The 22 Megapixel Laptop

Daniel Stødle* John Markus Bjørndalen† Otto J. Anshus‡
Department of Computer Science
University of Tromsø, Norway

Figure 1: *(a) Configuring virtual displays to match a 28-tile display wall. (b) Extending a display to a portable device. (c) Using the 22 megapixel laptop. (d) One laptop extended with both a display wall and a portable device, for a total display area of 22 megapixels.*

Abstract

Displays are everywhere. To utilize them efficiently, we introduce the notion of the Network Accessible Display (NAD). A user can use displays on nearby computers as if they were physically connected to his computer, including displays on handheld devices and tiled display walls. We present a system adhering to the NAD-model, and demonstrate it by extending a laptop with up to 30 NADs with an area of 22 MPixels connected using both a wireless network and gigabit Ethernet. The system can support one display at 25 Hz and 30 displays at 1 Hz. Even with a refresh rate of only 1 Hz, the system remains useful for displaying relativly static content.

CR Categories: I.3.2 [Computer Graphics]: Graphics Systems—Distributed/network graphics; I.3.4 [Computer Graphics]: Graphics Utilities—Virtual device interfaces;

Keywords: Network Accessible Display, display wall

1 Introduction

The rapid progress in development of computer-related technologies has resulted in a commoditization of computers, storage, displays and other types of hardware. This development has given rise to approaches for building larger systems of cheap components, including hard disk RAIDs, Beowulf/NoW-style computer clusters and tiled displays. As this development continues, displays with processing power can be used as Network Accessible Displays (NADs), offering display services to nearby networked computers.

*e-mail: daniels@cs.uit.no
†e-mail: jmb@cs.uit.no
‡e-mail: otto@cs.uit.no

EDT 2007, San Diego, California, August 04, 2007.
© 2007 ACM 978-1-59593-669-1/07/0008 $5.00

We have built a software system enabling a desktop computer or laptop to utilize tens of displays as if they were directly connected to the computer.

Laptops can typically use both their built-in display and an external display. High-end workstations may be equipped with one or two quad-head graphic cards, capable of supporting up to eight displays in total. Wall-sized, high-resolution, tiled display walls have a pixel area of anywhere between 10 megapixels and 100 megapixels [Li et al. 2000; Stolk and Wielinga 2006], and are built using clusters of computers with displays or projectors. These approaches are lacking in several ways: (i) A laptop can only support one additional display, (ii) a workstation supporting eight displays is expensive, (iii) the number of supported displays is fixed, and (iv) using available, nearby displays from a laptop or workstation is impractical. Finally, a variety of "portable displays" - from watches, to mobile phones, PDAs and tablet computers - are not easily used as extended displays as there is no way of connecting them to computers using regular display cables. An increasing number support networking, however, potentially enabling them to act as NADs.

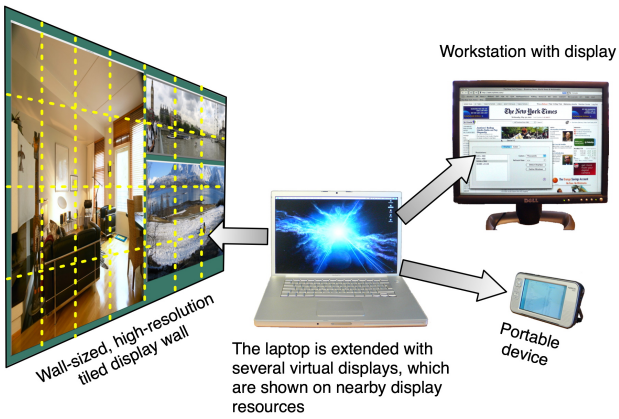

Figure 2: *Example of a laptop extending its local display to utilize the high resolution made available by a tiled display wall, as well as the display resources offered by a workstation and a portable device.*

Software like ZoneScreen, MaxiVista and Screen Recycler[1] lets

[1] http://www.zoneos.com/zonescreen.htm, http://www.maxivista.com

users extend their local desktops to a single other display. These products not only share the user's local display, but *extends it*, essentially making a remote display appear as a secondary local display. MaxiVista can support up to three additional displays, with a resolution up to 4800x2400 pixels. These solutions are lacking in (i) their ability to scale to many displays, (ii) no awareness of the physical arrangement of available display resources, and (iii) limited display resolution.

