clients choose from what projects they will compute for based on some local policy, where in WallScope the live data set has central control and chooses which compute resource will get the compute message based on a global view. Further, the customization phase for dynamic compute nodes in WallScope requires no permanent installation of software on the compute node, and the code is completely removed from the compute node when the user quits the NAC service. In addition, users have complete control over what data is shared, and can choose this data from their personal computer on a per data-element basis, for example only page 1 and 3 of a 10 page document. This also includes complete control over the output format such as pixels, PDF, original source, etc. Finally, the live data set supports local and remote compute resource like clusters and

supercomputers, which are not supported by BOINC focusing exclusively on public computing.

Condor [160] focuses on distributing work tasks to idle workstations. However, the system is batch oriented and does therefore not support the on-demand computation model used by WallScope.

XtremWeb [161], another platform supporting global computing, uses a set of XtremWeb root servers to host binaries for the clients (workers). Workers register with a root server, get a vector of available servers that host tasks, and request jobs from these servers. Similar to WallScope, XtremeWeb has pre-compiled binaries hosted for the workers (XtremWeb) / compute resource (WallScope). However, in contrast to XtremWeb, which uses one compute thread per computer, WallScope's dynamic NAC-software supports several compute processes per computer. This has some important advantages: (i) The current trend is towards more CPU cores per computer; and (ii) some APIs are not thread safe, and thus running these in separate address spaces avoids race conditions. Further, the live data set sends compute requests to compute resource on-demand, instead of having workers requesting jobs. Finally, the use of a live data set, which has the ability to adapt to the protocol used by a compute resource, allows for both local and remote compute resources without requiring any custom code to be running on this type of compute resources.

XtremWeb-CH [162] is an upgraded version of XtremWeb. It comprises two versions called XWCH-sMs, which uses a centralized communication method, and XWCH-p2p, which allows nodes to communicate without central coordination. XWCH-sMs uses a coordinator similar to the live data set used in WallScope. However, this version only supports what is referred to as dynamic compute nodes in WallScope. In WallScope, the live data set adapts to the protocols used by compute resources that the system cannot install software on, and can therefore support supercomputers, grids and other compute systems that accept job requests, but are protected by strict security policies, such as the ability to establish outgoing connections.

In [163] and [164] job distribution middlewares for P2P environments is described sharing some of the characteristics as XWCH-p2p. While P2P allows for decentralized control and therefore removes some of the potential bottlenecks and components that can cause a single point of failure, a centralized scheduler allows for distributing tasks with a global view, which in WallScope is preferred over a decentralized model.

WallScope also shares characteristics with Minimum Intrusion Grid (MIG) [165]. However, while MIG focuses on making it easy for users to utilize available computational resources, WallScope focuses on domain-specific computation of data for a set of visualization clients running on resources ranging from laptops to high-resolution tiled display walls.

In addition to the data-processing and rendering systems already presented in this section, there exists several cluster-based parallel rendering systems for display walls. Several remote rendering systems exists in addition to the ones presented earlier in this section, among others Chromium Renderserver [166], OpenGL Vizserver [167], HP Remote Graphics Software [168] and ThinAnywhere [169]. These systems utilize the rendering capabilities of remote clusters, but not the graphics capabilities of the computers receiving the rendered pixels. There are several systems that can utilize the local rendering capabilities of a display wall. These include Chromium [33] (based on WireGL [65]), Equalizer [170], VR Juggler [171], ClusterJuggler [172], NetJuggler [173], Garuda [174], OpenGL Multipipe SDK [175], and CGLX [176]. In addition to these general frameworks, there has been conducted much research on visualization and parallel rendering on display walls among others in [177] [178] [179] [180] [181] [182] [183] [184] [185] [186] [187] [188] [189] [190] [191] [192] [193]. However, these systems mainly focus on distributing, parallelizing and synchronizing the data and rendering workload between the display nodes. WallScope extends this by utilizing a live data set to provide visualization systems with domain specific data from local and remote data- and compute-resources, transparent to the display-side of the system.

7.1.2 ARCHITECTURE

This section describes the architecture of WallScope. The architecture is a realization of the main idea behind WallScope: Separate compute resources and display resources using a data set containing data customized for the particular application domain of the display-side's visualization systems. The architecture of WallScope is shown in figure 7.2.

A visualization client runs on each display node (the NAD comprises the combination of a physical node and the visualization client). A separate state server provides each client with view state. By combining the view received from the view-state server with the position in the display grid, a visualization client requests the content needed to display its part of the view from the live data set. If previous requests for the same data have been performed, and the data is not outdated, the live data set returns a cached copy of the content for the request. Otherwise, a message for processed data is sent to a network accessible compute resource for domain specific data generation for the visualization client.

The display-side of the system can query the live data set to get meta-data. The meta-data describes all the data the network accessible compute resources can produce on behalf of the display-side. From the visualization nodes point of view, the live data set contains all the data pre-processed. However, the live data set will only actually contain data that has been processed, and all requests for data that has not been processed will be sent to a compute node that can produce this data. This is done transparently to the display-side, and thus hides all computations to the visualization clients.

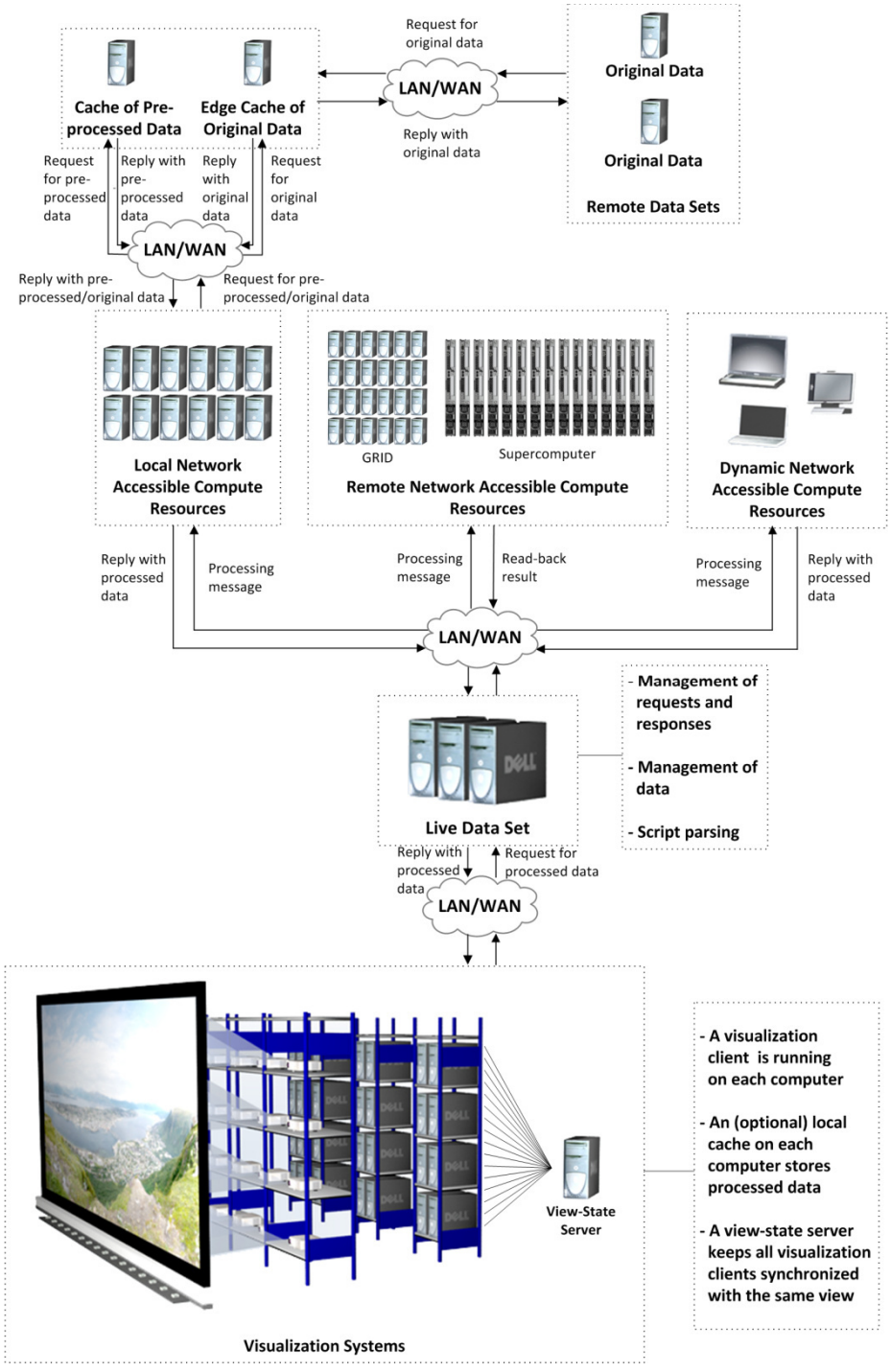

Figure 7.2: WallScope architecture.

As mentioned in the introduction, compute nodes are divided into static and dynamic. The static compute nodes are compute resources that are considered permanent to the system once added. A local compute cluster (WallCompute) is set up as the main provider of images, maps and elevation data. The compute nodes in the cluster uses data requested from a cache of pre-processed data and from an edge cache of original data. The edge cache is the compute resources entry point to the outside data sets. Each compute node in the local cluster has a local cache for the cache of pre-processed data and for the cache of original data.

A super computer (part of the remote compute resources) is running WallWeather's simulation for on-demand weather forecasts.

The dynamic compute nodes are compute resources that can be dynamically integrated with the system. This class comprises workstations, laptops, etc. These compute resources are dynamic in the sense that they can register on-the-fly with the live data set to become compute nodes in the system. The dynamic compute nodes can be used to provide resolution independent display-side-friendly formats for the visualization clients, for example to enable visualization of a large number of pages from a word document.

7.1.3 Design

Display-side Components

The display-side components of WallScope are the network accessible display resources and the live data set.

The network accessible display resources are responsible for performing the final visualization that is output to the physically connected display(s). The state of these visualizations is provided by a separate state server, which broadcasts this state at a configurable number of times per second. Based on this state and a visualization client's position in the display grid, a client requests the data needed to complete the visualization from the live data set.

Three display-side visualization systems have been developed. These are WallGlobe, WallView and LDSView. The visualization systems follow the same internal software design. The main internal software components of the visualization systems are the rendering engine, the request queue, and the request threads (shown in figure 7.3).

The rendering thread executes at a configurable number of iterations per second. For multi-resolution data sets, a tile-based representation is used to describe each data set. Each tile has a list of data that comprises that tile. This data is configured at the initial loading of the visualization system or discovered through queries to the live data set. For each new level of detail, tiles are divided into four. The result is a quad tree where the visible tiles can be calculated by iterating the tree.

The algorithm for calculating the visible parts of the quad tree is as follows [194]: Starting at the lowest level of detail, view-frustum clipping and back-face culling are performed. If the details of the layers comprising a tile is too low for the current camera view (for example the textures comprising the tile have too low resolution to be written to the frame buffer without being scaled), and the tile is inside the view-frustum and passes the back-face culling test, the tile is divided into four new sub-tiles and the same test is performed recursively. If a tile passes the view-frustum clipping and back-face culling, and have enough details for the layers it comprises, it is marked for rendering and no further sub-tiles of that tile are processed. A tile that does not pass the view-frustum clipping or the back-face culling is marked as not visible, and no more sub-tiles of that tile are processed.

When a tile is marked for rendering, the rendering thread checks if the data for the tile is available. If the data for the tile is not available, the rendering engine requests data for the tile through the request queue. Request threads fetch requests from this queue and invoke a request handler, which performs the actual request. Each request has an associated flag which tells the rendering thread when the data is available. When the request is completed, the request thread indicates this by setting this flag to true. The rendering thread checks the flag every iteration of the rendering loop (every frame). If it is set, the rendering thread loads the requested data into the selected layers of the tile. The number of request threads is configurable.

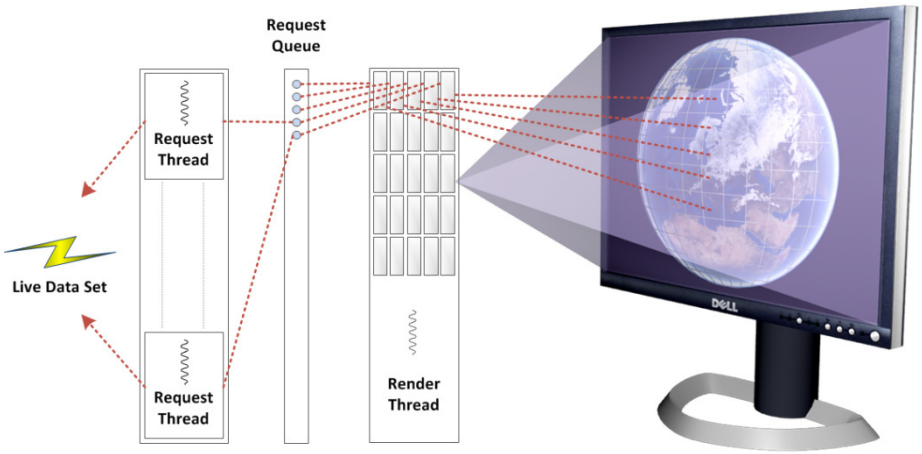

Figure 7.3: The main components of the visualization systems. The render thread requests data through a request queue. Request threads fetch data from this queue, and perform the actual request.

WallGlobe

WallGlobe is a system for visualizing planets. The data used by WallGlobe is requested from NACs through the live data set. Currently there are two different data types used by WallGlobe: (i) Elevation data (discrete values organized into a grid, describing the height-level in relation to the ocean), and image data (satellite and aerial photography).

The rendering engine supports alpha blending between layers, and can render layers at different height levels. This functionality is used to augment WallGlobe with functionality for weather forecast renderings. Clouds, wind and other data layers can be superimposed the regular satellite images at different height levels. This functionality is used to render data requested through the live data set from the WallWeather NACs.

WallView

WallView is a system for visualizing high-resolution images. The system supports both tiled and regular images. These are requested from the available NACs through the live data set. WallView follows the previously described design.

LDSView

LDSView is a system for visualizing the entire content of the live data set. It combines WallGlobe and WallView to create a system able to visualize virtual globes, gigapixel images, and other content produced for the display-side. LDSView requests meta-data from the live data set to learn its content. This meta-data includes a description of the resolutions of each of the data sets, and the elements they comprise.

Live Data Set

The live data set is the display-side's entry-point to the compute resources. It comprises data set elements that the visualization clients can request. A visualization client can query the live data set for the data set elements it comprises. An element in the live data set is not actually stored in the live data set unless it has recently been requested and cached. An element represents a computation.

The live data set comprises two main components:

1. The front-end cache.
2. The back-end logic.

These two components are running in separate address spaces as shown in figure 7.4.

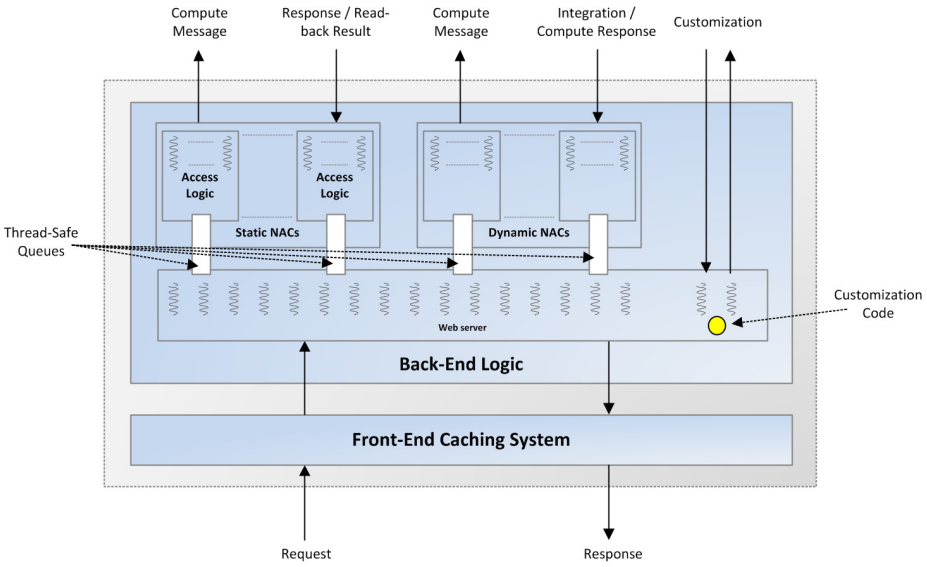

Figure 7.4: Live data set design.