The above applications use remote desktop software, like Virtual Network Computing (VNC) [Richardson et al. 1998], THINC [Baratto et al. 2005] and Microsoft Remote Desktop, to transfer an extended desktop's pixels to a remote host. VNC shares displays by sending the shared display's pixels to clients, while in THINC better performance is achieved by more efficiently coding the drawing operations used to generate pixels. Another way of sharing display contents is to transmit only drawing operations ("draw string", "fill rectangle", etc.) as used in Microsoft Remote Desktop and the X Window System [Scheifler and Gettys 1986]. Our system makes use of a custom component similar to VNC, but with support for sharing several extended displays.

To support the model of NADs, the system creates virtual displays and shows them on displays ranging from portable displays to tiled display walls. The system extends the local desktop of a laptop running Mac OS X with up to 30 additional, virtual displays of arbitrary resolution[2]. The system then discovers nearby NADs and configures the virtual displays to utilize the available display resources. Our experimental testbed consists of a display wall comprised of 7x4 tiles for a total resolution of 7168x3072 pixels, several workstations and a Nokia N800 "internet tablet" acting as a portable display with a resolution of 800x480. Figure 2 illustrates this setup.

Our main contribution with this paper is the development of the Network Accessible Display model, and in particular: (i) a scalable display sharing model and implementation based on virtual displays, (ii) dynamic mapping of virtual displays to match available display resources, (iii) a system that will fit both the traditional view of displays connected directly to computers, and our vision of the display of the future - the NAD, and (iv) an evaluation of system's performance.

2 Design

The NAD system we developed consists of a number of distinct components: (i) A display service running on computers whose displays we wish to utilize, (ii) a GUI frontend, (iii) a VNC-like display sharing daemon and (iv) a kernel extension to create and maintain a set of virtual displays. Figure 3 illustrates the design.

Any computer wishing to provide its display as a NAD, runs a display service. The display service maintains properties related to the display(s) on the computer it runs[3], and exposes them to clients through a network-based discovery mechanism.

The GUI frontend runs on computers that want to utilize NADs. It discovers nearby display services and queries their properties. Currently available displays and their relative locations are presented to the user, before the user selects the displays he wishes to use. The frontend then configures the virtual displays and tells the daemon to push screen contents to the selected display services.

The display sharing daemon accepts commands from the GUI frontend. It sends the contents of the virtual displays to display services

and http://www.screenrecycler.com.

[2]Limited only by available memory; the system has been tested with resolutions up to 16384x6144.

[3]Bit depth, resolution, location and more.

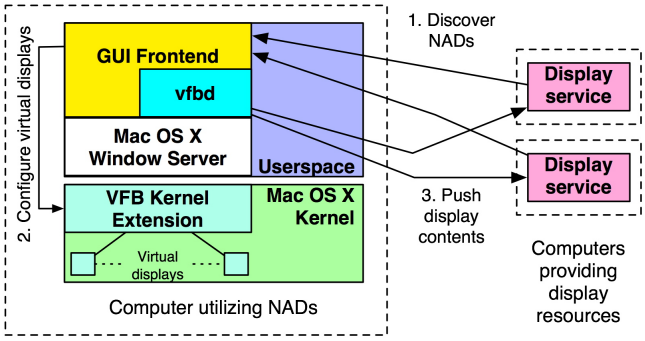

Figure 3: *The system design. Display services running on a set of computers are discovered by the GUI frontend, which proceeds to configure the number, resolution and arrangement of the virtual displays. It then instructs the display sharing daemon (vfbd) to push each display's contents to its associated display service.*

as raw pixel data. The first update consists of all pixels for a given virtual display, while further updates consist of pixels from areas that have changed on the virtual display (incremental updates).

The kernel extension creates a set of virtual displays when the computer boots. The virtual displays appear to the rest of the operating system as real, physically connected displays, but are in reality just a set of memory buffers. The GUI frontend communicates with the kernel extension to configure the number of virtual displays, and uses the window server to configure their resolution, bit depth and arrangement in relation to each other.

3 Implementation

The display service was implemented in C using SDL[4], and BSD sockets for network communication. It currently runs on Linux and Mac OS X. On startup, the display service is configured with the properties for the display resources it should provide. For a regular workstation with a single display, the properties consist of the local display's resolution and bit depth, as well as name and location. For display services running on tiled display walls, the configuration also includes information about the display wall, including the service's location in the grid of display tiles.