When a request for an element is made, the front-end cache checks the request against the locally cached data. If the data for the request is cached and has not expired, the data is returned back to the requesting client. Otherwise, the request is forwarded to the back-end logic. The front-end cache can be configured to directly load-balance requests to a set of available back-end compute nodes without going through the back-end logic.

When a request is forwarded to the back-end logic, the request is inspected by a thread in a web server to find a compute resource that can produce the data for the requested element. The live data set contains a list of all connected NACs and the data they can produce. This list is searched to find a NAC that can fulfill the request. Once an available NAC is found, a request is posted to a queue from where a NAC handler responsible for that NAC can read, translate and perform the actual request.

For static NACs, each NAC is assigned a queue and a number of threads for the actual communication. Threads fetch requests from this queue and request content from the NAC. The maximum number of threads per NAC, and thus the maximum number of communication channels to the NAC, is pre-assigned. Therefore, the live data set will not violate a communication protocol for NACs that might have an upper bound for the number of simultaneous connections accepted. The threads communicating with the NAC is responsible for converting the unified request from the web server to a NAC-specific request, including performing the actual request over the NAC-specific communication protocol. The functionality for doing this is pre-configured for each thread. However, such

functionality could also be added using for example configuration files. Since this complexity is handled by the live data set, the visualization systems developed can have unified data access logic.

Dynamic NACs are customized by the live data set. The live data set contains customization code hosted through a web server. This code is downloaded to a dynamic NAC once it connects. The customization code enables the production of customized data on the NAC, and integrates the NAC with the live data set. After the integration, the NAC is ready to accept requests from which it can produce customized data based on its installed software environment. This is achieved by hosting a set of plugins on the live data set that is transferred to the NAC in the customization process. As part of the integration process, the NAC validates each plugin and reports to the live data set which plugins are supported. In addition, a dynamic NAC reports to the live data set what data it will share with the system. This data can be a subset of data found on the NAC, including subsets of files. This assures that the NAC can share data at a fine-grained level, at the same time not touching files that the user wants to keep private. Dynamic NACs are handled by a server executing in a separate thread. This server accepts incoming connections from dynamic NACs. Once a connection is made, the server handles the initial handshake where parameters such as number of connections to the NAC, the data shared with the system, and compute capabilities are negotiated. This information is written to a list describing all NACs connected to the live data set. When this information has been updated, the server accepts input from the number of connections negotiated, and new threads are created from each new connection made. All these threads pick requests from a shared queue. If a request for an element residing on a dynamic NAC is made to the live data set, the request is put on the NAC's queue, and one of the threads waiting on the queue picks it up and performs the actual request to the dynamic NAC, at which point the thread handling the request from the client is suspended. Once data is received, the location for the data received is passed to the client thread handler, which in turn returns the content from the request to the client.

Compute-side Components

The compute-side components of WallScope are the static and dynamic network accessible compute resources.

The static network accessible compute resources are compute resources that are permanently available to the display-side. No code from the display-side is physically installed on these compute resources. Instead, the customization for the display-side is done virtually in the live data set based on the compute resources' access protocols. Compute resources in this category are clusters, grids and supercomputers.

Two compute systems have been developed as part of the static NACs. These are WallCompute and WallWeather.

WallCompute

WallCompute is the main provider of image data in the system. These images range from regular resolution images to high-resolution gigapixel images. In addition, WallCompute can compute maps from vector data, mask satellite images from vector data, and provide elevation data. WallCompute comprises three main components:

1. Caches of pre-processed data.
2. Caches of original data.
3. A compute system computing processed data using data from the aforementioned caches.

The interplay between the main components is shown in figure 7.2. A single edge cache of original data is used for the following reasons:

1. To make the system appear as one single entity to the outside world.
2. To limit the number of outgoing connections to prevent external servers from being overloaded or banning WallScope due to excessive resource usage.
3. To authenticate and keep track of external session state required by external servers.

WallWeather

WallWeather is the system used for weather forecasting in WallScope. This system can create high-resolution weather forecast images from user-selected regions on-demand. The system is comprised of two different types of NACs. The first type of NAC is responsible for doing the main weather forecast simulation for user-selected regions. The results are used by another type of NAC, which processes the simulation data and creates images at varying resolutions for the visualization systems. WallGlobe is integrated with WallWeather to enable on-demand weather forecasts (such as wind-arrows and clouds) projected onto the Earth.

Dynamic Compute Resources

A user wanting to register a computer as a dynamic compute resource can do so by contacting the live data set through a web browser. A web page is presented containing a button for launching the NAC-side software. By pressing the button, a two-phase customization phase is started. In the first phase, customization code is downloaded to the computer and launched. In the second phase, the customization code enables usage of the computer as a network accessible compute resource. The second phase also includes plugin validation, where all

plugins developed for the system are validated against the software install on the local computer. The plugins developed for the system are available for different computational resources, operating system platforms and installed software in general. Plugins are run in a separate address space to utilize multiple CPU cores and support non-thread-safe APIs. Based on the type of plugins the compute node supports, a list of supported data types is generated and sent to the live data set, which then stores the compute node and associated data types for future requests from visualization clients. The last step of the second phase is user-selection of data that will be shared with the system. The user selection of data happens at a sub-file level, meaning a user can select parts of a file for sharing, leaving the rest of the file untouched. For example, a user can select only certain slides of a presentation for sharing with the visualization systems, leaving the rest of the presentation private.

7.1.4 Implementation

Display-side Components

As detailed in the design section, visualization clients use a single thread for rendering and a variable number of request threads for communication with the live data set. A single rendering thread avoids race conditions with graphics libraries such as OpenGL.

The state server that provides the visualization systems with view state is implemented in C++. This state server receives events from the Shout event system (described in chapter 2, section 2.3.2), translates these events in to view state and broadcasts this state to the visualization systems. The sources for the events are different interaction systems (such as the device-free multi-touch interaction system presented in chapter 2, section 2.3).

WallGlobe

WallGlobe has been implemented in two different versions. One version is implemented in Java. Lessons learned from this implementation were used to create a new version in C++. Both implementations follow the same internal software design. The current versions of WallGlobe have 10x5 tiles for the first level of detail. Every planet has a description of the layers that comprises the planet, and at which level of detail these layers are active. Information about layers can be loaded from files or directly specified in the source code. At the lowest level of detail, each tile is 36 degrees in latitude and longitude.

WallView

WallView is implemented in C++. It can render images ranging from regular images (PNG, JPG, BMP), to high-resolution gigapixel images. Images are subject to the same view-frustum clipping algorithm as described in the design section. For regular images, the bounding box of the image is used for view-

frustum clipping. For gigapixel images, the image is divided into several tiles. Each of these tiles uses the quad-tree algorithm detailed in the design section.

LDSView

LDSView is based on WallView and the C++ version of WallGlobe. It uses the same software design, with a single rendering thread, a request queue, and multiple request threads for applying the rendering thread with data from the live data set. LDSView also includes a data set element discovery thread, which is used to provide LDSView with updates from the live data set. This element discovery thread supplies the rendering engine with the state of the live data set. This state comprises the elements that are present in the live data set. Based on the state of the live data set (all the data sets available), the rendering engine requests elements that are visible in the current view.

Live Data Set

As mentioned in the design section, the live data set comprises a front-end caching system and a back-end system for communication with the compute resources. The front-end cache is implemented using Squid [195], an existing caching system. Squid was chosen for several reasons: (i) It is open-source; (ii) it is cross-platform; and (iii) several Squid servers can be configured to cooperate in various cache hierarchies.

A request to the live data set is first handled by Squid to check if the element has been recently cached. If it has, Squid handles the request and responds with the cached element. If the request is not in the cache, or has expired, the request is forwarded to the back-end logic.

The back-end logic is implemented in C++. This logic handles the communication with the NACs, and the input from the cache system. The logic comprises a lightweight web server, and a number of threads, which handles the actual communication with the NACs as described in the design section.

Compute-side Components

WallCompute

WallCompute comprises a cache of pre-processed data, a cache of original data, and a compute system for computing processed data from the aforementioned caches.

The cache of pre-processed data is realized using memory-mapping and the Network File System [196]. The cache of original data is realized using the Squid caching system in combination with a Java-based system used for accessing remote data sets requiring authentication. A local version of the caches can be run on each compute node if a single instance of each cache becomes a bottleneck in

the system. However, for data requested from remote data sets, the caches still need to go through the central edge cache of original data to avoid overloading the remote data sets.

The mapping system of WallCompute is implemented in Common Lisp[9] [197]. The system providing images (both regular images and gigapixel images) and elevation data is implemented in C++. Each system is executed as a separate process on every compute node dedicated for WallCompute. Both systems accept HTTP requests from the live data set. The mapping system of WallCompute memory-maps the cache of pre-processed data over NFS when initially started. This cache cannot be updated when the system is executing. The Squid cache is accessed over HTTP by both systems. Elements from this cache can be updated when the system is running (for example as a consequence of elements being updated at the remote data sets). WallCompute requests elevation data, images, and vector data from the two caches, and uses this data to generate processed data for the visualization systems.

WallWeather

WallWeather comprises two different types of NACs. The first type is responsible for running the weather simulation. This type executes an implementation of the Weather Research and Forecasting (WRF) model [198]. Two different NACs implement this type: (i) A supercomputer [199]; and (ii) a cluster. These two NACs can generate weather simulations for user-selected regions, on-demand. The second NAC type is responsible for generating images of the results produced by the first NAC type. These images are used by the display-side's visualization systems. (Further details about WallWeather can be found in appendix A, section A.6).

Dynamic Compute Resources

The dynamic compute resources are implemented in Java 2 Standard Edition. Java was chosen to enable transparent code transfer from the live data set to the dynamic compute nodes and for cross-platform NAC software compatibility.

The first phase of the customization protocol is handled by Java Web Start. The live data set contains a Java JAR file that contains the code needed for a dynamic compute resource to communicate with the live data set, as well as all the plugins developed for the system. A user initiates the customization of a compute node by clicking on a link in a browser. Once the code (the JAR file) is downloaded and started, the plugin validation is started. The JAR file includes a set of executable files (plugins) that can be used to compute data for processing by the NAC software. The plugin validation phase executes these plugins to find the ones compatible for the current operating system and install. Some of the plugins

[9] The Mapping part of WallCompute was implemented by Espen Skjelnes Johnsen.

implemented are plugins to compute processed data from DOC, DOCX, XLS, XLSX, PPT, PPTX, PDF and various 3D formats. These plugins utilize the desktop applications already installed on the computer, for example using Microsoft Component Object Model (COM) [200] to orchestrate a document conversion to a format that can be processed by the NAC software. The processed data ranges from image-tiles and PDFs to 3D models that the visualization clients can load. Since the dynamic resources initiate contact with the live data set, the compute resources are available to the system even though they might be behind Network Address Translation (NAT) or a firewall.

7.1.5 EXPERIMENTS

To evaluate WallScope, 15 series of experiments were designed and conducted. The first 11 experiment series were conducted to evaluate the static NACs of the system and the cache system. The last 4 experiments were conducted to evaluate the dynamic NACs in the system, also including the cache system.

METHODOLOGY

For experiment series 1 to 3, a custom Java application was developed. The Java application was designed to perform 900 (512x512-pixel) image requests. This corresponds to 236 megapixels, in total 31.84 megabytes of JPG image data. In addition, the application was designed to enable/disable decoding of the requested images. Each image-tile had the Landsat [138] data set as the base layer with the ocean masked with data from the Blue Marble [137] data set. The original data sets that comprised the processed images were pre-fetched to the edge-cache of original data. The vector data used in the masking of the ocean was replicated from the cache of pre-processed data to every compute node.

For the first experiment series, the purpose was to measure the effect of caching in the system. The custom Java application described in the previous paragraph was used, and the time to perform the 900 image requests was measured. The factor was whether images were cached in the live data set or if they had to be computed by a single static compute node.

For the second series of experiments, the time used to request the same 900 image-tiles from a local cache residing on the same computer was measured. The factor for the experiment series was whether images were decoded or not. The purpose was to determine the time used to decode data at a display node versus the time used to only request this data from the local cache.

The third series of experiments measured the speedup when adding compute nodes to the system. The same 900 image requests were performed (caching disabled). The factor was the number of compute nodes used to produce processed data. This factor was varied from 1 to 26 with a step of 2. The purpose was to document the scalability of the live data set and the static compute

resources. (Load-balancing was done in the front-end cache system, bypassing the live data set back-end logic). All requests were divided between the compute nodes in a round-robin fashion.

Experiment series 4 to 11 (inclusive) used the WallGlobe visualization system (the Java version) and a camera trace to measure WallScope's performance under different configurations and loads. The trace consisted of a set of waypoints. A waypoint is given by the position and rotation of the camera. The camera's position and rotation was then interpolated over a spline curve calculated between the waypoints. The factor for each experiment series was the time used to interpolate between the waypoints. The trace started with the entire Earth visible in the view frustum, and then zoomed into a specific part of the Earth. All waypoints were picked in such a way that the camera only visited new tiles of the Earth. The time spent interpolating between waypoints was 30, 25, 20, 15, and 10 to 1 second. The state server was configured to multicast the global state derived from the camera trace to each of the WallGlobe clients 50 times per second (matching the refresh rate of the projectors). The total number of requests generated by the trace was 8585. 5111 (59.53%) of these requests were requests for images and 3474 (40.47%) were requests for elevation data. All the image-tiles had a resolution of 512x512 pixels. This corresponds to approximately 1340 megapixels of image updates. The different cache- and compute-configurations used for these experiments are listed in table 7-1.

Table 7-1: Configurations for experiment series 4 to 11

Experiment Series	Cache (Local)	Cache (Live Data Set)	Compute Nodes
4	Disabled	Disabled	1
5	Disabled	Disabled	2
6	Disabled	Disabled	4
7	Disabled	Disabled	8
8	Disabled	Disabled	16
9	Disabled	Disabled	26
10	Disabled	Containing All	0
11	Containing All	Disabled	0

Experiment series 12 to 15 (inclusive) measured the time used to load and simultaneously display a 350-page PDF document with the purpose of documenting the performance of the dynamic compute nodes and the cache

system. The 350-page PDF document was rasterized into image-tiles on the compute-side. Each tile had a size of 512x512 pixels and every page of the document comprised six such tiles. This yields a total resolution of 550 megapixels for the 350 pages. The load-time is the time from the first request is performed from a display node until all requests have been completed by all display nodes. The total number of requests generated for all experiments was 2432. This number is larger than the number of tiles that comprised the document, and is caused by some of the image-tiles overlapping between displays and thus are requested at least 2 times. Image-tiles overlapping between display corners are requested by 4 display nodes.

For experiment series 12 and 13, the factor was the number of compute nodes used (caching disabled). This number was 1, 2, 4, 8, 16 and 28. For experiment series 12, PNG was used as the image-tile format. Experiment series 13 used JPG as the image format. Each of the compute nodes had 4 compute processes running to utilize all the cores (not including HyperThreading). Every display node was configured to perform 4 simultaneous requests to the live data set (allowing the display nodes to utilize all the 28x4 cores of the compute-side). The requests were load-balanced on the available compute nodes by the live data set. The purpose of these two experiment series was to document the speedup when adding dynamic compute nodes to the system.

For experiment series 14 and 15, the time used to request the PNG/JPG image-tiles from the caches was measured. The factor was the location of the cache from where the image-tiles were requested from (live data set or local). The purpose of these experiment series was to document potential bottlenecks in the cache system and the network bandwidth between the live data set and display nodes.

Table 7-2 summarizes all experiment series. The hardware platform for experiment series 1 to 11 is listed in table 7-3. The hardware platform for experiment series 12 to 15 is listed in table 7-4.

Table 7-2: Experiment summary

Experiment Series	Description	Factor
900 image requests from a single display node using a custom Java application		
1	Requesting from two different locations	Location (live data set's cache or from a single compute node)
2	Requesting from local cache	Decoding (on/off)
3	Requesting from the compute nodes	Number of compute nodes used (1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24 and 26)

Playback of a WallGlobe camera trace on 28 display nodes		
4	1 compute node	Time between camera trace waypoints (30, 25, 20, 15, and 10 to 1 seconds)
5	2 compute nodes	
6	4 compute nodes	
7	8 compute nodes	
8	16 compute nodes	
9	26 compute nodes	
10	All data available in the live data set's cache	
11	All data available in the local cache	
Load-time of a 350-page PDF document rasterized into 550 megapixels of image-tiles		
12	PNG image-tiles	Number of compute nodes used (1, 2, 4, 8, 16 and 28)
13	JPG image-tiles	
14	PNG image-tiles	Location of the cache (live data set or local)
15	JPG image-tiles	

Table 7-3: Hardware- and software-platform for experiment series 1 to 11

	NAC-Side	NAD-Side
Type	Compute Cluster	Display Cluster
Number of Nodes	26	28
CPU	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading
RAM	2 GB	2GB

Graphics Card	-	NVidia Quadro FX3400 w/256 MB VRAM
Operating System	Rocks Linux Cluster Distribution 4.0 (64-bit)	Rocks Linux Cluster Distribution 4.0 (32-bit)
Interconnect	Display cluster nodes were connected to a front-end over switched gigabit Ethernet. Compute cluster nodes were connected to another front-end over switched gigabit Ethernet. Front-ends were connected over switched gigabit Ethernet.	

Table 7-4: Hardware- and software-platform for experiment series 12 to 15

	NAC-Side	NAD-Side
Type	Cluster of desktop computers	Display Cluster
Number of Nodes	28	28
CPU	Intel Xeon Processor E5520 8 MB Cache 2.26 Ghz, 4 Cores w/HyperThreading	Intel Pentium 4 EM64T 3.2 GHz w/HyperThreading
RAM	2.5 GB	2GB
Graphics Card	NVidia Quadro FX 580 w/512 MB VRAM	NVidia Quadro FX3400 w/256 MB VRAM
Operating System	CentOS release 5.5 (32-bit)	Rocks Linux Cluster Distribution 4.0 (32-bit)
Interconnect	Display cluster nodes were connected to a front-end over switched gigabit Ethernet. Desktop computers were group wise connected to gigabit switches (6 compute nodes per group). These switches were connected to a gigabit switch connected to a router providing the link to the display cluster front-end	

The specification for the computer running the state server and the live data set was the same as for the NAD-side display cluster nodes.

Results and Discussion

Table 7-5 shows the measured times for experiment series 1 and 2. The first column gives the location of the requested data. The second column shows the time used to complete all the 900 requests. The third column shows the mean time per request. Decoding of JPG images on the display node was measured to be CPU-bound.

Table 7-5: Time used to request 900 512x512-pixel (236 megapixels) image-tiles (experiment series 1 and 2)

Data Location	Total Time Used	Mean Time Per Request
Compute Node	296.238 s	329.2 ms
Live data set	1.411 s	1.6 ms
Local cache	0.716 s	0.8 ms
Local cache (with decoding)	13.850 s	15.4 ms

From table 7-5 it can be seen that the time to request data from the local cache is twice as fast as requesting data from the live data set. However, the time used to request and decode images on one node is 13.85 seconds. This is over 9 times slower than just requesting the images from the live data set, and therefore contributes to a much larger part of the overall time, compared to local versus central storage. Requesting data stored in the live data set is two orders of magnitude faster than computing the data on one compute node. For each processed image-tile, the compute node must request the original images tiles that comprise the processed image-tile. This requires data from at least on image-tile from the Landsat data set and one image-tile from the Blue Marble data set. Further, the compute node must create a raster surface matching the specification from the client request (512x512 pixels) and fill the raster surface with data from both data sets based on the meta-information from the vector shape data. The results show the importance of caching in WallScope to reduce the latency for data that is frequently used.

Figure 7.5 shows the result of experiment series 3. In the figure, the actual speedup is plotted against a linear (1:1) speedup.

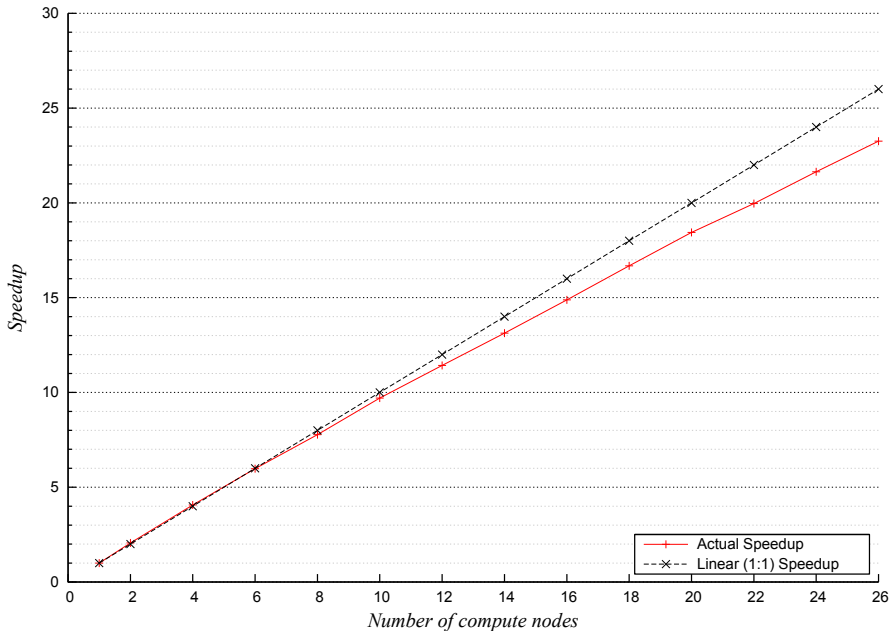

Figure 7.5: Speedup when going from 1 to 26 compute nodes.

From 1 to 6 compute nodes, the system has a near linear speedup. However, when the number of nodes increases beyond 6, the speedup is slightly reduced. This is caused by the round-robin work distribution. Because each data request has varying amounts of ocean to mask, the workload for each work request varies. Work is handed out in a round-robin fashion without any feedback mechanism. Therefore, some of the nodes will get more work than others, which is why the performance does not increase linearly with the number of compute nodes added.

During experiment series 4 to 11 all WallGlobe clients were measured to have the same frame rate as the refresh rate between the computers and the displays. This shows that the sort-first rendering approach use by running a separate client on each computer is sufficient for driving the 22-megapixel display wall.

Figure 7.6, 7.7, 7.8 and 7.9 show the result of experiment series 4 to 11. Figure 7.6 shows the displayed requests. A “displayed request” is a request that was loaded in memory and displayed at least one frame during the experiment. The y-axis is the number of requests that were displayed during the experiment. The x-axis is the seconds used between waypoints. The upper bold line is the total number of requests generated by the trace. The shaded are on the figure is where the expected performance of the system would be using all compute nodes with caching enabled. Different cache replacement policies will result in a system performance that will fall within this area. The data-points marked by an A in the upper right corner are shown in detail in figure 7.8.

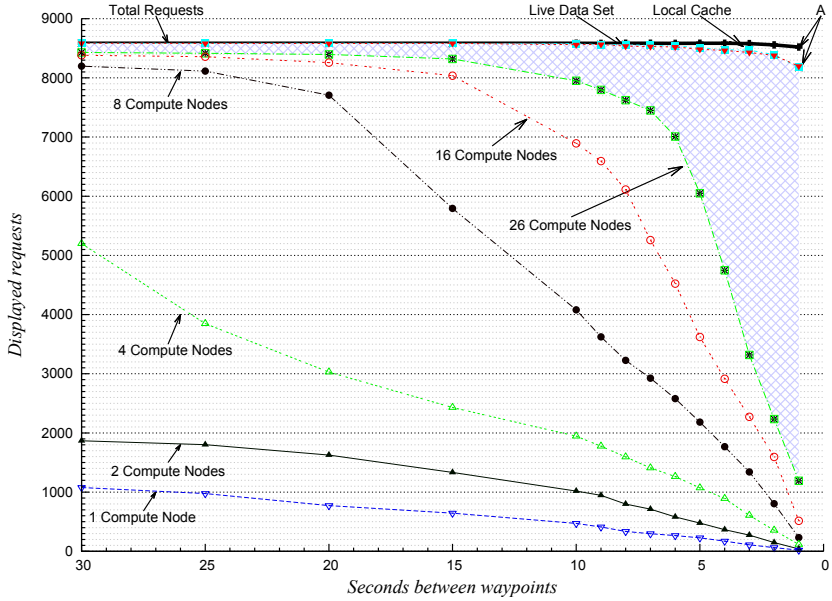

Figure 7.6: The total number of displayed requests (the number of requests that were loaded into memory and contributed to at least one frame). The shaded area marks the expected performance of the system using all compute nodes with caching enabled.

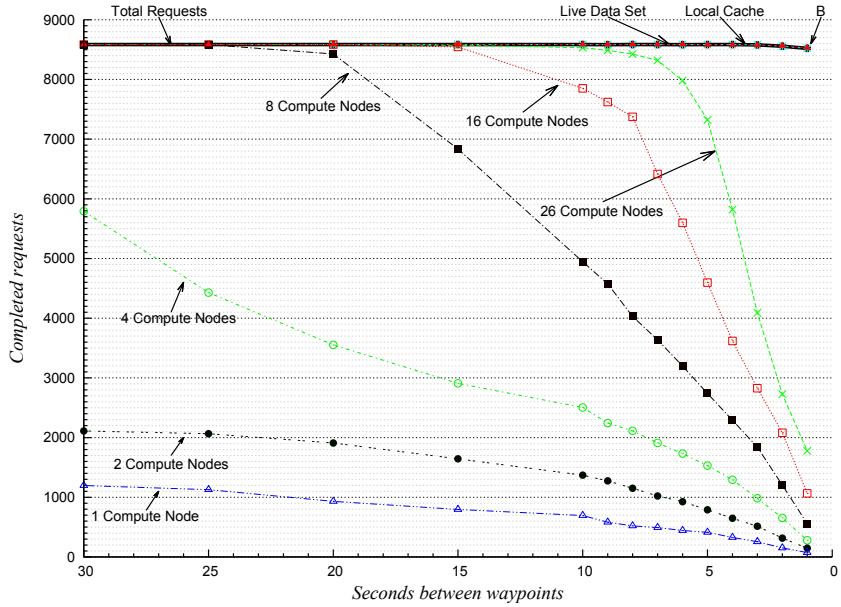

Figure 7.7: The total number of completed requests (the displayed and non-displayed requests). This graph does not include the shaded area shown in figure 7.6 because a completed request does not imply a request that was actually displayed.

Figure 7.7 shows the number of completed requests. This is the sum of the displayed requests and the requests that were downloaded and decoded but not displayed because the Earth's source tile for the request was outside the view-frustum at the time the data was available. This figure does not include the shaded area used in figure 7.6 because a completed request does not imply a request that was actually displayed during the experiment. The data-point marked by a B in the upper right corner is shown in detail in figures 7.8 and 7.9.

Figure 7.8 shows the requested, completed and displayed requests for the full local cache configuration using one second between waypoints. This graph was included to show a detailed trace for the configuration with the best performance for the highest load (the graph shows a detailed trace of the data-points marked by an A in the upper right corner of figure 7.6 and a B in the upper right corner of figure 7.7). The y-axis is the number of requests accumulated and the x-axis is the time in milliseconds.

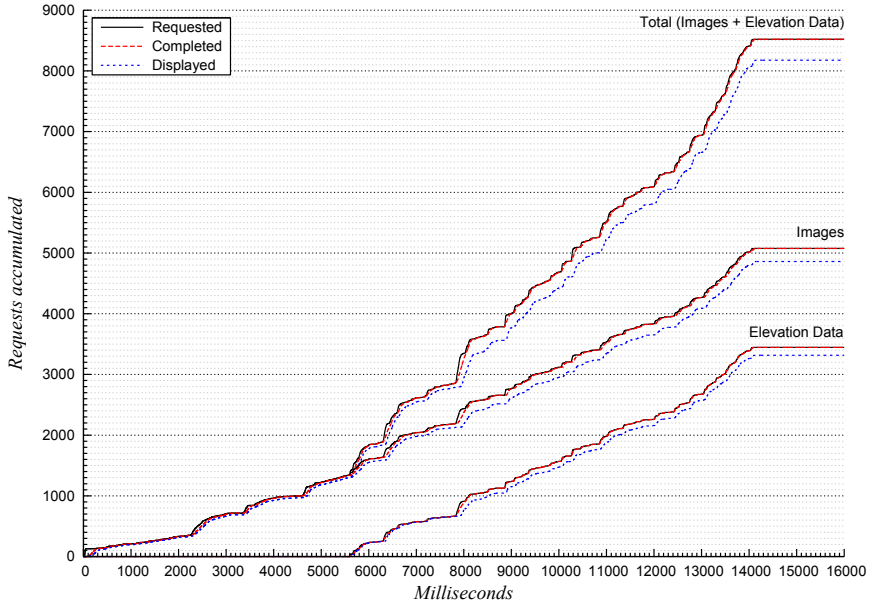

Figure 7.8: The cumulative number of requested, completed, and displayed requests with full local caches. The time between each waypoint is 1 second.

Figure 7.9 shows the completed requests of the same trace stacked in intervals of 50 milliseconds. This graph is a detailed trace of the data-point marked by a B in figure 7.7.

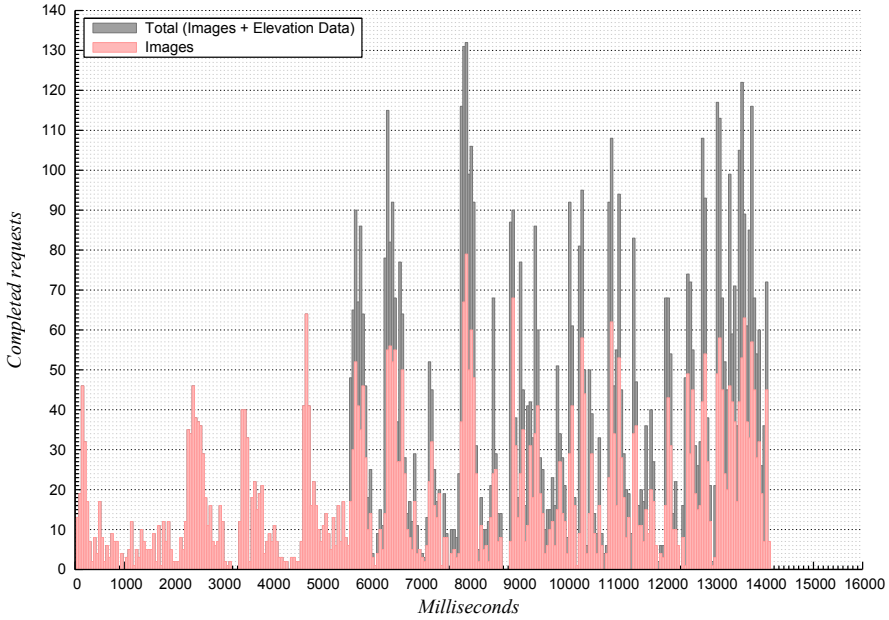

Figure 7.9: The number of completed requests for the full local cache configuration using 1 second between each waypoint. The requests are stacked in 50 milliseconds intervals.

From both figures 7.6 and 7.7 it can be seen that the main bottleneck of the system is the computation of customized data, illustrated by the difference between the graphs when everything is computed and the graphs when everything is stored in the live data set's cache or the local cache. As the graphs illustrate, the system benefits from increased number of compute nodes. The figures also show that the difference between local caching on each node and central caching using the live data set on one node is small. Thus, neither the network nor the live data set is a bottleneck of the system.

From 8 to 1 seconds the system does not display all requests, despite that fact that they are stored in the local cache (figure 7.6). This is explained by the time used to decode JPG images to create OpenGL textures, and the time used to parse elevation data to create geometry in the WallGlobe clients. From figure 7.8 it can be seen that around 8000 milliseconds the trace generates more requests than the WallGlobe clients are able to display, illustrated by the graphs showing displayed requests for both images and elevation data. When request threads cannot keep up with the frequency of the updates (because they are CPU bound decoding data), the amount of displayed data decreases. At 8000 milliseconds, the completed (decoded) requests peak at 132 requests (figure 7.9). This corresponds to 2640 requests per second (132 x (1000 ms / 50 ms)). Of these 132 requests, 79 are requests for images corresponding to a peak rate of 1580 images per second (414.2 megapixels/s), the equivalent of 18.83 updates per second on the 22-

megapixel display wall. As decoding of JPG images and parsing of elevation data is performed in separate request threads, the system will benefit from multi-core CPUs.