The GUI frontend was implemented in Objective-C using Cocoa on Mac OS X. It uses the CGDirectDisplay APIs in Mac OS X to configure virtual displays, including resolution and arrangement. The frontend uses property details from each display service when configuring the resolution and arrangement of virtual displays. Groups of display services that belong together, such as those running on a display wall, are presented together by the frontend, and not mixed with other "free-standing" displays.

The display sharing daemon uses the CGRemoteOperation APIs exported by Mac OS X' window server to access the raw pixels of the virtual displays. These APIs are also used to receive information about areas of the virtual displays where the pixels have changed, supporting incremental updates. The daemon performs run-length encoding of the pixels before sending them to connected display services, in order to reduce bandwidth usage. The daemon receives the network address for a display service from the frontend, then connects to the service and provides it with details about the virtual display. The service then starts accepting pixel data from the daemon.

[4]Simple Direct-Media Layer, a popular cross-platform library often used to develop games; http://www.libsdl.org/

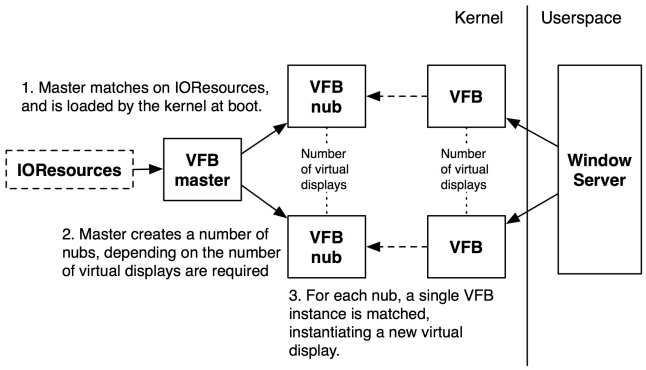

Figure 4: *The kernel extension design. The VFB master class is loaded at boot by the kernel, by matching on the class IOResources. It instantiates a number of VFB nubs. These nubs cause the kernel to start the matching procedure, and instantiate one instance of the VFB class for each nub. The virtual displays are then used by the window server when it starts up.*

The kernel extension, implemented in C++, consists of three classes: VFB (virtual framebuffer) master, VFB nub and VFB, as shown in Figure 4. The master accepts requests from userspace to configure properties of the virtual displays. In particular, it enables the GUI frontend to enable and disable virtual displays, without going through the window server[5]. The purpose of the nub is to provide an endpoint for Mac OS X' IOKit driver system to match and incorporate VFB instances into the kernel. When a nub is instantiated, it registers a service with IOKit. The VFB class matches on this service, making IOKit instantiate one instance of VFB for each nub created by the master.

The VFB class is a subclass of the IOKit class "IOFramebuffer." When it is instantiated, it allocates memory for a framebuffer of some pre-determined resolution (this can vary from instance to instance depending on configuration), before exposing its available resolutions and bit depths to the window server.

4 Evaluation

We document the performance of the system for different numbers of virtual displays. The hardware used was (i) a display cluster with 28 nodes (Intel Pentium 4 EM64T, 3.2 GHz, 2 GB RAM, Hyper-Threading enabled, NVIDIA Quadro FX 3400 with 256 MB Video RAM, running the Rocks Linux cluster distribution 4.0) connected to 28 projectors (1024x768, arranged in a 7x4 matrix), (ii) switched, gigabit Ethernet, and (iii) a MacBook Pro (2.33 GHz Intel Core 2 Duo, 3 GB RAM, Mac OS X 10.4.9).

4.1 Methodology

The MacBook Pro was configured with a number of virtual displays, where each virtual display had a resolution of 1024x768 at 32 bits per pixel. We varied the number of virtual displays between 1, 2, 4, 8, 16, 24 and 28. For each experiment, a window was created that fully covered all the virtual displays (this will be referred to as the "draw" process). The window was completely redrawn 300 times at an attempted rate of 10 Hz[6], after which statistics were gathered. To redraw the window, the draw process copies an image from memory to the window.

For each experiment, we measured the following statistics: (i) The total number of pixels updated by the display services, (ii) total number of bytes used to send pixel data to the display services, (iii) the CPU load both at kernel and user level for the draw process, display sharing daemon (vfbd) and Mac OS X window server.