Figures 7.10 and 7.11 show the time and speedup factor for experiment series 12 and 13. This time includes the rasterization of the 350-page PDF document into image-tiles on the compute-side, including the time spent encoding images to PNG or JPG, the transfer of these image-tiles from the compute-side, through the live data set, to the display-side, and the loading and rendering of the image-tiles on the display nodes.

Figures 7.12 and 7.13 show the per-node and average node utilization on the compute-side for experiment series 12 and 13.

Table 7-6 shows the average latency for one request in the system when using all 28 compute nodes. Table 7-7 shows the result of experiment series 14 and 15. The load-time using the LDS cache includes the time used to request data from the cache, the transfer of the images over the network from the computer running the LDS to the display nodes, and the local time at the display nodes used to decode and render the images. The load-time for the local cache includes the times used to request the tiles from the local cache, including the time to decode the images and render them.

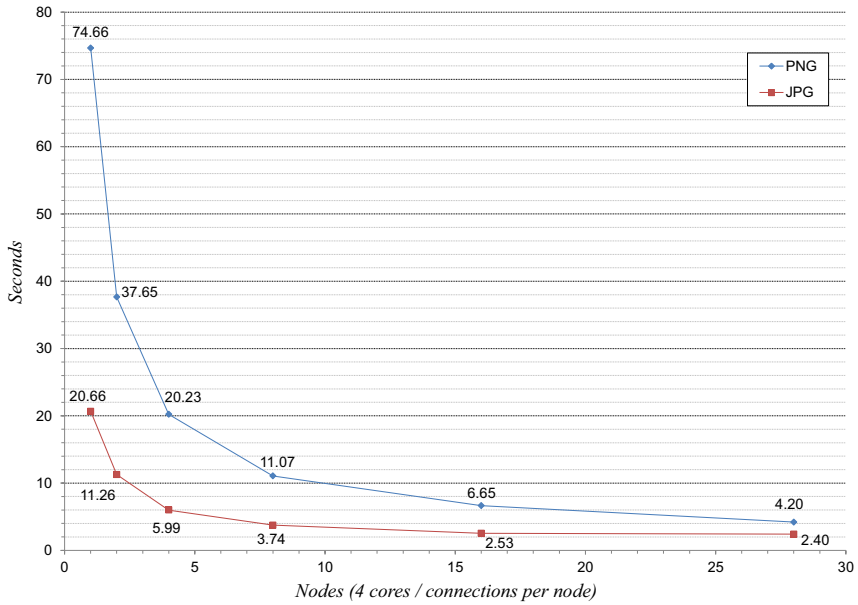

Figure 7.10: Time to request and simultaneously display 2432 JPG or PNG encoded image-tiles computed from a 350-page PDF document residing at the compute-side. (Compute nodes are increased from 1 to 28).

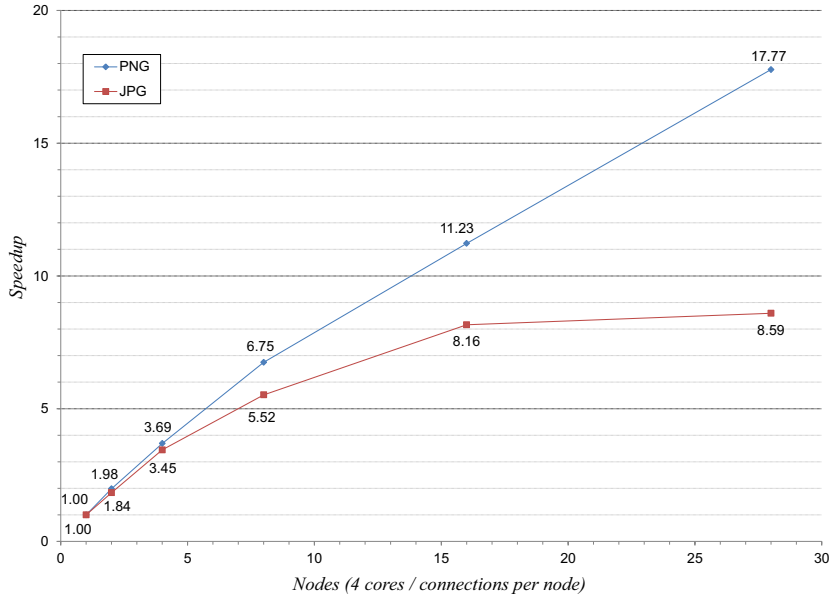

Figure 7.11: Speedup factor when requesting and simultaneously displaying 2432 JPG or PNG encoded image-tiles computed from a 350-page PDF document when going from 1 to 28 compute nodes.

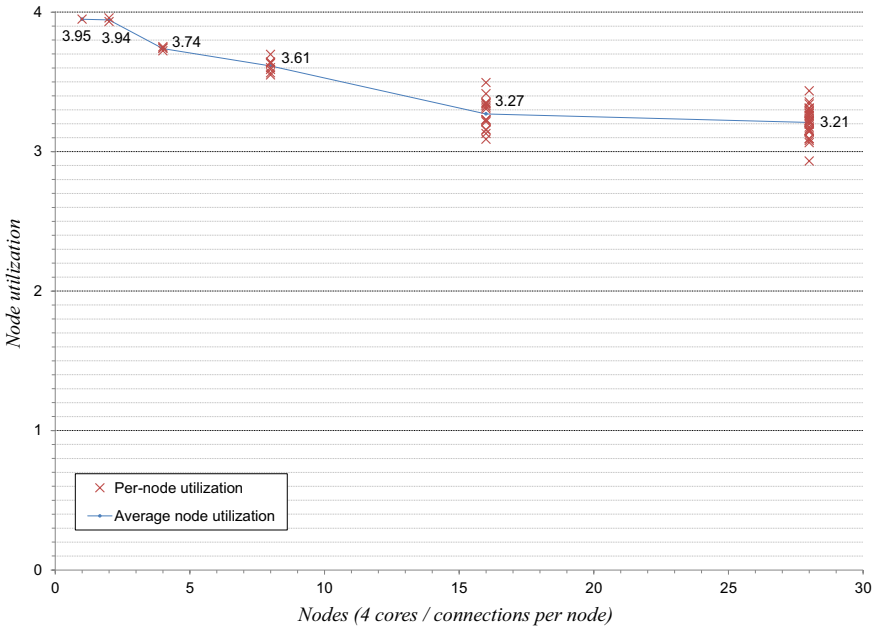

Figure 7.12: Compute node utilization when rasterizing the 350-page PDF document to PNG images.

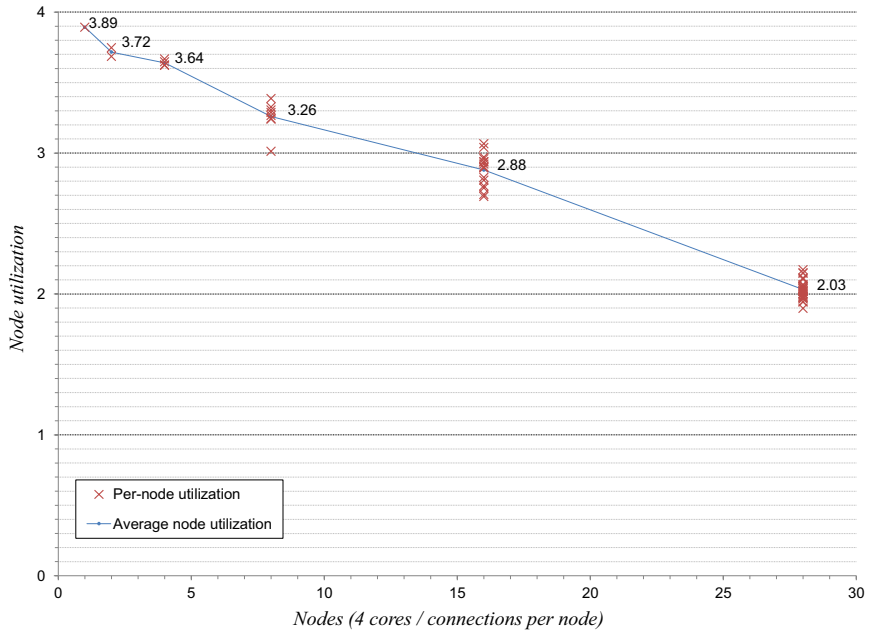

Figure 7.13: Compute node utilization when rasterizing the 350-page PDF document to JPG images.

Table 7-6: Average latency for a request to complete when using 28 compute nodes

Image Type	Display-side	Live Data Set	Compute-Side
PNG	0.1521 s	0.1456 s	0.1445 s
JPG	0.0856 s	0.0574 s	0.0533 s

Table 7-7: Time to request and simultaneously display 2432 JPG or PNG encoded image-tiles requested from the live data set's cache or from the local cache on each display node

Image Type	Load-Time LDS Cache	Load-Time Local Cache
PNG	1.694 s	0.908 s
JPG	1.305 s	0.923 s

As can be seen from both figures 7.10 and 7.11, the system benefits from increased number of dynamic compute nodes. When using PNG as the image format, the total load-time is 74.66 seconds using 1 compute node. When using

all nodes this time is reduced to 4.2 seconds, which translates to a speedup of 17.77. When using JPG as the image format the time to load the entire document using one compute node is 20.66 seconds. This time is reduced to 2.4 seconds using all compute nodes, translating to a speedup of 8.59. However, as both figures show, the load-time and speedup factor does not translate with a linear one-to-one factor with the use of additional compute nodes. In addition, for JPG, the speedup is approximately half of the speedup of PNG when using all nodes, and only increases with 0.43 when going from 16 to 28 nodes. This indicates a bottleneck in the system. When the produced image-tiles are located in the live data set's cache, the time used to load and display the document on the display-side is 1.694 seconds for PNG and 1.305 seconds for JPG (table 7-7). The reason that the compute system cannot produce data with this rate is a combination of the latency introduced by computing the image-tiles on the compute-side and the number of connections that are established from every node on the display-side.

During experiment series 11 to 15 (inclusive), every display node had 4 request connections. For PNG the average latency per request is 0.1521 seconds. When using 4 connections, this translates to a total average of 3.3 seconds per display node (((2432 / 28) / 4) x 0.1521). However, the compute nodes are idle some of this time. The result of this can be seen from figures 7.12 and 7.13. The utilization decreases as the number of nodes increases. When using PNG as the image format, the CPU core utilization is 3.95 using 4 cores on one node. This value is reduced to 3.21 using all nodes. For JPG the CPU core utilization using 1 node is 3.89, which is reduced to 2.03 using all nodes. To solve this situation the display-side could be configured to use more than 4 connections to the live data set. However, there is a tradeoff between the number of connections established from the display-side and the performance of the rendering engine. Request threads are responsible for decoding the data to the rendering engine. Decoding of images is CPU-bound and request threads will therefore compete with the rendering engine for CPU cycles on a single-core computer. This will in turn affect the frame rate of the visualization. However, this problem can be solved in several ways: (i) Separate request functionality from decoding functionality; (ii) pipeline requests; or (iii) use multi-core computers on the display-side with a dedicated core for the rendering engine. In addition to the display-side modifications, the connections between the live data set and compute nodes should allow for pipelining of requests to increase the node utilization and mask latency. The suggested improvements are all part of future updates to the system.

7.1.6 Conclusions

This chapter has presented WallScope, a pull-based NAD-NAC system, enabling interactive visualization of large-scale data sets on high-resolution tiled display walls. WallScope is based on a live data set architecture where display resources are separated from compute resources using a data set containing data customized for the particular application domain of the display-side's visualization systems. The live data set receives requests from visualization systems, translates these

into compute messages and forwards them to available compute nodes. The results of the computations are used as part of the visualization clients' final rendering.

Compute resources of the system are categorized into static and dynamic. Static compute resources are considered permanent to the system once added. Examples of such compute resources are clusters, grids and supercomputers. These compute resources are accessed according to their security policies and access protocols. Dynamic compute resources are volatile in the sense that they can register with the live data set on-the-fly to become compute nodes in the system, and then at a later point leave. Examples of dynamic compute resources are laptops and desktop computers.

Experiments conducted document that the sort-first rendering approach taken by the visualization systems implemented as part of WallScope, combined with a simple state server, enables each visualization client to keep the same frame rate as the refresh rate between the computers and their attached displays.

When visualizing the Earth by combining data from the Landsat data set with data from the Blue Marble data set, the bottleneck of the system is the merging process of the data sets on the static compute nodes. However, the time used to combine data sets decreases by a factor of 23 when increasing the number of compute nodes from 1 to 26. When all data is cached, the bottleneck of the system is decoding of JPEG images to create OpenGL textures and creating geometry from elevation data. WallGlobe, one of the visualization systems of WallScope, can decode 414.2 megapixels of image updates per second, resulting in 19 decoded frames per second when visualizing the Earth. The decoding process is multi-threaded, and higher frame rates are expected using multi-core computers. This tracks the current hardware trend towards more CPU cores.

The dynamic compute nodes in the system can be utilized in parallel to increase the overall performance of the system, improving the load-time of a PDF document from 74.66 to 4.2 seconds (PNG) and 20.66 to 2.4 seconds (JPG) when going from 1 to 28 compute nodes. This yields a resulting speedup of 17.77 (PNG) and 8.59 (JPG) using 28 compute nodes. These experiments show that the application output from personal desktop computers can be made interoperable with high-resolution tiled display walls, width good performance and without being limited to the resolution of the local desktop and display. The main bottleneck using the dynamic compute resources is the compute-side of the system combined with the number of connections that can be established from the display-side.

CHAPTER 8

DISCUSSION

This chapter presents a discussion of the contributions presented in this dissertation.

Three principles (chapter 1, section 1.5.1) have been formulated:

1. *The principle of establishing the end-to-end principle through customization* states that the end-to-end principle can be established between a client and a server by customizing one or both sides. For client-server communication, the normal way of establishing the end-to-end principle is to use pre-agreed protocols. In this dissertation, the end-to-end principle is established by having the display-side customizing the compute-side. The customization is accomplished in two different ways:

    a. Physically (customized behavior is accomplished by physically downloading code to the compute resource).

    b. Virtually (customized behavior is accomplished by using a third party for mapping between customized and compute resource behavior).

2. *The PC – PCR duality principle* states that a user's computer is both a personal computer and a personal compute resource. For normal usage, a user is sitting in front of the computer's display and applications are rendering to the frame buffer of the computer's graphics card. However, applications can also be instructed to produce output for visualization clients running on other computers without any modifications to the applications themselves. Thus, the computer is used as a personal compute resource for domain specific computation of data to extend the functionality of external visualization systems based on a user's custom software install.

3. *The principle of domain specific best-effort synchronization* states that for distributed visualization systems running on tiled display walls state handling can be performed using a best-effort synchronization approach,

where visualization clients will eventually get the correct state after a given period of time. Best effort synchronization has been researched in other contexts such as in [201]. However, this dissertation applies this principle for synchronization of visualization systems running on tiled display walls, which introduces properties that provide robustness for such visualization systems, such as removing multiple points of failure.

The principle of establishing the end-to-end principle through customization is used as a basis for all network accessible resources presented in this dissertation. In this dissertation, the principle implies that a display-side provides a compute-side with the information and software needed to communicate with it. This setup is inspired by and extends upon existing approaches for initiating rendering setups. For example, an X client using OpenGL for rendering to an X server needs to accept a frame buffer configuration for setting up an OpenGL context. The server provides the client with a set of frame buffer configurations that the client can choose from. If a valid configuration is not chosen, an OpenGL context cannot be created. This is comparable to the approach taken in the NAD system, where the NAD customizes the NAC by software that is uploaded from the NAD to the NAC. If the NAC accepts the software providing customized behavior (which is part of the two-phase customization protocol) output from the NAC's desktop can be mirrored onto the NAD. If not, the setup is aborted and the NAD cannot be used. For high-resolution tiled display walls, the choice of having a display-side customizing a compute-side comes from the shared aspect of the display wall. A display wall is a resource potentially shared by multiple simultaneous users, and thus requires that the compute resources that use the display wall respect a set of rules that protects the shared aspect of the resource.