4.2 Results

Figure 5 shows the target number of Mpixels updated per second compared to the system's actual update rate. With up to four displays, the system tracks the target update rate fairly well. Beyond four displays, the update rate is stable around 24 Mpixels/second, much less than the 30-210 MPixels/second needed to track the target rate. Using 24 virtual displays, the rate is 24.31 Mpixels/second, corresponding to a refresh rate of 1.35 Hz[7].

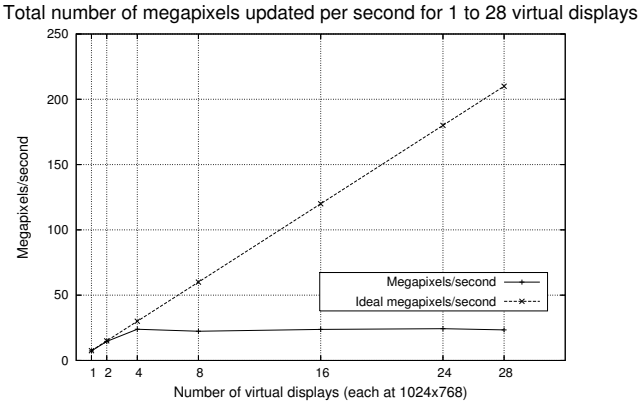

Figure 5: *The graph shows the actual update rate in megapixels/second, and compares it to the target update rate (10 full updates per second).*

Figure 6 shows the measured bandwidth. The bandwidth correlates well with the pixel update rate, with a peak bandwidth of 36.5 megabytes/second with 24 virtual displays. Figure 7 shows the kernel and user level CPU load for the different processes involved in generating and distributing data for the virtual displays. The majority of the CPU is used by the display sharing daemon, followed by the window server and finally the draw process. The combined load peaks at 175% with 24 displays (the MacBook Pro has a dual-core processor).

5 Discussion

The experiments demonstrate that there is a tradeoff between update rate and the size of the area being updated. In the experiments this area equals the combined resolution of the virtual displays. For smaller areas, the update rate can be quite high. As an example, a rate of 24 MPixels/second delivered to a virtual display with resolution 1024x768 corresponds to a refresh rate of 32 Hz. The same rate to a set of virtual displays with a total resolution of 7168x3072 (the size of the display wall used in the experiments) results in 1.14 Hz. Although not shown in the previous section, the best sustained refresh rate for full screen updates at 1024x768 in 16-bit color is 25

[5]The window server does not provide a mechanism to control whether a display is available or not.

[6]The actual rate was lower for most of the configurations, as discussed in the next section.

[7]24 virtual displays in a 6x4 grid results in a total resolution of 6144x3072 pixels; one full update is 18 megapixels, thus the refresh rate is 24.31/18 = 1.35.

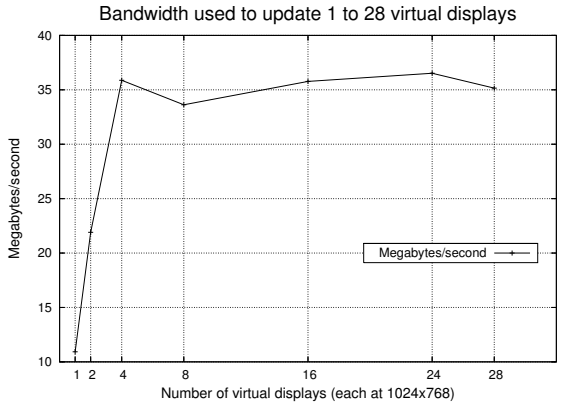

Figure 6: *The graph shows the bandwidth used to update the virtual displays.*

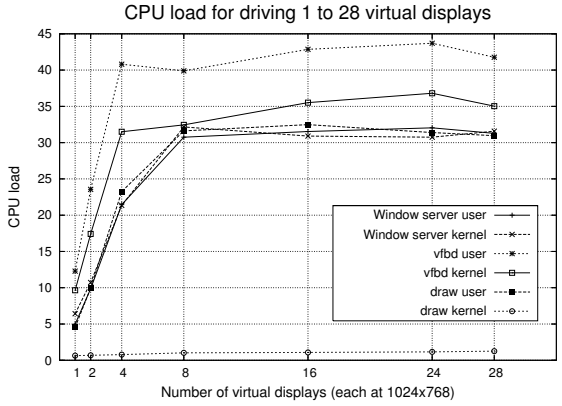

Figure 7: *The graph shows CPU load (in percent) for the display sharing daemon (vfbd), window server and draw process at both kernel and user level.*

Hz, and for 32-bit about 18 Hz. As the resolution increases, the system's performance goes down, but remains usable for mostly static content (images, documents, etc.).

The network is never saturated by the system - a transfer rate of 36 megabytes/second is less than half of the available bandwidth on a gigabit Ethernet. Thus, the network is not the main bottleneck. The CPU load measurements indicate that the main bottleneck is on the laptop. The load correlates well with the total resolution offered by the virtual displays, roughly doubling every time the resolution doubles, until the total CPU load goes beyond what the CPU can deliver at 4 virtual displays.

The draw process incurs little kernel level load, as it only copies pixel data from a buffer to its own window. The window server's CPU load tracks the load of the draw process well. Interestingly, this applies both to the window server's user *and* kernel level load, which indicates that the window server may be doing twice the work necessary to get the pixels to the virtual display (the data appears to be copied twice). The display sharing daemon spends about 55-60% of its time at user level, with the remaining time spent at kernel level. The time spent at user level is due to copying and compressing pixel data, while the time spent at kernel level comes from transferring pixel data over the network. The main bottleneck in the system as the total resolution offered by the virtual displays increases is copying data, and we hypothesize that improved performance can be achieved by eliminating redundant memory copies.

The Mac OS X window server is limited to 32 displays (virtual or not). In practice, the limit is 30, as there usually is a main display attached (a laptop's built-in display, for instance). In addition, the window server has a second, always-available virtual display with a resolution of 1x1 pixel which is always offline. The purpose of this display is unknown to the authors and to the authors' knowledge not documented. Even though the window server detects the presence of additional displays beyond the (practical) limit of 30, they are never used or exposed to clients. While the window server scales well, other parts of Mac OS X are not as scalable. Attempting to configure the virtual displays from System Preferences results in seeing an apparently random selection of at most 10 displays, and the display configuration menu only manages to show 16.

6 Conclusion

We have introduced the Network Accessible Display model, and presented the design and implementation of a system that adheres to the model. A NAD computer runs a display service that communicates with clients wishing to use the NAD. Clients discover NADs using a multicast-based discovery mechanism. We have used the system to extend a laptop with up to 30 virtual displays and map them to nearby physical displays, including a 22 Mpixel wall-sized, high resolution tiled display, and a 0.3 Mpixel portable device.

The bottleneck for increased resolution is copying pixel data locally on the client. When the number of pixels double, the client-side CPU load doubles. At a rate of 24 Mpixels/sec to the NADs, all available CPU is spent. We explain this by (i) load incurred compressing and transferring pixel data, and (ii) copying and compositing pixel data without graphics card hardware acceleration. Despite the low refresh rate for higher resolutions, the system is still useful for displaying static content like images and multiple documents.

Acknowledgements

Thanks to Tor-Magne S. Hagen and Espen S. Johnsen for discussions, and Ståle W. Nilsen for help with the video. Supported by the Norwegian Research Council, projects No. 159936/V30, SHARE - A Distributed Shared Virtual Desktop for Simple, Scalable and Robust Resource Sharing across Computer, Storage and Display Devices, and No. 155550/420 - Display Wall with Compute Cluster.

References

BARATTO, R. A., KIM, L. N., AND NIEH, J. 2005. THINC: a virtual display architecture for thin-client computing. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, ACM Press, New York, NY, USA, 277–290.

LI, K., CHEN, H., CHEN, Y., CLARK, D. W., COOK, P., DAMIANAKIS, S., ESSL, G., FINKELSTEIN, A., FUNKHOUSER, T., HOUSEL, T., KLEIN, A., LIU, Z., PRAUN, E., SAMANTA, R., SHEDD, B., SINGH, J. P., TZANETAKIS, G., AND ZHENG, J. 2000. Building and Using A Scalable Display Wall System. *IEEE Comput. Graph. Appl. 20*, 4, 29–37.

RICHARDSON, T., STAFFORD-FRASER, Q., WOOD, K. R., AND HOPPER, A. 1998. Virtual Network Computing. *IEEE Internet Computing 2*, 1, 33–38.

SCHEIFLER, R. W., AND GETTYS, J. 1986. The X window system. *ACM Trans. Graph. 5*, 2, 79–109.

STOLK, B., AND WIELINGA, P. 2006. Building a 100 Mpixel graphics device for the OptIPuter. *Future Gener. Comput. Syst. 22*, 8, 972–975.