Customization in a NAD – NAC context is handled in two ways, either physically by uploading customization code directly to the compute resource, or virtually by using a third party to handle the translation between NAD-side and compute resource behavior. By physically customizing a compute resource with uploaded software, the production of data can be customized without any intervention from a third party. However, the customization introduces trust and security concerns for users of the NAD, since the customization of the NAC requires execution of potentially untrusted code. A NAC downloads and executes software hosted by a potentially untrusted third party, the NAD. This dissertation approaches this problem using sandboxing in combination with code signing. The software uploaded from the NAD to the NAC is executed in a Java virtual machine, which is a sandboxed environment. In addition, the code is signed with a key authenticating the third party. Nevertheless, there are several contexts where a user could be reluctant to participate using the selected approach. For example, in meeting room contexts where a user needs to trust the person(s) responsible for maintaining the software hosted on the network accessible display(s).

The current way virtual customization has been carried out does not modify the processed data produced by a virtually customized resource. This is a limitation because the customization is limited to the presentation of the resource and not

the content of the data it produces. For example, the live data set presents computations as data sets to other visualization clients. However, the content of the data set is not customized since it would require processing the content before it is sent to the visualization clients. Processing the content in the live data set is not an option in the current architecture, since the role of the live data set is as an orchestrar of computations, rather than a compute resource. However, another approach that could be taken is to use other compute resources to further compute the content delivered from a virtually customized compute resource. This approach has not been fully investigated and such provides a promising path for future research in the WallScope system.

The PC – PCR duality principle states that a user's computer is both a personal computer and a personal compute resource. For normal usage, a user is situated in front of the computer's connected display(s), using input devices such as a keyboard and a mouse to interact with the computer. In addition to this usage, desktop applications on a user's computer can produce data for other clients in a system. The display-side can use the personal compute resource to produce compatible data from data that might be incompatible with the software installed on the display-side. This principle has been realized through the dynamic network accessible compute resources presented. The environment on a personal computer contains software and data installed that can complement a visualization system that lacks the necessary functionality for viewing the data. The visualization systems developed as part of this dissertation are simple compared to a complete desktop install. Consequently, the dynamic compute resources allow a broader variety of content to be shown on the display wall, while keeping the visualization clients simple.

The principle of domain specific best-effort synchronization is applied in this dissertation for handling state synchronization in distributed visualization systems running on tiled display walls. This principle requires the following two properties to be present in a system:

1. The participants of the state synchronization have established a pre-agreement on their arrangement (their position in the display wall grid).

2. Losing a state synchronization message does not affect state logic.

The principle allows for centralized control of frame rate, meaning the frame rate of every client in a distributed visualization system can be controlled from a single location. A consequence of this centralized push-based state synchronization is that a visualization system's load on a display wall can be controlled by throttling up or down the rate at which a state server broadcasts heartbeat messages. Another consequence of the principle is the effect an overloaded client will have on the performance of the entire visualization system. Other approaches, such as presented in [36] [43] [44], require synchronization through a barrier before the display is updated. This approach will in principle make all the visualization clients appear more harmonious, since each display of

the entire display wall is updated at the same time (within the overhead of performing the barrier). However, depending on the implementation of the barrier, the performance of the system is often limited by the most heavily loaded node. In contrast, the state synchronization approach used for most of the work in this dissertation does not require an explicit barrier between all clients between iterations, and thus every client can update the view based on their own initiative. Informal experience on the perceived impact of this does not indicate that it is a problem that makes the distributed visualization systems appear unharmonious. In addition, the approach removes the multiple point of failure a system using a barrier has. If a single node crashes or has performance problems, the rest of the visualization system can continue to work, and as such the crashed node does not have any impact on the other nodes.

The centralized state server realizing the best-effort synchronization principle simplifies the integration of visualization systems with various interaction devices. The single server broadcasts the state of the visualization, and the state abstracts away the type of interaction device used. For example, WallGlobe clients are not aware if the resulting viewing output is a consequence of interaction using a touch-free interaction device or a keyboard and a mouse. This enables a range of different input devices to be used with the visualization systems without affecting their implementation. In this regard, this approach adheres to the orthogonal interaction principle presented in [45].

WallScope has provided valuable experience for visualization of data from local and remote compute resources. Informal experience using the display wall with existing solutions for delivery of data, such as existing Web Map Services (WMS) [202] and Web Feature Services (WFS) [203], indicates that the order of magnitude more resolution provided by a display wall require data amounts that is beyond the processing capacity of these compute services. For this reason, pre-fetching data to the live data set has proven useful by providing the visualization systems with data close enough (both in terms of latency and bandwidth) to enable interactive performance. For some of the external systems used for data delivery in WallScope, the difference between having processed data cached or not, is waiting times within a couple of seconds instead of over 20 minutes.

The choice of using the HTTP protocol and the Squid cache system internally in WallScope could have resulted in bottlenecks being introduced. HTTP is a heavyweight protocol and as such using it in a high-performance system might seem counter-intuitive. However, experiments conducted have shown that neither the HTTP protocol nor the Squid cache system is a bottleneck in the system. In fact, the choice of using an existing highly available protocol significantly reduced the time used for testing and finding bugs in the system. WallScope is a complex system with several different components interacting to solve a common goal: Transforming data into a visual representation in form of pixels at interactive frame rates. Thus, a single component having bugs introduced from race conditions and/or timing issues could be hard to debug. Informal experience indicates that using existing systems such as web browsers and command line

HTTP tools can reduce the time to find these bugs compared to using experimental systems for the same purpose. The Squid caching system did not introduce any bottlenecks in the system, and in addition reduced the effort of debugging components in WallScope.

Experiments conducted using Squid did not document the performance of the system for different cache replacement policies. This is outside the scope of this dissertation. Instead, experiments have been conducted with and without caching enabled (using caches containing all requested data), to document the best- and worst-case performance scenarios. Different cache replacement policies will result in a system performance falling within this limit. Consequently, a mapping of the entire performance spectrum of the system has been presented, which can be used to predict the resulting performance of different application usages and cache- and compute-configurations.

The WallScope system and its associated static and dynamic NACs have been evaluated through a series of experiments. These experiments have documented the performance when integrating relatively static content into WallScope. A question that has not yet been addressed is how well the system will behave when visualizing data that requires frequent updates. For example, it is not clear if the current design could be used to stream video content. Such content will not benefit the cache to the same degree as static content. In addition, the shared links between the compute-side and the display-side would probably experience more traffic, and thus, it is not known whether the selected architecture and design could support the resulting load. There is also a question whether the centralized live data set will become a bottleneck for more dynamic content.

A problem with having a centralized live data set, that initially introduced bugs in the system caused by refused connections, is the number of simultaneous sockets that could be allocated by a single process. Using a centralized live data set simplifies the binding between the display-side and the compute-side since all requests and data to and from the display-side have to go through the same central location. However, the problem presented, combined with the fact that a centralized live data set introduces a bottleneck for the potential network bandwidth that could be used by having multiple display nodes, indicates that approaches for distributing the live data set to several computers should be investigated (a display wall comprising 28 computers could potentially transfer 28 gigabits per second with switched gigabit Ethernet technology). While the current version of the live data set has been implemented on a single computer, there are no limitations in the architecture for using more than a single computer running the live data set. One way of extending the live data set beyond a single computer is to use a single front-end, which transparently redirects requests from visualization clients to live data set nodes that are responsible for handling the content from a specific range or type of compute resources. For example, one live data set can be responsible for relaying video streams from a set of back-end video-transcoding compute services. Load-balancing could for example be done using techniques such as DNS load-balancing [204].

A limitation for the current design and implementation of WallScope is the fact that all visualization systems are consuming content from the static and dynamic network accessible compute resources, but none of the visualization systems update the original data sets. For example, a dynamic NAC can be used to show a Microsoft Word document on a display wall, by generating display friendly formats for the display-side visualization clients, such as image-tiles or PDFs. However, the selected approach does not yet support editing the original source for these image-tiles or PDFs. Thus, the current approach for visualization in WallScope enables interactive exploration of data from local and remote data resource, but does not yet give users any tools for modifying the original data sources. This would require a feedback mechanism to the NACs.

One problem that arises when adding functionality for updating local and remote data sources is how to update documents hosted by several NACs. As demonstrated in chapter 7, section 7.1.5, multiple NACs can be used in parallel to reduce the load-time of a PDF document. If a display-side visualization system would initiate a request for changing the content of a document, this would have to be taken care of by the system to make sure the document remains consistent between all NACs holding the document. Another issue that needs to be addressed in this regard is to equip users providing compute services through dynamic NACs with mechanisms to control what data is editable and what data is read only.

Another limitation for the current dynamic compute resources of WallScope is the fact that they require application support for using existing desktop applications to produce data that can be transformed into display-side friendly formats by the NAC software (for instance using Microsoft Component Object Model with Word or Excel). This requirement limits the number of applications that can be used to produce content for the display-side. One approach that can be taken as a last effort for applications that do not support this requirement is to invoke the applications and use a combination of input events and frame buffer captures to read the content from the applications as pixels, which then can be delivered to the visualization systems. This would require exclusive access to the computer where the applications are executed. In addition, it would be difficult to use more than a single process (and thus CPU core), and the output is limited to pixels. However, as a last effort approach it could still be used for applications that do not natively support other mechanisms.

The NAD system has been used on the display wall lab since the first prototype was developed. This system has enabled all users that wanted to mirror selected regions of the local desktop onto the display wall to do so without permanently installing third party software or opening firewall ports. This has been made possible by the customization phase that removes the need to have pre-installed software on the compute resource in order to utilize the display-side. There is a hidden assumption to the current version of the system, which also applies for the dynamic network accessible compute resources. Both systems need a Java Runtime Environment (JRE) installed. Since a JRE is not guaranteed to be

available for all software installs, it is possible that a user cannot use a NAD because the software needed for the customization phase is not present. From informal usage over almost three years, there has not been one incident where a user did not have the Java runtime environment installed. However, for users that do not have Java installed, the argument of not installing third party software does not hold. Another approach to this is to use binary files compiled for different operating systems, which can be manually downloaded by the user. However, this removes the sandboxing and transparent customization. A web browser with an extension would be a good approach for achieving transparent integration with a NAD. A web browser is included on every modern desktop install, and therefore would be a good approach for achieving customization in a NAD/NAC context. However, this would require frame buffer access from the web browser.

Another limitation of the current version of the NAD system is the lack of extending the resolution as a result of the increased resolution offered by a separate network accessible display. This has been successfully done in [47] where the desktop of a laptop has been extended up to 22 megapixels. However, currently there exists no way of extending a desktop cross-platform without modifying software in kernel space [47], including rebooting the computer to get access to the changes. Compared to the current approach used, such approaches are more intrusive. One way of achieving increased resolution with the current system, is to mirror the content from multiple NACs (potentially used by a single user) to a high-resolution NAD such as a display wall, and then interact with mirrored content from the NAD. Using this approach, the NAD can be thought of as a shared desktop (having multiple mirrored desktops) and the NAD would set the upper limit for the actual resolution that can be used.

The performance study of graphics processors conducted as part of this dissertation is based on the embarrassingly parallel Mandelbrot set computation. The study has shown how a modern graphics card can outperform an entire cluster of 28 computers computing on the CPUs and on previous generation GPUs. In fact, compared to the CPU versions of the Mandelbrot set computation, the single GPU could compute the entire data set and transfer the result to the cluster faster than computing the entire Mandelbrot set directly on the display nodes. However, the Mandelbrot set computation is limited in that it only benchmarks the performance of a computation that requires low amounts of data input (just the bounding box of the Mandelbrot set), does not consume much memory and can be fully distributed over all compute cores. Thus, the potential bottlenecks introduced by the bus connecting the graphics card with the CPU, the less memory available, and the impact of Amdahl's law [205] do not impose a huge penalty on the resulting performance. Consequently, the performance results presented gives an indication of the expected results for computations that share characteristics with the Mandelbrot set computation (low data input-size requirements, low memory profile, and embarrassingly parallel). Additionally, the computers used for the display wall were three years older than the GPUs used, and thus do not reflect state-of-the-art within CPU technologies.

One particular aspect that has not been addressed in this dissertation is security. The focus of this dissertation is not security, and thus security mechanisms have not been given much attention. The reason that security has not been explicitly addressed for the systems developed is that not all possible domains for the systems are known in advance. Understanding the domains require working prototypes, at which point domain specific security mechanism can be investigated. Nevertheless, security mechanisms are important for the presented systems for instance if used in a public context. The approach taken to security in this dissertation is to use existing mechanisms when security is needed. For example, the NAD system communicates with the NAC using a protocol based on pixels and events. A third party could easily get access to the pixels and keystrokes through eavesdropping. However, by tunneling the connection between the NAC and the NAD through SSH, the connection from the client to the server is secure and no explicit actions are needed in the protocol itself. The same applies for securing the communication channels between the components of the other systems developed.

The experiments conducted as part of this dissertation document the performance of the systems presented. No attempt is made on documenting their usability. The usability could be documented by designing a set of user-studies where users report their perception of the system(s) with respect to one or several factors (did the system make it easier to perform a certain task compared to other approaches? Was there any difficulties in using the system?). However, because of the systems methodology used throughout this dissertation this research direction has not been further pursued.

Chapter 9

Conclusions

This dissertation has presented Network Accessible Compute (NAC) resource and Network Accessible Display (NAD) resources for interactive visualization of data on displays ranging from laptops to high-resolution tiled display walls. A network accessible display is a display having functionality that enables usage over a network connection. Network accessible compute resources produce content for network accessible displays.

Several systems have been developed and evaluated based on push-based and pull-based network accessible compute- and display-resources. The idea, architecture, design, implementation and evaluation of the systems have resulted in three principles for interactive visualization on high-resolution tiled display walls. These are: (i) *Establishing the end-to-end principle through customization*, stating that the setup and interaction between a display-side and a compute-side in a visualization context can be achieved by customizing one or both sides; (ii) *Personal Computer (PC) – Personal Compute Resource (PCR) duality*, stating that a user's computer is both a personal computer and a personal compute resource, implying that desktop applications can be utilized locally using attached interaction devices and display(s), or remotely by other visualization systems for domain specific production of data based on a user's personal desktop install; and (iii) *domain specific best-effort synchronization* stating that for distributed visualization systems running on high-resolution tiled display walls, state handling can be performed using a best-effort synchronization approach, where visualization clients will eventually get the correct state after a given period of time.

One push-based NAC-NAD system has been developed enabling users to mirror multiple user-selected regions from their local computer's desktop onto nearby network accessible displays. The system is based on the principle of establishing the end-to-end principle through customization. This enables usage of nearby network accessible display resources without requiring: (i) Usage of DVI/VGA cables; (ii) permanent installation of third party software; and (iii) opening firewall ports.

The push-based system has been evaluated by measuring its performance when mirroring content from a desktop at different resolutions with applications exhibiting different output characteristics. At a resolution of 800 by 600 pixels, the system supports mirroring of dynamic content at 38.6 FPS. At 1600 by 1200 pixels, the refresh rate is 12.85 FPS. For static content such as images and slideshow presentations, the system's bandwidth usage is within the capacity of an 11 Mbit/s wireless network. For dynamic content such as videos and games, the system requires at least a 100 Mbit/s connection. The bottleneck of the system is frame buffer capturing and encoding/decoding of pixels. Pipelining the main parts of the NAC-side and NAD-side software would most likely give a frame rate increase in all cases, and by a factor of two in the best case.

One pull-based system, WallScope, has been developed comprising a range of visualization- and compute-systems, enabling interactive visualization of local and remote data sets on high-resolution tiled display walls. WallScope separates display resources from compute resources using a live data set containing data customized for the particular application domain of the display-side's visualization systems. The live data set receives requests from the display-side, translates these into compute messages and forwards them to available compute resources. Results of the computations are used as part of the final rendering on the display-side. The compute resources of the system are categorized into static, such as clusters, grids and supercomputers, and dynamic, such as laptops and desktop computers. Static compute resources are accessed according to their security policies and access protocols. Dynamic compute resources are customized on-the-fly, to become compute resources in the system.

Experiments conducted document that the sort-first rendering approach taken by the visualization systems implemented as part of WallScope, combined with a simple state server, enables each visualization client to keep the same frame rate as the refresh rate of their attached displays.

When visualizing the Earth by combining data from the Landsat data set with data from the Blue Marble data set, the bottleneck of the system is the merging process of the data sets on the compute nodes. However, the time used to combine data sets decreases by a factor of 23 when increasing the number of compute nodes from 1 to 26. When all data is cached, the bottleneck of the system is decoding of JPEG images to create OpenGL textures and creating geometry from elevation data. The visualization system can decode 414.2 megapixels per second, resulting in 19 decoded frames per second. The decoding process is multi-threaded, and higher frame rates are expected using multi-core computers. This tracks the current hardware trend towards more CPU cores.

The dynamic compute nodes in the system can be utilized in parallel to increase the overall performance of the system, improving the load-time of a PDF document from 74.66 to 4.20 seconds (PNG) and 20.66 to 2.40 seconds (JPG) when going from 1 to 28 compute nodes. This translates to a resulting speedup of 17.77 (PNG) and 8.59 (JPG) using 28 compute nodes. This shows that the

application output from personal desktop computers can be made interoperable with high-resolution tiled display walls, with good performance and without being limited to the resolution of the local desktop and display. The main bottleneck using the dynamic compute resources is the compute-side of the system combined with the number of connections that can be established from the display-side.

The work presented in this dissertation has resulted in several contributions for solving challenges related to interactive visualization on high-resolution tiled display walls.

Synchronization of display nodes are handled according to the principle of best-effort state synchronization, implemented using a state server and a push-based heartbeat protocol. Compared to systems using lock-step approaches and/or barriers [36] [43] [44], the presented approach has several advantages: (i) Centralized state control, allowing the state of a visualization to be computed using the same clock, less likely resulting in divergence between visualization clients; (ii) centralized load control, allowing for easy control of load from a single location; (iii) avoiding multiple points of failure; and (iv) avoiding global slowdown constrained by the most heavily loaded node.

Display nodes and compute nodes follow the principle of establishing the end-to-end principle through customization. This principle has enabled utilization of network accessible displays from different types of compute nodes (such as desktops and laptops) without requiring pre-installed pre-configured systems. Combined with a live data set the principle allows for separating domain specific computations from visualization systems, keeping each client of a distributed visualization system simple, while implementing domain specific functionality at the compute-side.

The principle of PC – PCR duality has enabled usage of existing desktop applications by visualization systems running on tiled display walls, by decoupling the application output from the resolution of the local desktop and display. This allows for visualization on displays with potentially much higher resolutions, without requiring modifications to the local applications or the operating systems. The principle has been realized by the dynamic compute resources in WallScope. In combination with the live data set architecture, this allows for utilization of desktop applications from distributed visualization systems running on tiled display walls.

Compared to state-of-the-art systems presented in the literature, the contributions of this dissertation enable utilization of a broader range of compute resources from a display wall, while at the same time providing better control over where to provide functionality and where to distribute workload between compute-nodes and display-nodes in a visualization context.

CHAPTER 10

FUTURE WORK

This section outlines some of the possible future directions that can be researched based on the current status of the work presented in this dissertation.

The push-based NAD system presented enables users to mirror content from their local desktop onto nearby network accessible displays. This work has focused on a simple way of displaying the content of a desktop on a remote display, as well as interacting with the local desktop from the remote location. However, a direction that has not yet been fully investigated is sharing of content between remote displays. This usage might not be obvious on stand-alone displays and projectors. However, on high-resolution tiled display walls, multiple desktops can be simultaneously mirrored to the same display surface. Sharing content between these desktops could be useful, for example, by dragging a file from one desktop to another, thereby copying the file between the computers. This could be a promising research direction for future work.

WallScope and its associated visualization systems comprise a platform that can be used for future research. The current version of WallGlobe and WallScope has already been used to conduct research on interactive weather forecasting. This can be extended to include other type of visualizations, such as earthquakes, avalanches, and other weather-related simulations.

Incorporating "live" objects into WallScope is another research direction that could be investigated. Data feeds describing objects in the real world has already been incorporated in WallGlobe (this work is included in appendix B). This work has documented how object state can be separated from the visualization of the objects, allowing multiple object-state servers to be used for state tracking. The work presented outlines some interesting future research directions that could be investigated, among others per-visualization node object state tracking. In addition, adding physics using both CPU and GPU (such as PhysX [206]) approaches could be investigated in this context.

The network accessible compute resources developed as part of this work produce customized data for the visualization systems. The use of separate compute resources allows for more computational power to be added if needed,

but most importantly allows the visualization systems to be kept simple by incorporating functionality for transforming data on the compute-side. In this sense, the functionality of the visualization systems heavily depends on the functionality of the compute resources. Thus, more functionality can be added to WallScope by adding this to the static or dynamic compute resources.

The dynamic network accessible compute resources have demonstrated how the application output from local desktop applications can be made independent of the resolution of the local desktop and display by transforming desktop computers into compute services accepting requests for content such as vector data or tiled output, that can utilize the resolution of high-resolution tiled display walls. More plugins can be implemented to support a wider range of desktop applications. The current version contains among others functionality for transforming office documents into display friendly formats for visualization systems running on display walls. Other formats could be added as well.

A research direction that has not yet been investigated is to use dynamic compute resources to provide further computations on data returned from virtually customized compute resources. For example, a virtually customized compute resource might provide data that still needs processing before it can be used on the display-side. Instead of performing this computation on the display-side, a physically customized compute resource could perform the computation before sending the processed data to the requesting display-side visualization system.

The number of connections that can be established from the display-side without affecting the performance of the rendering engine has been demonstrated to introduce lack in compute-side utilization. Thus, one future research direction that could be investigated is separating decoding and requesting of data. By separating decoding functionality from requesting functionality, more connections can be established from the display-side without affecting the performance of the rendering engine caused by having decoding threads competing with the rendering engine for CPU cycles. For multi-core compute nodes, this performance decrease can be controlled by assigning the rendering engine to run on a separate core. However, the display wall presented in chapter 2, section 2.3 have single core CPUs (with HyperThreading) and thus would benefit an approach where requesting and decoding are separated.

A research direction that has not been fully investigated is streaming of dynamic content from the compute resources to the visualization systems. The current version could support a similar approach by requesting new content from the compute nodes at regular intervals, an action that has been documented to support a refresh rate of 19 frames per second at the display-side. However, other approaches could be investigated for displaying videos and other content that have dynamic update characteristics.

With the recent trends in GPU hardware architectures and based on performance measurements conducted among others as part of this dissertation (chapter 5,

section 5.4.2), GPUs have been demonstrated as promising devices for data-parallel computations. One possible research direction that could be further investigated is using GPUs as NACs (both static and dynamic).

References

[1] Gordon Bell, Tony Hey, and Alex Szalay, "Beyond the Data Deluge," *Science*, vol. 323, no. 5919, pp. 1297-1298, 2009.

[2] Tony Hey, Stewart Tansley, and Kristin Tolle, *The Fourth Paradigm: Data-Intensive Scientific Discovery*.: Microsoft Research, 2009.

[3] Peter Fox and James Hendler, "Changing the Equation on Scientific Data Visualization," *Science Magazine*, vol. 331, no. 6018, pp. 705-708, February 2011.

[4] Intel Corporation. (2011, February) Excerpts from A Conversation with Gordon Moore: Moore's Law. [Online]. ftp://download.intel.com/museum/Moores_Law/Video-transcripts/Excepts_A_Conversation_with_Gordon_Moore.pdf

[5] Angela C. Sodan, Jacob Machina, Arash Deshmeh, Kevin Macnaughton, and Bryan Esbaugh, "Parallelism via Multithreaded and Multicore CPUs," *Computer*, vol. 43, no. 3, pp. 24-32, March 2010.

[6] Spiral. (2007, April) Evolution of Intel Platforms. [Online]. http://www.spiral.net/graphics/evolution.gif

[7] John L. Manferdelli, Naga K. Govindaraju, and Chris Crall, "Challenges and Opportunities in Many-Core Computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 808-815, May 2008.

[8] Tilera. (2011, February) TILE-Gx Processor Family. [Online]. http://www.tilera.com/sites/default/files/productbriefs/PB025_TILE-Gx_Processor_A_v3.pdf

[9] NVIDIA. (2011, February) Geforce GTX 580. [Online]. http://www.nvidia.com/object/product-geforce-gtx-580-us.html

[10] Atlas Computing Division Rutherford Laboratory. (2010, June) PERQ History. [Online]. http://www.chilton-computing.org.uk/acd/sus/perq_history/part_1/c3.htm

[11] EIZO. (2011, February) ColorEdge CG303W. [Online]. http://www.eizo.com/global/products/coloredge/cg303w/index.html

[12] Canon. (2011, February) Digital Compact Cameras. [Online]. http://www.canon.com/camera-museum/camera/dcc/data/2009-2010/2010_ixy_400f.html?p=2

[13] Bruce H. McCormick, Thomas A. DeFanti, and Maxine D. Brown, "Visualization in Scientific Computing," *Computer Graphics*, vol. 21, no. 6, November 1987.

[14] Nahum Gershon, Stephen G. Eick, and Stuart Card, "Information visualization," *Interactions*, vol. 5, no. 2, pp. 9-15, March 1998.

[15] R. B. Haber and D. A. McNabb, "Visualization Idioms: A Conceptual Model for Scientific Visualization Systems," in *Visualization in Scientific Computing*, Bruce Shriver, Ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 61-73.

[16] Roy S. Kalawsky, "Gaining Greater Insight through Interactive Visualization: A Human Factors Perspective," in *Trends in Interactive Visualization*, Lakhmi Jain et al., Eds.: Springer London, 2009, pp. 1-36, 10.1007/978-1-84800-269-2_6.

[17] Microsoft. (2011, April) Excel. [Online]. http://office.microsoft.com/en-us/excel/

[18] Oracle. (2011, February) OpenOffice.org Calc. [Online]. http://www.openoffice.org/product/calc.html

[19] X.org Foundation. (2011, January) Documentation for the X Window System Version 11 Release 7.6 (X11R7.6). [Online]. http://www.x.org/releases/X11R7.6/doc/

[20] (April, 2011) GIMP Toolkit. [Online]. http://www.gtk.org

[21] Nokia Corporation. (2011, April) Qt. [Online]. http://qt.nokia.com/

[22] The GNOME Project. (2011, April) GDK 2 Reference Manual. [Online]. http://developer.gnome.org/gdk/

[23] Khronos Group. (2011, April) OpenGL - The Industry's Foundation for High Performance Graphics. [Online]. http://www.opengl.org/

[24] Silicon Graphics Inc. (2011, April) OpenGL Graphics with the X Window System. [Online]. http://www.opengl.org/documentation/specs/glx/glx1.4.pdf

[25] Brian Paul. (2007, June) Mesa. [Online]. http://mesa3d.sourceforge.net/

[26] Matthias Hopf. (2010, June) compiz: The Next Generation Desktop. [Online]. http://www.vis.uni-stuttgart.de/~hopf/pub/LinuxTag2007_compiz_NextGenerationDesktop_Slides.pdf

[27] Oliver G. Staadt, Justin Walker, Christof Nuber, and Bernd Hamann, "A survey and performance analysis of software platforms for interactive cluster-based multi-screen rendering," in *Proceedings of the workshop on Virtual environments*, New York, NY, USA, 2003, pp. 261-270.

[28] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs, "A Sorting Classification of Parallel Rendering," *IEEE Computer Graphics and Applications*, vol. 14, no. 4, pp. 23-32, July 1994.

[29] Jorge Luis Williams, "Sort-middle stream-based real-time rendering on commodity clusters," University of Idaho, Moaxcow, Ph.D. 3347517, 2009.

[30] Michael J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948-960, September 1972.

[31] Robert W. Scheifler and Jim Gettys, "The X window system," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79-109, April 1986.

[32] Yong Liu and Otto J. Anshus, "Improving the performance of VNC for high-resolution display walls," in *Proceedings of the 2009 International Symposium on Collaborative Technologies and Systems*, Washington, DC, USA, 2009, pp. 376-383.

[33] Greg Humphreys et al., "Chromium: a stream-processing framework for interactive rendering on clusters," in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2002, pp. 693-702.

[34] (2004, June) Distributed Multihead X Project. [Online]. http://dmx.sourceforge.net/

[35] 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*, vol. 39, New York, NY, USA, 2005, pp. 277-290.

[36] 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*, vol. 5, no. 10, November 2008.

[37] T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper, "Virtual Network Computing," *IEEE Internet Computing*, vol. 2, no. 1, pp. 33-38, January 1998.

[38] OnLive Inc. (2011, February) OnLive. [Online]. www.onlive.com

[39] Calit2. (2011, February) HIPerWall. [Online]. http://hiperwall.calit2.uci.edu/?q=node/1

[40] Doug Ramsey. (2008, July) UC San Diego Unveils World's Highest-Resolution Scientific Display System. [Online]. http://www.calit2.net/newsroom/release.php?id=1332

[41] Thomas A. DeFanti et al., "The OptIPortal, a scalable visualization, storage, and computing interface device for the OptiPuter," *Future Generation Computer Systems*, vol. 25, no. 2, pp. 114-123, 2009.

[42] Doug Ramsey. (2008, September) California scientists demonstrate how to use advanced fiber-optic backbone for research. [Online]. http://www.universityofcalifornia.edu/news/article/18717

[43] Ping Yin, Xiaohong Jiang, Jiaoying Shi, and Ran Zhou, "Multi-screen Tiled Displayed, Parallel Rendering System for a Large Terrain Dataset," *The International Journal of Virtual Reality (IJVR)*, vol. 5, no. 4, pp. 47-54, 2006.

[44] Sungwon Nam et al., "Multi-Application Inter-Tile Synchronization on Ultra-High-Resolution Display Walls," in *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, New York, NY, USA, 2010, pp. 145-156.

[45] Daniel Stødle, "Device-Free Interaction and Cross-Platform Pixel Based Output to Display Walls," Department of Computer Science, Faculty of Science, University of Tromsø, Tromsø, Ph.D. Dissertation 2009.

[46] Microsoft. (2011, February) Windows. [Online]. http://www.microsoft.com/windows/

[47] Daniel Stødle, John Markus Bjørndalen, and Otto J. Anshus, "The 22

megapixel laptop," in *Proceedings of the 2007 workshop on Emerging displays technologies*, New York, NY, USA, 2007, pp. 1-4.

[48] NVIDIA. (2011, February) CUDA C Programming Guide (Version 3.2). [Online]. http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf

[49] Thomas Sørensen. (2010, March) NRK Nordnytt - Verdens største iPhone? [Online]. http://fil.nrk.no/nyheter/distrikt/troms_og_finnmark/1.6911043

[50] Vlad Savov. (2010, March) engadget - Tromso students put together the best interactive display wall we've seen yet. [Online]. http://www.engadget.com/2010/03/24/tromso-students-put-together-the-best-interactive-display-wall-w/

[51] Chris Davies. (2010, March) slashgear - 22MP Display Wall for gigapixel images created. [Online]. http://www.slashgear.com/22mp-display-wall-for-gigapixel-images-created-video-2378652/

[52] Øyvind Solstad. (2010, March) NRKbeta - Verdens største multitouchskjerm er norsk. [Online]. http://nrkbeta.no/2010/03/23/verdens-stoerste-multitouchskjerm-er-norsk/

[53] Caleb Kraft. (2010, March) HACK A DAY - Massive no-touch physically-interfaced display. [Online]. http://hackaday.com/tag/gigapixel/

[54] (2010, March) Ubergizmo - Monstrous physical interactive display requires no touch at all. [Online]. http://www.ubergizmo.com/2010/03/monstrous-physical-interactive-display-requires-no-touch-at-all/

[55] Daniel Stødle, Bård Fjukstad, and Otto Anshus. (2010, September) Nordlys - Morgendagens spåkuler. [Online]. http://www.nordlys.no/kronikk/article5325750.ece

[56] Falko Kuester et al. (2011, February) The Highly Interactive Parallelized Display Space project (HIPerSpace). [Online]. http://vis.ucsd.edu/mediawiki/index.php/Research_Projects:_HIPerSpace

[57] K. Li et al., "Building and Using A Scalable Display Wall System," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 29-37, July/August 2000.

[58] Grant Wallace et al., "Tools and Applications for Large-Scale Display

Walls," *IEEE Computer Graphics and Applications*, vol. 25, no. 4, pp. 24-33, July 2005.

[59] AMD. (2011, April) AMD Eyefinity Technology. [Online]. http://www.amd.com/us/products/technologies/amd-eyefinity-technology/Pages/eyefinity.aspx

[60] Tristan Richardson. (2010, November) The RFB Protocol. [Online]. http://www.realvnc.com/docs/rfbproto.pdf

[61] RealVNC Limited. (2011, February) RealVNC. [Online]. http://www.realvnc.com/

[62] (2011, February) TightVNC. [Online]. http://www.tightvnc.com/

[63] UltraVNC. (2011, February) UltraVNC. [Online]. http://www.uvnc.com/

[64] (2009, April) Xvnc - the X VNC server. [Online]. http://www.realvnc.com/products/free/4.1/man/Xvnc.html

[65] Greg Humphreys et al., "WireGL: a scalable graphics system for clusters," in *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2001, pp. 129-140.

[66] Byungil Jeong et al., "High-performance dynamic graphics streaming for scalable adaptive graphics environment," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2006, pp. 24-24.

[67] Microsoft. (2011, February) Direct3D Graphics. [Online]. http://msdn.microsoft.com/en-us/library/bb153256%28v=VS.85%29.aspx

[68] NVIDIA. (2011, April) Cg Toolkit. [Online]. http://developer.nvidia.com/cg-toolkit

[69] NVIDIA. (2011, April) CUDA Zone. [Online]. http://www.nvidia.com/object/cuda_home_new.html

[70] (2011, April) TOP. [Online]. http://www.linuxmanpages.com/man1/top.1.php

[71] Luis Martin Garcia. (2011, March) TCPDUMP & LIBPCAP. [Online]. http://www.tcpdump.org/

[72] Daniel Stødle, Tor-Magne Stien Hagen, John Markus Bjørndalen, and Otto

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*, 2007, pp. 75-83.

[73] Tor-Magne Stien Hagen, Espen Skjelnes Johnsen, Daniel Stødle, John Markus Bjørndalen, and Otto Anshus, "Liberating the Desktop," in *First International Conference on Advances in Computer Human Interaction, ACHI*, 2008, pp. 89-94.

[74] Tor-Magne Stien Hagen, Oleg Jakobsen, Phuong Hoai Ha, and Otto Anshus, "Comparing the Performance of Multiple Single-Cores versus a Single Multi-Core," in *Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA*, 2008.

[75] Tor-Magne Stien Hagen, Phuong Hoai Ha, and Otto Anshus, "Experimental Fault-Tolerant Synchronization for Reliable Computation on Graphics Processors," in *Proceedings of the 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA*, 2008.

[76] Tor-Magne Stien Hagen, Daniel Stødle, and Otto Anshus, "On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls," in *Proceedings of the International Conference on Information Visualization Theory and Applications*, 2010, pp. 112-119.

[77] Tor-Magne Stien Hagen, Daniel Stødle, John Markus Bjørndalen, and Otto Anshus, "A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls," *to appear in the Proceedings of the 11th IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS*, 2011.

[78] Bård Fjukstad et al., "Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes," *to appear in the Proceedings of the 10th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA*, 2011.

[79] Microsoft. (2011, February) Remote Desktop Protocol. [Online]. http://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx

[80] Dean Macri, "The Scalability Problem," *Queue*, vol. 1, no. 10, pp. 66-73, 2004.

[81] id. (2011, March) id Software. [Online]. http://www.idsoftware.com/

[82] Relic Entertainment. (2010, October) HomeworldSDL.org. [Online].

http://www.homeworldsdl.org/

[83] id Software. (2011, March) ioquake3. [Online]. http://ioquake3.org/

[84] Relic Entertainment. (2011, March) Relic Entertainment. [Online]. www.relic.com

[85] Nintendo. (2011, February) Wii Controllers. [Online]. http://www.nintendo.com/wii/console/controllers

[86] Jeffrey Jacobson and Zimmy Hwang, "Unreal tournament for immersive interactive theater," *Communications of the ACM - Internet abuse in the workplace and Game engines in scientific research*, vol. 45, no. 1, pp. 39-42, January 2002.

[87] (2010, January) Message Passing Interface Forum. [Online]. http://www.mpi-forum.org/

[88] Intel. (2011, March) Intel® Developer Network for PCI Express Architecture. [Online]. http://www.intel.com/technology/pciexpress/index.htm

[89] Intel. (2005, March) Intel® E8500 Chipset North Bridge (NB). [Online]. http://www.intel.com/assets/pdf/datasheet/306745.pdf

[90] Anton Lokhmotov, Alan Mycroft, and Andrew Richards, "Delayed Side-effects Ease Multi-core Programming," in *Proceedings of the 13th International Euro-Par Conference*, Rennes, France, 2007, pp. 641-650.

[91] Jason Mcguiness, Colin Egan, Bruce Christianson, and Guang Gao, "The Challenges of Efficient Code-Generation for Massively Parallel Architectures," in *11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, Shanghai, China, 2006, pp. 416-422.

[92] Felix Putze, Peter Sanders, and Johannes Singler, "MCSTL: the multi-core standard template library," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming* , New York, NY, USA, 2007, pp. 144-145.

[93] A. Corradi, L. Leonardi, and F. Zambonelli, "Dynamic load distribution in massively parallel architectures: the parallel objects example," in *Proceedings of the First Conference on Massively Parallel Computing Systems*, Ischia, Italy, 1994, pp. 318-322.

[94] V. Drakopoulos, N. Mimikou, and T. Theoharis, "An overview of parallel

visualisation methods for Mandelbrot and Julia sets," *Computers & Graphics*, vol. 27, no. 4, pp. 635-646, 2003.

[95] Zhe Fan, Feng Qiu, Arie Kaufman, and Suzanne Yoakum-Stover, "GPU Cluster for High Performance Computing," in *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, Washington, DC, USA, 2004, pp. 47--.

[96] Hsi-Yu Schive, Chia-Hung Chien, Shing-Kwong Wong, Yu-Chih Tsai, and Tzihong Chiueh, "Graphic-Card Cluster for Astrophysics (GraCCA) - Performance Tests," *New Astronomy*, vol. 13, no. 6, pp. 418-435, August 2008.

[97] Nolan Goodnight, Cliff Woolley, Gregory Lewin, David Luebke, and Greg Humphreys, "A multigrid solver for boundary value problems using programmable graphics hardware," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Aire-la-Ville, Switzerland, 2003, pp. 102-111.

[98] Isaac Rudomín, Erik Millán, and Benjamín Hernández, "Fragment shaders for agent animation using finite state machines. Simulation Modelling Practice and Theory," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 741-751, November 2005.

[99] Manuel Ujaldon and Joel Saltz, "The GPU on irregular computing: Performance issues and contributions," in *Proceedings of the Ninth International Conference on Computer Aided Design and Computer Graphics*, vol. 0, Washington, DC, USA, 2005, pp. 442-450.

[100] NVIDIA. (2011, February) NVIDIA CUDA Compute Unified Device Architecture (Version 1.0). [Online]. http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf

[101] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset, "General purpose molecular dynamics simulations fully implemented on graphics processing units," *Journal of Computational Physics*, vol. 227, no. 10, pp. 5342-5359, May 2008.

[102] S. Williams, J. Carter, L. Oliker, J. Shalf, and K. Yelick, "Lattice Boltzmann simulation optimization on leading multicore platforms," in *IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, 2008, pp. 1-14.

[103] B. Franke and M.F.P. O'Boyle, "A complete compiler approach to auto-

parallelizing C programs for multi-DSP systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 3, pp. 234-245, March 2005.

[104] Paul K. Sherard, "Julia sets and quasi-stable orbits in the complex plane," *Computers & Graphics*, vol. 17, no. 2, pp. 175-184, March 1993.

[105] NVIDIA. (2011, March) CUDA GPUs. [Online]. http://www.nvidia.com/object/cuda_gpus.html

[106] Phuong Hoai Ha, Philippas Tsigas, and Otto J. Anshus, "Wait-free Programming for General Purpose Computations on Graphics Processors," in *22nd IEEE International Symposium on Parallel and Distributed Processing, IPDPS*, Miami, Florida, USA, 2008, pp. 1-12.

[107] Maurice Herlihy, "Wait-free synchronization," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 13, no. 1, pp. 124-149, January 1991.

[108] Clyde P. Kruskal, Larry Rudolph, and Marc Snir, "Efficient synchronization of multiprocessors with shared memory," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 10, no. 4, pp. 579-601, October 1988.

[109] TightVNC Group. (2011, February) TightProjector. [Online]. http://www.tightvnc.com/projector/

[110] Bartels Media GmbH. (2011, March) MaxiVista. [Online]. http://www.maxivista.com/

[111] Vasily Tarasov. (2011, February) ZoneOS. [Online]. http://www.zoneos.com/zonescreen.htm

[112] Patrick Stein. (2010, November) ScreenRecycler. [Online]. http://www.screenrecycler.com/ScreenRecycler.html

[113] Microsoft. (2011, February) Windows Network Projector Overview. [Online]. http://msdn.microsoft.com/en-us/library/aa934598.aspx

[114] Aras Bilgen and Keith W. Edwards, "inSpace Projector: An Accessible Display System for Meeting Environments," in *Workshop on Usable Ubiquitous Computing in Next Generation Meeting Rooms: Design, Architecture, and Evaluation. Ubicomp Conference*, 2006.

[115] Grant Wallace and Kai Li, "Virtually shared displays and user input devices," in *2007 USENIX Annual Technical Conference on Proceedings*

of the USENIX Annual Technical Conference, Berkeley, CA, USA, 2007, pp. 1-6.

[116] UPnP Forum. (2008, October) UPnP - Device Architecture Document. [Online]. http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf

[117] Oracle. (2011, February) Java. [Online]. http://www.java.com/en/

[118] Oracle. (February, 2008) Java Web Start Technology. [Online]. http://java.sun.com/javase/technologies/desktop/webstart/

[119] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman, "Benchmarking Java against C and Fortran for scientific applications," in *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, New York, NY, USA, 2001, pp. 97-105.

[120] Thomas J. Lynch, *Data compression: techniques and applications*, 0534034187th ed. Belmont, CA, USA: Lifetime Learning Publications, 1985.

[121] Google. (2011, February) Google Maps. [Online]. http://maps.google.com

[122] Google. (2011, February) Google Maps API Family. [Online]. http://code.google.com/apis/maps/index.html

[123] Google. (2011, February) Google Earth. [Online]. http://www.google.com/earth/index.html

[124] Fay Chang et al., "Bigtable: a distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, USA, 2006, pp. 205-218.

[125] Sanjay Ghemawat, Howard Gobioff, and Shun T. Leung, "The Google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, vol. 37, New York, NY, USA, 2003, pp. 29-43.

[126] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, Berkeley, CA, USA, 2004, pp. 137-150.

[127] Chris Williams. (2007, June) Display Wall with Google Earth on Mac OS X. [Online]. http://blog.irisink.com/2007/06/14/display-wall-with-google-

earth-on-mac-os-x/

[128] University of California. (2011, February) Google Earth on HIPerWall. [Online]. http://hiperwall.calit2.uci.edu/?q=node/5

[129] Google. (2011, February) Liquid Galaxy. [Online]. http://www.google.com/earth/explore/showcase/liquidgalaxy.html

[130] Microsoft. (2011, February) Bing Maps APIs, SDK, and application development tools. [Online]. http://www.microsoft.com/maps/

[131] Microsoft. (2011, February) Bing Maps 3D. [Online]. http://www.bing.com/maps/help/ve3dinstall/

[132] NASA. (2011, February) World Wind. [Online]. http://worldwind.arc.nasa.gov/download.html

[133] Microsoft. (2011, February) Visual C# Developer Center. [Online]. http://msdn.microsoft.com/en-us/vcsharp/default

[134] NASA. (2011, February) World Wind Java SDK. [Online]. http://worldwind.arc.nasa.gov/java/

[135] Geosoft Inc. (2011, February) Dapple. [Online]. http://dapple.geosoft.com/

[136] Inc. The Institute for the Application of Geospatial Technology at Cayuga Community College. (2011, February) SERVIR Viz. [Online]. http://www.iagt.org/downloads.aspx/

[137] NASA. (2011, February) Blue Marble next generation. [Online]. http://earthobservatory.nasa.gov/Features/BlueMarble/

[138] NASA. (2011, February) The Landsat Program. [Online]. http://landsat.gsfc.nasa.gov/

[139] esri. (2011, February) ArcGis Engine. [Online]. http://www.esri.com/software/arcgis/arcgisengine/index.html

[140] Hong Liang, Raj Arangarasan, and Larry Theller, "Dynamic visualization of high resolution GIS dataset on multi-panel display using ArcGIS engine," *Computers and Electronics in Agriculture*, vol. 58, no. 2, pp. 174-188, September 2007.

[141] Tahsin Kurc, Ümit Çatalyürek, Chialin Chang, Alan Sussman, and Joel Saltz, "Visualization of Large Data Sets with the Active Data Repository,"

IEEE Computer Graphics and Applications, vol. 21, no. 4, pp. 24-33, July 2001.

[142] Michael D. Beynon et al., "Distributed processing of very large datasets with

[143] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz, "Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments," *Parallel Computing*, vol. 33, no. 7-8, pp. 497-520, August 2007.

[144] Wagner T. Correa, James T. Klosowski, Chrisopher J. Morris, and Thomas M. Jackmann, "SPVN: a new application framework for interactive visualization of large datasets," in *ACM SIGGRAPH 2007 courses*, New York, NY, USA, 2007.

[145] Peggy Li, "Supercomputing Visualization for Earth Science Datasets," in *Proceedings of 2002 NASA Earth Science Technology Conference*, Pasadena, CA, USA, 2002.

[146] M. Thiebaux, H. Tangmunarunkit, K. Czajkowski, and C. Kesselman, "Scalable Grid-based Visualization Framework," USC/Information Sciences Institute, Technical report ISI-TR-2004-592, 2004.

[147] Daniel S. Katz et al., "Accessing and Visualizing Scientific Spatiotemporal Data," in *16th International Conference on Scientific and Statistical Database Management*, Washington, DC, USA, 2004, pp. 107 - 110.

[148] P. Peggy Li, William H. Duquette, and David W. Curkendall, "RIVA: A Versatile Parallel Rendering System for Interactive Scientific Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 2, no. 3, pp. 186-201, September 1996.

[149] Rajvikram Singh, Byungil Jeong, Luc Renambot, Andrew E. Johnson, and Jason Leigh, "TeraVision: a distributed, scalable, high resolution graphics streaming system," in *IEEE International Conference on Cluster Computing*, 2004, pp. 391-400.

[150] Chong Zhang, Jason Leigh, Thomas A. DeFanti, Marco Mazzucco, and Robert L. Grossman, "TeraScope: Distributed visual data mining of terascale data sets over photonic networks," *Future Generation Computer Systems*, vol. 19, no. 6, pp. 935-943, August 2003.

[151] Robert Kooima, "Planetary-scale Terrain Composition," Computer Science, Graduate College of the University of Illinois, Chicago, Ph.D.

Dissertation 2008.

[152] Charles Zhang, "OptiStore: An On-Demand Data Processing Middleware for Very Large Scale Interactive Visualization," Computer Science, Graduate College of the University of Illinois, Chicago, Ph.D. Dissertation 2007.

[153] Larry L. Smarr, Andrew A. Chien, Tom DeFanti, Jason Leigh, and Philip M. Papadopoulos, "The OptIPuter," *Communications of the ACM*, vol. 46, no. 11, pp. 58-67, November 2003.

[154] Nut Taesombut et al., "Collaborative data visualization for earth sciences with the OptIPuter, Future Generation Computer Systems 22 (8," *Future Generations Computer Systems*, vol. 22, no. 8, pp. 955-963, January 2006.

[155] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer, "SETI@home: An Experiment in Public-Resource Computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56-61, November 2002.

[156] (2006, February) Predictor@Home. [Online]. http://www.boinc-wiki.info/Predictor@Home

[157] Ian Baker et al., "Atomistic protein folding simulations on the submillisecond time scale using worldwide distributed computing," *Peter Kollman Memorial Issue, Biopolymers*, vol. 68, p. 2003, 2003.

[158] David Stainforth et al., "Climateprediction.net: Design principles for public-resource modeling research," in *14th IASTED International Conference Parallel and Distributed Computing and Systems*, Cambridge, MA, USA, 2002, pp. 32-38.

[159] David P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," in *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Washington, DC, USA, 2004, pp. 4-10.

[160] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor-a hunter of idle workstations," in *8th International Conference on Distributed Computing Systems* , San Jose, CA, USA, 1988, pp. 104-111.

[161] Gilles Fedak, Cecile Germain, Vincent Neri, and Franck Cappello, "XtremWeb: A Generic Global Computing System," *IEEE International Symposium on Cluster Computing and the Grid*, pp. 582-587, May 2001.

[162] Nabil Abdennadher and Regis Boesch, "Towards a Peer-To-Peer Platform for High Performance Computing," *High Performance Computing and Grid in Asia Pacific Region, International Conference on*, vol. 0, pp. 354-361, July 2005.

[163] Thomas Fischer, Stephan Fudeus, and Peter Merz, "A Middleware for Job Distribution in Peer-to-Peer Networks," in *Applied Parallel Computing. State of the Art in Scientific Computing*, 2008, pp. 1147-1157.

[164] Luigi Gallo and Antonio Coronato, "Pervasive distributed volume rendering in a lightweight multi-agent platform," in *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, New York, NY, USA, 2009, pp. 750-755.

[165] Brian Vinter, "The Architecture of the Minimum intrusion Grid (MiG)," in *The 28th Communicating Process Architectures Conference*, Amsterdam, 2005, pp. 189-201.

[166] Brian Paul et al., "Chromium Renderserver: Scalable and Open Remote Rendering Infrastructure," *IEEE Transactions on Visualization and Computer Graphics*, vol. 14, no. 3, pp. 627-639, May 2008.

[167] Silicon Graphics Inc. (2007) OpenGL Vizserver. [Online]. http://www.sgi.com/

[168] Hewlett-Packard. (2011, January) HP remote Graphics Software. [Online]. http://h20331.www2.hp.com/hpsub/cache/286504-0-0-225-121.html

[169] Mercury International Technology. (2011, May) 3D ThinAnywhere. [Online]. http://www.thinanywhere.com/

[170] Stefan Eilemann, Maxim Makhinya, and Renato Pajarola, "Equalizer: A Scalable Parallel Rendering Framework," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 3, pp. 436-452, May-June 2009.

[171] Allen Bierbaum et al., "VR Juggler: A Virtual Platform for Virtual Reality Application Development," in *Proceedings of the Virtual Reality 2001 Conference (VR'01)* , Washington, DC, USA, 2001, pp. 89-96.

[172] Eric Charles Olson, "Cluster Juggler - PC cluster virtual reality," Iowa State University, Iowa, Master Thesis 2002.

[173] Jeremie Allard, Valérie Gouranton, Loïck Lecointre, Emmanuel Melin, and Bruno Raffin, "Net Juggler and SoftGenLock: Running VR Juggler with Active Stereo and Multiple Displays on a Commodity Component Cluster," in *Proceedings of IEEE Virtual Reality Conference*, Orlando, FL , USA, 2002, pp. 273-274.

[174] Nirnimesh, Pawan Harish, and P.J. Narayanan, "Garuda: A Scalable Tiled Display Wall Using Commodity PCs," *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 5, pp. 864-877, September-October 2007.

[175] Praveen Bhaniramka, Philippe C.D. Robert, and Stefan Eilemann, "OpenGL Multipipe SDK: A Toolkit for Scalable Parallel Rendering," in *IEEE Visualization*, vol. 0, Minneapolis, MN, USA, 2005, p. 16.

[176] Kai-Uwe Doerr and Falko Kuester, "CGLX: A Scalable, High-Performance Visualization Framework for Networked Display Environments," *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 3, pp. 320-332, March 2011.

[177] N.K. Krishnaprasad et al., "JuxtaView - a tool for interactive visualization of large imagery on scalable tiled displays," in *Proceedings of the 2004 IEEE International Conference on Cluster Computing*, vol. 0, Washington, DC, USA, 2004, pp. 411-420.

[178] Wagner T. Corrêa, James T. Klosowski, and Cláudio T. Silva, "Out-of-core sort-first parallel rendering for cluster-based tiled displays," in *Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization* , Aire-la-Ville, Switzerland, 2002, pp. 89-96.

[179] Steven Molnar, John Eyles, and John Poulton, "PixelFlow: high-speed rendering using image composition," *ACM SIGGRAPH Computer Graphics*, vol. 26, no. 2, pp. 231-240, July 1992.

[180] John S. Montrym, Daniel R. Baum, David L. Dignam, and Christopher J. Migdal, "InfiniteReality: A Real-time Graphics System," in *SIGGRAPH*, Los Angeles, CA, USA, 1997, pp. 293-302.

[181] Greg Humphreys and Pat Hanrahan, "A distributed graphics system for large tiled displays," in *Proceedings of the conference on Visualization '99: celebrating ten years*, Los Alamitos, 1999, pp. 215-223.

[182] Rudrajit Samanta, Jiannan Zheng, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh, "Load balancing for multi-projector rendering systems," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS*

workshop on Graphics hardware, New York, NY, USA, 1999, pp. 107-116.

[183] Carl Mueller, "The sort-first rendering architecture for high-performance graphics," in *Proceedings of the 1995 symposium on Interactive 3D graphics*, New York, NY, USA, 1995, pp. 75-84.

[184] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh, "Hybrid sort-first and sort-last parallel rendering with a cluster of PCs," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, New York, NY, USA, 2000, pp. 97-108.

[185] Daniel R. Schikore et al., "High-Resolution Multiprojector Display Walls," *IEEE Computer Graphics and Applications*, vol. 20, no. 4, pp. 38-44, July 2000.

[186] Kai Li, Matthew Hibbs, Grant Wallace, and Olga Troyanskaya, "Dynamic Scalable Visualization for Collaborative Scientific Applications," in *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, Washington D.C., USA, 2005, p. 225.

[187] Visualization and MultiMedia Lab, "Two Methods for driving OpenGL Display Walls," Department of Informatics, University of Zürich, Zürich, White Paper 2008.

[188] Shalini Venkataraman, Werner Benger, Amanda Long, Byungil Jeong, and Luc Renambot, "Visualizing Hurricane Katrina - Large Data Management, Rendering and Display Challenges," in *Proceedings of the 4th international conference on Computer graphics and interactive techniques in Australasia and Southeast Asia*, New York, NY, USA, 2006, pp. 209-212.

[189] Kunihiro Nishimura, Tomohiro Tanikawa, and Michitaka Hirose, "Human Genome Data Visualization Using a Wall Type Display," in *Proceedings of the 8th Asia-Pacific conference on Computer-Human Interaction*, Seoul, Korea, 2008, pp. 175-182.

[190] Zhenni Li, "Building a High-Resolution Scalable Visualization Wall," Auburn University, Auburn, Alabama, USA, Master Thesis 2006.

[191] Moohyun Cha, Jaikyung Lee, and Soonhung Han, "A Distributed Visualization Module and its Applications Using Tiled Display Wall," in *Proceedings of the 9th ACM SIGGRAPH Conference on Virtual-Reality Continuum and its Applications in Industry*, New York, NY, USA, 2010, pp. 63-66.

[192] So Yamaoka, Kai-Uwe Doerr, and Falko Kuester, "Visualization of high-resolution image collections on large tiled display walls," *Future Generation Computer Systems*, vol. 27, no. 5, pp. 498-505, May 2011.

[193] Sebastian Thelen, "Advanced Visualization and Interaction Techniques for Large High-Resolution Displays," in *Visualization of Large and Unstructured Data Sets - Applications in Geospatial Planning, Modeling and Engineering (IRTG 1131 Workshop)*, Dagstuhl, Germany, 2011, pp. 73-81.

[194] James H. Clark, "Hierarchical geometric models for visible surface algorithms," *Communications of the ACM*, vol. 19, no. 10, pp. 547-554, October 1976.

[195] (2011, April) squid-cache.org - Optimising Web Delivery. [Online]. http://www.squid-cache.org/

[196] S. Shepler et al. (2003, April) Network File System (NFS) version 4 Protocol. [Online]. http://tools.ietf.org/html/rfc3530

[197] (2011, February) Steel Bank Common Lisp. [Online]. http://www.sbcl.org/

[198] (2011, April) WRF - The Weather Research & Forecasting Model. [Online]. http://www.wrf-model.org

[199] notur. (2011, April) stallo - Resource description. [Online]. http://docs.notur.no/uit/stallo_documentation/user_guide/key-numbers-about-stallo

[200] Microsoft. (2011, March) COM: Component Object Model Technologies. [Online]. http://www.microsoft.com/com/default.mspx

[201] Chris Olston and Jennifer Widom, "Best-effort cache synchronization with source cooperation," in *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, New York, NY, USA, 2002, pp. 73-84.

[202] Open Geospatial Consortium Inc. (2011, April) Web Map Service. [Online]. http://www.opengeospatial.org/standards/wms

[203] Open Geospatial Consortium Inc. (2011, April) Web Feature Service. [Online]. http://www.opengeospatial.org/standards/wfs

[204] T. Brisco. (2011, February) DNS Support for Load Balancing. [Online].

http://tools.ietf.org/html/rfc1794

[205] Gene M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS '67 (Spring) Proceedings of the April 18-20, 1967, spring joint computer conference*, New York, NY, USA, 1967, pp. 483-485.

[206] NVIDIA. (2011, March) PHYSX. [Online]. http://www.nvidia.com/object/physx_new.html

[207] Microsoft. (2011, February) Understanding the Remote Desktop Protocol (RDP). [Online]. http://support.microsoft.com/kb/186607

Appendix A

Papers

In this section, all papers that the research presented in this dissertation is based on are included. All papers have been peer-reviewed and accepted for publication in international conferences and journals. For conference papers that later have been published in journals, only the journal paper is included. However, citations for both papers are shown.

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, 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.

A.2 Liberating the Desktop

Citation: Tor-Magne Stien Hagen, Espen Skjelnes Johnsen, Daniel Stødle, John Markus Bjørndalen, and Otto Anshus. Liberating The Desktop. In Proceedings of the First International Conference on Advances in Computer Human Interaction, ACHI, pages 89-94, February 2008.

Abstract

We report on a system supporting cross-platform mirroring of user-selectable regions from one or multiple computer desktops onto nearby network accessible projectors and displays (NADs). The purpose is a simple and flexible use of nearby display resources requiring no permanent installation of new software on the desktop computer.

The NAD system architecture consists of a NAD side and a desktop side. The desktop software is downloaded to the desktop computer on demand, from a web server running on the NAD. The desktop and NAD software handle the integration of user-selectable desktop regions and remote control between the desktop computer and the NAD. The system is implemented in Java 1.6.

At a resolution of 800 by 600 pixels the system supports mirroring of dynamic content at 38.6 FPS. At 1600 by 1200 pixels the refresh rate is 12.85 FPS. For static content such as images and slideshow presentations the system's bandwidth usage is within the capacity of a 11 Mbit/s wireless network. For dynamic content such as videos and games the system requires at least a 100 Mbit/s connection.

A.5 On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls

Citation: Tor-Magne Stien Hagen, Daniel Stødle, and Otto Anshus. On-Demand High-Performance Visualization of Spatial Data on High-Resolution Tiled Display Walls. In Proceedings of the International Conference on Information Visualization Theory and Applications, pp. 112–119, May 2010.

Abstract

Visualization of large data sets on high-resolution display walls is useful and can lead to new discoveries that would not have been noticeable on regular displays. However, exploring such data sets with interactive performance is challenging. This paper presents live data sets, a scalable architecture for visualization of large data sets on display walls. The architecture separates visualization systems from compute systems using a live data set containing data customized for the particular visualization domain. Experiments conducted show that the main bottleneck is the compute resources producing data for the visualization side. When all data is cached in the live data set, the main bottleneck (decoding images to create OpenGL textures and constructing geometry from raster data) is on the visualization side. On a 22 megapixel, 28 node display wall, the visualization system can decode 414.2 megapixels of images (19 frames) per second. However, the decoding is multi-threaded, and increased performance is expected using multi-core computers.

A.6 Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes

Citation: Bård Fjukstad, Tor-Magne Stien Hagen, Daniel Stødle, Phuong Hoai Ha, John Markus Bjørndalen and Otto Anshus. Interactive Weather Simulation and Visualization on a Display Wall with Many-Core Compute Nodes. To appear in the Proceedings of the 10th International Workshop on State-of-the-Art in Scientific and Parallel Computing, PARA, 2011.

Abstract

Numerical Weather Prediction models (NWP) used for operational weather forecasting are typically run at predetermined times at a predetermined resolution and a fixed geographical region. The period between each run is a function of waiting for observational data and the availability of compute resources. The resolution is a function of the geographical region, the available processing power and operational forecasting time constraints. The geographical region is defined by being a region with known need or interest for forecasts. These characteristics make it hard to interactively produce and visualize on-demand high-resolution forecasts for a small and arbitrarily located region. This paper documents a system achieving this, using a high-resolution tiled 22 mega pixel display wall, a 16 node PC cluster and a HP BL 460c blade server with two quad core processors. We document the performance characteristics experimentally. The results show that using 10 km resolution background data, the system produces a 6 hour forecast for a 117 x 123 km small region with 3 km resolution, in 3 minutes. Visualizing the forecast takes between 3 - 75 seconds. An informal survey among operational forecasters indicate that the majority is willing to wait up to minutes for higher resolution forecasts. This paper identifies and documents some of the bottlenecks and computational challenges created by combining interactivity and traditional batch oriented computing. The main bottlenecks in the system are identified as the execution time of the NWP and the preparation of data for visualization.

A.7 A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls

Citation: Tor-Magne Stien Hagen, Daniel Stødle, John Markus Bjørndalen and Otto Anshus. A Step towards Making Local and Remote Desktop Applications Interoperable with High-Resolution Tiled Display Walls. To appear in the proceedings of the 11th IFIP International Conference on Distributed Applications and Interoperable Systems, DAIS, 2011.

Abstract.

The visual output from a personal desktop application is limited to the resolution of the local desktop and display. This prevents the desktop application from utilizing the resolution provided by high-resolution tiled display walls. Additionally, most desktop applications are not designed for the distributed and parallel architecture of display walls, limiting the availability of such applications in these kinds of environments. This paper proposes the Network Accessible Compute (NAC) model, transforming personal computers into compute services for a set of display-side visualization clients. The clients request output from the compute services, which in turn start the relevant personal desktop applications and use them to produce output that can be transferred into display-side compatible formats by the NAC service. NAC services are available to the visualization clients through a live data set, which receives requests from visualization nodes, translates these to compute messages and forwards them to available compute services. Compute services return output to visualization nodes for rendering. Experiments conducted on a 28 nodes, 22 megapixel, display wall show that the time used to rasterize a 350 page PDF document into 550 megapixels of image-tiles and display these image-tiles on the display wall is 74.7 seconds (PNG) and 20.7 seconds (JPG) using a single computer with a quad core CPU as a NAC service. When increasing this into 28 quad core CPU computers, this time is reduces to 4.2 seconds (PNG) and 2.4 seconds (JPG). This shows that the application output from personal desktop computers can be made interoperable with high-resolution tiled display walls, with good performance and independent of the resolution of the local desktop and display.

Appendix B

WallScope – Additional Resources

This appendix includes a special curriculum based on and extending upon the WallScope system. It shows how the system can be extended to visualize data feeds describing objects in the real world. None of the work presented in this section has been presented or used in this dissertation.

B.1 Interactive Visualization of Data Feeds on High-Resolution Tiled Display Walls

Abstract

High-resolution tiled display walls provide orders of magnitude more resolution than regular desktop displays. The combination of size and resolution enables a person to get overviews by looking at the display wall from a distance, while at the same time being able to walk up close to look at details. This makes a display wall a promising device for usage in control room / surveillance contexts.

The work presented in this document extends WallScope, a system for interactive computation and visualization of data, and WallGlobe, a visualization system in the WallScope system, to support interactive visualization of local and remote data feeds describing objects in the real world. The system supports among others AISSAT, a system for surveillance of maritime activities in the High North. The high resolution of a display wall enables visualizations of data collected from data feeds with much higher fidelity than what is possible with a normal resolution display; instead of simply marking a boat's location on the map, it can be visualized using a high resolution graphical model.

The architecture of the system comprises a display-side and a compute-side. The display-side comprises a set of visualization clients, a set of state servers and a live data set. The state servers provide the visualization clients with view, time and object state. Based on the visualization clients' view, and the state provided by the state servers, visualization clients request data from the live data set, which is then used as part of the final rendering. The data set is live in that it accepts

client requests, which it translates into compute related messages and forwards to local and remote compute resources.

By separating state handling from the actual visualization, and using client-side interpolation of object transformations, the system is able to include new data feeds without any modifications to the visualization systems, while at the same time keeping network usage lower than a corresponding server-side state interpolation approach.

With an acceptable lower limit frame rate of five frames per second, the system is able to manage and display between 21 600 and 350 000 static objects, depending on the number of visible objects. When animating 22 500 objects the frame rate drops about one frame per second compared to displaying the same number of static objects. The system supports updating 14 516 objects per second from a single state server. The bottleneck in propagating object updates is the underlying centralized event system.

ISBN xxx-xx-xxxx-xxx-x